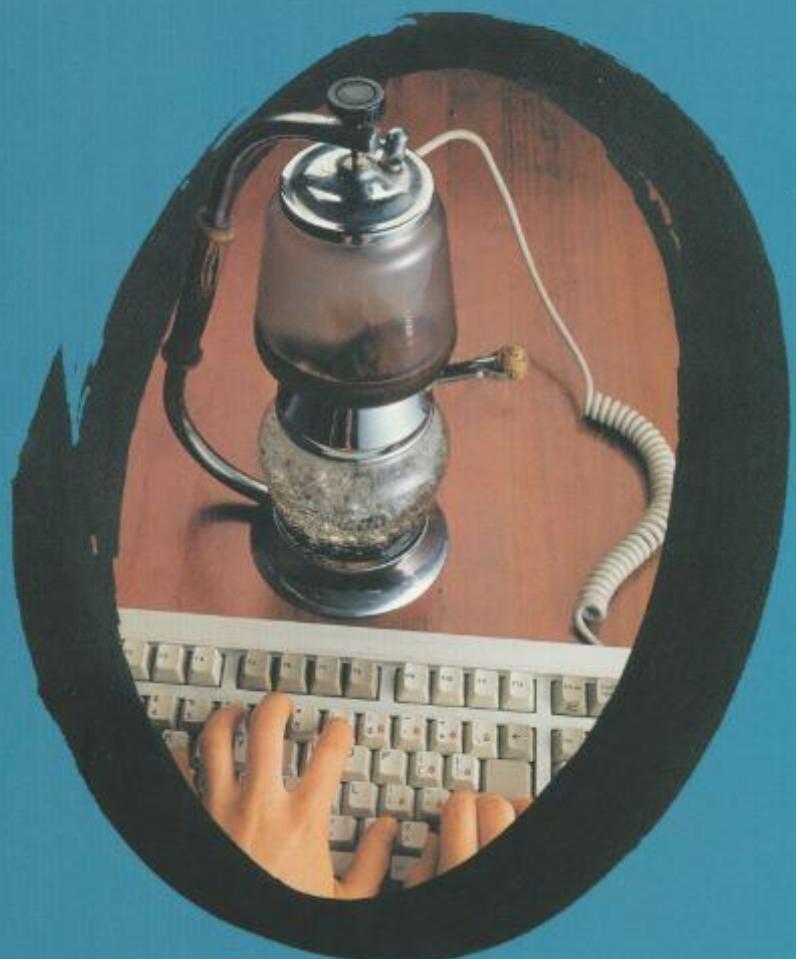


ANGSTER ERZSÉBET

2

OBJEKTUMORIENTÁLT TERVEZÉS ÉS PROGRAMOZÁS



Objektumorientált technikák + Grafikus felhasználi interfész

+ Állománykezelés + Végyes technológiák + Adatszerkezetek, kollekciók

Java

© Angster Erzsébet, 2002

Második, átdolgozott kiadás, 2004
(Első kiadás: 2002)

Minden jog fenntartva.

A könyv ezen online (PDF) változata szabadon terjeszthető változatlan formában.

Kiadja: 4KÖR Bt.

Szakmai lektor: Andor Gergely és Seres Iván
Nyelvi lektor: Seres Iván

Szerkesztő: Angster Erzsébet
Borítóterv: Bérczi Zsófia

ISBN 963 216 513 6

Akadémiai Nyomda, Martonvásár
Felelős vezető: Reisenleitner Lajos

Előszó

Az Objektumorientált tervezés és programozás, Java című könyv második kötete egyenes folytatása az elsőnek. Annak az volt a célja, hogy – konzolos környezetben – megismertesse az olvasót az objektumorientált tervezés, valamint a Java nyelv és osztálykönyvtár alapgondolataival és alapelemeivel; a második kötet főleg a grafikus felhasználói interfész készítésével és a programozás technológiájával foglalkozik. Azokat a témaikat igyekeztem összegyűjteni ebben a két kötetben, amelyek ismeretében átlagos, de már valóban élvezetes és színvonalas programok állíthatók össze. Legfőképpen az általános alapelvek bemutatására törekedtem – hogy az Olvasó már megtervezhessen és felhasználói interfésszel együtt létrehozhasson egyszerűbb alkalmazásokat.

A könyvet nem szántam Java referenciakönyvnek – az API osztályok ismertetésében nem is igyekeztem mindenre kitérni. Az érdeklődő Olvasó a teljes leírást megtalálhatja a referenciakönyvekben, a fejlesztőkörnyezet Helpjében és a JDK megfelelő dokumentumaiban.

Bizonyos mintafeladatok ebben a kötetben is konzolos környezetben íródtak – a grafikus felület programozása ugyanis sokszor elvonta volna a figyelmet az éppen bemutatandó téma lényegéről. Az itteni ismeretek később felhasználhatók egy-egy teljesebb alkalmazás összeállításához.

A programokat kipróbáltam az JDK 1.3.1-es verziójával JBuilder 7.0 környezetben, valamint a JDK 1.4.2-es verziójával JBuilder-X környezetben. A JBuilder alapváltozata ingyenes, és a Borland honlapjáról letölthető. Sajnos a projektállományok környezetfüggők, de a programokat más környezetben is használhatja.

Köszönetnyilvánítás

Elsősorban köszönöm férjemnek a sok segítséget és lelki támogatást. Köszönöm szüleimnek a sok hetes igazi alkotói lékgört Pécssett, az ősi házban.

Köszönöm Andor Gergőnek, hogy még az utolsó hajrában is olyan érdeklődéssel és készséggel olvasta és tette jobbá a könyvet. Köszönöm Seres Ivánnak is, hogy hasznos észrevételeivel tovább gazdagította a könyvet. Köszönöm hallgatóimnak, Zétényi Emesének, Smigura Antal-

nak, Tatár Zsoltnak, Csípő Ildikónak, Kuhn Balázsnak és Fehérvári Attilának, hogy még a megjelenés előtt kiszűrték a könyvből a legtöbb hibát.

2002. augusztus

2., átdolgozott kiadás

A könyv első megjelenése óta két év telt el. Igyekeztem kijavítani az első kiadás hibáit, és az összegyűlt tapasztalatok alapján átdolgoztam a könyv különböző részeit. Más hangsúlyt kapott az 1., Csomagolás, projektkelés című fejezet vége, és két új fejezet került a könyvbe: az **Applet** és a **Többszálú programozás**. Ezeknek a megtanulása után már internetes programokkal is „kedveskedhetünk” barátainknak, kollegáinknak. A könyv végén elkülönítve beiktattam egy **Feladatok** részt is. A legnagyobb feladat ugyan eddig is benne volt a könyvben, de az Olvasók észrevételei szerint elég jó „rejtve”. A Feladatok fejezetéhez két kisebb feladatot is hozzáadtam: ezekkel próbára teheti magát az Olvasó. Visszajelzések szerint ehhez hasonló feladatokat 2 óra alatt meg lehet oldani.

Külön szeretnék köszönetet mondani Seres Ivánnak a 2. kiadás átdolgozásában vállalt áldozatos munkájáért és tanácsaiért.

2004. augusztus

Kedves Olvasó!

Kérem, tanulmányozza át a Tanulási és jelölési útmutatót!

A könyvvel kapcsolatos észrevételeit szívesen fogadom a következő címen:

angster.erzsebet@gmail.com

És ha a sok programozás miatt nem tudja jól irányítani a testét, akkor látogasson el ide:

<https://angstererzsebet.hu>



Tartalomjegyzék

I. RÉSZ. OBJEKTUMORIENTÁLT TECHNIKÁK	1
1. Csomagolás, projektkezelés.....	3
1.1. Csomagolás.....	3
1.2. Projektkezelés a JBuilderben.....	11
1.3. JAR-állomány készítése	19
1.4. Java program futtatása	25
Tesztkérdések	28
Feladatok	30
2. Öröklődés.....	31
2.1. Az öröklődés fogalma.....	31
2.2. Mintafeladat – Hengerprogram.....	34
2.3. Az objektumreferencia statikus és dinamikus típusa	43
2.4. Az utódosztály adatai és kapcsolatai	47
2.5. Metódus felülírása, dinamikus és statikus kötés	50
2.6. this és super referencia	53
2.7. this és super konstruktorok – konstruktorok láncolása	54
2.8. Polimorfizmus	56
2.9. Absztrakt metódus, absztrakt osztály	57
2.10. Láthatóság	62
2.11. Összefoglalás – metódusok nyomkövetése.....	63
Tesztkérdések	65
Feladatok	67
3. Interfészek, belső osztályok.....	69
3.1. Interfész	69
3.2. Belső osztály.....	75
3.3. Névtelen osztály	79
Tesztkérdések	83
Feladatok	84
4. Kivételkezelés	85
4.1. Kivételek, hibák.....	85
4.2. Kivételek keletkezése és szándékos előidézése – throw	89
4.3. A kivétel továbbadása – throws	94

4.4.	A kivétel elkapása, kezelése	96
4.5.	Saját kivételek használata	99
	Tesztkérdések	101
	Feladatok.....	102
II. RÉSZ. GRAFIKUS FELHASZNÁLÓI INTERFÉSZ		103
5. A felhasználói interfész felépítése		105
5.1.	Komponensek és tulajdonosi hierarchiájuk	106
5.2.	AWT- és Swing-osztályhierarchia	109
5.3.	Swing mintaprogram.....	114
5.4.	Jellemzők	119
5.5.	Pont, méret, téglalap	119
5.6.	Koordinátarendszer	122
5.7.	Szín, betű	124
5.8.	Az absztrakt JComponent osztály	128
5.9.	Container osztály	133
5.10.	java.awt.Window osztály	135
5.11.	JFrame osztály	136
	Tesztkérdések	139
	Feladatok.....	140
6. Elrendezésmenedzserek		141
6.1.	Az elrendezésmenedzserek tulajdonságai	141
6.2.	FlowLayout – sorfolytonos elrendezés	144
6.3.	GridLayout – rácsos elrendezés	146
6.4.	BorderLayout – határ menti elrendezés	148
6.5.	JPanel, az összefogó konténer.....	150
	Tesztkérdések	153
	Feladatok.....	154
7. Eseményvezérelt programozás		157
7.1.	Mintaprogram	157
7.2.	Eseményosztályok	160
7.3.	Alacsony és magas szintű események.....	163
7.4.	Eseménydelegációs modell	166
7.5.	A felhasználói felület tervezése	169
7.6.	Eseményadapterek	175
	Tesztkérdések	179
	Feladatok.....	180
8. Swing-komponensek.....		181
8.1.	Swing-konstansok – SwingConstants	182
8.2.	Címke – JLabel	183
8.3.	A gombok öse – AbstractButton.....	186
8.4.	Nyomógomb – JButton	187
8.5.	Jelölőmező – JCheckBox	193
8.6.	Rádiógomb – JRadioButton, csoportosítás	196
8.7.	Kombinált lista – JComboBox.....	199
8.8.	MVC-modell, dokumentumkezelés	202

8.9.	A szövegek öse – JTextField	205
8.10.	Szövegmező – JTextField	207
8.11.	Szövegterület – JTextArea	210
8.12.	Lista – JList	214
8.13.	Görgetősáv – JScrollPane	220
8.14.	Menüsor – JMenuBar	224
8.15.	Ablak – JWindow	229
8.16.	Dialógusablak – JDialog, kész dialógusok – JOptionPane	230
8.17.	Időzítő – Timer	240
	Tesztkérdések	243
	Feladatok	244
9.	Grafika, képek	251
9.1.	Rajzolás	251
9.2.	Mintaprogram	254
9.3.	A Graphics osztály	256
9.4.	Sokszög rajzolása – a Polygon osztály	260
9.5.	Képek – az absztrakt Image osztály	263
	Tesztkérdések	269
	Feladatok	270
10.	Alacsony szintű események	273
10.1.	Az alacsony szintű események osztályhierarchiája	273
10.2.	Komponensesemény – ComponentEvent	275
10.3.	Fókuszesemény – FocusEvent	277
10.4.	Billentyűesemény – KeyEvent	285
10.5.	Egéresemény – MouseEvent	293
	Tesztkérdések	303
	Feladatok	304
11.	Belső eseménykezelés, komponensgyártás	307
11.1.	Esemény keletkezése és életútja	307
11.2.	Események feldolgozása	311
11.3.	Komponensgyártás – feladatok	316
	Tesztkérdések	322
	Feladatok	323
12.	Applet	325
12.1.	Mi az applet?	325
12.2.	Applet-futtató környezetek	330
12.3.	Az Applet és a JApplet osztály	334
12.4.	Az alkalmazás átalakítása appletté	336
12.5.	Hanglejátszás – AudioClip	338
12.6.	Az applet életciklusa	343
12.7.	Az applet paraméterei	345
12.8.	Biztonság	346
	Tesztkérdések	347
	Feladatok	348

III. RÉSZ. ÁLLOMÁNYKEZELÉS	351
13. Állományok, bejegyzések.....	353
13.1. A java.io csomag.....	353
13.2. Útvonalak.....	355
13.3. A File osztály	357
13.4. Állományműveletek.....	363
13.5. Szűrés – FilenameFilter interfész.....	365
13.6. Állománykiválasztó dialógus – JFileChooser	367
13.7. Könyvtár felderítése rekurzióval.....	372
Tesztkérdések	374
Feladatok.....	375
14. Folyamok.....	377
14.1. A folyam fogalma	377
14.2. Bájtfolyam	381
14.3. Karakterfolyam, szöveges állomány	387
14.4. Adatfolyam	394
14.5. Pufferező folyam.....	398
14.6. Objektumfolyam	402
Tesztkérdések	412
Feladatok.....	413
15. Közvetlen hozzáférésű állomány	415
15.1. Állományszervezési és -hozzáférési módok	415
15.2. A RandomAccessFile osztály	418
Tesztkérdések	423
Feladatok.....	423
IV. RÉSZ. VEGYES TECHNOLÓGIÁK	425
16. Rekurzió	427
16.1. A teljes indukció elve	427
16.2. Rekurzív feladat.....	428
16.3. Rekurzív eljárás, függvény	430
16.4. A rekurzió megállítása	432
16.5. Feladat – Hanoi tornyai.....	433
16.6. Feladat – Gyorsrendezés	436
Tesztkérdések	438
Feladatok.....	438
17. Többszálú programozás.....	439
17.1. A programszál fogalma.....	439
17.2. A Thread osztály és a Runnable interfész	444
17.3. Szinkronizáció: wait, notify	446
17.4. Programszálak appletben	453
Tesztkérdések	456
Feladatok.....	456

18. Nyomtatás	459
18.1. A nyomtatás technikája	459
18.2. Mintaprogram – PrintHello	462
18.3. Printable interfész, PrinterJob osztály	463
18.4. Oldalformázás – PageFormat osztály	467
18.5. Megjelenítés és nyomtatás	471
Tesztkérdések	473
Feladatok	474
19. Hasznos osztályok	477
19.1. Időpont – Date	477
19.2. Környezet – Locale	480
19.3. Időeltolás – TimeZone	482
19.4. Naptár – GregorianCalendar	484
19.5. Dátumformázás – DateFormat	488
19.6. Számformázás – NumberFormat	490
19.7. Megfigyelés – Observer, Observable	491
19.8. Klónozás – Cloneable	497
19.9. Rendszerjellemzők – System	502
19.10. Külső program futtatása – Runtime	503
Tesztkérdések	505
Feladatok	506
V. RÉSZ. ADATSZERKEZETEK, KOLLEKCIÓK	509
20. Klasszikus adatszerkezetek	511
20.1. Az adatszerkezetek rendszerezése	511
20.2. Absztrakt tárolók	516
20.3. Tömb	521
20.4. Tábla	522
20.5. Verem	525
20.6. Sor	526
20.7. Fa	527
20.8. Irányított gráf, hálózat	531
Tesztkérdések	534
Feladatok	535
21. Kölleció keretrendszer	537
21.1. A köllecíó keretrendszer felépítése	538
21.2. A Collection interfész és leszármazottai	541
21.3. A HashSet osztály – hasítási technika	545
21.4. A TreeSet osztály – Comparator	549
21.5. Iterátor	554
21.6. A List interfész implementációi	556
21.7. A Map és a SortedMap interfész	559
21.8. A Hashtable osztály	561
21.9. A TreeMap osztály	563
Tesztkérdések	566
Feladatok	567

FELADATOK	569
1. feladat: Témák rögzítése	571
2. feladat: Csempetervező	575
3. feladat: Címjegyzék	581
4. Esettanulmányok	588
FÜGGELÉK	589
A tesztkérdések megoldása	591
Irodalomjegyzék.....	593
Tárgymutató	595

Tanulási és jelölési útmutató

Tanulási útmutató

A tananyag a könyv első kötetének ismeretére épül.

A könyv távoktatási célra is használható, vagyis **önállóan is meg lehet tanulni**. A téma környezetében kellően részletes; a kezdők persze lassabban fognak benne haladni – nekik többször is el kell olvasni egy-egy fejezetet; azok, akik már tudnak programozni, gyorsabban is haladhatnak. Egy tanfolyam mindenéppen rövidebbé teszi a megtanuláshoz szükséges időt.

A könyvön kívül Önnek a következőkre szüksége lesz:

- ◆ **A könyv forráskód mellékletére (javaproj.zip):** Ebben van a könyvben tárgyalt feladatok forráskódja, illetve a fejezetvégi feladatok megoldása. A melléklet letölthető az Internetről; a címét ott találja a könyv hátán.
- ◆ **Fejlesztőkörnyezetre:** A JBuilder alapverziója szintén letölthető az Internetről és ingyenesen használható. A könyv majdnem független a fejlesztőkörnyezettől; de ha a környezetre is példát mutat, akkor mindenkor a JBuilderre támaszkodik.

Ajánlott tanulási sorrend – kihagyható részek

A könyvnek vannak mindenéppen elsajátítandó részei, más részei viszont átugorhatók. Részletesebben:

- ◆ **I. rész** (Objektumorientált technikák): Ez a rész megalapozza a grafikus felhasználói interfész készítéséhez szükséges ismereteket. Ha itt nem sikerül mindenötökéletesen megértenie, akkor később a II. részről nyugodt lélekkel lapozzon vissza ide, és tisztázza, amit még kell!
- ◆ **II. rész** (Grafikus felhasználói interfész): Ennek a kötetnek ez a java. Az 5–8. fejezet tartalmazza a legfontosabb ismereteket – ezek nélkül egy egyszerű grafikus alkalmazást sem fog tudni Javában megírni. A 9–12. fejezet speciális feladatokat tárgyal: a rajzolást, a képeket, a billentyűkezelést, appleteket – ezekre nemigen támaszkodnak a könyv későbbi fejezetei.
- ◆ **III. rész** (Állománykezelés): A 13. fejezetben vannak az állománykezeléssel kapcsolatos legfontosabb ismeretek. A 14. fejezet, a Folyamok egy kicsit nehezebben emészt-

hető, de bizonyos részeire gyakran szükség lehet. A 15. fejezet nem nehéz, és használható ismereteket ad.

- ◆ **IV. rész** (Vegyes technológiák): A 16–19. fejezet egymástól független témákat dolgoz fel, s azok nem hiányozhatnak az igényes programozó eszköztárából (egy jó program nyomtatni is tud, több szalon fut, és figyel a nemzetközi beállításokra). A könyv többi része nem támaszkodik erre a tudásanyagra.
- ◆ **V. rész** (Adatszerkezetek, kollekciók): Ez a rész szinte teljesen független a többitől. Ha konténerre van szükség, tömbbel és `vector`-ral – ha esetleg nehézkesen is – szinte bármilyen feladat megoldható. A kollekció keretrendszer használata azonban hatékonyá és „profivá” teszi a programot.

Ellenőrizze tudását!

Minden fejezet végén **tesztkérdéseket** és **feladatokat** talál: ellenőrizheti velük a tudását. A tesztkérdések megoldását a könyv függelékében nézheti meg, a gyakorlati feladatok megoldását pedig a forráskód mellékletben. Ha a teszteket már sikerült megoldania, akkor a feladatok megoldásával mélyítse tovább a tudását! A feladatokat **nehézségi szint szerint osztályoztuk**:

- ◆ **(A)** Rutinfeladat. Ha ezt nem tudja megoldani, ne menjen tovább!
- ◆ **(B)** Könnyű feladat. Néhány alapfogást kombinálni kell, de valószínűleg nem okoz majd sok fejtörést.
- ◆ **(C)** Nehezebb feladat. Gondolkodtatón, összetettebb vagy munkás feladat.

Ha a feladatot megoldotta (megtervezte és lekódolta), akkor vesse össze az eredményt az Internetről letölthető megoldással!

Tanulmányozza a könyv mellékletében található **Esettanulmányokat** is!

Jelölési útmutató

A tankönyv a következő jelöléseket használja:

Ilyen dobozokban olvashatók **tömören összefoglalva az elméleti ismeretek**.

A normál szövegen magyarázatokat talál; azért van szükség rájuk, mert a doboz túlságosan „tömény”.

Megjegyzés: Ez csak megjegyzés, sokkal kevésbé fontos, mint a többi információ. Mégsem árt elolvasni.

● Az ilyen jelölés mellett valami „bomba” van elrejtve; érdemes résen lennie!

Szintaktika

A Java deklarációk, utasítások szintaktikai szabályait így adjuk meg:

```
<típus> <változó1>[=<kifejezés1>] [ ,<változó2>=<kifejezés2>... ] ;
```

A <> „kacsacsörben” kitöltendő dolgok vannak; a [] nagy zárójelbe írt részeket nem kötelező megadni. Ahol három pontot lát, ott folytatható a felsorolás. Egy példa:

```
int a=b=12*Math.PI, c=1, d;
```

Java metódusok

A Java szintaktika szerinti metódusokat (például a Java osztálykönyvtár metódusait) úgy adjuk meg, hogy a metódusok fejét egy kis háromszöggel jelöljük meg, s utána közöljük a metódus leírását:

► void metodus(int par) // ez egy metódusfej

Itt következik a metódus magyarázata.

Osztályok megadása

A különféle osztályok bemutatásakor csak a fontosabb, publikus deklarációkat ismertetjük, a következő csoportokban:

- ◆ mezők (adatok)
- ◆ jellemzők (beállítható és lekérdezhető adatok)
- ◆ konstruktörök
- ◆ metódusok

A public módosítót sehol sem írjuk ki.

Feladatok és megoldások

Feladat

Az itt megfogalmazott feladatot meg is oldjuk...

Ha kell, akkor a feladatspecifikáció alapján meg is tervezzük a programot. Mindig megadjuk a megoldás teljes forráskódját, és szükség szerint kiegészítjük a futási eredménnyel és a program elemzésével. A magyarázatban a forráskód megjegyzéseiben megadott számokra hivatkozunk.

Jó tanulást, örömteli programozást!

Angster Erzsébet



I.

I. OBJEKTUMORIENTÁLT TECHNIKÁK

1. Csomagolás, projektkezelés
2. Öröklődés
3. Interfészek, belső osztályok
4. Kivételkezelés

II. GRAFIKUS FELHASZNÁLÓI INTERFÉSZ

5. A felhasználói interfész felépítése
6. Elrendezésmenedzserek
7. Eseményvezérelt programozás
8. Swing-komponensek
9. Grafika, képek
10. Alacsony szintű események
11. Belső eseménykezelés, komponensgyártás
12. Applet

III. ÁLLOMÁNYKEZELÉS

13. Állományok, bejegyzések
14. Folyamok
15. Közvetlen hozzáférésű állomány

IV. VEGYES TECHNOLÓGIÁK

16. Rekurzió
17. Többszálú programozás
18. Nyomtatás
19. Hasznos osztályok

V. ADATSZERKEZETEK, KOLLEKCIÓK

20. Klasszikus adatszerkezetek
21. Kollekció keretrendszer

FELADATOK

FÜGGELÉK

- A tesztkérdések megoldása
Irodalomjegyzék
Tárgymutató

1. Csomagolás, projektkezelés

A fejezet pontjai:

1. Csomagolás
 2. Projektkezelés a JBuilderben
 3. JAR-állomány készítése
 4. Java program futtatása
-

A fejezetben egy egyszerű mintaprogramon keresztül mutatjuk be a csomagolás és a projektkezelés technikáját. Osztályokat és interfések fogunk egymásba ágyazott csomagokba tenni. Először egy mintaalkalmazás osztálydiagramján megmutatjuk a csomagkészítés szabályait. Ezután tisztázzuk a projekt fogalmát, majd egy JBuilder projekt segítségével implementáljuk mintaprogramunkat. Végül az egész alkalmazást egy futtatható JAR-állományba tesszük, hogy azt igényes módon adhassuk át a megrendelőnek.

1.1. Csomagolás

A **csomag** (package) valamilyen szempontból összetartozó osztályok és interfések csoportja. Csomagok használatával a program áttekinthetővé válik, egyszerűbb lesz a deklarációk azonosítása és védelme. A csomagok egymásba ágyazhatók, s így tetszőleges mélységű csomagstruktúra építhető fel. Egy szintre akárhány osztály, interfész, illetve csomag tehető.

A csomag fogalma logikai és fizikai szinten is értelmezhető: az osztálydiagramok, a forrásállományok és a lefordított bájtkódok mind csomagolhatók.

Egy alkalmazást a következők miatt szokás csomagokra bontani:

- ◆ A sok forrásállomány és bájtkód áttekinthetetlen, ezért a logikailag összetartozó elemeket csoportosítjuk. Szokásos hármas felosztás például a következő: felhasználói interfések (az UML-ben határosztályok), adatok (információhordozó osztályok) és alkalmazáslogika (vezérlő osztályok).
- ◆ Más alkalmazás is szeretné használni az osztályokat. A csomagok (könyvtárstruktúrába szervezett osztályok) könyvtár formájában vagy egyetlen jár állományba tömörítve

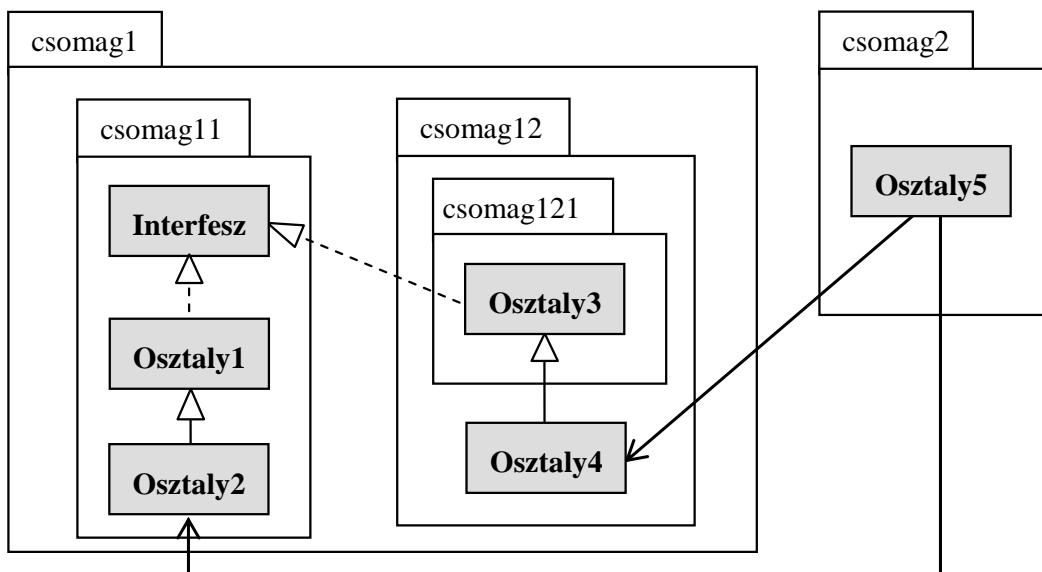
I. RÉSZ. OBJEKTUMORIENTÁLT TECHNIKÁK

adhatók át további felhasználásra. Ilyen csomagrendszer a JDK szabványos osztálygyűjteménye, az `rt.jar`, valamint a könyv mellékleteként készült `javalib` osztálykönyvtár (`javalib.jar`).

Feladat – CsomagApp

Az 1.1. ábrán egy osztálydiagram látható. A fő (a main metódust tartalmazó) osztály az `Osztaly5`. Készítsük el a terv forráskódjának vázát! Az osztály- és interfészdeklarációk belsejét nem kell megírni, hiszen nincs is konkrét feladat!

Megjegyzés: A feladat csak felületesen érinti az öröklést és interfészkészítést – ezekkel a témaikkal mélyebben a 2. és 3. fejezet foglalkozik majd. Most a csomagolási és projektkészítési technikákra koncentrálunk.



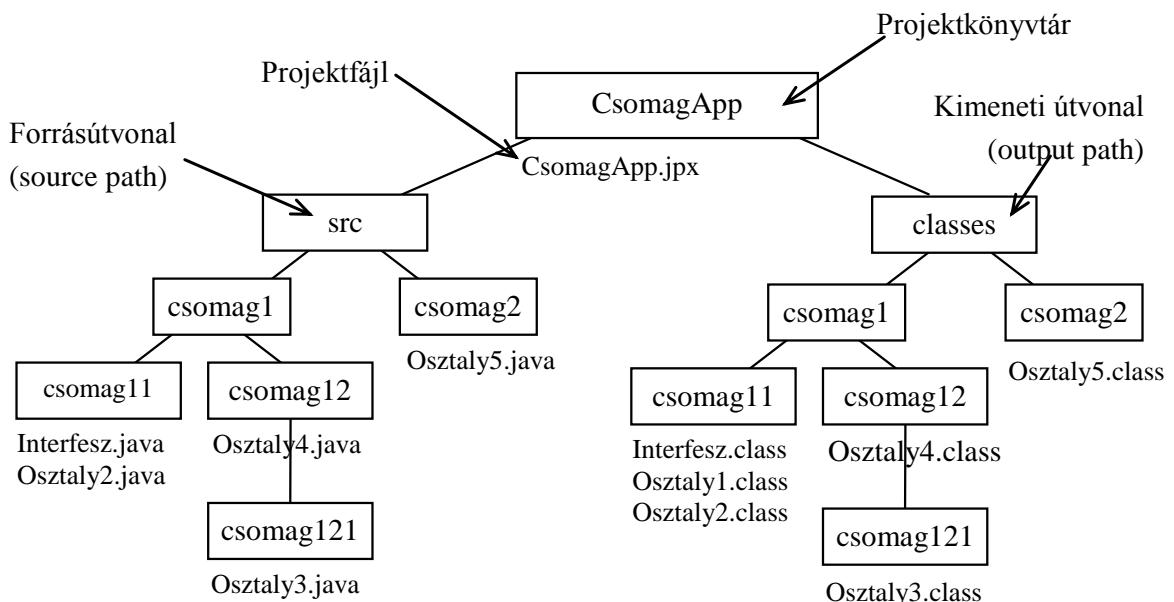
1.1. ábra. A CsomagApp osztálydiagramja

Az első – a legfelső – szinten két csomag van: `csomag1` és `csomag2`. A `csomag1` két csomagot tartalmaz, a `csomag11`-et és `csomag12`-t. A `csomag121`-et a `csomag12`-ben helyeztük el. A csomagok osztályai, illetve interfészei között implementálási (üres háromszögfejű, szaggattott nyíl), öröklési (üres háromszögfejű, folytonos nyíl) és társítási (normális nyíl) kapcsolatok vannak.

Megjegyzés: Az áttekinthetőség kedvéért itt a csomagazonosítót így képezzük: szülőcsomag azonosítója + egyjegyű sorszám. Az azonosítóban szereplő számjegyek száma jelzi tehát az egymásba ágyazás szintjét: `csomag1` és `csomag2` az első, `csomag11` és `csomag12` a második szinten van, a `csomag121` a harmadik szinten.

Könyvtárstruktúra

A terv alapján el kell készítenünk a forrásállományokat. El kell döntenünk, hogy pontosan hány forrásállomány készüljön, milyen deklarációk szerepeljenek bennük, hogyan nevezzük a forrásállományokat, és hol helyezzük el őket a tárolón.



1.2. ábra. Az alkalmazás könyvtárstruktúrája

A csomaghierarchiának egy egyértelmű könyvtárstruktúra feleltethető meg a tárolóeszközön. A csomag- és könyvtárnevek párhuzamba állíthatók egymással, a fordítási egységek (forrásállományok) és a lefordított bájtkódok tárolásának is ezt a könyvtárstruktúrát kell követnie (1.2. ábra)!

A fordítási egységek (forrásállományok) gyökérkönyvtárát **forrásútvonalnak** (source path), más szóval forráskönyvtárnak nevezzük, a lefordított bájtkódok gyökérkönyvtárát pedig **kimeneti/cél útvonalnak** (output path/destination path), másképpen kimeneti/cél könyvtárnak. A forrás- és a kimeneti könyvtárrendszer tükröképe egymásnak. A kimeneti könyvtár struktúráját a fordító automatikusan készíti el a forráskódokban megadott csomagdeklaráció (package) alapján. A forráskódok könyvtárrendszerét a csomaghierarchia alapján kell kialakítani; a fejlesztőkörnyezet megköveteli a tükrözést. A szimmetria áttekinthetővé teszi a programot.

A fordítással interfészenként és osztályonként egy-egy class állomány keletkezik a kimeneti könyvtárban.

Az egy csomagban szereplő deklarációk fordítási egységeit szükségképpen ugyanabba a könyvtárba kell tenni! Az osztályokat és az interfészeket ajánlott külön fordítási egységek-

ben elhelyezni, de ez nem kötelező. A forrásútvonal és a kimeneti útvonal egybeeshet éppen (ha kis programokról van szó), de jobb, ha különbözik egymástól.

Mintaprogramunkban a forráskód gyökérkönyvtára a `CsomagApp/src`, a lefordított bájtkódok gyökérkönyvtára pedig a `CsomagApp/classes`. A `CsomagApp` alkalmazás könyvtárstruktúráját az 1.2. ábra mutatja. Figyelje meg, hogy a `csomag11` könyvtárban nem szerepel az `Osztaly1.java` állomány! Ez azért van, mert az `Osztaly1` és `Osztaly2` osztályt egy forrásállományba, az `Osztaly2.java`-ba tettük.

A csomag azonosítása

Egy csomagot a szülőcsomagjával minősítünk úgy, hogy a szülőcsomag és a csomag neve közé az UML-ben két kettőspontot, a Javában pedig egy pontot teszünk:

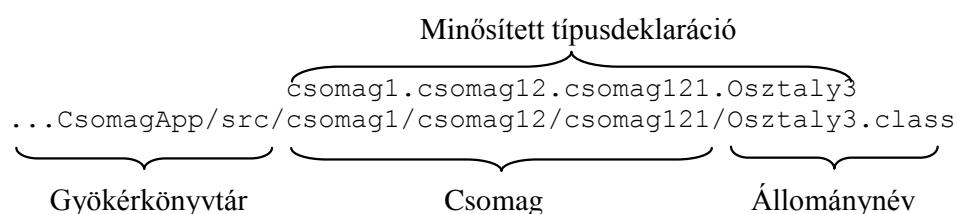
UML:	<code>csomagnév [:: csomagnév [:: csomagnév] ...]</code>
Java:	<code>csomagnév [. csomagnév [. csomagnév] ...]</code>

Elnevezési konvenció: a csomag neve egyetlen – csupa kisbetűs és egyes számban álló – szó.

Például: Az 1.1. ábrán látható `Osztaly3` osztályt közvetlenül tartalmazó csomag azonosítója Javában:

`csomag1.csomag12.csomag121`

A csomag azonosítójában levő pont a könyvtárstruktúrában a könyvtárelválasztó jelnek felel meg; ez a jel Unixban a `/`, Windowsban a `\`. A minősített típusdeklaráció (osztály- vagy interfészdeklaráció) és az állományspecifikáció közötti összefüggést az 1.3. ábra mutatja.



1.3. ábra. Típusdeklaráció, állományspecifikáció

Megjegyzések:

- A könyvtárstruktúrák jelölésében a könyvtárak elválasztására a `/` jelet fogjuk használni (Unix-konvenció), mert a Java programban az útvonal karakterláncát így szokás megadni.

- Az extra.util csomag gyökérkönyvtára c:/javaprog/lib/javalib, a csomag útvonala tehát c:/javaprog/lib/javalib/extra/util. Nézze meg a mellékletben! (A javalib könyvtár a javalib.jar állományba van becsomagolva.)
- Internetes programok fejlesztésénél a következő csomagazonosítási konvenció alakult ki: a gyártó számítógépének domainnevét fordított sorrendben beolvastjuk a struktúrába – így biztosan egyedi nevet kapunk. A Borland által szállított TextEditor mintaalkalmazás esetében például az osztályok a com.borland.samples.texteditor csomagban vannak, mert a domainnev borland.com. A forrásállományok könyvtára: .../samples/TextEditor/src/com/borland/samples/texteditor.

Csomag megadása és importálása

Csomag megadása:

Egy Java forrásállomány összes típusdeklarációja (osztálya és interfész) szükségképpen ugyanabba a csomagba kerül. A csomagot a fordítási egység (forrásállomány) első utasításaként adjuk meg, a package kulcsszóval:

```
package csomagnév [.csomagnév [.csomagnév] ...];
```

A csomagot teljes útvonalával azonosítjuk a csomag gyökérkönyvtárától (a legfelső csomag bennfoglaló könyvtárától) számítva. A csomagrendszert (könyvtárat) általában becsomagolják egy JAR-állományba.

A forráskód package deklarációjában szereplő csomagnév és a forráskód könyvtára meg kell, hogy feleljen egymásnak!

Ha egy forrásállományban nem adunk meg csomagot (nincs benne package kulcsszó), akkor a lefordított class állományok az alapértelmezés szerinti csomagba (default package) kerülnek; annak a célútvonal gyökérkönyvtára felel meg.

Csomag importálása:

A forrásállomány elején (a package deklaráció után) importálhatjuk azokat az osztályokat, interfészeket, amelyekre a forrásban hivatkozni szeretnénk:

```
import csomagnév [.csomagnév [.csomagnév] ...].Osztálynév|Interfésznév;  
import csomagnév [.csomagnév [.csomagnév] ...].*;
```

A * a csomag összes deklarációját importálja (alcsomagét nem). A csomagnévvel minősített deklarációkra importálás nélkül is hivatkozhatunk.

Példaként vizsgáljuk meg a CsomagApp mintaprojekt forrásállományait. Ehhez töltse be a projektet a JBuilder környezetben: *File/Open Project.../*

c:/javaprog/_OOTPJava2/Mintaprogramok/01CsomagProjekt/CsomagApp/CsomagApp.jpx

Bár az osztályok és interfészek pontos megadása nem volt feladat, a jobb érhetőség kedvéért mégis egy teljes, „értelmes” programot írunk. Egy olyan kódolási szabályt adunk meg, amelynek eredményeképpen a fő osztály konzolra való kiíratásával a kapcsolatok mentén kiíródik az összes osztály és interfész neve is – vagyis bejárjuk az osztálydiagramot.

A kódolási szabály a következő:

- ◆ Az `Interfesz1` egyetlen metódust definiál:

```
public String interfesz1();
a visszaadott szöveg az implementációban: "Interfesz1 "
```

- ◆ minden osztálynak megadjuk a `toString()` metódusát, éspedig a következőképpen:

```
osztálynév +
super.toString() (Csak ha van ősosztály. Legfeljebb egy lehet.) +
interfesz1() (Csak ha van az osztálynak interfésze. Lehet több is.) +
kapcsolati objektumok (Csak ha vannak kapcsolati objektumok. Lehet több is.)
```

Vizsgáljuk meg tehát a forráskódot, de közben tartsuk szem előtt az osztálydiagramot (1.1. ábra)!

Forráskód

Projekt: CsomagApp

Interfesz1.java

```
package csomag1.csomag11;
public interface Interfesz1 {
    public String interfesz1();
}
```

`Interfesz1` a `csomag1.csomag11` csomagban van. Importálás nincs, hiszen a forráskódban nem hivatkozunk más csomag deklarációira (csak a `java.lang.String`-re). `Interfesz1` egyetlen metódust definiál, s azt az őt implementáló osztálynak kell majd kifejtenie.

Osztaly2.java

```
package csomag1.csomag11;
class Osztaly1 implements Interfesz1 {
    public String interfesz1() {
        return "Interfesz1 ";
    }
    public String toString() {
        return "Osztaly1 " + interfesz1();
    }
}

public class Osztaly2 extends Osztaly1 {
    public String toString() {
        return "Osztaly2 " + super.toString();
    }
}
```

Importálásra tehát itt sincs szükség: Osztaly1 hivatkozik ugyan Interfesz1-re, de az ugyanebben a csomagban van. Osztaly2 hivatkozik Osztaly1-re, de ők végképp egy csomagban (egy forrásállományban is) vannak.

Osztaly1 implementálja Interfesz1-et, ezért kifejti az interfesz1() metódust. Osztaly2 Osztaly1-ből származik (extends Osztaly1) – felülírjuk annak toString() metódusát úgy, hogy az osztály nevéhez hozzá tesszük az ősosztály toString metódusát (így az osztálynevek összeadódnak).

Osztaly3.java

```
package csomag1.csomag12.csomag121;
import csomag1.csomag11.Interfesz1;

public class Osztaly3 implements Interfesz1 {
    public String interfesz1() {
        return "Interfesz1 ";
    }
    public String toString() {
        return "Osztaly3 " + interfesz1();
    }
}
```

Osztaly3 hivatkozik Interfesz1-re, az azonban nem ebben a csomagban van. Interfesz1-et tehát importálnunk kell!

Osztaly4.java

```
package csomag1.csomag12;
import csomag1.csomag12.csomag121.Osztaly3;

public class Osztaly4 extends Osztaly3 {
    public String toString() {
        return "Osztaly4 " + super.toString();
    }
}
```

Osztaly4-nek Osztaly3 az őse, és az eggyel lejjebb csomagban van. Osztaly3-at tehát importálnunk kell!

Osztaly5.java

```
package csomag2;
import csomag1.csomag12.*;
import csomag1.csomag11.*;
import extra.Console;
```

```

public class Osztaly5 {
    Osztaly2 obj2 = new Osztaly2();
    Osztaly4 obj4 = new Osztaly4();

    public String toString() {
        return "Osztaly5 " + obj2 + "" + obj4;
    }
    public static void main(String[] args) {
        System.out.println("CsomagApp fut...");
        System.out.println(new Osztaly5());
        Console.pressEnter();
    }
}

```

Osztály5 hivatkozik Osztaly2-re és Osztaly4-re is, köztük társítási kapcsolatok vannak. Egyszerűség kedvéért a csomag11 és csomag12 csomagok összes deklarációját importáljuk, abból baj nem lehet. A Console osztályt is importáljuk, mert a program utolsó utasítása a Console.pressEnter().

A program futása

CsomagApp fut...
Osztaly5 Osztaly2 Osztaly1 Interfesz1 Osztaly4 Osztaly3 Interfesz1
<ENTER>

Látható, hogy a kiírt osztályok és interfések a program tervén (UML-ábráján) feltüntetett kapcsolatok feltérképezése (bejárása, kiterítése, sorba fejtése).

A csomag elemeinek láthatósága

Egy **osztály vagy interfész** láthatósága (hozzáférési módja, védelme) **publikus** vagy **csomagszintű** aszerint, hogy megadjuk-e a `public` módosítót vagy nem. Ha megadjuk, akkor az importáló fordítási egység látja őt, ha nem adjuk meg, akkor csak a saját csomagjában látható.

Egy **osztály tagjai** (adatai, metódusai) láthatóság szempontjából a következő csoportok egyikébe sorolhatók:

- **publikus** (`public`, +): minden kliens hivatkozhat rá.
- **védett** (`protected`, #): az őt tartalmazó csomagban bárki hivatkozhat rá, más csomagból csak az utódosztályok.
- **privát** (`private`, -): csak a deklarációt tartalmazó osztály hivatkozhat rá.
- **csomagszintű** (nincs jelölése, ez az alapértelmezés): az őt tartalmazó csomagban mindenki hivatkozhat rá, a csomagon kívülről viszont senki.

Megjegyzések:

- Egy csomagot általában egyetlen programozó készít el, a csomagon belül tehát minden felelősség a programozóé. Ezért a védelemnek sokkal nagyobb jelentősége van csomagon kívülről: a már kész, lefordított csomagok eladhatók, azoknak mindenképpen elronthatatlannak kell lenniük!
- A protected védelem csomagon belül nem változtat semmin, ígyekezzünk a protected védelmet mégis tiszteletben tartani csomagon belül is, hiszen sosem lehet tudni, hogy mikor akarunk bizonyos osztályokat kiemelni a csomagból. A láthatóságok akkor másiképp fognak működni!
- Egy osztálynak összesen háromféle módosítója lehet: public, abstract és final.

1.2. Projektkezelés a JBuilderben

Egy kezdő programozó általában egyetlen forrásállományba teszi a maga egy vagy két osztályát. Egy nagyobb alkalmazás kifejlesztéséhez azonban több forrásállományon kell dolgozni párhuzamosan, és sokszor több idegen könyvtárat (csomagot) is be kell építenünk a szoftverbe. Egy integrált fejlesztőkörnyezetnek minden lehetséges eszközzel támogatnia kell a fejlesztő munkáját. A fejlesztői munka egyszerűsítése és összehangolása érdekében a fejlesztő ún. projekteken dolgozik.

A **projekt (project)** valamely szoftver fejlesztéséhez használt, logikailag összetartozó állományok és környezeti beállítások gyűjteménye. minden projekthez tartozik egy projektállomány; az tartalmazza a szükséges adatokat. A fordító-futtató rendszer a projektállomány adatait használja az állományok betöltésekor, mentésekor, fordításakor, futtatásakor stb.

A projektállomány (projektfájl) tartalma:

- ◆ Állománylista. A projekt elemeinek (állományainak) listája; ezeken az állományokon dolgozik a fejlesztő. Az állomány bármilyen típusú lehet (például java, txt, doc, html, xml, jpg, gif, wav stb.)
- ◆ A projekt tulajdonságai (project properties):
 - JDK-profil
 - A szerző adatai
 - Alapvető útvonalak: Kimeneti útvonal (Output path), Biztonsági másolatok útvonala (Backup path), Munkakönyvtár (Working directory), Forrásútvonalak (Source path), Dokumentációs útvonalak (Doc path), Felhasznált könyvtárak (Required libraries)
 - Futtatási (Run), nyomkövetési (Debug) és fordítási (Build) beállítások
 - stb.

A JBuilderrel csak projektben dolgozhatunk. A projekt kiterjesztése JBuilderben jpx. Az alkalmazásbongészőben egyszerre több projekt is nyitva lehet. Egy projektet – a hozzá tartozó

összes állományt – külön könyvtárban, az ún. projektkönyvtárban szokás elhelyezni (1.2. ábra). A projektállomány közvetlenül a projektkönyvtárban van, a forráskódok és a bájtkódok külön könyvtárban (forrásútvonal és kimeneti útvonal). A projektadatok relatívak, ezért a teljes projektkönyvtár akárhol is átmozgatható.

Mielőtt elmerülnénk a részletekben, tisztáznunk kell a Java fordító által használt főbb útvonalak jelentését. Az **aktuális könyvtár** környezetenként más: általában az éppen futó class állomány könyvtára, JBuilderben a Working directory.

Alapútvonalak

Kimeneti útvonal – output path

A lefordított bájtkódok gyökérkönyvtára. Ebben a könyvtárban helyezi el a fordító a class állományokat a megadott csomagstruktúra szerint.

Osztályútvonal – class path

A hivatkozásokban szereplő típus- (osztály-, illetve interfész-) deklarációk gyökérkönyvtárai. Összetett útvonal, vagyis több könyvtár is megadható. Az útvonalak lehetnek könyvtárnevek és JAR-állományok is. A fordító ezekben a könyvtárakban keresi a hivatkozásokban megadott típusokat (class állományokat). A keresés a felsorolás sorrendjében halad. Az osztályútvonal érvénytelenné teszi a rendszer CLASSPATH környezeti változóját. Ha nem adjuk meg, akkor a fordító az aktuális könyvtárban keresi a hivatkozásokat. A JBuilderben ez a Required libraries (szükséges könyvtárak).

Forrásútvonal – source path

A forrásállományok gyökérkönyvtára. Összetett útvonal, vagyis több könyvtár is megadható. Az útvonalak lehetnek könyvtárnevek és JAR-állományok is. A fordító a felsorolás sorrendjében keresi a forrásokat. Ha nem adunk meg forrásútvonalat, akkor a fordító a classpath-on keres. A fordító a classpath-on talált bájtkódot szükség estén újrafordítja, ha megtalálja a hozzá tartozó forráskódot.

Kész projektek fordításáról és futtatásáról már az első kötetben is szó volt. Most magunk készítünk projektet. Létrehozzuk a CsomagApp projektet a c:/javaprog/_MyPrograms könyvtárban. A kész projekt megtalálható a mellékletben, a c:/javaprog/_OOTPJava2/Mintaprogramok/01CsomagProjekt mappában.

Projekt létrehozása

Hozzuk létre a CsomagApp projektet a mintaprogram elkészítéséhez! Ehhez hívjuk meg a projektvarázslót:

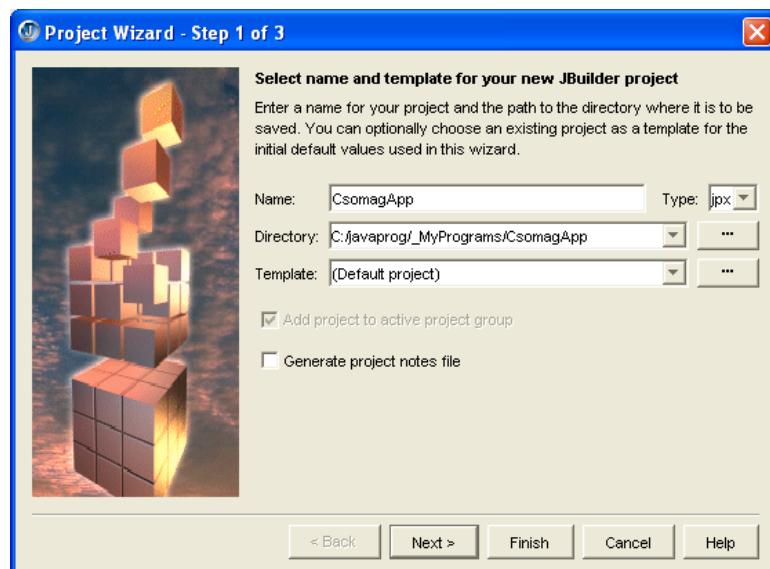
File/New Project...vagy File/New.../Project

Megjelenik a Project Wizard, és 3 lépéser osztja fel a teendőket. A projekt könyvtára legyen c:/javaprog/_MyPrograms/CsomagApp, a projektfájl neve pedig legyen CsomagApp.jpx.

Az egymás utáni lépésekben töltök ki a vastagon szedett paramétereket, minden másat hagyunk változatlanul!

1. lépés (1.4 ábra):

- ◆ Name: **CsomagApp**. Ekkor a [Directory] utolsó eleme automatikusan CsomagApp lesz. Ez jó nekünk.
- ◆ Type: jpx
- ◆ Directory: **c:/javaprog/_MyPrograms/CsomagApp**. A [...] gombbal a szülőkönyvtárat választjuk ki; a JBuilder ehhez automatikusan hozzáírja a [Name] mezőben megadott szöveget, a kettőt / jelrel elválasztva.
- ◆ Template: **(Default project)**. Később majd választhatunk más, már elkészített projektet, hogy eleve annak a beállításaiból indulhassunk ki.



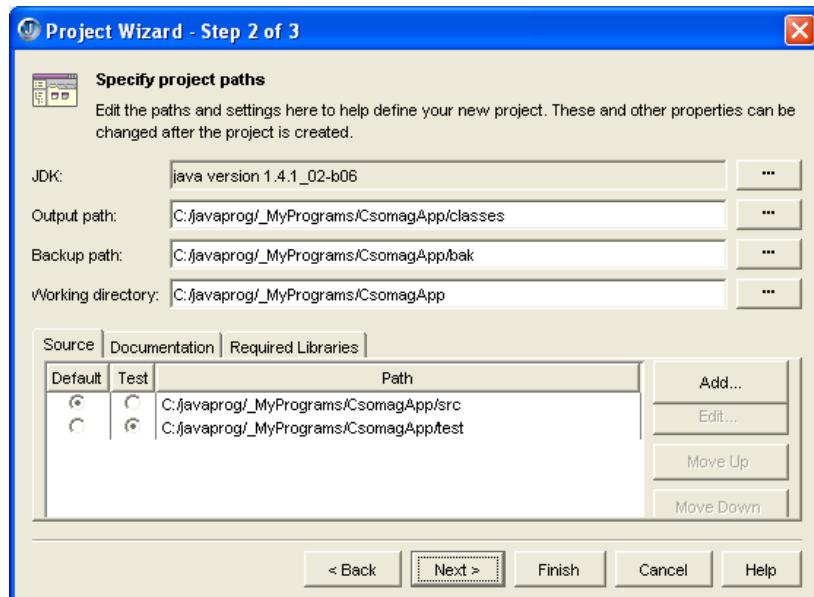
1.4. ábra. Projekt létrehozása – 1. lépés

2. lépés (1.5 ábra): A felkínált könyvtárak az előző lépésben kiválasztott [Directory] projektkönyvtár alatt keletkeznek. Mindent hagyunk változatlanul, csak a *Required Libraries* elemein kell módosítanunk! A mezők jelentése:

- ◆ JDK: Az aktuális JDK.
- ◆ Output path: **[Directory/]classes**. Kimeneti útvonal. Ide kerülnek a bájtkódok.
- ◆ Backup path: **[Directory/]bak**. Ide kerülnek a biztonsági másolatok.
- ◆ Working Directory: **[Directory/]** Ez lesz az aktuális könyvtár. Innen indulnak a projekt relatív útvonalai.

Alsó fülek:

- ◆ Source: **[Directory/]src**. Forrásútvonal. Ide kerülnek a forrásállományok.
- ◆ Documentation: **[Directory/]doc**. A dokumentumállományok útvonala.
- ◆ Required Libraries. Itt kell felvenni a projektben használt külső könyvtárakat, például az extra csomagot magába foglaló javalib könyvtárat. Kattintsunk rá a *Required Libraries* fülre, és válasszuk ki az *Add* funkciót! Ha a javalib még nincs ott a könyvtárak listájában, akkor a *New* funkcióval vigyük fel a javalib könyvtár adatait (nevét: javalib és állományspecifikációját: c:/javaproj/lib/javalib.jar)! Ezután válasszuk is ki a javalib könyvtárat, hogy felkerüljön a *Required Libraries* listára!



1.5. ábra. Projekt létrehozása – 2. lépés

3. lépés: Ez a lépés nekünk nem lényeges. Nyomja meg a *Finish* gombot!

A varázslás befejeztével létrejön a projektkönyvtárban a projektállomány. A projekt tulajdon-ságai később is állíthatók: *Project/Project properties* vagy *Kattintás jobb egérgombbal a projektállományon/Properties*

Most írjuk be sorban a CsomagApp projekt osztályait, interfészeit! Először az *Interfesz1* nevű interfész kódoljuk, mert a többi osztály függ tőle.

Új interfész létrehozása

Új interfész a következőképpen hozhatunk létre:

File/New.../General/Interface

Megjelenik az interfész varázsló (Interface Wizard, 1.6. ábra, felül):

- ◆ Package: **csomag1.csomag11**. Itt kell megadnunk az interfést tartalmazó csomagot. Választhatunk a felkínált csomagokból is, ha már előzőleg megadtunk csomagokat.
- ◆ Interface name: **Interfesz1**. Az interfész neve.
- ◆ Base interface: **<none>**: Az ősinterfész neve.
- ◆ Generate header comments. Ha ezt bejelöljük, akkor általános információk jelennek meg a forráskód elején, megjegyzés formájában.

Az OK gomb lenyomása után visszakerülünk az alkalmazásböngészőbe. A tartalompanelen megjelenik az interfész generált váza:

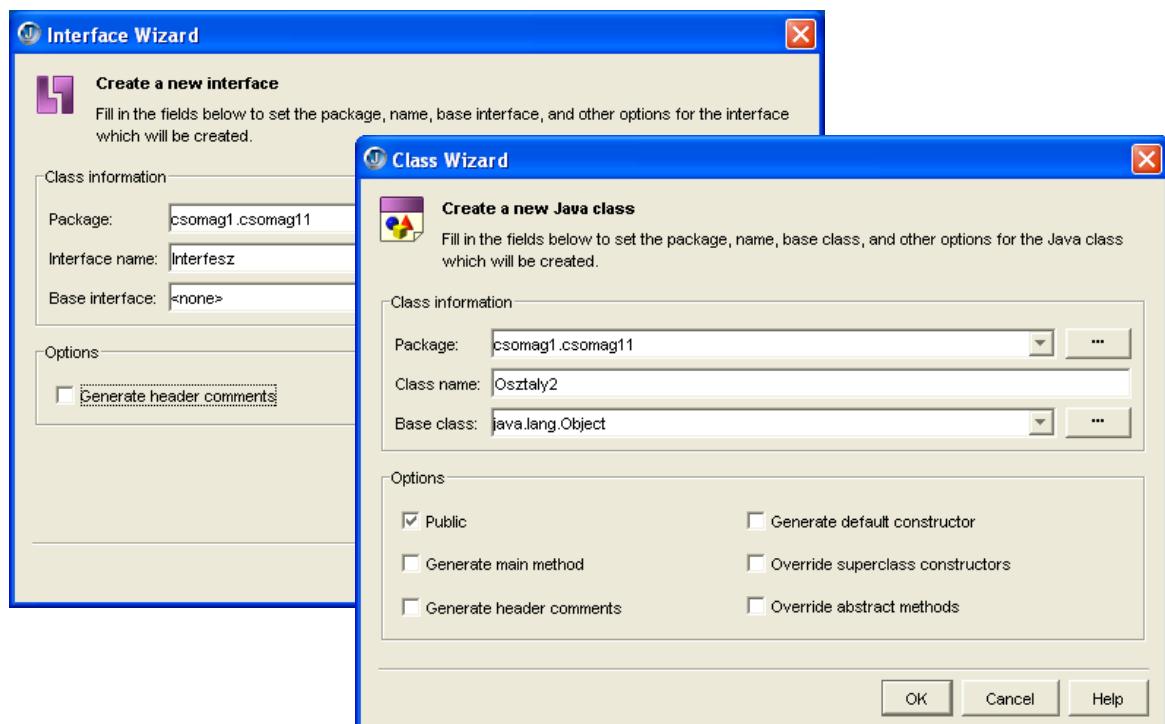
```
package csomag1.csomag11;

public interface Interfesz1 { }
```

Egészítsük ki a kódot! Adjuk meg az interfész egyetlen metódusfejét:

```
package csomag1.csomag11;

public interface Interfesz1 {
    public String interfesz1();
}
```



1.6. ábra. Interfész- és osztályvarázsló

Új osztály létrehozása

Új osztályt a következőképpen hozhatunk létre:

File/New.../General/Class

vagy

File/New class...

Megjelenik az osztályvarázsló (Class Wizard, 1.6. ábra, alul):

- ◆ Package: **csomag1.csomag11**. Az osztályt tartalmazó csomag neve. Választhatunk a felkínált csomagokból is, ha előzőleg már megadtunk csomagokat.
- ◆ Class name: **Osztaly2**. Az osztály neve.
- ◆ Base class: **java.lang.Object**. Az ősosztály neve. Ha felülírjuk, akkor ez szerepel majd az extends kulcsszó után a forráskódban. Ha **Osztaly1** még nem létezik (és most még nem létezik), akkor az öröklési kapcsolatot kézzel kell majd beírnunk.

A dialógus alsó részében szereplő jelölőnégyzetek közül most csak a Public-ot jelöljük be; az osztály ezáltal publikus lesz. A többi jelölő üresen marad, vagyis a kódgenerátor nem generál main metódust; nem lesz megjegyzés; nem lesz alapértelmezés szerinti paraméter nélküli konstruktur; nem lesznek az őskonstruktor paramétereinek hasonló konstruktorok; nem lesznek metódusok az ősosztály absztrakt metódusainak felülírásához.

Az OK gomb lenyomására elkészül az **Osztaly2.java** forráskódja:

```
package csomag1.csomag11;

public class Osztaly2 { }
```

Az osztály- és interfészvarázslók felteszik, hogy egy fordítási egységben csak egy definíciót (osztályt vagy interfész) akarunk elhelyezni. A létrehozott fordítási egységekbe azonban lehetők más osztályok, interfések is. Ezeket a további deklarációkat kézzel kell beírnunk, és nem tehetjük őket publikussá. Az **Osztaly1** deklarációját például az **Osztaly2** osztályvarázslóval elkészített fordítási egységbe tesszük. Egészítük ki az **Osztaly2.java** forráskódját:

```
package csomag1.csomag11;

class Osztaly1 implements Interfesz1 {
    public String interfesz1() {
        return "Interfesz1 ";
    }

    public String toString() {
        return "Osztaly1 " + interfesz1();
    }
}
```

```
public class Osztaly2 extends Osztaly1 {
    public String toString() {
        return "Osztaly2 " + super.toString();
    }
}
```

Készítsük el sorban a `CsomagApp` projekt összes osztályát és interfészét! Az `import` beírása-kor a JBuilder környezet automatikusan felkínálja a lehetséges csomagokat.

Projekt fordítása és futtatása

Projekt fordítása

- ◆ *Project/Make Project "[Projektfájl]"* (vagy *Ctrl+F9*, vagy *Fő eszköztár/Sárga téglalap/Make*). Lefordítja a projektben az időközben módosított forrásállományokat (azokat, amelyeknek a dátuma későbbi, mint a már lefordított class állományé). Ha szükséges, akkor az összes forráskódot lefordítja. Fordítás előtt elmenti a forráskódot, ha azt meg-változtattuk a szövegszerkesztőben.
- ◆ *Project/Rebuild Project "[Projektfájl]"* (vagy *Fő eszköztár/Sárga téglalap/Rebuild*). Lefordítja a projekt összes forrásállományát, feltétel nélkül. A megváltozott forráskódot előbb elmenti.

A nem `class` kiterjesztésű állományokat erőforrásoknak (`resources`) nevezzük, ilyenek például a `jpg`, `gif`, `png`, `properties` stb. állomány. A fordító az engedélyezett erőforrásokat a forráskönyvtárból átmásolja a kimeneti könyvtárba. Az engedélyezett erőforrások listája megtekinthető a *Tools/Preferences/Browser/File Types* ablakban.

Projekt futtatása

- ◆ *Run/Run Project (F9, Fő eszköztár/Zöld háromszög/[Futási konfiguráció])*. Futtatja a projektet. Ha a projekt forráskódjai még nincsenek lefordítva, akkor rendszer lefordítja őket.

Futtatási konfiguráció beállítása

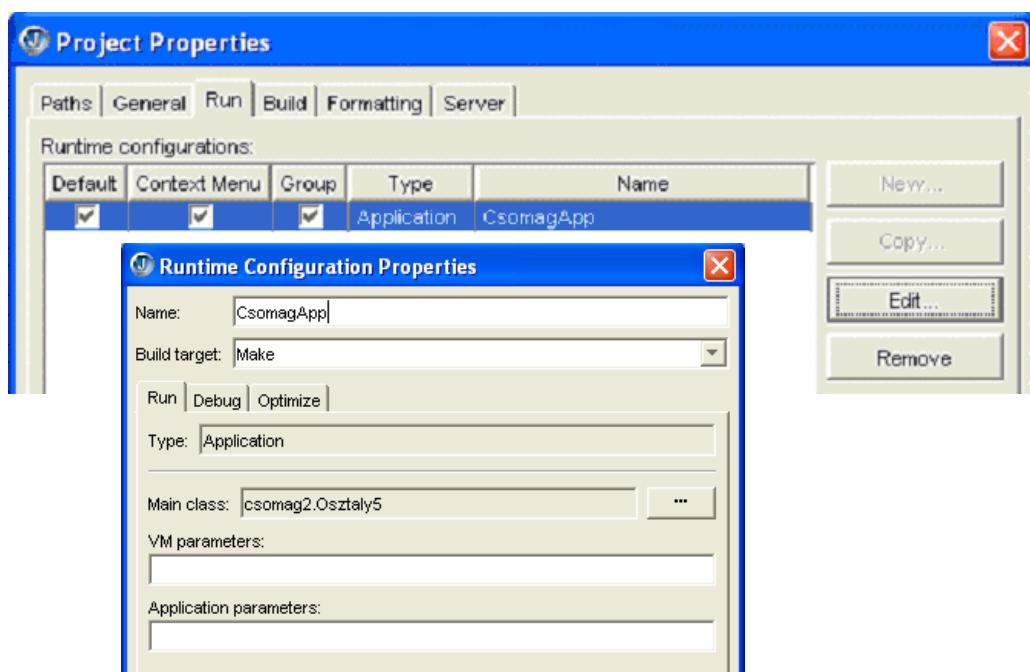
Futtatási konfigurációnak nevezzük a futtatásra jellemző tulajdonságok névvel ellátott együttesét. Egy projektben több futtatási konfigurációt is kialakíthatunk; közülük minden csak egy aktuális. Első futtatáskor be kell állítanunk egy futtatási konfigurációt. A konfigurációk listáját így érhetjük el (1.7. ábra, felül):

Project/Project Properties.../Run fül

Új konfiguráció létrehozásakor (*New*), illetve a már meglévő szerkesztésekor (*Edit*) megjelenik a *Runtime Configuration Properties* dialógus; ez valamelyik futtatási konfiguráció tulajdonságait mutatja (1.7. ábra, alul). A futtatási konfigurációra jellemző a neve (*Name*), a futtatás típusa (*Application/Applet...*); ha alkalmazásról van szó, akkor a *main* metódus osztálya (*Main*

class), a virtuális gép paraméterei (VM parameters), az alkalmazás paraméterei (Application parameters) stb. is.

Több futtatási konfigurációra akkor lehet szükségünk, ha gyakran akarjuk változtatni a futtatási paramétereket.



1.7 ábra. A futtatási konfiguráció beállítása

A következő feladattal olyankor találkozunk, ha futtatni akarunk egy Java alkalmazást, de nincs meg hozzá a JBuilder projektállományunk. Az Interneten keresgélve bizony ez gyakran előfordulhat.

Feladat

A lemezen helyesen felépített csomag, illetve könyvtárhierarchiában megvan egy alkalmazás összes forráskódja, és tudjuk, hogy a fordítási egységek közül melyik tartalmazza a main metódust. Készítsünk projektállományt ehhez az alkalmazáshoz, és futassuk a projektet!

Példaként vegyük a c:\javaprogram\OOTPJava2\Esettanulmanyok\KissEditor forrásállományait, és készítsünk egy projektet a _MyPrograms mappában! Végezzük el a következő lépéseket:

1. Hozzunk létre egy `KissEditor` nevű projektet, legyen `c:/javaproj/`
`_MyPrograms/KissEditor` a projektkönyvtára! A projekt tulajdonságainak beállításakor ügyeljünk az útvonalak (Output path, Source path...) helyes megadására!
2. A projektkönyvtár alatt hozzunk létre egy `src` könyvtárat, és másoljuk ide változatlan könyvtárstruktúrában az alkalmazás összes forrásállományát!
3. Frissítsük a projektet (*Projekt eszköztár/Refresh*)! A JBuilder feltérképezi a csomagokat, és a benne levő állományokat rátesszi a projektre. Vigyázzunk, mert a JBuilder csak akkor látja a forrásállományokat, ha a forráskönyvtár helyesen van beállítva!
4. Be kell állítanunk a *Project/Project Properties.../Paths/Required Libraries* fülön a `javalib` könyvtárat.
5. Adjunk meg egy futtatási konfigurációt, s abban a főosztályt: `Main class = kisseditor.KissEditor`! A rendszer ezt kérni is fogja majd az első futtatási kísérletkor.

A frissítés a projekt létrehozásakor is lezajlik, vagyis egy lépést megtakarítunk, ha előbb másolunk, és azután hozzuk létre a projektet.

1.3. JAR-állomány készítése

Miután elkészítettünk egy Java programot (alkalmazást, appletet stb.) vagy egy segédkönyvtárat (olyat, mint a `javalib`), át kell adnunk azt a felhasználónak (esetleg sajátmagunknak). A felhasználó kényelmesen szeretné telepíteni és használni az átadott anyagot, természetesen most már a fejlesztőkörnyezet nélkül. Az átadandó program, illetve könyvtár tartalmazhat bájtkódokat, erőforrásokat (képeket, meghajtókat, adatokat, dokumentumokat...) és különböző, szabványos `JDK API` és egyéb könyvtárakat. A „kellékek” száma meglehetősen nagy is lehet. Át lehet adni őket sok különálló állományban is, de ekkor a telepítés és a környezeti beállítás nagyon nehézkes lesz. Sokkal kényelmesebb megoldás, ha a futtatóhoz szükséges állományokat összecsomagoljuk, és egyetlen állományként adjuk át a „vevőnek”. – a Java programok esetén ez egy speciális PKZIP-formátumú, tömörített, JAR kiterjesztésű állományban. Bizonyos JAR-állományok futtathatók (például a `CsomagApp.jar`), mások meg nem (például a `javalib.jar`). A futtatható JAR-állománynak kell, hogy legyen egy belépési pontja, vagyis egy `main`-t tartalmazó fő osztálya – erről a JAR-ba csomagolt, ún. aláírásállomány ad információt.

Ebben a pontban először megvizsgáljuk a JAR archív állomány felépítését, majd megmutatjuk, hogyan lehet egy JAR-állományt elkészíteni az Archive Builder segédprogrammal vagy parancssorból, „kézi módszerrel”. A futtatható JAR-állomány futtatásával a következő pont foglalkozik majd.

A JAR-állomány felépítése

A **JAR** (Java ARchive) egy Java projekt állományait tömörítő állomány. Formátuma megegyezik a PKZIP-formátummal, de `jar` a kiterjesztése, és speciális belső könyvtárstruktúrája van. A JAR platformfüggetlen, vagyis bármely operációs rendszeren használható, futtatható.

A JAR-állomány három részből tevődik össze (bármelyik rész elhagyható):

- Tartalom (content): az `Output path` alatt levő osztályok (class kiterjesztésű állományok) és erőforrások (nem class állományok, mint jpg, gif, wav, html stb.)
- Felhasználandó könyvtárak (library dependencies): a projekt által használt külső könyvtárak, például az `rt.jar`, a `javalib.jar`. Ezek a könyvtárak a classpath elemei.
- Aláírásállomány (manifest file): Szöveges állomány, felvilágosítást ad a fejlesztés körülményeiről (verzió, szerző...), valamint arról, hogy melyik a `main` metódust tartalmazó `class` állomány. Egy JAR-állományt csak akkor lehet futtatni, ha van benne aláírásállomány, és az megadja a `main-t` tartalmazó osztály nevét. Az aláírásállomány neve általában `manifest.mf`.

Mit érdemes becsomagolni egy JAR-ba?

- ◆ A tartalmat általában becsomagoljuk; tartalom nélkül a JAR nem sokat ér.
- ◆ A felhasznált könyvtárak becsomagolhatók a JAR-ba, de külön is átadhatók. Könnyebb kezelni az alkalmazást, ha minden egyben van; másfelől ugyanazt a könyvtárat helypazarlás több JAR-ba is belepréselni.
- ◆ Az aláírásállományt csak futtatható JAR-ba kötelező belecsomagolni, de más esetben is ajánlatos betenni, mert hasznos információt ad. Az aláírásállomány jellemző sorai a következők:

```
Manifest-Version: 1.0
Main-Class: <class-name>
Created-By: XY
```

Példaként vizsgáljuk meg a `CsomagApp.jar` archív állomány felépítését!

Az 1.8. ábra jobb oldalán látható a tömörített `CsomagApp.jar`. Bal oldalon vannak a „források”, innen szedtük össze a JAR elemeit. A tartalom a `CsomagApp` projekt lefordított bájtkódjai, ezek a `classes` könyvtárban találhatók. A `javalib` felhasznált könyvtár, hiszen a tartalom használja az `extra.Console` osztályt. Ami a felhasználandó könyvtár becsomagolását illeti, a következő lehetőségeink vannak:

- ◆ Az `extra.Console` osztályt nem csomagoljuk be a JAR-állományba. Ez esetben futtatáskor gondoskodnunk kell az osztály jelenlétééről. De mint a következő pontban látni

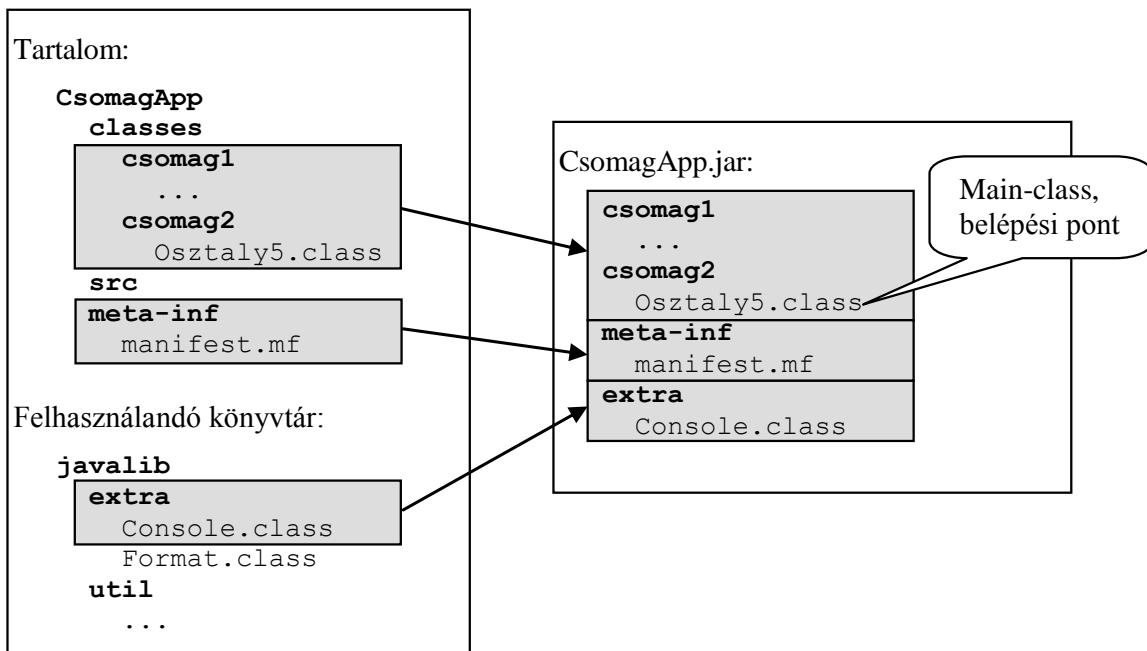
fogjuk, a JAR futtatásakor nincs classpath, ezért a `javalib` csomagot a futtatókörnyezetbe kell majd beleágazni.

- ◆ Az `extra.Console` osztályt becsomagoljuk a JAR állományba. Ezt úgy tehetjük meg, hogy becsomagolás előtt átmásoljuk az `extra\Console.class` állományt (könyvtárastól!). Ha csak kevés osztályra kell támaszkodnunk, akkor ez az ideális megoldás.
- ◆ A teljes `javalib` könyvtárat becsomagoljuk a JAR-állományba. Fontos azonban tudni, hogy JAR-t nem lehet JAR-ba pakolni, vagyis átmásolás előtt ki kell csomagolni a felhasználandó könyvtárakat! Ezt a megoldást akkor érdemes alkalmazni, ha a csomagból sok osztályt használunk, és sokat kellene bajlódni az osztályok kiválogatásával, szűrésével.

A JAR-állománynak egyébként mindegy, honnan másoljuk bele az elemeket. Az aláírásállományt is bárhol elkészíthetjük, és nem is mindenki kell tárolnunk: az Archive Builder például „röptében” elkészíti.

Ahhoz, hogy a `CsomaggApp.jar` futtatható legyen, mindenkiéppen be kell csomagolnunk egy aláírásállományt, és abban meg kell adnunk a fő osztályt. A `CsomaggApp.jar` aláírásállománya (`manifest.mf`) így fest:

```
Manifest-Version: 1.0
Main-Class: csomag2.Osztaly5
```



1.8. ábra. A `CsomaggApp.jar` felépítése

JAR készítése Archive Builder-rel

A JAR-állományt „kézi” módszerrel is le lehet gyártani a JDK `jar.exe` programjával. Szerencsére azonban szinte minden valamirevaló fejlesztőkörnyezetben van intelligens JAR-készítő segédprogram, a fejlesztőnek nem kell tehát „szenvednie”.

Az **Archive Builder**, a JBuilder segédprogramja a projekt alapján elkészíti a JAR-állományt. Használata egyszerű: egy varázslón végigszaladva beállítjuk a szükséges paramétereket, és már megjelenik a projekt könyvtárában az archív állomány.

Az Archive Builder bemutatásához példaként hozzunk létre a `CsomagApp` projekthez egy futtatható JAR-állományt!

Az Archive Builder használata:

Wizards/Archive Builder...

A varázslás 6 lépésből áll:

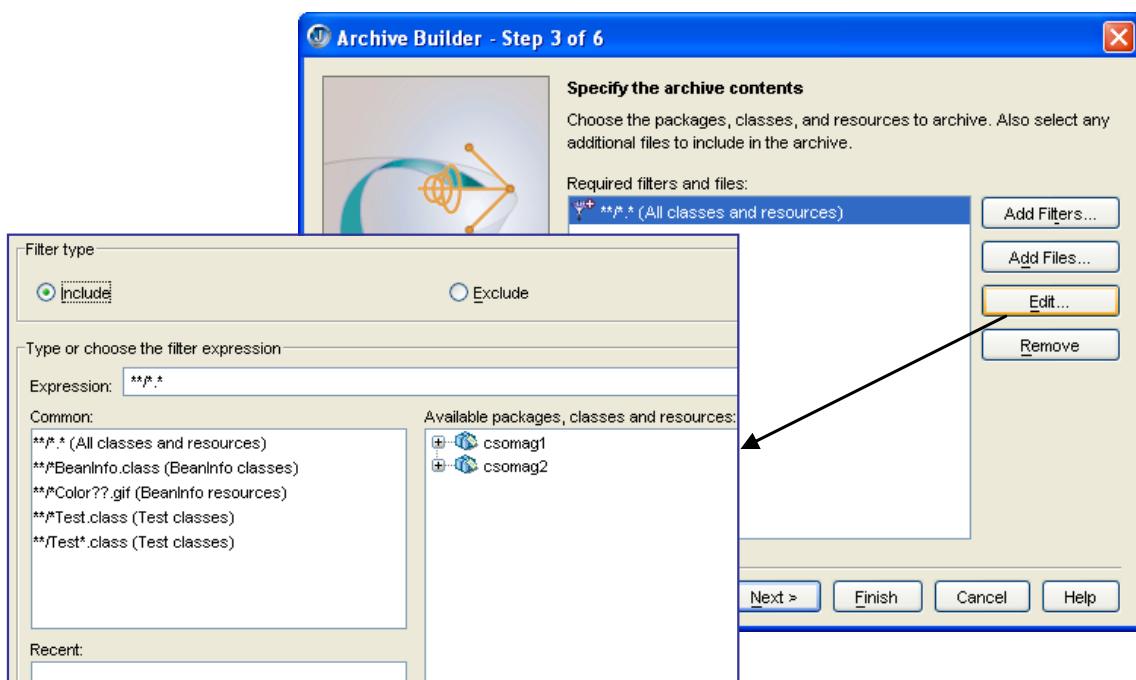
1. lépés: **Archive type** (Jar típusa): *Application*. A JAR típusa lehet Applet JAR, Application, Source stb. Mivel alkalmazást szeretnénk készíteni, válasszuk ki az Applicationt!
2. lépés:
 - **Name** (név): *Application*. A JAR-készítési eljárás logikai neve – ez jelenik meg a projektfán. Meghagyhatjuk az alapértelmezést.
 - **File** (JAR-állomány) : *.../01CsomagProjekt/CsomagApp/CsomagApp.jar*
A keletkező JAR-állomány neve. Hagyjuk meg az alapértelmezést!
 - **Always create archive when building the project**: Hagyhatjuk bejelölve. Ekkor a JAR-állomány minden fordításkor újra elkészül.
3. lépés (1.9. ábra): **Required filters and files** (szükséges szűrők és állományok): *All classes and resources*. Itt válogatjuk össze a tartalmat. Az állományok szűrőkkel és egyenként is megadhatók. Az alapértelmezés meghagyásával a classes könyvtár alatti összes állományt betesszük a JAR-be (ekkor nem kell szerkesztenünk a tartalmat).
4. lépés: **Dependencies** (felhasználandó könyvtárak): *Include required classes and known resources*. Megjelennek a projektben szükséges könyvtárak (Project Properties/Required Libraries). minden könyvtárhoz be kell jelölni a következő lehetőségek valamelyikét:
 - Never include any classes or resources: A JAR ebből a könyvtárból semmit sem fog tartalmazni. Futtatáskor majd elérhetővé kell tenni a könyvtárat.
 - Include required classes and known resources: A JAR csak a hivatkozásokban szereplő osztályokat és az ismert erőforrásokat fogja tartalmazni.
 - Include required classes and all resources: A JAR tartalmazza majd a felhasználandó könyvtárból a hivatkozásokban szereplő osztályokat és mindegyik erőforrást.

- Always include all classes and resources: A JAR tartalmazni fogja a felhasználandó könyvtár összes osztályát és erőforrását. Például a teljes javalib bekerül a JAR-ba.

A CsomagApp projekt feldolgozásakor ebben a lépésben csak a javalib könyvtár jelenik meg. Jelöljük be a második lehetőséget! Így a javalib osztályai közül csak azok az osztályok és erőforrások (képek) kerülnek bele a JAR-ba, amelyekre ténylegesen hivatkozunk, vagyis minden az extra\Console.class lesz becsomagolva. Ezzel a JAR-állomány a lehető legkisebb lesz és a javalib könyvtárat nem kell a felhasználó gépére telepíteni.

5. lépés: Manifest: (aláírásállomány): Megadjuk, hogy legyen-e aláírásállomány. Meghagyhatjuk az alapértelmezést – ekkor az aláírásállományba két sor kerül: a Manifest-Version és a Main-Class.
6. lépés: Main (fő osztály): Hagyjuk meg az alapértelmezést! Ha jól volt beállítva a futtatási konfiguráció (és alighanem jól volt, ha a projekt futott már), akkor a projekt automatikusan felderíti a main-t tartalmazó osztályt.

A legközelebbi fordításkor létrejön a projektkönyvtárban a CsomagApp.jar állomány.



1.9. ábra. Az Archive Builder 3. lépése: a tartalom összeállítása

Ellenőrzésképpen bontsa ki a CsomagApp.jar állományt (ez úgy is elvégezhető, hogy a kiterjesztését ideiglenesen zip-re változtatja), és vizsgálja meg a tartalmát! Négy könyvtárat talál

benne: csomag1, csomag2, extra és meta-inf. Az aláírásállomány nem a projektkönyvtár-ból való, azt a JBuilder maga állította elő. A projektkönyvtárba a kézi összeállításhoz tettük be az aláírásállományt.

A JAR-készítés tulajdonságait később módosíthatjuk:

Projektpanel/Application/Jobb egér/Properties..

Megjegyzések:

- A Java alkalmazást át lehet alakítani a különböző platformok futtatható állományaivá, de erre csak a JBuilder Enterprise változata ad módot. A JBuilder futtatható állomány készítője a Native Executable Builder.
- Megtehetjük, hogy a JDK API-ból használt osztályokat (rt.jar, Runtime Library) is hozzákeresztjük a JAR-hoz. Ehhez először a *Project/project Properties.../Required Libraries* listájába fel kell vennünk a könyvtárat (válasszuk ki a JAVA_HOME alatti jre\lib\rt.jar állományt); majd a JAR-készítő Application tulajdonságainak Dependencies pontjában a könyvtárhoz az *Include required classes and known resources* pontot kell választanunk. Általában azonban nem jó ötlet az API-t hozzákereszteni a programhoz, mert így borzasztó nagy lesz a JAR-állomány (10 MB-os is lehet), és rengeteg lesz a duplikáció. Az API a JRE-ben minden felhasználó gépen jelen van: ahol a JVM (java.exe) ott van, ott API is van.

JAR készítése parancssorból

Előfordulhat, hogy a JAR-állományt „kézi” módszerrel, parancssorból kell összeállítanunk, mert például nincs kéznél JAR-készítő program. Ilyenkor a JDK jar.exe programját használhatjuk; hívásának szintaktikája a következő:

```
jar <opción> <jar-file> <manifest-file>
      [-C <dir>] [<file> ...] [@<file>]
      ...
```

A jar program elkészíti a <jar-file> állományt úgy, hogy becsomagolja a <manifest-file> aláírásállományt, majd a felsorolt <file> állományokat. -C -vel ideiglenesen átkapcsolhatunk egy másik mappába, ha így egyszerűbb megadni az állományokat. @file egy állománylistát tartalmazó szöveges állomány.

A fontosabb opciók jelentése:

- c: Új archívumot (JAR-állományt) hoz létre. Az esetlegesen már létezőt törli.
- v: Bő, szószátyár (verbose) kimenetet ad. Csomagolás közben kiírja, hogy éppen mit ad az archív állományhoz, és hány százaléknál tart.
- f: Jelzi, hogy a második parancssori paraméter a JAR-állomány neve.
- m: Hozzáad a JAR-állományhoz egy már meglevő aláírásállományt. A JVM csak akkor futtathatja a JAR-állományt, ha abban van szabályos aláírásállomány.

Például:

A CsomagApp projektkönyvtárában szerepel egy `makeJar.bat` állomány, benne egy parancssor a `CsomagApp.jar` elkészítésére:

```

    1.          2.          3.
    jar cvfm CsomagApp.jar meta-inf\manifest.mf -C classes .
    -C c:\javaprog\lib\javalib extra\Console.class
    4.
  
```

1. A `jar` parancs a `cvfm` opciókkal (lásd felül) elkészíti a `CsomagApp.jar` állományt. Ehhez egy képzeletbeli `CsomagApp` mappába bemásolja a következő dolgokat:
2. a `meta-inf\manifest.mf` aláírásállományt (ez az aktuális könyvtárban található, vagyis a parancs indításának könyvtárában);
3. a `classes` mappa összes állományát (.). Ehhez ideiglenesen lekapcsol a `classes` mappába;
4. a `javalib` mappából az `extra\Console.class` állományt. Ehhez ideiglenesen átkapcsol a `javalib` mappába.

Végül az egészet becsomagolja a `CsomagApp.jar` állományba, és azt beteszi az aktuális könyvtárba.

Megjegyzés: A könyv mellékletében oktatási célból jó néhány projekt könyvtárában ott van a JAR-készítő parancsállomány.

1.4. Java program futtatása

A Java alkalmazás osztályok (class kiterjesztésű állományok) és erőforrások együttese, részben vagy egészben JAR-állomány(ok)ba csomagolva. A Java alkalmazást a JVM (Java Virtual Machine, Java virtuális gép) elindításával futtathatjuk. A JVM a Java futtatókörnyezet (JRE, Java Runtime Environment) része. A számítógépen több futtatókörnyezet is lehet – a `JAVA_HOME` környezeti változó az aktuális JRE könyvtárára mutat. A virtuális gép a `jre/bin` könyvtárban található: Windowsban ez a `java.exe` vagy `javaw.exe`; Linuxban a futtatható `java` állomány.

A JVM hívása

A Java alkalmazás futtatásának (a JVM hívásának) szintaktikája a következő:

<code>java [<opcionálk>] <class> [<paraméter>...]</code>	<code>//1</code>
<code>javaw [<opcionálk>] <class> [<paraméter>...]</code>	<code>//2</code>
<code>java -jar [<opcionálk>] <jar-file> [<paraméter>...]</code>	<code>//3</code>
<code>javaw -jar [<opcionálk>] <jar-file> [<paraméter>...]</code>	<code>//4</code>

A `javaw` parancs hasonló a `java` parancshoz; a különböző az, hogy a `javaw`-hez nem tartozik konzolablak. A JVM elindíthat `class` (//1 és //2) és `jar` (//3 és //4) kiterjesztésű állományt is (a `class` állományoknak csak a nevét kell megadni, a kiterjesztést nem – sőt nem is szabad). A `class` állománynak fő osztálynak kell lennie (olyannak tehát, amelyben van `main`); a futtatható JAR-állománynak pedig tartalmaznia kell egy aláírásállományt, amely megmondja, melyik a fő osztály.

A fő osztályban szerepelnie kell egy szabványos `main` metódusnak, a következő fejjel:

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

A JVM betölti a fő osztályt, és meghívja annak `main` metódusát.

`<opción>` a futtatónak szóló parancsok; bármilyen sorrendben megadhatók; `<paraméter>`... a `main` metódus aktuális paraméterei.

Java alkalmazáson a futtatott osztályok együttesét értjük; a nevük `<class>`, illetve `<jar>` (A CsomagApp alkalmazásnak például része az Osztaly4 is).

Fontosabb opciók

-classpath <classpath> -cp <classpath>	Osztályútvonalak megadása. A JVM ezeken az útvonalakon keresi a hivatkozásokban szereplő osztályokat és erőforrásokat. <classpath>: könyvtárak, JAR-/ZIP-állományok, pontosvesszővel (Linuxban kettősponttal) elválasztva. JAR esetén nincs hatása!
-verbose	A futtató bő, „szószátyár” tájékoztatást ad a futás menetéről. Alapértelmezés: nincs bővebb tájékoztatás.
-showversion	Kiírja a JRE verzióját, és folytatja a futást.

Fontos tudni, hogy `-jar` mellett a virtuális gép figyelmen kívül hagy minden explicit `classpath` beállítást. Ha vannak használandó osztályok, akkor a következő lehetőségeink vannak:

- ◆ a használandó könyvtárakat (a szükséges osztályokkal) bepakoljuk a JAR-állományba;
- ◆ bevetünk egy kis trükköt: `-jar` nélkül futtatunk: `java [<opción>] <class>....`
Az opciókban a `classpath`-ra tesszük a futtatandó JAR-állományt, így a futtatórendszer meg fogja találni a `<class>` fő osztályt (lásd 4. példa);
- ◆ a használandó könyvtárakat a JRE részévé tesszük (lásd következő alpont: Használandó osztályok keresése).

Megjegyzés: Futtatás előtt ajánlatos beállítani a PATH, JAVA_HOME és CLASSPATH környezeti változót. A rendszer a JVM-et a PATH alapján kutatja fel; sok parancsállomány a JAVA_HOME környezeti változó kiegészítésével ad teljes állományspecifikációt; a CLASSPATH-nak pedig a „csupasz” class állományok futtatásakor lehet jelentősége.

Nézzünk néhány példát!

1. példa (01CsomagProjekt\CsomagApp\run.bat):

```
cd classes  
java -cp .;c:\javaprogram\javalib csomag2.Osztaly5
```

A becsomagolatlan CsomagApp projektet futtatja, ehhez előbb belép a classes könyvtárba. Főosztály: csomag2.Osztaly5; CLASSPATH: .;c:\javaprogram\javalib.

Vigyázat! Ne tegyük szóközt a classpath útvonalai közé, mert azzal lezárjuk a cp opciót!

Feltesszük, hogy az operációs rendszer PATH környezeti változója tartalmazza a java.exe elérési útvonalát.

2. példa (01CsomagProjekt\CsomagApp\runJar.bat):

```
%JAVA_HOME%\bin\java -jar CsomagApp.jar
```

Futtatja a CsomagApp.jar állományt; abban benne van a futáshoz szükséges Console.class osztály.

Feltesszük, hogy az operációs rendszer JAVA_HOME környezeti változója a JDK könyvtárnevét tartalmazza (pl. C:\JBuilderX\jdk1.4), és így a java.exe állomány megtalálható a %JAVA_HOME%\bin könyvtárban.

3. példa (01CsomagProjekt\ProgParamTeszt\runJar.bat):

```
java -jar ProgParamTeszt.jar Yellow Submarine Yesterday
```

A programnak itt paramétereit vannak. A „bemutatóhoz” kölcsönvesszük az 1. kötet egy programját: _OOTPJava1\Mintaprogramok\18Tomb\ProgParamTeszt.java, mely kiírja a program futási paramétereit. Vizsgálja meg a ProgParamTeszt könyvtárat! Ott megtalálja a JAR.készítő parancsállományt is.

Feltesszük, hogy az operációs rendszer PATH környezeti változója tartalmazza a java.exe elérési útvonalát.

4. példa (_OOTPJava2\Esettanulmanyok\KissDraw\runJar2.bat):

```
java -cp KissDraw.jar kissdraw.KissDraw
```

Mivel JAR-állomány futtatásakor a JVM figyelmen kívül hagyja a classpath-t, azért most kicselezzük, és a futtatást -jar nélkül végezzük. A JAR állományt ráte tesszük a classpath-ra, így a futtató eléri a fő osztályt ☺.

Használandó osztályok keresése

A JVM három helyen keresi a hívott osztályokat:

- ◆ Boot classpath: A Java futtatási környezet könyvtára: `jre/lib`. Itt található az `rt.jar` (runtime library).
- ◆ Extension classpath: A Java futtatási környezet kiterjesztett könyvtára: `jre/lib/ext`. A JVM az itt található JAR-állományokat is látja. Ha ide beteszünk egy JAR-állományt (például a `javalib.jar-t`), akkor azt a futtatórendszer felismeri.
- ◆ User classpath: Ez a klasszikus, környezeti változóként és a JVM paramétereként is megadható felhasználói CLASSPATH. A `-jar` kapcsoló mellett nem jut szóhoz.

Indítás dupla kattintással

Egy futtható, grafikus JAR-állomány úgy is fut, ha kettőt kattintunk az állomány nevén. Konzolos alkalmazásokra ez nem működik, mert a Windows alapértelmezésben a `javaw-t` hívja meg, az pedig nem nyit konzolablakot, a konzolra kiírt szövegek tehát elvesznek.

A konzolos alkalmazást csak parancssorral vagy parancsállományból futthatjuk, a `java` parancccsal.

A `CsomagApp.jar` például konzolos alkalmazás (a program kiviteli eszköze a konzol), dupla kattintásra tehát nem indul el. Futtassuk parancssorral vagy parancsállományból! A parancsállomány végére tegyük egy `pause` parancsot, másképp futás után a konzol rögtön becsukódik:

```
%JAVA_HOME%\bin\java -jar CsomagApp.jar
pause
```

A grafikus JAR-alkalmazások dupla kattintásra is futnak. Próbálja meg futtatni például a következő alkalmazások JAR-állományát:

- ◆ OOTPJava1/08Esettanulmany/GyusziJatszik alkalmazás;
- ◆ OOTPJava2/Mintaprogramok/20Esettanulmanyok összes alkalmazása (csak a `javalib` nem fut).

Ezekbe az alkalmazásokba be vannak csomagolva a `javalib`nek a futáshoz szükséges osztályai.

Megjegyzés: Érdemes tanulmányozni az OOTPJava2 könyv esettanulmányait, a JAR-állományokat, és a projektkönyvtárak parancsállományait!

Tesztkérdések

1.1. Jelölje be az összes igaz állítást!

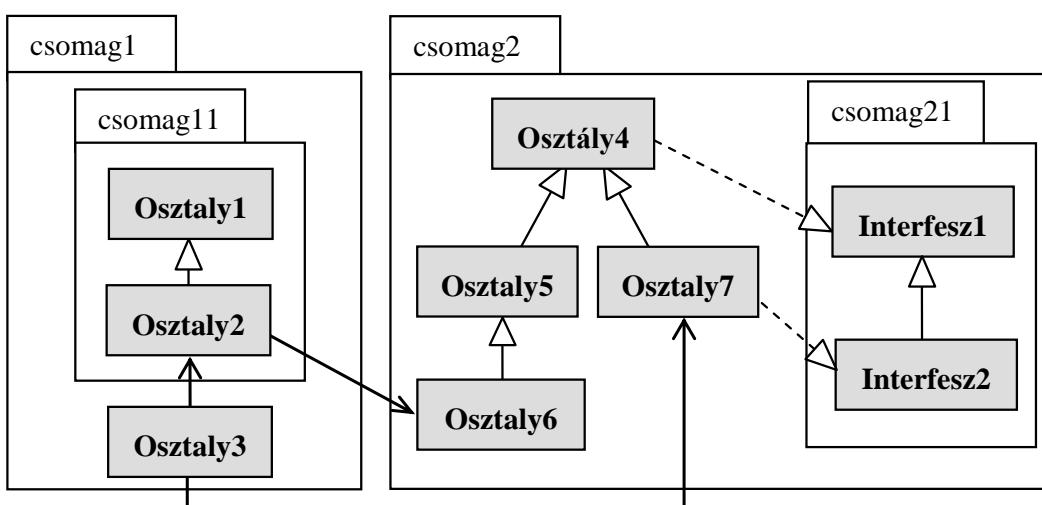
- a) A csomag csak logikai szintű csoportosítás.
- b) Egy csomag csak publikus osztályokat tartalmazhat.

- c) A lefordított bájtkódok gyökérkönyvtárát kimeneti útvonalnak nevezzük.
d) A forrásútvonal és kimeneti útvonal ugyanaz a könyvtár is lehet.
- 1.2. Mi alapján épül fel a célkönyvtár struktúrája? Jelölje be az egyetlen helyes választ!
- a) a forrásútvonal alapján
 - b) a forráskódbeli package deklarációk alapján
 - c) a projektpanel struktúrája alapján
 - d) egyéb szempont alapján
- 1.3. Jelölje be az összes igaz állítást!
- a) Az ugyanabban a csomagban szereplő deklarációk fordítási egységeit minden ugyanabba a könyvtárba kell tenni.
 - b) Egy fordítási egységben csak egy osztályt lehet deklarálni.
 - c) A forrásútvonal és a kimeneti útvonal könyvtárfelépítése szimmetrikus kell, hogy legyen.
 - d) Egy csomag akárhány csomagot tartalmazhat.
- 1.4. Jelölje be az összes igaz állítást!
- a) Elnevezési konvenció szerint a csomag neve nagybetűvel kezdődik, a többi kisbetű.
 - b) Egy fordítási egység kötelezően tartalmaz package deklarációt.
 - c) Egy fordítási egység kötelezően tartalmaz import deklarációt.
 - d) A package deklaráció csak első utasítás lehet a fordítási egységen.
- 1.5. Jelölje be az összes igaz állítást!
- a) Egy osztálynak láthatóság szerint kétféle módosítója lehet: private vagy public.
 - b) Ha az osztály egy deklarációjában nem adunk meg láthatóságot, akkor arra a deklarációra a csomag más osztályaiból hivatkozni lehet.
 - c) A Java fordítóprogram osztályútvonala a hivatkozott deklarációk gyökérkönyvtárai.
 - d) A Java fordítóprogram kimeneti útvonala a forrásállományok gyökérkönyvtára.
- 1.6. Jelölje be az összes igaz állítást!
- a) A projekt logikailag összetartozó állományok és környezeti beállítások gyűjteménye.
 - b) A projektállomány tartalmazza a projekt alapvető útvonalait.
 - c) A projekt csak java és class kiterjesztésű állományokat tartalmazhat.
 - d) A Java forráskódokat a projektpanelen lehet szerkeszteni.
- 1.7. Jelölje be az összes igaz állítást!
- a) A JAR a Java Application Runner rövidítése.
 - b) A JAR-állomány egy Java projekt állományait tömöríti.
 - c) A JAR-állományba csak class állományok tehetők be.
 - d) minden JAR-állománynak kötelezően tartalmaznia kell egy aláírásállományt; ennek az állománynak a neve tetszőleges lehet.
- 1.8. Jelölje be az összes igaz állítást!
- a) Futtatható JAR-állományban minden szerepel aláírásállomány.
 - b) A JAR-állományba minden esetben bele kell csomagolni a használandó könyvtárakat.
 - c) Az aláírásállomány nem olvasható formátumú.
 - d) Az Archive Builder Java forráskódok fordítására való.

- 1.9. Jelölje be az összes igaz állítást!
- A JVM arra való, hogy JAR-állományt állítsunk össze vele.
 - A `java` parancs a futtatáshoz nyit egy konzolablakot.
 - A JVM-nek minden esetben meg kell adni egy szabványos main metódust tartalmazó fő osztályt.
 - JAR-állományt -jar kapcsoló nélkül is futthatunk.

Feladatok

- 1.1. (A) Keressen a lemezen JBuilder-projekteket, és futtassa őket! A könyv mellékletében és a JBuilder samples könyvtárában például vannak projektek.
- 1.2. (A) A tankönyv 1. kötetében levő néhány feladatot írja át projektkörnyezetre! Válassza ki például a következő mintafeladatokat:
14. fejezet, `Bank.java` (*Bank.jpx*)
 17. fejezet, `RaktarProgram.java` (*RaktarProgram.jpx*)
 20. fejezet, `AutoEladas.java` (*AutoEladas.jpx*)
- 1.3. (B) Hozzon létre egy projektet a következő osztálydiagram alapján! A terv és a fejezetben használt kódolási szabály alapján készítsen kódot, azt fordítsa le, és készítse el a futtatható JAR-állományt is! (*CsomagApp2.jpx*)



- 1.4. (A) Készítsen a lemez melléklet valamely esettanulmányához egy projektet úgy, hogy abból csak az `src` könyvtárat ismeri. A `_MyPrograms` könyvtár alatt dolgozzon! Futtassa a projektet!
- 1.5. Az előző feladathoz hasonlóan készítsen projektet a Borland `TextEditor` mintaprogramjához!

2. Öröklődés

A fejezet pontjai:

1. Az öröklődés fogalma
 2. Mintafeladat – Hengerprogram
 3. Az objektumreferencia statikus és dinamikus típusa
 4. Az utódosztály adatai és kapcsolatai
 5. Metódusok felülírása, dinamikus és statikus kötés
 6. this és super referencia
 7. this és super konstruktörök – konstruktörök láncolása
 8. Polimorfizmus
 9. Absztrakt metódus, absztrakt osztály
 10. Láthatóság
 11. Összefoglalás – metódusok nyomkövetése
-

Az öröklődés fogalmáról már volt. szó az 1. kötet 7. fejezetében. Ebben a fejezetben átismételjük és kibővítjük e fogalmakat, valamint bevezetjük az ide tartozó Java kódolási szabályokat. A szabályok tárgyalásában mindenkor a fejezet elején található mintaprogramra fogunk hivatkozni.

2.1. Az öröklődés fogalma

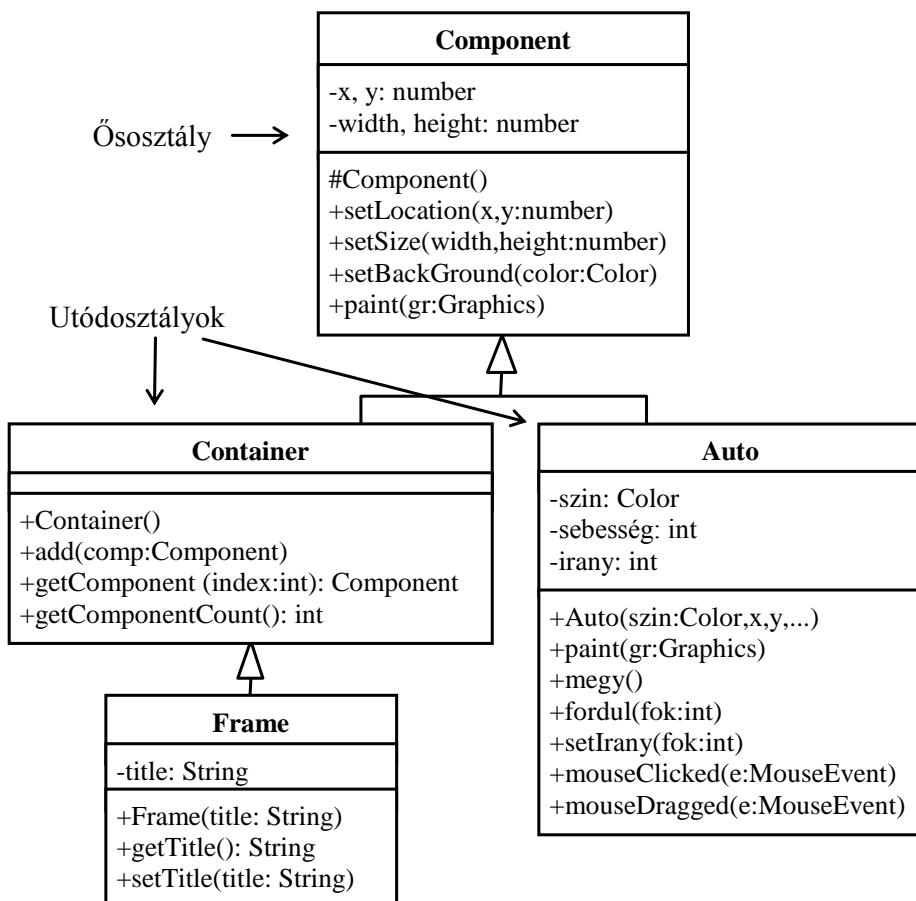
Az öröklődéssel, más néven kiterjesztéssel továbbfejlesztünk egy osztályt – újabb tulajdonságokat és metódusokat adhatunk hozzá, valamint átírhatjuk már meglévő metódusait. Az öröklődés technikáját két esetben használjuk: specializáláskor egy már meglévő osztályt veszünk alapul, általánosításkor pedig egymáshoz hasonlító osztályokból közös tulajdonságokat emelünk ki, és az így kialakított osztályt fejlesztjük tovább.

Specializálás (specialization): Az a folyamat, amellyel egy doleg (objektum) leírásához egyedi jellemzőket adunk hozzá.

Általánosítás (generalization): Az a folyamat, amellyel több doleg (objektum) leírásából kiemeljük a közös jellemzőket.

Öröklődés (inheritance) más néven **kiterjesztés** (extension): Egy már meglévő osztály kiterjesztése, továbbfejlesztése. A már meglévő osztály az **ősosztály**, a kiterjesztett osztály pedig a leszármazott, más szóval származtatott osztály, **utódosztály**. Az utódosztály az ősosztály specializálása. Az öröklőést az osztályok közötti „az egy”, vagy „**olyan, mint**” kapcsolatnak is nevezik.

Az UML-ben az utódosztályból egy nyíl mutat az ősosztályra; a nyíl hegye egy üres, zárt háromszög (2.1. ábra). A nyíl iránya a függéséget fejezi ki: az utódosztály függ az ősosztálytól. Több utód esetén rajzolhatunk külön nyílakat is, de össze is húzhatjuk őket az ábrán látható módon.



2.1. ábra. Öröklődés

Az ábrán az autó „az egy” komponens – az `Auto` osztály a `Component` osztály utódja, specializációja. A képernyön az autónak van bal felső sarka (`x, y`), mérete (`width, height`), színe, sebessége és iránya. A `Frame` (keret) szintén a `Component` osztály leszármazottja – van bal felső sarka (`x, y`) és mérete (`width, height`). A `Frame` egyúttal `Container` is, vagyis egyéb komponensek tárolóhelye – az `add` metódussal lehet más komponenseket beleenni. A `Frame`-nek az ösökhöz képest van egy további adata is, a keret címe (`title`).

Megjegyzések:

- A `Component`, a `Container` és a `Frame` az API osztálya. Az ábrán csak néhányat tüntetünk fel a metódusaik közül.
- Az ősosztályt általában az utódosztály felett szokták elhelyezni úgy, hogy a nyíl felfelé mutasson. Ez azonban egyáltalán nem szabály, a nyíl bármilyen irányba mutathat; az a fontos, hogy az ábra érthető legyen.

A Java nyelvben az utódosztály fejében az osztálynév után meg kell adnunk az `extends` (kiterjeszt) kulcsszót és az ősosztály nevét:

```
class <OsztályAzonosító> extends <OsztályAzonosító> ...
```

Például:

```
class Auto extends Component {  
    ...  
}
```

Osztályhierarchia-diagram: Csak öröklési kapcsolatokat ábrázoló osztálydiagram.

Egyszeres öröklés: Az öröklés egyszeres, ha egy osztálynak csak egy közvetlen őse lehet.

Többszörös öröklés: Az öröklés többszörös, ha egy osztálynak több őse is lehet.

Az öröklés szabályai:

- Egy osztályból több osztály is származtatható.
- Egy osztályhierarchia mélysége elvileg tetszőleges lehet, de a sok szint könnyen áttekinthetetlenné válik.
- Az öröklés tranzitív: Ha B örökli A-t és C örökli B-t, akkor C örökli A-t is.

Szabályok a Javában:

- Egy osztálynak csak egy közvetlen őse lehet (egyszeres öröklés).
- Az `Object` implicit őse minden osztálynak.

A következő feladat egy öröklést bemutató mintafeladat; végig erre fogunk hivakozni a fejezetben.

2.2. Mintafeladat – Hengerprogram

Feladat – Hengerprogram

Különböző hengereket szeretnénk nyilvántartani:

- vannak olyan hengerek, amelyek csupán mértani testek. Ezeknek van sugaruk és magasságuk;
- vannak rudak, melyeknek súlyuk is van (tömör hengerek);
- végül vannak csövek: valamekkora falvastagságú tömör, lyukas hengerek.

Készítsük el a megadott hengerek osztályait! Inicializáláskor a hengereknek értelemszerűen megadhatók a következő adatok: sugár, magasság, fajsúly és falvastagság. A fajsúlyt nem kötelező megadni, az alapértelmezésben legyen 1 (ez műanyagnak vagy nehezebb fának felelhet meg)! minden henger legyen képes visszaadni alapadatait, térfogatát és esetleges súlyát! A `toString()` metódus adja meg a henger szöveges reprezentációját!

A fent leírt osztályokból példányosítunk néhányat, és tegyük bele őket egy konténerbe! Végezzük el a következő feladatokat:

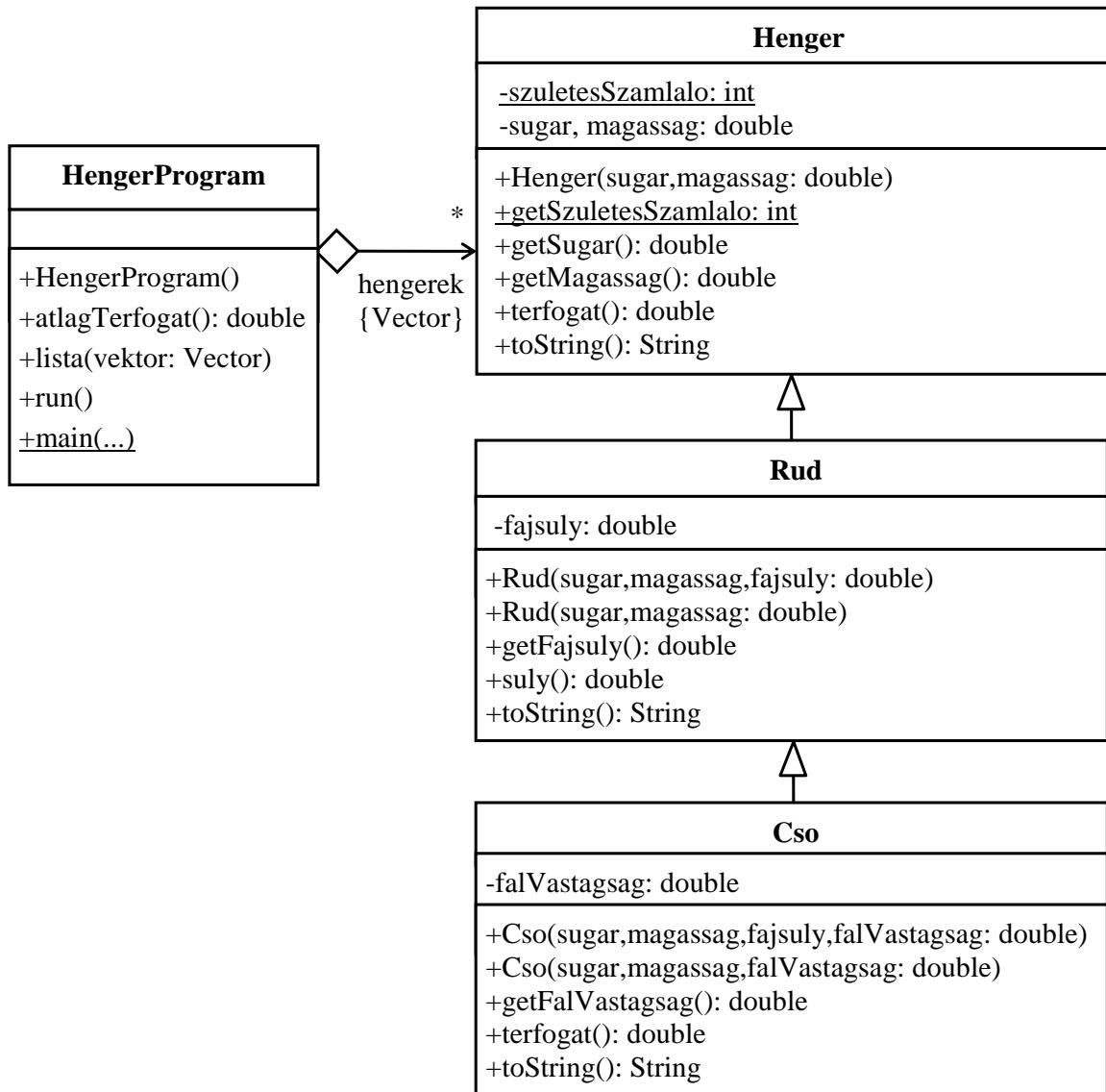
- Írjuk ki a hengerek számát, és listázzuk ki a hengerek adatait!
- Számítsuk ki a konténerben található hengerek átlagtérfogatát!
- Listázzuk ki a csövek adatait, adjuk meg a csövek számát és összsúlyukat!

Számoljuk és a program végén írjuk ki a program futása közben született hengerek számát!

A feladat olvasásakor azonnal lehet látni, hogy itt háromfélé hengerről van szó: egyszerű hengerről, rúdról és csőről. A következőket állíthatjuk:

- ◆ a rúd „az egy” henger, de súlya is van;
- ◆ a cső „az egy” rúd, de belül lyukas.

Ezen állítások alapján megpróbáljuk osztályainkat öröklés útján előállítani. A `HengerProgram` osztálydiagramja az 2.2. ábrán látható. Az osztályleírások után megadjuk a program forráskódját, majd a forráskód részletes elemzését. Az osztályokat külön forrásállományokban helyezzük el; így a teljes forráskód áttekinthetőbb.



2.2. ábra. A HengerProgram osztálydiagramja

Osztályleírások

Henger

Az egyszerű hengernek csak sugara és magassága van. A **szuletesSzamlalo** egy osztályadat: ebben tartjuk nyilván a program futása alatt született példányok számát. Az objektum meg tudja mondani a jellemzőit, valamint a térfogatát. Szöveges reprezentációja, a **toString()** is ezeket adja vissza. A **Henger** osztályt már eddig gyakorlatunkkal is elkészítetjük.

Rud

A `Rud` osztályú objektum olyan henger, amelynek súlya is van, ezért az eddigi adatokon kívül szüksége van egy `fajsuly` adatra is. A rúdnak két konstruktora van: ha nem adjuk meg a fajsúlyt, akkor az 1 lesz. Egy `Rud` mindenzt megmondhatja, amit egy egyszerű `Henger`, sőt megmondja fajsúlyát és súlyát is. Több adata lévén, szöveges reprezentációja is bővül a hengerhez képest.

Cso

A `Cso` osztályt a `Rud` osztályból származtatjuk. Az eddigi adatok kiegészülnek a falvastagsággal; azt a konstruktorkban is meg kell adni. Egy cső már minden meg tud mondani: sugarát, magasságát, fajsúlyát, súlyát, falvastagságát, térfogatát. Szöveges reprezentációja is felsorolja mindezt.

HengerProgram

A hengereket a `HengerProgram` osztályban manipuláljuk. Az osztálynak nincs explicit ōse (az `Object` az ōse), és egy-sok kapcsolatban áll a hengerekkel. A kapcsolat közvetlenül a `HengerProgram` és a `Henger` osztály között van, de a kapcsolatnak minden olyan objektum részese lehet, amelynek osztálya a `Henger` osztályon „ló” (annak utódja). Az egy-sok kapcsolatot a `Vector` segítségével hozzuk létre.

Forráskód

Projekt: `HengerProgram`
Csomagok: -

Henger.java

```
import extra.Format;

class Henger { //1
    private static int szuletesSzamlalo=0; //2
    private double sugar, magassag;

    public Henger(double sugar, double magassag) { //3
        this.sugar = sugar;
        this.magassag = magassag;
        szuletesSzamlalo++;
    }

    public static int getSzuletesSzamlalo() { //4
        return szuletesSzamlalo;
    }

    public double getSugar() { //5
        return sugar;
    }
}
```

```

public double getMagassag() { //6
    return magassag;
}

public double terfogat() { //7
    return sugar*sugar*Math.PI*magassag;
}

public String toString() { //8
    return "\n"+getClass().getName() +
        "\nSugár: "+Format.right(getSugar(),0,2) +
        " Magasság: "+Format.right(getMagassag(),0,2) +
        " Térfogat: "+Format.right(terfogat(),0,2);
}
}

```

Henger.java elemzése

- ◆ //1: A Henger osztály feje. A Henger-nek csak az Object az őse.
- ◆ //2: A szuletesSzamlalo osztályadat; ebben tartjuk nyilván a program futása alatt a Henger osztályban vagy valamelyik leszármazottjában létrejött objektumok számát. A sugar és a magassag a Henger példányadatai.
- ◆ //3: Konstruktor. Beállítja a sugar és magassag példányadatokat, majd megnöveli a szuletesSzamlalo osztályváltozót. Ez a konstruktor minden egyes példány megszületésekor végrehajtódik, függetlenül attól, hogy Henger vagy annak egy utódja született meg. A szuletesSzamlalo aktuális értéke tehát megadja, hogy összesen hány ilyen objektum született a program futása alatt.
- ◆ Figyelem! Az ideiglenesen létrehozott (és azonnal elengedett) objektumok is megnövelik a számláló értékét!
- ◆ //4-//6: Adatlekérdező (get) metódusok.
- ◆ //7: Megadja a henger térfogatát.
- ◆ //8: Szövegesen reprezentálja a hengert. Ahogy az a forráskódban látható, elsőként az osztály nevét adjuk meg a getClass().getName() metódus meghívásával. A getClass() az Object osztályban van definiálva, és visszaad egy Class típusú objektumot a megszólított objektum osztályának adataival, egyebek között az osztály nevét. Ezután megadunk minden lehetséges adatot: a henger sugarát, magasságát és térfogatát.

Rud.java

```

import extra.Format;

class Rud extends Henger { //1
    private double fajsuly; //2
}

```

```

public Rud(double sugar, double magassag,
           double fajsuly) {                                //3
    super(sugar,magassag);
    this.fajsuly = fajsuly;
}

public Rud(double sugar, double magassag) {            //4
    this(sugar, magassag, 1);
}

public double getFajsuly() {                           //5
    return fajsuly;
}

public double suly() {                                //6
    return terfogat()*fajsuly;
}

public String toString() {                            //7
    return super.toString() +
        "\nFajsuly: "+Format.right(getFajsuly(),0,2) +
        " Suly: "+Format.right(suly(),0,2);
}
}

```

Rud.java elemzése

- ◆ //1: A Rud osztály a Henger osztályból származik (Kiterjeszti a Henger-t, mászzóval a Henger utódja). Ezt az osztály fejében az extends (kiterjeszti) kulcsszóval adjuk meg: Rud extends Henger.
- ◆ //2: A fajsuly egy további adat. A sugár és magasság adaton kívül a rúdnak fajsúlya is van.
- ◆ //3: Konstruktort nem lehet örökölni, azt minden osztályban meg kell írnunk. A rúdnak két konstruktora is van. Ebben a konstruktorkban a paraméterek száma három. A sugár és a magasság beállításához meghívjuk az ōskonstruktort (a super minősítő segítségével), majd beállítjuk az új adatot, a fajsúlyt.
- ◆ //4: Kétparaméteres konstruktur. Ha az objektum létrehozásakor csak két paramétert adnak meg, akkor a fajsúlyt 1-re kell állítanunk. Egyszerűen meghívjuk az osztály másik, háromparaméteres konstruktörét, s abban 1-nek vesszük a fajsúlyt.
- ◆ //5: A getFajsuly() egy új kiolvasó metódus.
- ◆ //6: A suly() metódus kiszámolja a súlyt a térfogat és a fajsúly alapján. A hívott terfogat() metódus nem ebben az osztályban van, azt az ōosztály definiálta.
- ◆ //7: Van ugyan egy örökolt toString() metódusunk, de azt nem tudjuk egy az egyben használni, hiszen a rúdra a henger eddigi adatain kívül a fajsúly és a súly is jellemző. Meghívjuk a super.toString() metódust, hogy állítsa össze az ōben definiált adatokat, majd hozzáteszük az osztály saját adatait.

Cso.java

```

import extra.Format;

class Cso extends Rud { //1
    private double falVastagsag; //2

    public Cso(double sugar, double magassag,
               double falVastagsag, double fajsuly) { //3
        super(sugar,magassag,fajsuly);
        this.falVastagsag = falVastagsag;
    }

    public Cso(double sugar, double magassag,
               double falVastagsag) { //4
        this(sugar,magassag,falVastagsag,1);
    }

    public double getFalVastagsag() { //5
        return falVastagsag;
    }

    public double terfogat() { //6
        Henger belso = new Henger(
            getSugar()-falVastagsag,getMagassag());
        return super.terfogat()-belso.terfogat();
    }

    public String toString() { //7
        return super.toString() +
            " Falvastagsag: "+Format.right(getFalVastagsag(),0,2);
    }
}

```

Cso.java elemzése

- ◆ //1: A Cso osztály a Rud osztályt terjeszti ki.
- ◆ //2: A falVastagsag új adat: a lyukas hengernek falvastagsága is van.
- ◆ //3: Négyparaméteres konstruktur. A falvastagságot az öskonstruktorig képest a magasság és a fajsúly közé tesszük. Általában nem ajánlatos a paraméterek eredeti sorrendjét megváltoztatni, de most kivételesen indokolt, hiszen ennek az osztálynak a másik konstruktorig a fajsúlyt hagyjuk el, és középről paramétert elhagyni szintén zavaró lehet. Az implementációban a sugár, a magasság és a fajsúly beállításához a super minősítővel meghívjuk az öskonstruktort, ezután beállítjuk az új adatot, a falvastagságot.
- ◆ //4: Háromparaméteres konstruktur. A fajsúly alapértelmezésben 1. Az adatok beállításához meghívjuk az osztály négyparaméteres konstruktorig, 1 fajsúlytal.
- ◆ //5: A getFalVastagsag() saját kiolvasó metódus.

- ◆ //6: A Henger osztálynak van egy `terfogat()` metódusa – az a Rud osztályban megfelelt, de most nem használható. A cső térfogata más: ott a teljes térfogatból le kell vonnunk a lyuk térfogatát. A teljes térfogatot az ugyanolyan nevű ös metódus, a `super.terfogat()` ki tudja számolni. A lyuk térfogatának kiszámolásához létrehozunk egy `belso` nevű segédhengert; annak sugara a teljes sugár és a falvastagság különbsége, magassága pedig ugyanaz (a sugár privát adat, azt csak metódussal lehet kiolvasni az ösosztályból).
- ◆ //7: Egy csövet a rúdban deklarált adatokon kívül a falvastagság is jellemző. Az ös `toString()` által visszaadott adatokhoz tehát hozzá tesszük a falvastagságot is.

HengerProgram.java

```

import extra.Format;
import java.util.Vector;

public class HengerProgram {                                         //1
    private Vector hengerek = new Vector();                           //2

    public HengerProgram() {                                         //3
        hengerek.add(new Henger(1,4));
        hengerek.add(new Rud(0.5,4,2));
        hengerek.add(new Rud(0.5,4));
        hengerek.add(new Cso(5,5,0.5));
        hengerek.add(new Cso(5,5,0.5,2));
    }

    public double atlagTerfogat() {                                     //4
        Henger henger;
        double osszTerfogat = 0;
        for (int i=0; i<hengerek.size(); i++) {
            henger = (Henger)hengerek.get(i);
            osszTerfogat += henger.terfogat();
        }
        int meret = hengerek.size();
        return meret==0? 0 : osszTerfogat/meret;
    }

    public void lista(Vector vektor) {                                    //5
        for (int i=0; i<vektor.size(); i++) {
            System.out.println(vektor.get(i));
        }
    }

    public void run() {                                                 //6
        // Hengerek listázása:
        System.out.println("Hengerek száma: "+hengerek.size());
        lista(hengerek);

        // Átlagtérfogat számítása:
        System.out.println("\nÁtlagtérfogat: "+
Format.right(atlagTerfogat(),0,2));
    }
}

```

```

// Csövek listázása, összsúlyuk számítása:
double suly = 0; //7
System.out.println("\nCsövek listája");
for (int i=0; i<hengerek.size(); i++) {
    Object obj = hengerek.get(i);
    if (obj instanceof Cso) {
        System.out.println(obj);
        suly += ((Cso)obj).suly();
    }
}
System.out.println("\nCsövek súlya összesen: "+
    Format.right(suly,0,2));

// Született hengerek száma:
System.out.println("\nSzületett hengerek száma: "+
    Henger.getSzuletesSzamlalo()); //8
}

public static void main (String args[]) {
    new HengerProgram().run();
    Console.pressEnter();
}
}

```

HengerProgram.java elemzése

- ◆ //1: A HengerProgram a main metódust tartalmazó fő osztály: ez dolgozza fel a program hengereit.
- ◆ //2: A hengerek konténer hozza létre az egy-sok kapcsolatot. A vektorba elvileg bármilyen objektum betehető, de mi csak hengereket fogunk beleenni.
- ◆ //3: A konstruktorban létrehozunk egy egyszerű hengert, két rudat és két csövet. Ezeket sorra hozzáadjuk a hengerek vektorhoz.
- ◆ //4: Átlagtér fogat kiszámítása. A vektorban levő hengereket a get(i) üzenettel sorra kikérjük a konténertől. Mivel a kiadott objektumok referenciai Object típusúak, rájuk kell kényszerítenünk a Henger osztályt. Ez elegendő, mert a Henger-ben megtalálható a terfogat() metódus. Hogy konkrétan melyik terfogat() hajtóidik majd végre, azt a mutatott objektum valódi osztálya határozza meg: ha Rud-ra mutatunk, akkor a Rud osztály (örökolt) terfogat() metódusa fog futni, ha Cso-re mutatunk, akkor a Cso osztályé. A program mindenkor a jó térfogattal fog számolni.
- ◆ //5: A lista metódus bármilyen vektort kilistáz, tehát a hengerek vektort is: soronként kiírja az elemek toString() metódusát.
- ◆ //6: A run metódus több részből áll:
 - Először kilistázzuk a vektort, majd kiírjuk az átlagtér fogatot.
 - Ezután //7-ben kilistázzuk a csöveket, és párhuzamosan összegezzük a súlyukat. Végighaladunk a vektor minden objektumán: ha annak osztálya Cso (ezt az instanceof operátorral tudjuk lekérdezni), akkor kiírjuk az adatait és gyűjtjük a

súlyát. A súly kikéréséhez az objektumra rá kell kényszerítenünk a Cso osztályt. Ezt nyugodtan megtehetjük, hiszen az objektum osztálya biztosan Cso, most győződünk meg róla. A Rud osztályt is elegendő lett volna rákényszeríteni, hiszen már annak is van súlya.

- Végül //8-ban kiírjuk a program futása idején született objektumok számát. Látható, hogy bár a vektorméret 5, a született objektumok száma mégis 17. Ennek oka, hogy a Cso a térfogat kiszámításához minden alkalommal ideiglenesen létrehoz egy Henger objektumot (Cso, //6). Ez a henger nem hosszú életű ugyan, de konstruktora lefut, s így megnöveli a születésszámlálót.

A program futása

```
Hengerek száma: 5

Henger
Sugár: 1.00 Magasság: 4.00 Térfogat: 12.57

Rud
Sugár: 0.50 Magasság: 4.00 Térfogat: 3.14
Fajsúly: 2.00 Súly: 6.28

Rud
Sugár: 0.50 Magasság: 4.00 Térfogat: 3.14
Fajsúly: 1.00 Súly: 3.14

Cso
Sugár: 5.00 Magasság: 5.00 Térfogat: 74.61
Fajsúly: 1.00 Súly: 74.61 Falvastagság: 0.50

Cso
Sugár: 5.00 Magasság: 5.00 Térfogat: 74.61
Fajsúly: 2.00 Súly: 149.23 Falvastagság: 0.50

Átlag térfogat: 33.62

Csövek listája

Cso
Sugár: 5.00 Magasság: 5.00 Térfogat: 74.61
Fajsúly: 1.00 Súly: 74.61 Falvastagság: 0.50

Cso
Sugár: 5.00 Magasság: 5.00 Térfogat: 74.61
Fajsúly: 2.00 Súly: 149.23 Falvastagság: 0.50

Csövek súlya összesen: 223.84

Született hengerek száma: 17

<ENTER>
```

2.3. Az objektumreferencia statikus és dinamikus típusa

Egy objektum csak egyetlen osztályhoz tartozhat, s már a születéskor eldől, hogy melyikhez. Egy `Rud`-ból tehát sosem lesz `Cso`. De mint tudjuk, egy objektum referenciája (mutatója) nem szükségképpen olyan típusú, mint a mutatott objektum osztálya. Egy objektumot bármely űstípusú referenciaival azonosíthatunk. Igazak a következők:

- ◆ Egy objektumra több, különböző típusú referenciával is hivatkozhatunk. Egy `Rud` osztályú objektumot például azonosíthatunk `Object`, `Henger` vagy `Rud` referenciával, `Cso` típusúval viszont nem.
- ◆ Fordítva, egy adott típusú referenciával több, különböző osztályú objektumot azonosíthatunk. Egy `Rud` típusú referenciával például azonosíthatunk `Rud` és `Cso` osztályú objektumokat, de nem azonosíthatunk `Object` vagy `Henger` osztályú objektumokat.

Minden objektumreferenciának van egy **statikus típusa**; azt a deklarálásakor határozzuk meg. Ez a referencia a program futása közben különféle osztályú objektumokat azonosíthat. Az objektumreferencia által azonosított objektum osztálya a referencia valódi, más szóval **dinamikus típusa**.

Fontos, hogy a referencia statikus típusa a dinamikus típussal egyező legyen, vagy annak egy őse, mert különben futási hiba következhet be.

Értékadási kompatibilitás

Értékadási kompatibilitás: Értékadáskor a jobb oldalon álló objektumreferencia értékadás szempontjából akkor kompatibilis a bal oldalon álló objektumreferenciával, ha típusa megegyezik a bal oldaiéval vagy annak leszármazottja. Ellenkező esetben

- fordítási hiba keletkezik, ha ez már fordításkor kiderül, vagy
- futási hiba (`ClassCastException`) áll elő, ha ez csak futáskor, egy rossz üzenet küldésekor derül ki.

Ha az értékadás jobb oldalán referencia típusú változó van, akkor a fordító annak csak a statikus típusát vizsgálhatja. Hogy mi „lapul” a hivatkozás mögött, azt a fordító nem tudhatja.

Tekintsük például a következő deklarációkat:

```
Object obj;           // obj statikus típusa Object
Henger h1, h2;       // h1 és h2 statikus típusa Henger
Rud rud;             // rud statikus típusa Rud
Cso cso;             // cso statikus típusa Cso
```

Szintaktikailag helyes értékkedások a következők:

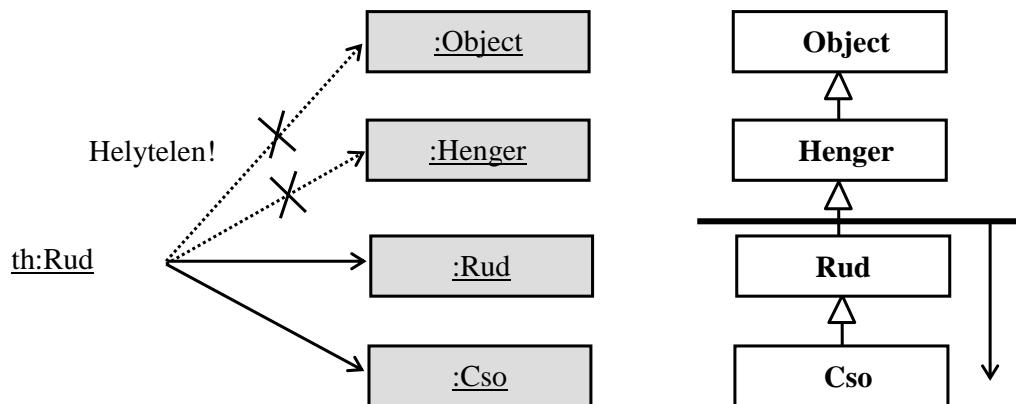
```
obj = new Henger(1,3);           // obj dinamikus típusa Henger
h1 = new Rud(2,2,1.3);          // h1 dina. típusa Rud
rud = new Cso(1,4,2);           // rud dinamikus típusa Cso
obj = h1;                      // obj dina. típusa Rud
h2 = h1;                       // h2 dina. típusa Rud
```

Szintaktikailag helytelen értékkedások:

```
// cso = new Henger(2,6); cso stat. típusa bővebb, mint a Henger!
// h1 = obj; h1 stat. típusa bővebb, mint az obj deklarált típusa!
// rud = h1;   rud stat. típusa bővebb, mint a h1 deklarált
típusa!
```

Példa futási hibára:

```
obj = new Henger(1,5); // obj mögött Henger van!
cso = (Cso) obj; // Kényszerítés - a cso mögött Henger van!
// Szintaktikailag jó, mert van Henger.terfogat():
double t = cso.terfogat();
// Futási hiba, mert nincs Henger.getFalVastagsag()!:
// double v = cso.getFalVastagsag();
```



2.3. ábra. Az objektumreferencia statikus és dinamikus típusa

Az 2.3. ábráról a következő olvasható le: ha a `rud` referencia statikus típusa `Rud`, akkor dinamikus típusa csak `Rud`, vagy annak egy leszármazottja lehet. A dinamikus típus tehát csak `Rud` vagy `Cso` lehet, `Object` vagy `Henger` nem. Típuskényszerítés révén persze hivatkozhatunk őstípusú objektumra, de ebben az esetben könnyen támadhat futási hiba.

instanceof operátor

Az `instanceof` (magyarul: példánya) operátorral lekérdezhető a mutatott objektum típusa:

```
<objektum> instanceof <Típus>
```

Az operátor bal oldalán egy objektum szerepel, jobb oldalán pedig egy típus (osztály vagy interfész). Az eredmény egy boolean típusú érték: ha az objektum típusa (az objektum-referencia dinamikus típusa) a megadott Típus vagy annak egy leszármazottja, akkor `true`, egyébként `false`.

Megjegyzés: Az `instanceof` operátorban kis o betű szerepel.

Például:

```
Object obj = new String("Valami");
if (obj instanceof Object) ...      // true
if (obj instanceof String) ...     // true
if (obj instanceof Integer) ...    // false
if ("Semmi" instanceof String) ... // true
```

A következő példában csak az első két kiírás hajtódiik végre:

```
Object obj = new Rud(2,2,1.3);
if (obj instanceof Henger)           // -> true
    System.out.println("obj egy Henger");
if (obj instanceof Rud)             // -> true
    System.out.println("obj egy Rud");
if (obj instanceof Cso)            // -> false
    System.out.println("obj egy Cso");
```

Automatikus típuskonverzió felfelé (upcasting)

Automatikus típuskonverzió felfelé: Egy utódtípusú referencia egyszerűen értékül adható egy őstípusú referenciának. Ilyen esetben az utódreferencia őstípusúvá konvertálódik (általánosabb lesz), vagyis lebutul, elveszti valódi „énjét”. A típuskonverzió (casting) automatikusan felfelé (up), az őstípus felé irányul.

Egy objektumnak csak olyan üzenetek küldhetők, amelyek a referencia statikus osztályában deklarálva vannak. A rossz üzenetküldés már fordításkor kiderül.

Például:

```
String str = "Kakukk";
System.out.println(str.toLowerCase());        // OK
Object obj = str;                          // String -> Object típuskonverzió
// System.out.println(obj.toLowerCase());    Fordítási hiba!
```

Az `obj=str` értékkadáskor `str`-re nem kell rákényszeríteni az `Object` osztályt, hiszen a `String` „az egy” `Object`. Az értékkadás után minden referencia, `obj` és `str` is ugyanazt a `String` objektumot azonosítja. Az `obj` referencia azonban egy kicsit buta, és nem látja a `String` valódi képességeit: `obj`-on keresztül csak azok az üzenetek küldhetők `str`-nek, amelyekhez már az `Object` osztályban is tartoznak metódusok. Egy `String` típusú referencia azonban jól bánik a `String` objektummal, látja az a `String` minden publikus képességét.

A `HengerProgram`ban automatikus típuskonverzió történik, amikor egy hengert a konténerbe dobunk, hiszen a `Vector` osztály `add(Object obj)` metódusának paramétere `Object` típusú, így az bármilyen `Object` leszármazottat képes fogadni. A konténerbe betett minden objektum `Object` típusúvá konvertálódik:

```
hengerek.add(new Rud(0.5, 4));
hengerek.add(new Cso(5, 5, 0.5));
```

Típuskényszerítés lefelé (downcasting)

Típuskényszerítés lefelé: Egy őstípusú referencia nem adható egyszerűen értékül egy utódtípusú referenciának. Ha ezt mindenkor meg akarjuk tenni, akkor az ősre rá kell kényszerítenünk az utódtípust, mert az értékkadás jobb oldalán mindenkor „bövebb” típusnak kell lennie. Ilyenkor a programozónak kell vigyáznia, hogy az objektum futás közben csak olyan üzenetet kaphasson, amely benne van az objektum osztályában. Futás-kor kiderülhet, hogy a kényszerítés hamis; ilyenkor a program `ClassCastException` kivételt ejt.

Például:

```
Object obj = "Szoveg";
String str = (String) obj; //1
// String str = obj; //2 Fordítási hiba!
```

//1-ben az `obj`-ra rákényszerítjük a `String` típust, így értékül adható `str`-nek. A rákényszerítés nem okoz majd hibát, hiszen `obj` mögött igazából egy `String` bújik meg. //2 szintaktikailag hibás, mert a fordító nem tudhatja, hogy az `obj` referencia valójában `String`-et azonosít.

A következő példában szintaktikailag jók a típuskényszerítések. Az `obj` kezdetben egy `String`-et azonosít, majd később rákényszerítjük az `Integer`-t (a fordító ezt megengedi, ilyen csalafintásokat már nem tud kibogozni). Az `obj` mögött azonban egy `String` húzódik meg, ezért az `Integer`-nek szánt üzenettől „meghibban”:

```
Object obj = "Kakas";
Integer iObj = (Integer) obj;
System.out.println(iObj.doubleValue()); // ClassCastException!
```

A HengerProgramban a konténerből kikért objektum hivatkozása le van butítva, szintaktikailag csak Object típusú. De mi tudjuk, hogy az objektum legalább Henger típusú, s ezt a típust rá is kényszerítjük a hivatkozásra, mert különben nem kérdezhetnénk meg tőle a térfogatát:

```
Henger henger;
double osszTerfogat = 0;
for (int i=0; i<hengerek.size(); i++) {
    henger = (Henger)(hengerek.get(i));
    osszTerfogat += henger.terfogat();
}
```

2.4. Az utódosztály adatai és kapcsolatai

A leszármazott osztály örökli az ősosztályban levő adatokat és kapcsolatokat. Az adatok lehetnek primitív típusúak vagy objektumok, példány- vagy osztályadatok. A kapcsolatok speciális adatok, mert a kapcsolatokat objektumreferenciákkal hozzuk létre: az egy–egy kapcsolatot az objektummal, az egy–sok kapcsolatot egy konténerobjektummal.

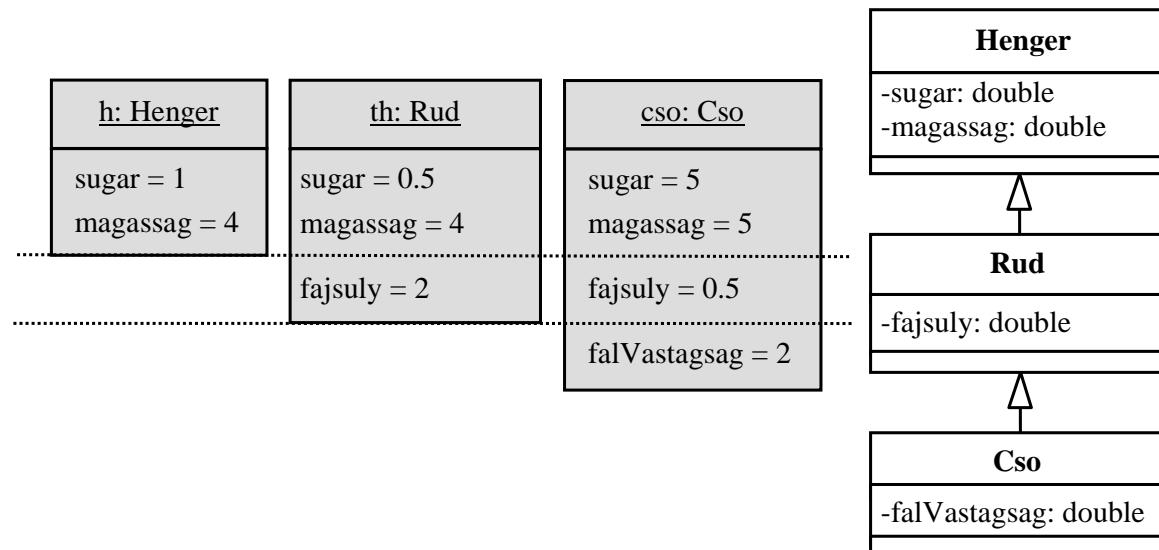
Példányadatok, az objektum memória foglalása

Egy példányban fizikailag benne van az összes – az objektum osztályában és annak valamely ősosztályában deklarált – példányadat. Ha tehát a Henger-nek van sugara és magassága, akkor a Rud-nak is megvannak ugyanezek az adatai, de a rúdnak ezeken kívül fajsúlya is van. Az 2.4. ábrán látható, hogy egy egyszerű henger két adatot tárol, a rúd egyetlen többet, a cső pedig már négyet. Egy objektum gyakorlatilag annyi memóriahelyet foglal le, amennyire szüksége van a példányadatok tárolásához. A tárfoglalás a deklaráció sorrendjét követi: először az ősádatok kapnak helyet, azután az utódadatok. Az utódosztályból létrehozott objektum mérete tehát legalább akkora, mint az ősosztályból létrehozott objektumé.

Egy osztály példányában a következő adatok vannak jelen: az ősosztály(ok)ban deklarált példányadatok + a saját osztályban deklarált példányadatok.

Az osztályadatok, mint tudjuk, nem kapnak helyet a példányokban. Egy osztályadatból csak egy van, és az magában az osztályban van meg. minden osztályadat addig él, ameddig maga az osztály.

● Figyelem! Az utódosztály akkor is örökli az ősosztály adatait és kapcsolatait, ha nem látja őket (private az elérhetőségük).



2.4. ábra. Az objektum memória foglalása

Hivatkozás az adatokra

Hivatkozás kívülről: Egy osztályban deklarált bármely publikus példányadat, illetve osztályadat manipulálható az objektum, illetve az osztály megszólításával, akár a maga osztályában, akár annak egy űsében van deklarálva, kivéve azt az esetet, ha az adatot eltakarja egy ugyanolyan nevű másik adat (lásd Adatok takarása, következő oldal).

Az adatok általában privát elérhetőségűek, hiszen egy manipulálható adat könnyen megsértheti az objektum zártságát. Így a `Henger` és `Rud` osztályok adatai is privátok. De ha publikusak lennének, akkor szintaktikailag helyesek lennének a következő utasítások:

```
// Rud rud;
// rud.sugar = 0;
// rud.fajsuly = 0.5;
// Henger.szuletesSzamlalo++;
// rud.szuletesSzamlalo++;
```

Hivatkozás osztályból: Bármely űsadatra hivatkozhatunk annak nevével, kivéve, ha az űsadat privát vagy ha eltakarja egy ugyanolyan nevű másik adat.

Példa a hivatkozásra (az adatok takarását lásd a következő oldalon):

```
// HivAdat.java
class C1 {
    static String s = "C1";
    int a = 5;
}

class C2 extends C1 {
    int b = 10;
    void kiir() { System.out.println(s+" "+a+" "+b); }
}

...
C2 obj = new C2();
obj.kiir(); // -> C1 5 10
```



Hivatkozhatunk az ōsadatokra is

A hivatkozásokkal mindenkor a referencia statikus osztályában, vagy annak egy ōsében deklarált adatra hivatkozunk. Az alábbi deklarációk szerint `obj1` és `obj2` ugyanarra az objektumra mutat ugyan, de az `obj1.b`-re való hivatkozás mégis hibás, mert `obj1` statikus típusában, `C1`-ben nincsen `b`:

```
C2 obj2 = new C2();
C1 obj1 = obj2;
System.out.println(obj2.b); // -> 10
//System.out.println(obj1.b); // -> szintaktikai hiba!
```

Adatok takarása

Adatokat nem lehet felülírni (ugyanolyan néven újabb memóriaterületet foglalni), hiszen attól az ōsosztály esetleg helytelenül működne. **Egy adatot** azonban **el lehet takarni** (be lehet árnyékolni) **egy ugyanolyan nevű másik adat deklarálásával**, ahogyan a metódus lokális változójával is eltakarhatunk egy osztályszintű adatot.

Az ōsben deklarált, eltakart adatot minősítéssel lehet elérni az utódosztályból, feltéve, hogy az adat nem privát:

- példányadat esetén a `super` (felette levő) minősítővel hivatkozhatunk az ōsadatra. Ezzel csak a példányadat felett álló legközelebbi ugyanolyan nevű adatot érjük el (nem biztos persze, hogy az a közvetlen ōsben van deklarálva);
- osztályadat esetén az osztály megnevezésével hivatkozhatunk az ōsadatra. Ha a megnevezett osztályban nincs ilyen nevű adat, akkor a legközelebbi felette álló adatot kapjuk.

Például:

```
// AdatokTakarasa.java
class C1 {
    static String s = "C1 ";
    int a = 1;
    int b = 10;
}
```

```

class C2 extends C1 {
    int a = 2;
}

class C3 extends C2 {
    static String s = "C3 ";
    int b = 30;
    public void kiir1() { System.out.println(C1.s+C2.s+s); }
    //public void kiir2() { System.out.println(C1.a+C2.a+C3.a); }
    public void kiir3() { System.out.println(super.a+a); }
    public void kiir4() { System.out.println(super.b+b); }
}

```

Elemezzük a C3 osztály kiíró metódusait:

- ◆ kiir1: Helyes. Statikus adatot osztállyal kell minősítenünk. Kiírás: C1 C1 C3
- ◆ kiir2: **Szintaktikailag hibás!** Példányadatot nem lehet osztállyal minősíteni!
- ◆ kiir3: Itt a super.a és az a is a C2-ben deklarált a-t jelenti. Kiírás: 4
- ◆ kiir4: A super.b most a C1-beli b-t jelenti, mert az öröklési ágon ott van a legközelebb b. Kiírás: 40

Egy C3 osztályú objektumban a következő változók foglalnak helyet: a C1-beli a és b, a C2-beli a és a C3-beli b. A statikus C1.s és C3.s változót az objektumok nem tárolják, mindkettő a maga osztályában foglal helyet.

❖ Csak indokolt esetben takarjuk el az adatokat, mert az erősen zavaró!

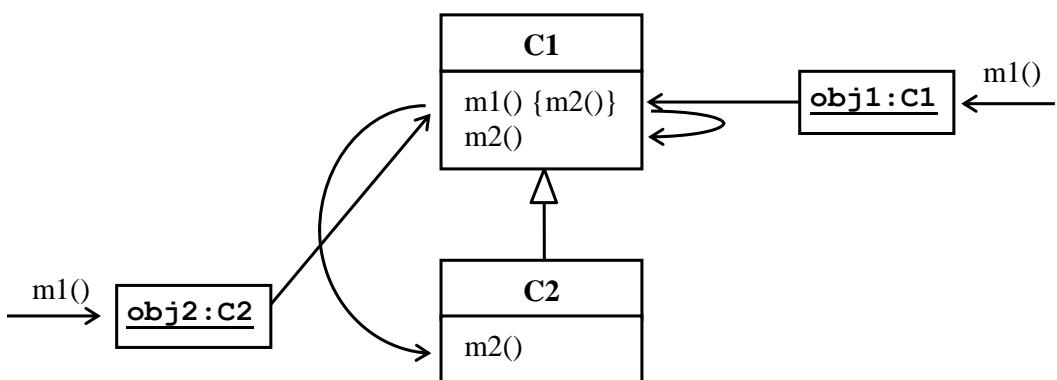
2.5. Metódus felülírása, dinamikus és statikus kötés

Példánymetódus felülírása (overriding): Az utódosztály elvileg az ősosztály bármely példánymetódusát felülírhatja. A felülírással az eredeti metódust az új metódussal cseréljük fel. Szabályok:

- Csak akkor beszélhetünk felülírásról, ha a metódusok szignatúrája megegyezik (különben csak különböző, túlterhelt metódusokról van szó).
- A felülírt metódus visszatérési típusának meg kell egyeznie az eredetivel.
- Egy metódust az utódosztályban nem lehet felülírni, ha az a) private, hiszen akkor nem is lehet látni. Lehet viszont egy ugyanolyan nevű új metódust írni (ez hiba forrása is lehet!); b) final, mert az ősosztály megalkotója így rendelkezett (a final metódus végleges, nem lehet felülírni); vagy ha c) static, mert csak példánymetódust lehet felülírni (a statikus metódus az osztály sajátja).

Dinamikus kötés: Az osztályhierarchia bármely pontjáról hívunk is meg egy példánymetódust, mindenkor a megszólított objektum osztályában deklarált metódus fog végrehajtódni (sőt ha ott nincs deklarálva ez a metódus, akkor az osztály leszármazási ágában legkésőbb

deklarált ilyen nevű metódus). Mivel ez csak futáskor derülhet ki, a rendszer futáskor határozza meg a pontos címet. Ezt a jelenséget **futás alatti kötésnek** (**runtime binding**), másnéven késői kötésnek (late binding) vagy dinamikus kötésnek (dynamic binding) nevezik. A keresés folyamatát **dinamikus metóduskeresésnek** (dynamic method lookup) nevezik.



2.5. ábra. A példánymetódus dinamikus kötése

A következő példában (2.5. ábra) a C1 osztály m2 () metódusát C2-ben felülírtuk. Ha az obj1-nek, illetve obj2-nek az m2 üzenetet küldenénk, akkor természetesen a megfelelő objektum osztályában deklarált metódus hajtódna végre. A dolog akkor kezd érdekessé válni, amikor a két objektumnak az m1 () üzenetet küldjük. Mivel C2-ben nincs m1 () metódus, azért minden esetben a C1.m1 () hajtódik végre, s az majd meghívja m2 () -t. Igen ám, de melyiket? Ezt a fordító nem tudhatja; honnan is tudhatná, hogy milyen objektumnak küldött üzenetből érkezett ide a hívás? Ha a megszólított objektum osztálya

- ◆ C1, akkor a C1.m2 () metódust kell végrehajtania;
- ◆ C2, akkor a C2.m2 () metódust kell végrehajtania.

A forráskódban a kiírások egyértelműen bizonyítják, hogy a kötés futás alatt jön létre, hiszen az obj1-nek és obj2-nek küldött m1 üzenetek más kiírást adnak. //1-ből látszik, hogy obj2.m1 lefutásakor az m1 metódus **visszanyúl** a maga osztályában deklarált m2 metódushoz:

```
// DinaKotes.java
class C1 {
    void m1() { m2(); }
    void m2() {
        System.out.println("C1.m2-ben vagyok!");
    }
}
```

```

class C2 extends C1 {
    void m2() {
        System.out.println("C2.m2-ben vagyok!");
    }
}

public class Feluliras {
    public static void main (String args[]) {
        C1 obj1 = new C1();
        C2 obj2 = new C2();
        obj1.m2();      // -> C1.m2-ben vagyok!
        obj2.m2();      // -> C2.m2-ben vagyok!
        obj1.m1();      // -> C1.m2-ben vagyok!
        obj2.m1();      // -> C2.m2-ben vagyok!           //1
    }
}

```

Futtassa a programot lépésenként!

Statikus kötés: Osztálymetódus hívásakor a fordító egyértelműen eldöntheti, hogy melyik osztály metódusát hajtsa végre. A végrehajtandó statikus metódust tehát már a fordító megkeresi és a kóhoz köti. A keresés a megadott osztálytól kezdődik (ha nincs minősítés, akkor a hívást indító osztálytól) és a **hierarchián felfelé**, az ősosztályok irányába halad.

A következő forráskód minden össze annyiban különbözik az előző, DinaKotes.java-tól, hogy m2 itt statikus. //2-ből látszik, hogy az obj2.m1 lefutásakor az m1 metódus **nem nyúl vissza** a maga osztályában deklarált m2 metódusárt:

```

// StatKotes.java
class C1 {
    void m1() { m2(); }
    static void m2() {
        System.out.println("C1.m2-ben vagyok!");
    }
}

class C2 extends C1 {
    static void m2() {
        System.out.println("C2.m2-ben vagyok!");
    }
}

public class StatKotes {
    public static void main (String args[]) {
        C1 obj1 = new C1();
        C2 obj2 = new C2();
        obj1.m2();      // -> C1.m2-ben vagyok!
        obj2.m2();      // -> C2.m2-ben vagyok!
        obj1.m1();      // -> C1.m2-ben vagyok!
        obj2.m1();      // -> C1.m2-ben vagyok!           //2
    }
}

```

Mintapéldánkban a `terfogat()` és a `toString()` felülírt példánymetódus; statikus metódust nem írtunk felül.

Megjegyzések:

- A felülírás nem tévesztendő össze a túlterheléssel (overloading)!
- Adatok esetén nincs dinamikus kötés.

2.6. **this** és **super** referencia

A `this` objektumreferenciáról már a könyv első kötetében is szó volt. A `this` az objektum példánymetódusainak rejttett paramétere; egy memóriacím – a megszólított objektum referenciaja. A példánymetódus ebből tudja, hogy éppen melyik példányon dolgozik (a `this` által azonosított objektumon). A `this`-t implicit paraméterként minden konstruktor és példánymetódus megkapja. A `this` osztálya minden az éppen megszólított objektum osztálya.

A példánymetódusok minden valamilyen objektumon dolgoznak, s ha nem adjuk meg közvetlenül a célobjektumot, akkor a fordító a `this`-t veszi annak. A következő két utasítás ekvivalens egymással:

```
kiir();  
this.kiir();
```

A `super` referencia a `this`-hez hasonlóan az éppen működő objektum lebutított referenciaja: típusa a közvetlen ősosztály. A `super` révén hivatkozhat a programozó az osztály feletti példányadatokra, illetve példánymetódusokra. De ezzel a módszerrel csak az osztály feletti első ősadatra/metódusra lehet hivatkozni (`super.super` nincsen).

A `this` és a `super` referenciát a valamelyen módon eltakart példánydeklarációkra való hivatkozáshoz használjuk:

- Az objektum saját adatát (akár a maga osztályában van deklárálva, akár annak egyik ősében) eltakarhatja a metódus paramétereként vagy lokális változójaként deklárált adat. Ilyenkor a saját adatra így hivatkozhatunk:
`this.adat`
- Az ősben deklárált adatot vagy metódust eltakarhatja az osztályban deklárált ugyanolyan nevű adat vagy metódus. (Csak az ugyanolyan szignatúrájú metódust lehet eltalálni!) Egy ősdeklarációra így hivatkozhatunk:
`super.adat`
`super.metódus()`

A következő, `terfogat()` metódus a `Cso` osztályban volt deklárálva:

```
public double terfogat() {
    Henger belso = new Henger(getSugar()-falVastagsag,
        getMagassag());
    return super.terfogat()-belso.terfogat();
}
```

A cső térfogatát másképp számoljuk ki, mint az egyszerű hengerét. A számításhoz azonban fel szeretnénk használni az ősben megadott térfogatszámítást. Ha a `terfogat()` elő nem írnánk ki a `super` kulcsszót, akkor rekurzív lenne a hívás (a metódus önmagát hívna meg), így viszont a `Cso` osztály feletti legközelebbi `terfogat()` metódus hajtódiik végre, itt tehát a `Henger` osztály `terfogat()` metódusa. A `belso.terfogat()` csak a `Henger`ben deklarált metódusra utalhat, mert a `belso` objektum `Henger` típusú.

Statikus adatra, illetve metódusra sohasem hivatkozhatunk a `this`, illetve `super` referenciával. Osztálymetódus futtatásához nincs szükség objektumra. S ha nincs objektum, akkor nincs referenciája sem. A fordító csak a példánymetódusokhoz teszi hozzá az implicit `this` paramétert!

2.7. this és super konstruktorkor – konstruktorkor láncolása

A konstruktur feladata, hogy inicializálja a létrejövő objektum adatait, kapcsolatait, és megtegye a működéséhez szükséges kezdeti lépéseket. **A konstruktur deklarálásának szabályai Javában:**

- Az osztály konstruktornak neve megegyezik az osztály nevével.
- A konstruktorkor egy adott osztályban túlterhelhetők, vagyis az osztálynak számos különböző paramétere zérus konstruktora lehet.
- **Minden osztálynak van saját konstruktora.** Ha egy osztályban nem deklarálunk konstruktort, akkor egy alapértelmezés szerinti, paraméter nélküli konstruktur lép érvénybe. Ha van deklarált konstruktur, akkor nincs alapértelmezés szerinti.
- A konstruktur nem örökölődik, és nem is lehet felülírni. minden osztály saját konstruktorkorral gondoskodik objektumok létrehozásáról.

Konstrukturhívási lánc

Egy objektum megkonstruálásakor egymás után meghívódik az öröklési ág minden osztályának legalább egy konstruktora, a létrehozandó objektum osztályától kezdve egyesével egészen az alap (`Object`) osztályig. Ez a biztosíték arra, hogy az objektum kezdeti állapota összhangban legyen a deklarációkkal, hiszen így minden adatot az őt deklaráló osztály konstruktora állít be.

Minden osztálynak van konstruktora, ha más nem, az alapértelmezés szerinti. A `new` operátorral implicit módon meghívjuk az osztály megadott szignatúrájú konstruktorkorát. Ezután az

ugyanebben az osztályban levő túlterhelt konstruktorkód hívhatják egymást a `this()` segítségével. A konstruktornak nem kötelező az osztály egy másik konstruktort hívni (`this`), de akkor meg kell hívnia a közvetlen ösosztály adott szignatúrájú konstruktort (`super`). A konstruktorkód tehát láncban hívják egymást. Ez a folyamat addig tart, amíg meg nem hívódik a legfelső (a legősibb) osztály legalább egy konstruktora.

A konstruktorkódok lánctalánának szabályai:

- Osztályon belül az egyik konstruktorból a másik így hívható:
`this(paraméterek)`
- Egy konstruktorból a közvetlen ös konstruktorkód így hívható:
`super(paraméterek)`
- minden konstruktornak tartalmaznia kell **pontosan egy** `this()` vagy `super()` hívást, azt is legelső utasításként (adatdeklaráció sem lehet előtte). A `this` tehát kizáraja a `super-t` és viszont. Ha egyik sem szerepel, akkor a fordító betesz egy alapértelmezett, paraméter nélküli `super()` hívást, feltéve, hogy az ösosztálynak van ilyen konstruktora; ha nincs, akkor a fordító hibát jelez.

Példának tekintsük a következő kódot. A hibás konstruktorkódok megjegyzéseként szerepelnek:

```
// KonstruktorProba.java
class C1 {
    C1(int a) {}                                //1
}

class C2 extends C1 {
    C2(int a, int b) {super(a);}                //2
    C2(int a) {this(a,1);}                      //3
    C2() {this(2);}                            //4
}

class C3 extends C2 {
    C3(int a, int b) {}                        //5
    C3(int a) {this(a,3);}                      //6
}

class C4 extends C3 {
    int a;
    C4(int a) {super(a);}
    //C4() {super();}                         // hiba: C3-ban nincs C3()
    //C4() {}                               // hiba: C3-ban nincs C3()
    //C4() {a=1; super(a);}                  // hiba: super nem első!
    //C4() {this(); super(5);}               // hiba: kettő nem lehet!
}

public class KonstruktorProba {
    public static void main (String args[]) {
        new C3(1);                           //7
    }
}
```

//7-ben konstruálunk egy C3 osztályú objektumot. Ilyenkor meghívódik a //6, s az meghívja a maga osztályában deklarált másik konstruktort. Mivel //5-ben explicit módon nem adtunk meg konstruktorhívást, alapértelmezés szerint az össztest paraméter nélküli, //4 konstruktora lép működésbe. Ha C2-ben nem lenne ilyen, akkor fordítási hiba állna elő. //4 meghívja //3-mat, az pedig //2-t. A //2 meghívja //1-et, az meg az Object konstruktőrét, és ezzel véget ér a hívási lánc.

A C4 osztályban a paraméter nélküli konstruktőrök szintaktikailag minden hibásak. A hiba oka a megfelelő sor megjegyzésében olvasható.

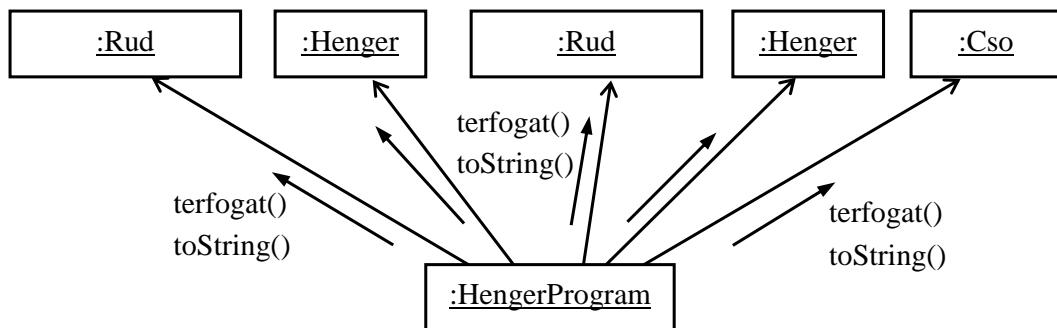
Fontos tanácsok

- ◆ Az űsödöket és kapcsolatokat lehetőség szerint az űskonstruktőr hívásával inicializáljuk! minden konstruktőr csak a maga dolgát végezze!
- ◆ A konstruktőrhívási láncban lehetőség szerint ugyanaz legyen a paraméterek sorrendje. Fontos, hogy a felhasználó könnyen megjegyezhesse a sorrendet!

2.8. Polimorfizmus

A **polimorfizmus** (polymorphism, többalakúság) azt jelenti, hogy ugyanarra az üzenetre különböző típusú objektumok különbözőképpen reagálnak – minden objektum a maga (az üzenetnek megfelelő) metódusa szerint.

A HengerProgram feladatban például a konténerben háromféle henger is van (2.6. ábra), és mindegyiknek ugyanazt üzenjük: terfogat(). Az üzenet hatására az objektum osztálya szerint más-más metóduslánc fut le: a Henger térfogatát másképp számoljuk, mint a Cso térfogatát. A terfogat() üzenet különböző objektumok esetén más és más alakot ölt.



2.6. ábra. Polimorfizmus

A `toString()` üzenet a `terfogat()`-hoz hasonlóan polimorf: egy `Rud` például több adatot mond magáról, mint a `Henger`, a `Cso` pedig a `Rud`-nál is többet. A `toString()` metódust a `System.out.println()` metódus hívja meg, implicit módon.

A `HengerProgram`-ból minden hengernek ugyanazt az üzenetet küldjük, mit sem törödve azzal, hogy az objektum hogy fogja végrehajtani a feladatot:

```
for (int i=0; i<hengerek.size(); i++) {
    henger = (Henger) (hengerek.get(i));
    osszTerfogat += henger.terfogat();
}
```

Fontos, hogy a statikus referencia osztályában (`Henger`) már meglegyen a megfelelő metódus. A program a referencia dinamikus típusától függően más és más metódusláncot fog végrehajtani. Ha egy utódosztályban felülírtunk egy metódust, akkor a megfelelő részfeladatot természetesen az fogja végrehajtani. Ezért van szoros összefüggés a polimorfizmus és a futás alatti kötés között. A kettőt azonban nem szabad összetéveszteni: polimorfizmus futás alatti kötés nélkül is létezik.

2.9. Absztrakt metódus, absztrakt osztály

Az osztály fejlesztője nyitva hagyhatja bizonyos metódusok implementálását. Ez akkor lehet hasznos például, ha egy közös ös metódusnak egyszerűen nem lehet alapértelmezést adni. Ilyenkor a fejlesztő elvégzi a munka dandárját, de az osztály csak akkor lesz működőképes, ha az utóban megadják a nyitva hagyott, absztrakt metódust.

Absztrakt metódus: Csak örökítési cérla való üres metódus.

Absztrakt osztály: Örökítési célt szolgáló, rendszerint absztrakt metódust tartalmazó osztály. Példány nem hozható létre belőle.

Az UML-ben az absztrakt osztály és az absztrakt metódus nevét dölt betűvel szedjük.

Java kódban az absztrakt osztály és az absztrakt metódus elő az `abstract` módosítót írjuk:

```
abstract class <OsztályAzonosító> {
    ...
    abstract <metódusAzonosító>(paraméterek);
    ...
}
```

Szabályok Javában:

- Az absztrakt metódusnak nincsen blokkja.
- Egy absztrakt osztályban akárhány absztrakt metódus lehet (nulla is).

- Absztrakt metódust tartalmazó osztály csak absztrakt lehet (tartalmazhat persze nem absztrakt metódust is).
- Ha az utódosztály nem absztrakt, akkor abban minden absztrakt metódust implementálni kell!
- Absztrakt osztályból nem lehet példányt létrehozni.
- Absztrakt osztály, illetve metódus nem lehet végleges (`final`).

Megjegyzés: A `java.lang.Number` például absztrakt osztály:

```
public abstract class Number implements java.io.Serializable {
    public abstract int intValue();
    public abstract long longValue();
    ...
}
```

Az itt felsorolt absztrakt metódusokat az összes leszármazottban (`Byte`, `Integer`, `Long`, `Double`...) implementálták.

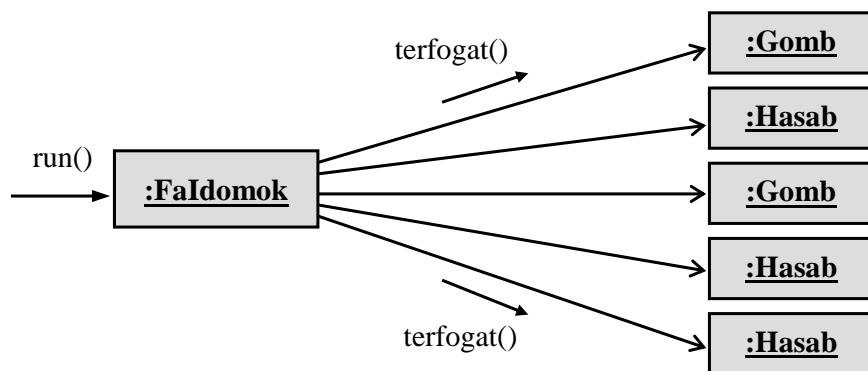
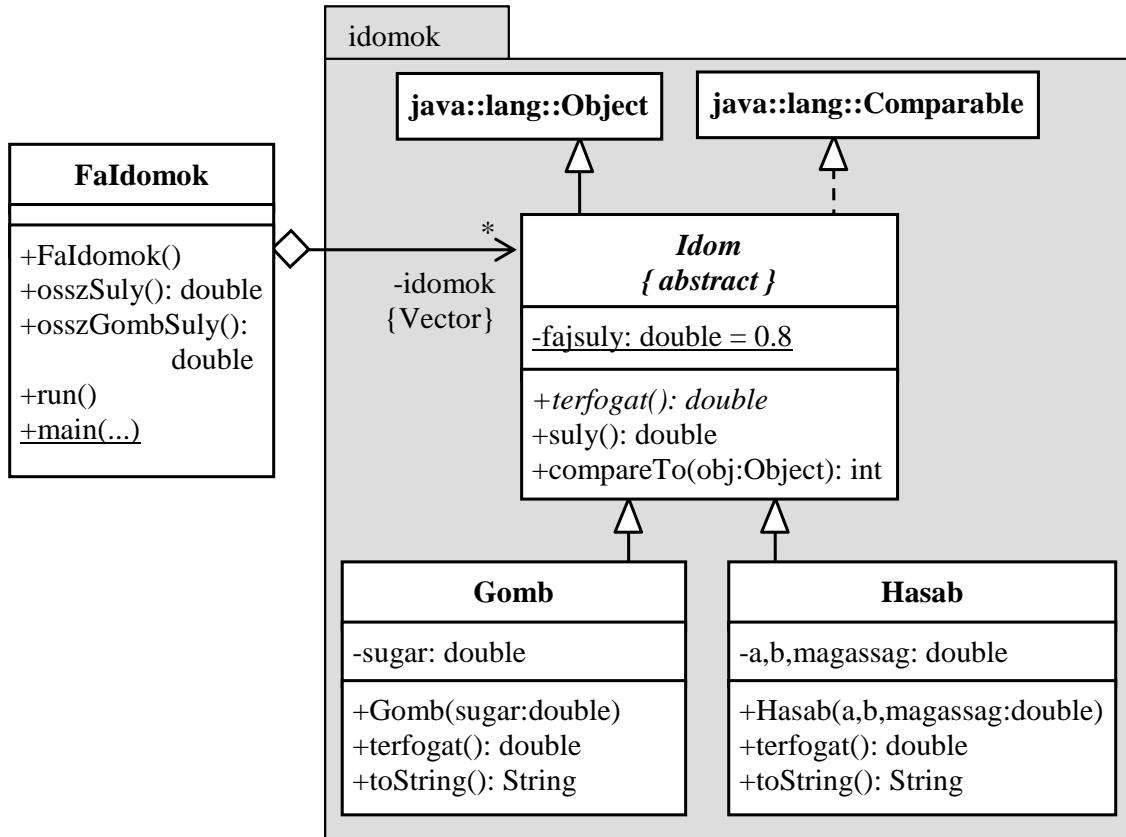
Az absztrakt metódust és az absztrakt osztályt egy feladat segítségével tessük szemléletessé.

Feladat – Faldomok

Egy fajátékokat tartalmazó dobozban kétféle idom van: gömb és hasáb, különböző méretekben. Készítsünk olyan programot, amellyel nyilvántarthatjuk különböző idomok adatait! Írjuk ki a következőket:

- az összes idom adatait a felvitel sorrendjében;
- az idomok összsúlyát;
- a gömbök összsúlyát;
- a legkisebb és a legnagyobb térfogatú idom típusát és adatait!

A gömb és a hasáb is idom, és azt is állíthatjuk, hogy minden idomnak van térfogata és súlya. De ahhoz, hogy a közös konténerbe betett idomuktól egységesen lekérdezhessük majd ezeket a tulajdonságokat, mindenkorban szükségünk van egy közös, ős `Idom` osztályra. Egy idom térfogatának azonban nincs alapértelmezése, azt másiképp számítjuk ki, ha gömbről van szó, és másiképpen, ha hasábról. Megadunk tehát egy `Idom` osztályt, s nyitva hagyjuk benne a `terfogat()` metódus kifejtését, azt majd a konkrét `Gomb` és `Hasab` osztályokban adjuk meg. A súly a térfogat alapján kiszámítható, a `suly()` metódus tehát nem absztrakt. A `compareTo()` úgyszintén nem absztrakt, hiszen az a térfogat metódust felhasználva egyértelműen megadható. Az `Idom` osztályból egyedül a térfogat hiányzik. A feladat tervét az 2.7. ábra mutatja. Az ábrán nem tüntettük fel, hogy az `Idom` osztály őse az `Object` osztály, de tudjuk, hogy ha nem így lenne, akkor az idomoknak nem küldhetnénk `toString()` üzenetet. A `terfogat()` üzenet csak akkor küldhető a konténer minden elemének, ha a hivatkozásukra szintaktikailag rákény-szerítjük az `Idom` osztályt (csak az `Idom`-nak van térfogata, az `Object`-nek nincs).



2.7. ábra. A FaIdomok program osztály- és együttműködési diagramja

Forráskód

Projekt: FaIdomok
Csomag: idomok

Idom.java

```
package idomok;
import extra.Format;
```

```

public abstract class Idom implements Comparable {
    private static double fajsuly = 0.8;
    public abstract double terfogat();

    public double suly() {
        return terfogat()*fajsuly;
    }

    public int compareTo(Object obj) {
        double t = ((Idom)obj).terfogat();
        if (t > terfogat())
            return -1;
        if (t < terfogat())
            return 1;
        return 0;
    }

    public String toString() {
        return " térfogat="+Format.right(terfogat(),0,1)+"
            " súly="+Format.right(suly(),0,1);
    }
}

```

Gomb.java

```

package idomok;
import extra.Format;

public class Gomb extends Idom {
    private double sugar;

    public Gomb(double sugar) {
        this.sugar = sugar;
    }

    public double terfogat() {
        return 4*Math.pow(sugar,3)*Math.PI/3;
    }

    public String toString() {
        return "Gömb, sugár="+Format.right(sugar,0,1)+"
            super.toString();
    }
}

```

Hasab.java

```

package idomok;
import extra.Format;

public class Hasab extends Idom {
    private double a,b, magassag;

```

```
public Hasab(double a, double b, double magassag) {
    this.a = a;
    this.b = b;
    this.magassag = magassag;
}

public double terfogat() {
    return a*b*magassag;
}

public String toString() {
    return "Hasáb, a="+Format.right(a,0,1) +
        " b="+Format.right(b,0,1)+"
        magasság="+Format.right(magassag,0,1)+"
        super.toString();
}
}
```

FaIdomok.java

```
import java.util.*;
import extra.Format;
import idomok.*;

public class FaIdomok {
    private Vector idomok = new Vector();

    public FaIdomok() {
        idomok.add(new Gomb(3));
        idomok.add(new Hasab(2,4,5));
        idomok.add(new Gomb(5));
        idomok.add(new Hasab(3,5,2));
        idomok.add(new Hasab(4,5,4));
    }

    public double osszSuly() {
        double osszes=0;
        Idom idom;
        for (int i=0; i<idomok.size(); i++) {
            idom = (Idom)idomok.get(i);
            osszes += idom.suly();
        }
        return osszes;
    }

    public double osszGombSuly() {
        double osszes=0;
        Idom idom;
        for (int i=0; i<idomok.size(); i++) {
            idom = (Idom)idomok.get(i);
            if (idom instanceof Gomb)
                osszes += idom.suly();
        }
        return osszes;
    }
}
```

```

public void run() {
    System.out.println("Idomok:");
    for (int i=0; i<idomok.size(); i++)
        System.out.println(idomok.get(i));
    System.out.println();

    System.out.println("Összsúly: "+
        Format.right(osszSuly(),0,1));
    System.out.println("Össz gömbsúly: "+
        Format.right(osszGombSuly(),0,1));
    System.out.println("Legkisebb térfogatú idom: "+
        Collections.min(idomok));
    System.out.println("Legnagyobb térfogatú idom: "+
        Collections.max(idomok));
}

public static void main(String[] args) {
    new FaIdomok().run();
}
}

```

A program futása

```

Idomok:
Gömb, sugár=3.0 térfogat=113.1 súly=90.5
Hasáb, a=2.0 b=4.0 magasság=5.0 térfogat=40.0 súly=32.0
Gömb, sugár=5.0 térfogat=523.6 súly=418.9
Hasáb, a=3.0 b=5.0 magasság=2.0 térfogat=30.0 súly=24.0
Hasáb, a=4.0 b=5.0 magasság=4.0 térfogat=80.0 súly=64.0

Összsúly: 629.4
Össz gömbsúly: 509.4
Legkisebb térfogatú idom: Hasáb, a=3.0 b=5.0 magasság=2.0
térfogat=30.0 súly=24.0
Legnagyobb térfogatú idom: Gömb, sugár=5.0 térfogat=523.6
súly=418.9

```

2.10. Láthatóság

Az öröklési ágon a láthatóság

- **nem szűkíthető:** Ha egy példánytag módosítója például `public`, akkor az utódosztályban nem írhatjuk felül `protected` védelemmel.
- **bővíthető:** Egy `protected` metódust például `public`-ká tehetünk. Egy `private` védeottségű deklaráció láthatóságát azonban nem bővíthetjük, hiszen a vele deklarált adattagot vagy metódust el sem érjük.

2.11. Összefoglalás – metódusok nyomkövetése

Foglaljuk most össze a fejezet pontjaiban megfogalmazott szabályokat oly módon, hogy nagyító alá veszünk egy objektumot! Legyen adva az objektum osztálya és annak összes őse. A következő dolgokat vizsgáljuk:

- ◆ Milyen üzenetek küldhetők az objektumnak?
- ◆ Van-e fogadó metódusa ennek vagy annak az üzenetnek?
- ◆ Mely adatok foglalnak maguknak memóriahelyet az objektumban?
- ◆ A különféle üzenetek hatására pontosan milyen metódusok hajtódnak végre?
- ◆ Melyek a helyes hivatkozások az objektum egyik-másik metódusában?

Tekintsük a 2.8. ábrán levő osztályhierarchiát. Tegyük fel, hogy a következő deklarációk más csomagban vannak, mint a C1 és C2 osztály (érvényes a `protected` védelem):

```
C1 obj1;
C2 obj2;
```

A következő feladatokban végig a C1 és C2 osztályokról, valamint az obj1 és obj2 hivatkozásról lesz szó.

1. Milyen üzenetek küldhetők az objektumnak?

Sorolja fel az obj2-nek küldhető összes üzenetet ábécé rendben!

Megoldás:

`m1()`, `m1(String)`, `m2()`, `m2(String)`, `m3(int)`

Az `m4` és `m5` védett, illetve privát metódus.

C1

-s1: String
-a1, a2: int
+C1()
+m1() { m2()}
+m1(String)
+m2() { m3(1) }
+m3(int)
#m4(String,int)
#m4() { m4("ja",1)}

C2

-s2: String
-b: int
+C2()
+m2() { m4() }
+m2(String) { m1() }
+m3(int)
#m4(String,int) { m5() }
-m5()

2.8. ábra

2. Van-e fogadó metódusa ennek vagy annak az üzenetnek?

A következő üzenetek közül jelölje be az összes olyan üzenetet, amelynek van fogadó metódusa!

- a) `obj2.m3(1)`
- b) `obj2.m1("ja")`
- c) `obj1.m5()`
- d) `obj2.m3()`
- e) `obj2.m2()`

Megoldás:

- a) Van, a C2-beli m3 (int) metódus.
- b) Van, a C1-beli m1 (String) metódus.
- c) Nincs, mert obj1 osztályában (C1-ben) nincs m5 () metódus.
- d) Nincs, mert obj2 osztályában és annak egyetlen ősében sincs paraméter nélküli m3 () metódus.
- e) Van, a C2-beli m2 () metódus.

3. Mely adatok foglalnak maguknak memóriahelyet az objektumban?

Sorolja fel az obj2-ben egymás után helyet foglaló adatokat!

Megoldás:

a1, a2, b

4. A különféle üzenetek hatására pontosan milyen metódusok hajtódnak végre?

Kövesse nyomon az obj2.m2("ja") üzenetet! A metódusokat az osztály nevével minősítse!

Megoldás:

C2.m2(String), C1.m1(), C2.m2(), C1.m4(), C2.m4(String, int),
C2.m5()

5. Melyek a helyes hivatkozások az objektum egyik-másik metódusában?

A C2.m3(int) metódusban mely utasítások helyesek szintaktikailag?

- a) super.m3(1);
- b) m1();
- c) C1.m3(1);
- d) C1.m2();
- e) s2="ja";

Megoldás:

- a) Nem helyes. Statikus metódusra nem lehet super-rel hivatkozni.
- b) Nem helyes. Statikus metódusból nem lehet példánymetódusra hivatkozni.
- c) Helyes.
- d) Nem helyes. Statikus metódusból nem lehet példánymetódusra hivatkozni.
- e) Helyes.

Tesztkérdések

- 2.1. Jelölje be az összes igaz állítást!
- Az osztályhierarchia-diagram az osztályokat és a közöttük fennálló társítási kapcsolatokat ábrázolja.
 - Minden osztály implicit őse az `Object`.
 - Javában egy osztálynak több közvetlen őse is lehet.
 - Egy objektumban helyet foglal az objektum osztályában és annak bármely ősében deklarált összes példányadat.
- 2.2. Milyen kötelező viszony van az objektumreferencia deklarált osztálya és az objektumreferencia által azonosított objektum osztálya között? Jelölje be az egyetlen helyes választ!
- Az objektumreferencia deklarált osztálya az azonosított objektum osztályának utódja.
 - Az objektumreferencia deklarált osztálya az azonosított objektum osztályának őse.
 - Az objektumreferencia deklarált osztálya azonos az azonosított objektum osztályával vagy annak őse.
 - Az objektumreferencia deklarált osztálya azonos az azonosított objektum osztályával, vagy annak utódja.
- 2.3. Jelölje be az összes igaz állítást!
- Egy őstípusú referencia értékül adható egy utódtípusú referenciának.
 - Egy utódtípusú referencia értékül adható egy őstípusú referenciának.
 - Automatikus típuskonverzió csak az őstípus felé lehetséges.
 - Felfelé való automatikus típuskonverzióban az őstípus konvertálódik utódtípussá.
- 2.4. Melyik kifejezésre igaz, hogy szintaktikailag helyes és értéke `true`? Jelölje be az összes jó választ!
- ```
Object szoveg = new String("Ures");
```
- `szoveg instanceof Object`
  - `String instanceof szoveg`
  - `szoveg instanceof String`
  - `"Ures" instanceof String`
- 2.5. Jelölje be az összes igaz állítást!
- A `this` egy objektumreferencia.
  - A `super` egy objektumreferencia.
  - Ha egy hivatkozásban nem adunk meg célobjektumot, akkor a fordító a `this`-t veszi célobjektumnak.
  - Ha űsalatra hivatkozunk, akkor használni kell a `this` minősítést.

2.6. C1 osztálydeklarációja két konstruktort tartalmaz, a következőképpen:

```
class C1 {
 C1(int n) { ... }
 C1(String s) { ... }
 ...
}
```

Jelölje be az összes igaz állítást!

- a) C1 utódjában kötelező konstruktort írni.
- b) C1 utódjában legalább két konstruktort kell írni.
- c) C1 utódjában kötelező paraméter nélküli konstruktort írni.
- d) C1 utódjában megadható paraméter nélküli konstruktor, és abból meghívható a `super()` paraméter nélküli konstruktor.

2.7. Jelölje be az összes igaz állítást!

- a) Az öröklési ágon a hozzáférhetőség szükíthető.
- b) Ha az `m1()` metódus `private` egy osztályban, akkor az osztály utódjában nem deklarálható `m1()` metódus.
- c) Egy `protected` védeeltségű metódus a csomagon belül bárhonnan elérhető.
- d) A fordító megköveteli, hogy egy adattagot `private`-nak deklarálunk.

2.8. Adva vannak a következő deklarációk:

```
class C1 { ... }
Object obj = new C1();

String dolgozat = "Dolgozat";
//XXX
```

Mely értékkedások helyesek szintaktikailag a  
//XXX helyén?

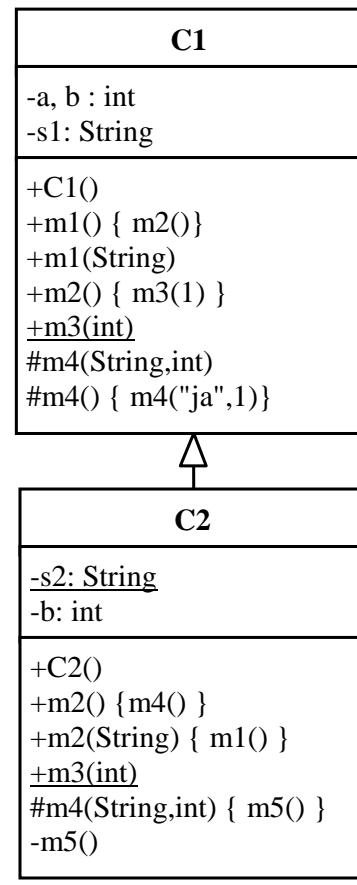
- a) `dolgozat = obj;`
- b) `obj = dolgozat;`
- c) `dolgozat = (String) obj;`
- d) `obj = (C1) obj;`

2.9. Tekintse az 2.9. ábrán levő osztályhierarchiát! Adva vannak a következő deklarációk:

```
C1 obj1 = new C1();
C1 obj2 = new C2();
...
```

Melyik üzenetnek van fogadó metódusa? Jelölje be az összes jó választ!

- a) `obj2.m3();`
- b) `obj2.m1("ja");`
- c) `obj1.m5();`
- d) `obj2.m3(1);`



2.9. ábra

- 2.10. Tekintse az 2.9. ábrán levő osztályhierarchiát! Az `obj2=new C2()` objektumnak kül-dünk egy `m2("Figyelj!")` üzenetet. Az alábbi metódusok közül melyek futnak le az üzenet végrehajtása során? A metódusokat osztályukkal minősítettük. Jelölje be az összes jó választ!
- a) `C1.m1()`
  - b) `C1.m2()`
  - c) `C1.m3(int)`
  - d) `C2.m4(String,int)`
- 2.11. Tekintse az 2.9. ábrán levő osztályhierarchiát! A `C2.m3(int)` metódusban mely utasítások helyesek?
- a) `super.m3(1);`
  - b) `m1();`
  - c) `C1.m3(1);`
  - d) `C2.s2="ja";`

## Feladatok

- 2.1 (A) Egészítse ki a `Faidomok` programot úgy, hogy az idomok közt hengerek is legyenek! Határozza meg a hengerek össztérfogatát és -súlyát! Határozza meg a legkisebb térfogatú hengert! (*Faidomok2.jpx*)
- 2.2. (B) Készítsen egy `Hasab` osztályt! Egy hasábot három adatával (szélesség, hosszúság és magasság) lehet inicializálni, s a hasáb meg tudja mondani térfogatát és felsínét! Ezután a `Hasab` osztályból származtasson egy `Kocka` osztályt! A kockát egyetlen adattal, a kocka oldalméretével lehet inicializálni. A kocka a hasábhoz hasonlóan meg tudja mondani térfogatát és felsínét!
- Tegyen be egy vektorba hasábokat és kockákat! Ezután:
- a) Írja ki a vektorban található összes hasáb és kocka jellemzőit!
  - b) Töröljön ki a vektorból minden olyan hasábot, amelynek térfogata egy kritikus érték alá esik!
  - c) Számolja meg, hogy összesen hány hasáb, illetve kocka születetik a program futása alatt!
  - d) Számolja meg, hogy összesen hány kocka van a vektorban!
  - e) Számolja meg, hogy összesen hány olyan hasáb van a vektorban, amely nem kocka! (*Hasabok.jpx*)
- 2.3. (B) Készítsen egy `Teglalap`, egy `Negyzet` és egy `Kor` osztályt! Mindegyik síkidom mondja meg a területét! Tegyen be egy konténerbe több ilyen síkidomot, majd
- a) Készítsen teljes listát a síkidomokról a bevitel sorrendjében! minden idomnak írja ki a fajtáját és jellemzőit (alapadatok, terület).
  - b) Határozza meg a síkidomok átlagterületét!
  - c) Határozza meg a legkisebb és legnagyobb területű síkidomot!

- d) Rendezze az idomokat terület szerint növekvő sorrendbe!
- e) Számolja meg, hány kör van a síkidomok között!

Tipp: Készítsen egy absztrakt `SikIdom` osztályt, amelyben már van terület! A `Negyzet` osztályt származtassa a `Teglalap` osztályból! (*SikIdomok.jpx*)

## 3. Interfészek, belső osztályok

---

A fejezet pontjai:

1. Interfész
  2. Belső osztály
  3. Névtelen osztály
- 

Interfészről már szó volt a könyv első kötetében – akkor az API-ban deklarált, kész interfész implementáltunk. Most röviden összefoglaljuk, majd kiegészítjük ismereteinket: interfész fogunk készíteni, örökíteni, valamint konstanst deklarálni benne.

Ezután az osztályon belül deklarált, ún. belső osztályokkal fogunk foglalkozni. Megtehetjük, hogy egy belső osztálynak nem is adunk nevet – az ilyen osztályt névtelen belső osztálynak nevezzük. Belső osztályokat elsősorban az eseményvezérelt programozásban fogunk alkalmazni.

### 3.1. Interfész

Az interfész (interface) konstansokat és metódusfejeket definiál. A metódusfejeket az implementáló osztály fogja implementálni, megvalósítani.

#### Az interfész deklarációja

```
[public] interface <InterfészAzon> [extends <InterfészAzon, ...>] {
 <konstansok, metódusfejek>
}
```

Szabályok:

- Az interfész alapértelmezésben `abstract`, az interfésemből nem lehet példányt létrehozni.
- Az interfésznek egyetlen módosítója adható meg, a `public` (ha nem adjuk meg, akkor csomagszintű a láthatósága).
- A konstansok módosítói alapértelmezésben `public`, `static` és `final`, akár deklaráljuk őket, akár nem (s csak dokumentációs céllal lehet őket megadni).

- A metódusfejek módosítói alapértelmezésben `public` és `abstract`, akár deklaráljuk őket, akár nem. Az interfész metódusai példánymetódusok.
- A metódusfejet pontosvesszővel zártuk.

### Az interfész implementálása

Az interfész implementáló osztályban meg kell írni az interfészben megadott metódusokat. Az implementáló osztály speciális utódja az implementált interfésznek. Szabályok:

- Az interfész implementáló osztálynak az interfészben megadott összes metódust implementálnia kell, különben a fordító hibát jelez, kivéve, ha az osztály `abstract` – ekkor az osztály utódjában befejezhetjük az implementálást.
- Egy osztálynak legfeljebb egy őse lehet, de interfész akárhányat implementálhat. Az implementált interfések összes metódusát meg kell írni!
- Az implementáló osztály az interfész metódusait örökölheti is más osztálytól.

### Az interfész öröklése

Az interfések öröklíthetők. Az utódinterfész öröklí az ősinterfész deklarációit. Az utód-interfész implementálásakor implementálnunk kell az ősinterfész metódusait is.

Az osztályuktól eltérően egy interfész akárhány interfészöt öröklőhet. UML-ben az interfész öröklését az osztály örökléséhez hasonlóan üres fejű nyíllal jelöljük, a 3.1. ábrán látható módon (a nyíl nem szaggatott). Az interfések öröklési szabályai hasonlóak az osztályokéhoz.

### Értékadási kompatibilitás

Az objektum statikus referenciája interfész típusú is lehet. Az implementáló osztály objektuma értékül, illetve paraméterül adható az interfész típusú statikus referenciának. Az interfész típusú referenciákra ugyanazok az értékadási szabályok érvényesek, mint az osztály típusú referenciákra.

Az interfések öröklését és implementálását egy példán keresztül tesszük érthetőbbé.

#### **Feladat – Sorok feldolgozása, QueueApp**

A sor (`queue`) szekvenciális, csak sorban feldolgozható konténer; csak a végére lehet új elemet tenni, s csak a legelsőnek betett elemet lehet kivenni belőle.

Készítsünk két interfést:

Az `IQueue` olyan sor, amely képes a következőkre:

- Betesz egy új objektumot a sor végére;
- Kiveszi és rendelkezésre bocsátja a sor legelső objektumát;
- Megmondja, hogy a sor üres-e;

Az `ICleverQueue` okos sor (clever queue), s az előbbieken túl a következőket is tudja:

- Kiveszi és megszünteti a sor első n objektumát;
- Megadja a sorban álló objektumok számát.

Készítsünk minden interfésből egy-egy osztályt! Az `ICleverQueue` implementációja legyen leszármazottja az `IQueue` implementációjának.

Az interfészeket és az osztályokat tegyük be az `extra.util` csomagba, hogy később is használhassuk őket!

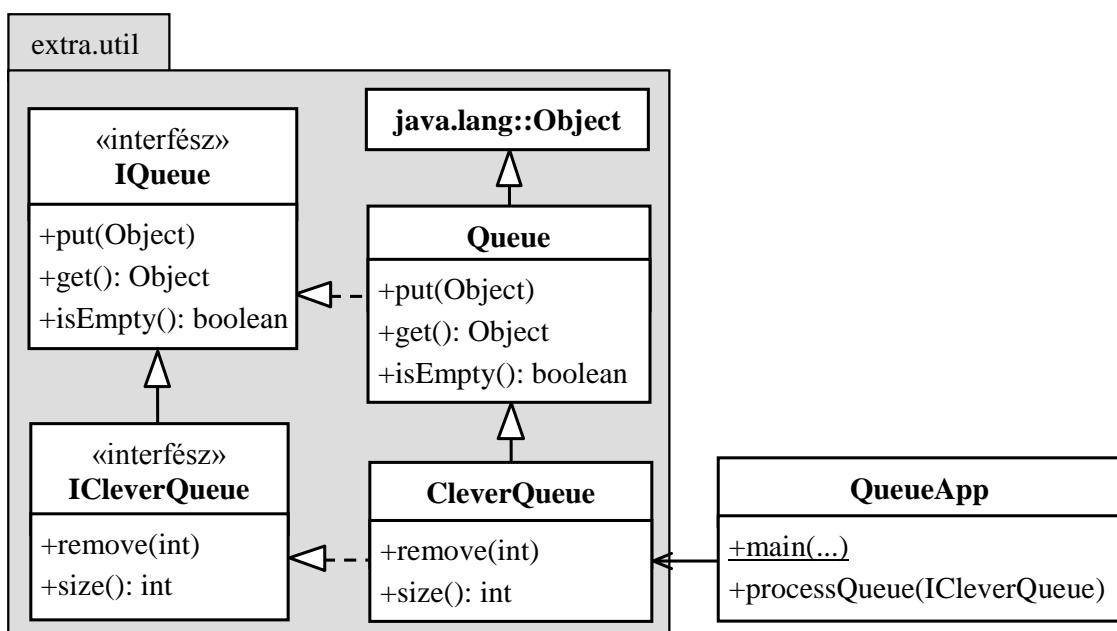
Végül készítsünk egy programot, amely menüvezérelten végzi a következő funkciókat:

- betesz a sorba egy szöveget;
- kivesz a sorból egy szöveget (ha van benne), és ki is írja;
- kitöröl a sorból három szöveget, ha van benne annyi!

*Megjegyzés:* Az interfész azonosítóját nem kötelező I betűvel kezdeni.

A feladat tervét a 3.1. ábra mutatja. Látható, hogy az `ICleverQueue` interfész az `IQueue` interfésből származik. Az őket implementáló `Queue` és `CleverQueue` osztály szintén öröklési viszonyban van egymással.

A `QueueApp` alkalmazás létrehoz egy `CleverQueue` sort, és átadja a `processQueue`-nak feldolgozásra. A metódus a megadott sorba menüvezérelten betesz, kivesz, illetve kitöröl szövegeket. Figyelje meg, hogy a metódus formális paramétere `ICleverQueue` interfész típusú! Ez az interfész elvileg minden olyan sort fogadhat, amely őt implementálta.



3.1. ábra. A QueueApp program osztálydiagramja

**Forráskódok**

```
Projekt: QueueApp
Csomaghierarchia:
 alapértelmezett (nevezetlen) csomag
 QueueApp
 extra
 util
 IQueue
 ICleverQueue
 Queue
 CleverQueue
```

**IQueue.java**

```
package extra.util;
public interface IQueue {
 void put(Object o);
 Object get();
 boolean isEmpty();
}
```

**ICleverQueue.java**

```
package extra.util;
public interface ICleverQueue extends IQueue {
 void remove(int n);
 int size();
}
```

**Queue.java**

```
package extra.util;
import java.util.*;

public class Queue implements IQueue {
 protected Vector v = new Vector();

 public void put(Object o) {
 v.add(o);
 }

 public Object get() {
 if (isEmpty()) return null;
 return v.remove(0);
 }

 public boolean isEmpty() {
 return v.isEmpty();
 }

 public String toString() {
 StringBuffer str = new StringBuffer("");
 for (int i=0; i<v.size(); i++)
 str.append(v.get(i)+" ");
 return str.toString();
 }
}
```

A Queue osztály implementálja az IQueue interfést. A sort egy belső vektor segítségével programozzuk be. A vektort protected módosítóval látjuk el, egy másik csomagból tehát csak az utódosztályok férhetnek hozzá. A put(o) metódus a sor végére teszi az o objektumot. A get metódus visszaadja a sor első objektumát, és mindenkor törli is a sorból (ha egyáltalán van első objektum). Az isEmpty metódus akkor ad vissza true értéket, ha a vektor elemeinek száma 0. A toString metódus a felvitel sorrendjében visszaadja a sor összes elemét.

### CleverQueue.java

```
package extra.util;

public class CleverQueue extends Queue implements ICleverQueue {
 public void remove(int n) {
 for (int i=0; i<n; i++) {
 if (isEmpty())
 break;
 get();
 }
 }

 public int size() {
 return v.size();
 }
}
```

A CleverQueue osztály implementálja az ICleverQueue interfést. A remove metódus n elemet kivesz a vektor elejéről (vagy ahány a sorban van). A size metódus visszaadja a sor hosszát, vagyis a vektor elemeinek a számát.

### QueueApp.java

```
import extra.Console;
import extra.util.*;

public class QueueApp {

 public void processQueue(ICleverQueue q) {
 char c;

 do {
 c = Character.toUpperCase(
 Console.readChar(
 "Érkezik/Sorra kerül/Töröl hármat/Végé: "));
 switch (c){
 case 'E': q.put(Console.readLine("Név: "));
 System.out.println("Sor: "+q);
 break;
 case 'S': if (!q.isEmpty())
 System.out.println(q.get());
 System.out.println("Sor: "+q);
 break;
 }
 }
 }
}
```

```

 case 'T': q.remove(3);
 System.out.println("Sor: "+q);
 break;
 }
} while (c != 'V');
}

public static void main (String args[]) {
 new QueueApp().processQueue(new CleverQueue());
} // main
} // QueueApp

```

A QueueApp okos sort hoz létre, és abba menüvezérelten betesz elemeket, illetve elemeket vesz ki belőle. Létrehozzuk a QueueApp alkalmazást, meghívjuk annak processQueue metódusát, és paraméterként átadjuk neki a frissen létrehozott okos sort.

### A program egy lehetséges futása

|                                                                                                                                                                                                                                                                                               |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Érkezik/Sorra kerül/Töröl hármat/Vége:E<br>Név: Piroska<br><b>Sor: Piroska</b><br>Érkezik/Sorra kerül/Töröl hármat/Vége:E<br>Név: Farkas<br><b>Sor: Piroska Farkas</b><br>Érkezik/Sorra kerül/Töröl hármat/Vége:S<br>Piroska<br><b>Sor: Farkas</b><br>Érkezik/Sorra kerül/Töröl hármat/Vége:V |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### Konstans az interfészben

Az interfészben konstansokat is definiálhatunk. E konstansok módosítói: `public`, `static` és `final`, akár megadjuk őket, akár nem. Az interfész konstansaira ugyanúgy lehet hivatkozni, mint egy osztály statikus konstansaira:

- ◆ Az interfészt implementáló osztály és az interfészt kiterjesztő újabb interfész minősítés nélkül hivatkozhat a kontransokra;
- ◆ A konstansokra bárhonnan lehet hivatkozni, ha az interfésszel minősítjük őket.

#### Feladat – Számla

Kérjünk be egy forintösszeget, majd írjuk ki az értékét euróban!

Az Euro interfészben összefogjuk az Euro jellemzőit (árfolyamot és pénznemet). Az interfészt a Szamla osztály implementálja, így ott hivatkozhatunk az euróra vonatkozó ARFOLYAM és PENZNEM konstansra. Ha amerikai dollárral szeretnénk számolni, akkor csak ki kell kicserélni az Euro interfészt az USD-re, amely a dollár ugyanilyen nevű adatait tartalmazza.

## Forráskód

```
// Szamla.java
import extra.*;

interface Euro {
 double ARFOLYAM = 230.0;
 String PENZNEM = "EURO";
}

public class Szamla implements Euro {
 public static void main (String args[]) {
 double ftOsszeg = Console.readDouble("Összeg (Ft): ");
 System.out.println(Format.left(ftOsszeg/ARFOLYAM, 0, 2) +
 " " +PENZNEM);
 }
}
```

### A program futása

|              |          |
|--------------|----------|
| Osszeg (Ft): | 25000    |
|              | 108 EURO |

Ha a Szamla osztály nem implementálná az Euro interfészét, akkor az ARFOLYAM és PENZNEM adatra csak minősítve hivatkozhatnánk: Euro.ARFOLYAM, és Euro.PENZNEM. Az interfész implementálásával kikerülhető a konstans minősítése.

## 3.2. Belső osztály

Egy osztályon belül deklarált másik osztályt **belső osztálynak** nevezzük. A belső osztályt csak az őt deklaráló külső osztály ismeri. A belső osztályból létrehozott objektum ismeri az őt létrehozó külső objektumot.

### Belső osztály deklarálása

```
class <KülsőOsztályAzon> ... {
 ...
 class <BelsőOsztályAzon> ... {
 ...
 }
}
```

Szabályok:

- A belső osztálynak nem lehetnek statikus deklarációi.
- A belső osztályból példányt létrehozni csak a külső osztály konstruktorából vagy valamely példánymetódusából lehet.
- A belső objektum hivatkozhat a külső objektum deklarációira; takarás esetén a KülsőOsztály.this referenciaival minősítjük őket.

- Belső osztályt nem kötelező azonosítani, ilyenkor **névtelen belső osztályról** beszélünk.
- A belső osztályt a fordítóprogram külső osztállyá alakítja, és aszerint, hogy az osztálnak van-e azonosítója vagy nincs, a class állomány neve a következő lesz:

KülsőOsztályAzon\$BelsőOsztályAzon.class  
 KülsőOsztályAzon\$n.class // n=1,2,... (névtelen osztály)

Belső osztályt a következő okok miatt szokás alkalmazni:

- ◆ A belső osztályt el akarjuk takarni a csomag más osztályai elől. Emlékezzünk arra, hogy egy osztály láthatósága csak publikus vagy csomagszintű lehet. A belső osztály automatikusan privát.
- ◆ A belső osztályú objektum hivatkozni szeretne az öt létrehozó külső osztályú objektum deklarációira. Ez kényelmesebb megoldás, mint a referencia átadása.

**• Csak indokolt esetben használunk belső osztályt!** A program veszíthet áttekinthetőségeből.

*Megjegyzés:* Mint később látni fogjuk, az eseményvezérelt programozásban kényelmes lesz belső osztályokat használni.

A belső osztály készítésének technikáját egy feladaton keresztül mutatjuk meg.

### Feladat – UdvarApp

Egy udvaron különböző négyzetes oszlopok vannak elhelyezve. Tartsuk nyilván az udvar és az oszlopok adatait, majd készítsünk róluk jelentést! Ne engedjünk olyan oszlopot létrehozni, amely az udvaron kívülre esik!

Az udvarnak van szélessége és hossza, és tárolja az oszlopokat. Egy oszlop ismeri a maga udvaron belüli helyzetét és a szélességét. Az udvar ismeri az oszlopokat, hiszen ő tárolja őket. Másfelől jó, ha az oszlopok is ismerik „gazdájukat”, az udvarat, hiszen csak így kívánhatjuk meg tőlük, hogy eleve rajta maradjanak. Legyen tehát az Oszlop az Udvar belső osztálya!

### Forráskód

```
// UdvarApp.java
import java.util.*;

class Udvar {
 private int szel=100, hossz=50;
 private Vector oszlopok = new Vector();

 public Udvar() { //1
 oszlopok.add(new Oszlop(25,30));
 oszlopok.add(new Oszlop(40,5));
 oszlopok.add(new Oszlop(97,10));
 }
}
```

```

class Oszlop { //2
 private int x, y; // az oszlop helye az udvaron
 private int szel=5; // az oszlop szélessége

 public Oszlop(int x, int y) {
 if (x>=0 && x<=Udvar.this.szel-szel &&
 y>=0 && y<=hossz-szel) { //3
 this.x = x; this.y = y; //4
 }
 else {
 this.x = 0; this.y = 0;
 }
 }

 public String toString() {
 return x+"-"+y;
 }
}

public String toString() { //5
 return "Udvar:"+szel+"x"+hossz+", Oszlopok:"+oszlopok;
}
}

public class UdvarApp {
 public static void main(String[] args) {
 Udvar udvar = new Udvar();
 System.out.println(udvar); //6
 }
}

```

### A fordításkor keletkezett állományok

Udvar.class  
 Udvar\$Oszlop.class  
 UdvarApp.class

### A program futása

| Udvar:100x50, Oszlopok:[25-30, 40-5, 0-0]

### A program elemzése

- ◆ //1: Az oszlopokat az Udvar osztály konstruktorában hozzuk létre. Az osztályon kívülről ezt nem lehetnék meg, hiszen az Oszlop osztály az Udvar-on kívül ismeretlen.
- ◆ //2: Az Oszlop az Udvar osztály belső osztálya.
- ◆ //3: Az Oszlop konstruktorában az Udvar osztály szel és hossz adatára kell hivatkoznunk – az oszlop szel adata azonban eltakarja az udvar szel adatát, ezért arra csak így hivatkozhatunk: Udvar.this.szel.
- ◆ //4-ben a this.x az Oszlop adattagja, a jobb oldali x pedig a paraméter.

- ◆ //5-ben a szel és hossz adat már egyértelműen az Udvar adata, hiszen itt már az oszlop osztályon kívül vagyunk.
- ◆ //6: Az udvar kiírásából látható, hogy a harmadik oszlopot nem sikerült létrehozni, az az udvaron kívül esett volna ( $97 + 5 = 102$ , az udvar szélessége pedig 100).

**Feladat – UdvarApp2**

Oldjuk meg az előző feladatot belső osztály nélkül!

**Forráskód**

```
// UdvarApp2.java
import java.util.*;

class Oszlop {
 private int x, y; // az oszlop helye az udvaron
 private int szel=5; // az oszlop szélessége

 public Oszlop(Udvar udvar,int x, int y) {
 if (x>=0 && x<=udvar.szel-szel &&
 y>=0 && y<=udvar.hossz-szel) {
 this.x = x; this.y = y;
 } else {
 this.x = 0; this.y = 0;
 }
 }

 public String toString() {
 return x+"-"+y;
 }
}

class Udvar {
 int szel=100, hossz=50;
 private Vector oszlopok = new Vector();

 public Udvar() {
 oszlopok.add(new Oszlop(this,25,30));
 oszlopok.add(new Oszlop(this,40,5));
 oszlopok.add(new Oszlop(this,97,10));
 }

 public String toString() {
 return "Udvar:"+szel+"x"+hossz+", Oszlopok:"+oszlopok;
 }
}

public class UdvarApp2 {
 public static void main(String[] args) {
 Udvar udvar = new Udvar();
 System.out.println(udvar);
 }
}
```

### A fordításkor keletkezett állományok

Oszlop.class  
Udvar.class  
UdvarApp2.class

### A program elemzése

Ebben a megoldásban az oszloból nincs automatikus referencia az udvarra. Ezért az udvar átadja magát az Oszlop konstruktörának: new Oszlop(this, 25, 30), hogy az ismerhesse az udvar szélességét és hosszúságát. Ezeket az adatokat az Udvarban most publikusra vagy csomagszintüre kell deklárnunk, vagy kiolvasó metódusokat kell írnunk (itt az előbböt választjuk).

## 3.3. Névtelen osztály

**A névtelen osztályt** az objektum létrehozásával egy időben deklaráljuk a new operátor szintaktikájának kiterjesztésével. Szabályok:

- A névtelen osztályra nem hivatkozhatunk az osztályon kívülről, mivel egy ilyen osztálnak nincs is neve.
- Névtelen osztályú objektumot kétféleképpen konstruálhatunk: egy osztály kiterjesztésével vagy adott interfész implementálásával (lásd külön pontok).

Névtelen osztályt általában akkor szokás létrehozni, ha az osztáyból csak egyetlen példányra van szükségünk és az osztály deklarációja egyszerű.

### Névtelen osztály példányosítása osztály kiterjesztésével

Egy névtelen osztály példányosítható egy már meglévő osztály kiterjesztésével; ennek a következő a szintaktikája:

```
new <ŐsosztályAzonosító> (<paraméterlista>) {
 <Névtelen osztály blokkja>
}
```

A létrejövő objektum osztálya az ősosztály itt megadott névtelen leszármazottja lesz. A paraméterlista az ősosztály konstruktörának adódik át. Az új osztálynak nem lehet konstruktora, hiszen neve sincs – az esetleges inicializáló műveleteket példányinicializáló blokkban adhatjuk meg. Az extends kulcsszó nem szerepel a deklarációban.

**Feladat – Névtelen kiterjesztés**

Induljunk ki egy `Ember` osztályból – egy embert nevével lehet inicializálni, és a `jellemzo()` metódusa a "Normális" szöveget adja vissza. Hozzunk létre `Ember` típusú példányokat úgy, hogy azok `jellemzo()` metódusát majd szükség szerint névtelen osztályokban felülírjuk!

Tegyük egy vektorba embereket különböző névvel és jellemzővel! Végül listázzuk ki a vektort!

**Forráskód**

```
// NevteleNkiterjesztes.java
import java.util.*;

class Ember { //1
 private String nev;

 public Ember(String nev) {
 this.nev = nev;
 }
 public String jellemzo() {
 return "Normális";
 }

 public String toString() {
 return nev + " ("+jellemzo()+")";
 }
}

public class NevteleNkiterjesztes {

 public static void main(String[] args) {
 Vector emberek = new Vector();
 emberek.add(new Ember("Zoli")); //2
 public String jellemzo() { //3
 return "Kövér";
 }
);
 emberek.add(new Ember("Laci")); //4
 emberek.add(new Ember("Jenő")); //5
 public String jellemzo() {
 return "Szemüveges";
 }
);
 System.out.println(emberek);
}
}
```

**A fordításkor keletkezett állományok**

`Ember.class`  
`NevteleNkiterjesztes.class`  
`NevteleNkiterjesztes$1.class`  
`NevteleNkiterjesztes$2.class`

### A program futása

| [Zoli (Kövér), Laci (Normális), Jenci (Szemüveges)]

### A program elemzése

- ◆ //1-ben deklaráljuk az Ember osztályt. Ezt az osztályt fogjuk névtelen osztályokban kiterjeszteni.
- ◆ //2-ben deklarálunk egy névtelen osztályt, az Ember leszármazottját. Egyetlen metódusát, a jellemzo-t írjuk felül //3-ban. A deklarált osztályból azonnal létrehozunk egy objektumot, és még „azon melegében” be is dobuk a konténerbe. //5-ben ugyanezt teszszük.
- ◆ //4-ben is létrehozunk egy névtelen Ember példányt, de mivel Laci normális, neki nem írjuk felül a jellemzo metódusát.

### Névtelen osztály példányosítása interfész implementálásával

A névtelen osztály példányosítható egy már meglévő interfész implementálásával. A szintaktika a következő:

```
new <InterfészAzonosító> () {
 < Névtelen osztály blokkja>
}
```

A létrejövő objektum osztálya az interfész itt megadott névtelen implementációja lesz. Paraméterlista itt nem adható meg, mert az ōs egy interfész, tehát nincs konstruktora. Az új osztálynak sem lehet konstruktora, hiszen neve sincs – az esetleges inicializáló műveleteket példányinicializáló blokkban adhatjuk meg. Az új osztály implicit módon az Object osztály kiterjesztése. Az implements kulcsszó nem szerepel a deklarációban.

#### Feladat – Névtelen implementáció

Induljunk ki egy Jellemzett interfésből; legyen ennek csak egyetlen metódusa, a jellemzo(). Tegyük egy vektorba objektumokat, s azok osztályában névtelen osztályokkal implementáljuk a jellemzo() metódust! Végül listázzuk ki a vektort!

#### Forráskód

```
// NevteleinImplementacio.java
import java.util.*;

interface Jellemzett {
 public String jellemzo();
}
```

```

public class NevtelenImplementacio {
 public static void main(String[] args) {
 Vector jellemzettek = new Vector();
 jellemzettek.add(new Jellemzett() {
 public String jellemzo() {
 return "Csúnya";
 }
 });
 jellemzettek.add(new Jellemzett() {
 public String jellemzo() {
 return "Szép";
 }
 });
 for (int i=0; i<jellemzettek.size(); i++) {
 Jellemzett jel =
 (Jellemzett)jellemzettek.get(i);
 System.out.println(jel.jellemzo());
 }
 }
}

```

### A fordításkor keletkezett állományok

Jellemzett.class  
 NevtelenImplementacio.class  
 NevtelenImplementacio\$1.class  
 NevtelenImplementacio\$2.class

### A program futása

|        |
|--------|
| Csúnya |
| Szép   |

#### Megjegyzések:

- Az osztályok névtelen kiterjesztésének használatára az eseményadapterek névtelen kiterjesztése is példát ad majd (lásd 7. fejezet, Eseményvezérelt programozás).
- A belső osztályok világából még számos lehetőséget használhatnak a programozók, például a többszörösen egymásba ágyazott osztályokat, vagy a lokális (blokkban deklarált) osztályt. Ezeket e könyv nem tárgyalja.

### Névtelen tömb

Névtelen tömböt is létrehozhatunk, ha létrehozás után az értékeket egy inicializáló blokkban soroljuk fel.

Egydimenziós névtelen tömb létrehozása:

```
new <elemtípus>[] {<elem>, <elem>, ...}
```

Kétdimenziós névtelen tömb létrehozása:

```
new <elemtípus>[][] {{<elem>, ...}, {<elem>, ...}, ...}
```

**Például:**

Egydimenziós névtelen tömb létrehozása, majd „röptében” paraméterként való átadása:

```
lista(new String[] {"Piros", "Fehér", "Zöld"});
```

Kétdimenziós névtelen tömb létrehozása, majd „röptében” paraméterként való átadása:

```
lista(new String[][] {
 {"Piros", "Fehér", "Zöld"},
 {"Egy", "Kettő"}
});
```

## Tesztkérdések

3.1. Jelölje be az összes igaz állítást!

- a) Az interfész fejében kötelezően ki kell tenni az `abstract` módosítót.
- b) Az interfészen nem adható meg konstans deklarációja.
- c) Ha az interfész deklarációjában egy metódusfej előtt nem adunk meg módosítót, akkor az alapértelmezésben `public` és `abstract`.
- d) Az interfész metódusai példánymetódusok.

3.2. Jelölje be az összes igaz állítást!

- a) Az interfészt implementáló osztálynak az interfészen megadott összes metódust implementálnia kell, vagy az osztályt absztraktnak kell deklarálni!
- b) Egy osztály legfeljebb egy interfész implementálhat.
- c) Egy interfész akárhány interfészt örökölhet.
- d) Az implementáló osztály objektuma értékül, illetve paraméterül adható az interfész típusú statikus referenciának.

3.3. Jelölje be az összes igaz állítást!

- a) Belső osztálynak nem lehetnek statikus deklarációi.
- b) A külső osztálynak nem lehetnek statikus deklarációi.
- c) A belső osztályt nem kötelező azonosítani (lehet névtelen is).
- d) A belső osztályt a fordítóprogram egy külső osztállyá alakítja.

3.4. Egy belső osztály `pd` adata eltakarja a `KO` nevű külső osztály `pd` példánydeklarációját.

Hogy hivatkozhatunk a külső osztály `pd` adatára? Jelölje be az egyetlen igaz állítást!

- a) `this.KO.pd`
- b) `this.pd`
- c) `KO.pd`
- d) `KO.this.pd`

3.5. Jelölje be az összes igaz állítást!

- a) Névtelen osztály csak egy objektum létrehozásával együtt deklarálható.
- b) A névtelen osztályra nem lehet hivatkozni.
- c) Egy névtelen osztályt kizárálag egy másik osztály kiterjesztésével hozhatunk létre.
- d) A névtelen osztály bájkódja része a külső osztály `class` állományának.

3.6. Mely utasítások helyesek szintaktikailag? Jelölje be az összes helyes utasítást!

- a) Object obj = new Object(extends Object) {
   
    public String toString() {return "Blabla";}
   
};
- b) Object obj = new Object(toString());
- c) Object obj = new Object()extends Object {
   
    public String toString() {return "Blabla"}
   
};
- d) Object obj = new Object(){
   
    public String toString(){return "Blabla";}
   
};

## Feladatok

3.1. (A) A `SikIdom` interfész tartalmazza a terület- és kerületszámítás metódusait, valamint a számításokhoz szükséges PI konstanst. Készítsen egy `Kor`, egy `Teglalap` és egy `Negyzet` osztályt, s azok minden implementálják a `SikIdom` interfészt!

Tegyen be egy konténerbe több ilyen síkidomot, majd

- a) Írja ki sorban a síkidomok jellemzőit!
  - b) Határozza meg a síkidomok átlag területét!
  - c) Számolja meg, hány kör van a síkidomok között!
- (*SikIdomok.jpx*)

3.2. (B) A verem (stack) olyan konténer, amelyben az elemek egymás tetején vannak. Elemet csak a verem tetejére lehet tenni (`push`), és minden csak a legutoljára betett elemet lehet kivenni (`pop`). Készítsen egy `IStack` interfészt, majd implementálja azt két-féleképpen:

- a) a `VectorStack` osztály megvalósítása vektorral történjen!
  - b) az `ArrayListStack` osztály megvalósítása tömbbel történjen!
- Használja is a vermeket! (*Verem.jpx*)

3.3. (A) Hozzon létre névtelen osztályú `Object`-leszármazottakat és írja át a `toString` metódusukat! Tegye őket egy vektorba, és listázza ki ezt a vektort!

(*ObjectLeszarmazottak.java*)

## **4. Kivételkezelés**

---

A fejezet pontjai:

1. Kivételek, hibák
  2. Kivételek keletkezése és szándékos előidézése – throw
  3. A kivétel továbbadása – throws
  4. A kivétel elkapása, kezelése
  5. Saját kivételek használata
- 

Minden programnak van egy tiszta logika szerinti működése, de azt számos körülmény, kivételes esemény, illetve rendszerhiba megzavarhatja. Kivételes esemény például a nullával való osztás, a túlindexelés vagy egy nem létező állomány megnyitására tett kísérlet. A sok kivétel egymásba ágyazott szelekciókkal való kezelése eltakarhatja a program igazi logikáját, és a program olvashatatlanná, kezelhetetlenné válhat tőle. A kivételes események, hiába a legnagyobb körültekintés, kifoghatnak még a legjobb programozón is. A Java hatékony mechanizmust kínál a kivételek következetes kezelésére. A fejezet ezt a kivételkezelési technikát mutatja be.

### **4.1. Kivételek, hibák**

A programot leállíthatja egy objektum, amely a program futása során keletkezik, valamely kivételes esemény (exception) vagy rendszerhiba (error) felléptekor. A kivételes eseményt a Javában kivételeknek nevezik, ezt a programozó a programban elfoghatja, és ellenlépéseket tehet; a rendszerhiba viszont nem kezelhető. A továbbiakban a helyrehozhatatlan hibát (error) minden rendszerhibának fogjuk nevezni; a kivételek voltaképpen helyrehozható hibák.

Kivételt kelthet (másként: kivételobjektumot ejthet, esetleg dobhat) egy futási hiba vagy a program valamilyen állapota. minden kivételobjektumnak jól meghatározott osztálya van. A kivétel egy konkrét metódusban keletkezik valamely utasítás végrehajtásakor – ez a metódus lehet a programozó által írt metódus és lehet az API metódusa is. A keletkezett kivétel bekerül a program vérkeringésébe, és jól meghatározott úton, a metódushívási láncon visszafelé „kiszáll” a programból: a metódusok továbbadják egymásnak. A programozó elkaphatja a

kivételek objektumot, amint az kifelé tart a programból, és lehetőség szerint kezelheti: megszüntetheti például a hibát kiváltó okot és újra nekirugaszkodhat a feladat megvalósításának, vagy ha a program menthetetlen, akkor megteheti, hogy a körülmenyekhez képest elegánsan és kultúrálisan adja fel a harcot. Végleges leállása előtt egy igényes program elrendezi a még menthető dolgokat, valamint értesíti a felhasználót a „tragédiáról” és teendőiről. Ha a programozó nem „kapja el” a futási hibákat, akkor a rendszer alapértelmezés szerint bánik a kivételes helyzettel: egyszerűen konzolra írja a kivétel nevét egy rövid angol nyelvű magyarázó szöveggel, valamint kiírja a hívott metódusok nevét, hogy a programozó kideríthesse a hiba helyét. Ez a megoldás azonban valószínűleg nem fog tetszeni a programért fizető felhasználónak.

A programot leállító események valamennyien objektumok, s osztályuk a `Throwable` (dobható, ejthető) osztályból származik. A „dobható” események két csoportra oszthatók:

- **Rendszerhiba** (`Error`): a program nem állítható helyre, óhatatlanul leáll.
- **Kivétel** (`Exception`): hibakezelés után a program folytatódhat.

|                                              |                          |
|----------------------------------------------|--------------------------|
| <code>Object</code>                          | <code>java.lang</code>   |
| <b>Throwable</b>                             | <code>java.lang</code>   |
| <b>Error</b>                                 | <code>java.lang</code>   |
| <code>LinkageError</code>                    | <code>java.lang</code>   |
| <code>VirtualMachineError</code>             | <code>java.lang</code>   |
| <code>ThreadDeath</code>                     | <code>java.lang</code>   |
| <code>AWTError</code>                        | <code>java.lang</code>   |
| <b>Exception</b>                             | <code>java.lang</code>   |
| <code>RuntimeException</code>                | <code>java.lang</code>   |
| <code>ArrayIndexOutOfBoundsException</code>  | <code>java.lang</code>   |
| <code>ArithmaticException</code>             | <code>java.lang</code>   |
| <code>ClassCastException</code>              | <code>java.lang</code>   |
| <code>IllegalArgumentException</code>        | <code>java.lang</code>   |
| <code>NumberFormatException</code>           | <code>java.lang</code>   |
| <code>...</code>                             |                          |
| <code>NegativeArraySizeException</code>      | <code>java.lang</code>   |
| <code>NoSuchElementException</code>          | <code>java.util</code>   |
| <code>...</code>                             |                          |
| <code>NullPointerException</code>            | <code>java.lang</code>   |
| <code>IOException</code>                     | <code>java.io</code>     |
| <code>EOFException</code>                    | <code>java.io</code>     |
| <code>FileNotFoundException</code>           | <code>java.io</code>     |
| <code>...</code>                             |                          |
| <code>AWTException</code>                    | <code>java.awt</code>    |
| <code>UnsupportedLookAndFeelException</code> | <code>javax.swing</code> |
| <code>...</code>                             |                          |

4.1. ábra. Rendszerhibák és kivételek osztályhierarchiája

A rendszerhibák és kivételek osztályhierarchiáját a 4.1. ábra mutatja. Az alapvető kivételek osztályai a `java.lang` csomagba kerültek, a speciális, adott témahez illeszkedő kivételek osztályai

pedig a témának megfelelő csomagba (a be- és kimenettel kapcsolatos kivételek például a `java.io` csomagban vannak). A hierarchia tetején a `Throwable` (dobható) osztály van és két leszármazottja, az `Error` és az `Exception`:

- ◆ `Error`: Rendszerhiba; a fordító vagy a lefordított bájtkód helyreállíthatatlan hibája vagy változása okozhatja. A rendszerhiba elkaptható ugyan, de ezután a program (programszál) mindenkorban leáll. Mindössze annyit tehetünk, hogy a körülményekhez képest elegánsan fejezzük be a programot. Az `Error` alosztályai:
  - `LinkageError`: Ilyen esemény akkor keletkezik például, ha egy régebben lefordított `C1` osztály egy másik, `C2` osztályra hivatkozik (függ tőle), de `C2`-t időközben megváltoztatták (újrafordították).
  - `VirtualMachineError`: A Java futtató sérült vagy alapvető erőforrásai hiányoznak.
  - `ThreadDeath`: Egy futó programszál meghalt.
  - `AWTError`: Helyreállíthatatlan Abstract Window Toolkit rendszerhiba.
- ◆ `Exception`: Kivétel; programhibák, vagy bizonyos külső körülmények okozhatják. A kivételt a program kezelheti úgy, hogy a program tovább fussen. Az `Exception` közvetlen alosztályai:
  - `RuntimeError`: Futási hiba keletkezik például rossz típuskényszerítés, tömb túlindexelése vagy nullával való osztás miatt.
  - `IOException`: Input/output (beviteli/kiviteli) hiba keletkezik például, ha meg akarunk nyitni egy nem létező állományt vagy az állomány végéhez értünk.

*Megjegyzés:* Az `Error` leszármazottait szintaktikailag hasonlóképpen kezeljük, mint az `Exception` leszármazottait, de a rendszerhibák programból helyreállíthatatlanok.

A továbbiakban csak a kivételekkel, vagyis az `Exception` osztály utódjaival foglalkozunk.

## A kivételosztályok deklarációi

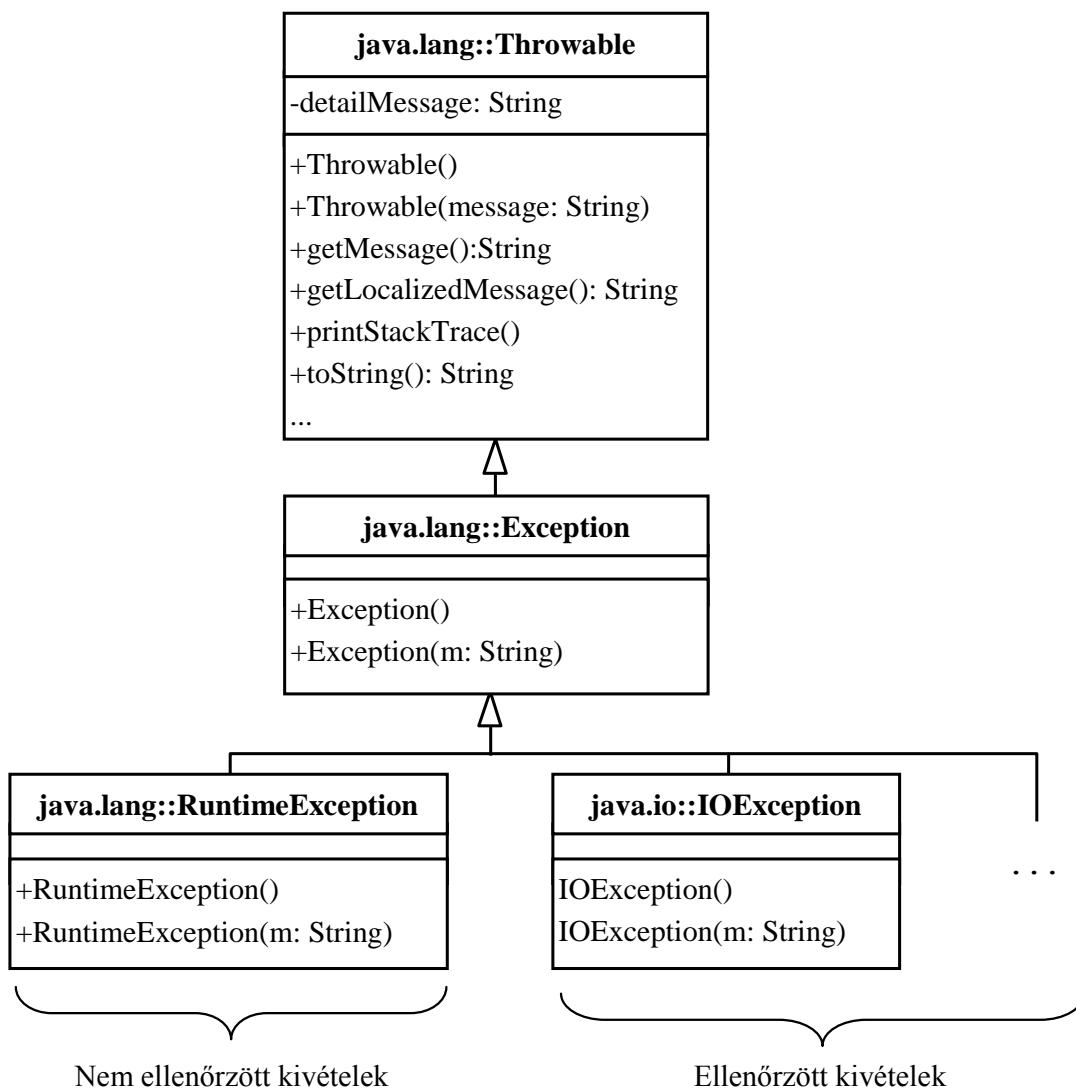
Vizsgáljuk most meg, milyen deklarációkat tartalmaznak a kivételeket leíró osztályok (4.2. ábra).

Az összes fontos adat és metódus a `Throwable` osztályban van implementálva:

- ◆ `detailMessage:String` Információs szöveg (részletes üzenet), a kivétel szöveges reprezentációja; például "/ by zero" vagy "Array index out of range". Az öröklő osztály konstruktora beállíthatja ezt az információs szöveget. A program kivétel miatti leállásakor ez a szöveg kerül a konzolra.
- ◆ Konstruktorok: Megadhatjuk a kivétel információs szövegét, a `detailMessage`-et. Ha a konstruktort paraméter nélkül hívjuk meg, akkor `detailMessage` értéke üres marad.
- ◆ `getMessage():String` Visszaadja a `detailMessage` szöveget.

- ◆ `getLocalizedMessage():String` Alapértelmezésben ugyanaz, mint a `getMessage()`. Felülírásával finomítható a hibaüzenet.
- ◆ `printStackTrace()` Konzolra írja a metódushívási láncot.
- ◆ `toString():String` Visszaadja a kivétesztály nevét és az információs szövegét: `getClass().getName() + detailMessage`.

A kivétesztályok lényegében jelölő szerepet játszanak, a konkrét kivétel magával az osztály-  
lyal azonosítható. Az utódosztályokban csak a konstruktorkor vannak átirva.



4.2. ábra. A kivétesztályok deklarációi

## Nem ellenőrzött és ellenőrzött kivételek

A kivételek (az `Exception` leszármazottai) logikailag két csoportra oszthatók:

- **Nem ellenőrzött kivételek** (unchecked exceptions): Ezek a futási hibák, vagyis a `RuntimeException` és annak leszármazottai. A nem ellenőrzött kivételeket a programozó nem köteles kezelní.
- **Ellenőrzött kivételek** (checked exceptions): Ide tartozik minden `Exception`, kivéve a `RuntimeException` és annak leszármazottai. Mint látni fogjuk, az ellenőrzött kivételeket a programozónak kezelnie kell. Ha nem kezeli, akkor a fordító hibát jelez.

## 4.2. Kivételek keletkezése és szándékos előidézése – throw

A kivétel olyan objektum, amelynek osztálya az `Exception` osztályból származik. Kivételek az API valamelyik metódusa vagy a programozó kelthet, a következőképpen:

```
throw new <Kivételosztály>(<információs szöveg>);
```

Vagyis létrehozunk egy megfelelő osztályú objektumot (`new`), és beijtjük (`throw`) a program vérkeringésébe. Ezzel az objektum a kivételkezelő mechanizmus felügyelete alá kerül. A kivétel a metódushívási láncon visszafelé haladva kiszáll a programból.

Például:

```
throw new ArithmeticException("Osztás nullával!");
throw new NullPointerException("A null nem objektum!");
throw new TulNagyException("A lottóban nincs ekkora szám!");
```

A harmadik példában saját kivételosztályból hoztunk létre egy objektumot.

## A kivétel útja

### Feladat – Kivétel útja

Keltsünk szándékosan futási hibát a 4.3. ábrán látható módon! Kövessük a kivétel kiszállásának útját!

### Forráskód

```
// KivetelUtja.java //1
class Masik { //2
 static void m2() { //3
 throw new RuntimeException("Rosszalkodás"); //4
 }
}

public class KivetelUtja { //5
 static void m1() { //6
 Masik.m2(); //7
 }
}
```

```

public static void main(String[] args) { //12
 m1(); //13
} //14
} //15

```

### A program futása

```

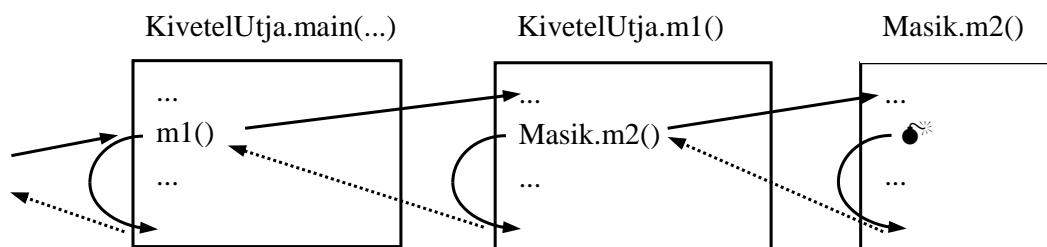
java.lang.RuntimeException: Rosszalkodás
 at Masik.m2(KivetelUtja.java:4)
 at KivetelUtja.m1(KivetelUtja.java:10)
 at KivetelUtja.main(KivetelUtja.java:13)
Exception in thread "main"

```

### A program elemzése

A programsorokat beszámoltuk, hogy követhető legyen a kivétel útja. A kivétel a `Masik.m2` metódusban keletkezett, //4-ben (ott, ahol az ábrán a kis bomba látszik). Ekkor a metódus végrehajtása megszakad, és a vezérlés a blokk végére, majd a hívó metódushoz kerül a kivétel-objektummal együtt. Az `m2` abnormálisan tér vissza (egy kivételejektumot hurcol), ezért megszakad a `KivetelUtja.m1` végrehajtása is: a vezérlés a metódusblokk végére, majd onnan a hívóhoz, a `main` metódushoz kerül. Mivel `m1` is abnormálisan tért vissza (ő is megkapja a kivételejektumot), azért a `main` metódus végrehajtása is megszakad. Itt azonban már megszakad a hívási lánc is, hiszen nincsen több hívó metódus. A program leáll, de előtte konzolra írja a következő információkat:

- ◆ A kivételestály neve, információs szöveg
- ◆ Metódushívási lánc: sorban a meghívott metódusok neve osztályukkal minősítve, zárójelben a fordítási egység neve és abban a hívott metódus sorának a száma.
- ◆ A szál neve, amelyikben a kivétel keletkezett (többszálú program futásakor nem csak a `main` szálban keletkezhet hiba).



4.3. ábra. A kivétel útja

## Példák

Nézzünk most néhány példát kivételobjektum keletkezésére!

### Feladat – Nullával való osztás

Tegyük a main metódusba egy műveletet, amelyben nullával osztunk! Mit ír ki a program a konzolra?

### Forráskód

```
// NullOsztas.java
public class NullOsztas {
 public static void main (String args[]) {
 int osszeg = 200;
 int darab = 0;
 int atlag = osszeg/darab; // Hoppá! Nullával osztás!
 // Ide már nem jut el a vezérlés:
 System.out.println("Minden rendben.");
 }
}
```

### A program futása

```
java.lang.ArithmetricException: / by zero
 at NullOsztas.main(NullOsztas.java:5)
Exception in thread "main"
```

### A program elemzése

A program main metódusában nullával osztunk. A vezérlés ebben a pillanatban a metódus végére ugrik; mivel a main nem kezelte a kivételt és a hívási láncnak vége van, azért lefut az alapértelmezés szerinti kivételkezelő, és a program leáll. Kiíródik a kivétoosztály neve és az információs szöveg:

```
java.lang.ArithmetricException: / by zero
```

majd a metódushívási lánc; annak most csupán egyetlen eleme van:

```
 at NullOsztas.main(NullOsztas.java:5)
```

végül megjelenik annak a számnak a neve, amelyben a kivétel keletkezett:

```
 Exception in thread "main"
```

A programban a "Minden rendben." nem kerülhet a konzolra.

**Feladat – Tömb túlindexelése**

A main által hívott metodusA-ban hivatkizzunk a tömb egy nem létező elemére! Mit ír ki a program a konzolra?

**Forráskód**

```
// TombTulindexeles.java
public class TombTulindexeles {
 static void metodusA() {
 int[] egeszek = new int[5];
 egeszek[5] = 12; // Hoppá! Túlindexelés!
 // Ide már nem jut el a vezérlés:
 System.out.println("Minden rendben a metodusA-ban.");
 }

 public static void main (String args[]) {
 metodusA();
 // Ide már nem jut el a vezérlés:
 System.out.println("Minden rendben a main-ben.");
 }
}
```

**A program futása**

```
| java.lang.ArrayIndexOutOfBoundsException
| at TombTulindexeles.metodusA(TombTulindexeles.java:4)
| at TombTulindexeles.main(TombTulindexeles.java:10)
| Exception in thread "main"
```

**A program elemzése**

A programban a kivétel a main által meghívott metodusA-ban keletkezik. Az 5 elemű tömb utolsó indexe 4, az egeszek[5] tehát érvénytelen hivatkozás. A hiba a Tulindexeles.java 4. sorában keletkezik, onnan a vezérlés ugyanennek a fordítási egységnek a 11. sorára kerül. Figyelje meg, hogy a metódushívási lánc kibővült!

**Feladat – Vektor túlindexelése**

Mi történik, ha a vektor nem létező elemére hivatkozunk?

**Forráskód**

```
// VektorTulindexeles.java
import java.util.*;

public class VektorTulindexeles {
 public static void main (String args[]) {
 Vector v = new Vector();
 v.get(0); // Hoppá! Nincs ilyen indexű elem!
 }
}
```

### A program futása

```
| java.lang.ArrayIndexOutOfBoundsException: Array index out of range:0
| at java.util.Vector.get(Vector.java:699)
| at VektorTulindexeles.main(VektorTulindexeles.java:6)
| Exception in thread "main"
```

### A program elemzése

A vektor használata közben könnyen beleeshetünk a túlindexelés hibájába, hiszen a `Vector` osztály több metódusa is ejthet `ArrayIndexOutOfBoundsException` kivételt. A `Vector` osztály `get` metódusát például a következőképpen kódolták:

```
public synchronized Object get(int index) {
 if (index >= elementCount)
 throw new ArrayIndexOutOfBoundsException(index);
 return elementData[index];
}
```

Ha tehát túl nagy indexet használunk, akkor a `get` metódus ejt egy `ArrayIndexOutOfBoundsException` kivételt. E kivétel konstruktora:

```
public ArrayIndexOutOfBoundsException(int index) {
 super("Array index out of range: " + index);
}
```

Az információs szöveg tehát az "Array index out of range: "+`index` lesz.

### További példák

```
String str = "45,7";
double d=Double.parseDouble(str); //-> NumberFormatException
Vector v; //-> v = null!!!
v.add("Eper"); //-> NullPointerException
Integer iObj = new Integer(5);
Double dObj = new Double(5);
int n = dObj.compareTo(iObj); //-> ClassCastException
```

Az eddig tárgyalt példákban a kivételek – egy-egy hibás operáció vagy metódushívás miatt – „maguktól” keletkeztek.

**Kivételt a programozó is könnyedén előidézhet a `throw` operátor segítségével.** A kivételkezelésben nem az a fontos, hogy mi történt valójában, hanem az, hogy a kivétes esemény révén milyen kivételobjektum került a rendszerbe.

#### Feladat – Nullával való osztás mesterségesen

Keltsünk nullával való osztás nélkül olyan kivételt, mintha nullával osztottunk volna!

### Forráskód

```
// MestersegesNullOsztas.java
public class MestersegesNullOsztas {
 public static void main (String args[]) {
 throw new ArithmeticException("Osztogatunk nullával?");
 }
}
```

### A program futása

```
| java.lang.ArithmetricException: Osztogatunk nullával?
| at MestersegesNullOsztas.main(MestersegesNullOsztas.java:3)
| Exception in thread "main"
```

### A program elemzése

Létrehozunk egy `ArithmetricException` kivételt, és a `throw` operátorral beejtjük a rendszerbe. Ezzel úgy teszünk, mintha nullával osztottunk volna. Az információs szöveg most más, mint az „eredeti”: a `/ by zero` szöveg helyett most az `Osztogatunk nullával?` szöveg íródik ki, hiszen a kivételobjektum létrehozásakor ezt a szöveget adtuk meg paraméterként.

## 4.3. A kivétel továbbadása – throws

Az eddigi példákban a keletkezett kivételek mindig a `RuntimeException` leszármazottai voltak, vagyis nem ellenőrzött kivételek.

**Az ellenőrzött kivételekkel** (vagyis azokkal a kivételekkel, amelyeknek az osztálya nem a `RuntimeException` utódja) a metódushívási lánc minden metódusában **foglalkoznunk kell** – ezt követeli a kivételkezelés mechanizmusa. Két lehetőségünk van:

- a metódusban **kezeljük** a kivételt (lásd következő pont),
- a kivételt **továbbadjuk** a hívó metódusnak: közöljük a rendszerrel, hogy ebben a metódusban ezt a kivételt nem akarjuk kezelní.

Ha a programozó egy metódusban nem akar kezelní némely ellenőrzött kivételt, akkor a metódusfej végére be kell tennie a következőt:

```
throws <Kivételosztály1, Kivételosztály2, ... >
```

A `throws` kulcsszó után több kivételosztály is felsorolható. A továbbadás az összes felso-rolt osztályra és azok utódaira vonatkozik. A `throws Exception` minden kivételt továbbad.

Ha egy metódus nem kezel egy hozzá érkezett ellenőrzött kivételt, akkor a metódusfej `throws` záradékában meg kell adni a kivétel osztályát vagy annak egy ősét, különben fordítási hiba keletkezik.

A kivételeket a hívási láncon felfelé minden metódusnak tovább kell adnia egészen addig, amíg valamelyik végül nem kezeli.

Minden kivételről nyilatkozni kell, ezért a **kivételelosztályok továbbadáskor nem szűkít-hetők!**

Ha egy metódus fejében például a `throws IOException` szerepel, akkor az őt hívó metódus fejében nem lehet `throws EOFException` (az `IOException` egy specializációja). A metódus által ejtett kivétel nem lehet bővebb, mint a kezelt, mert akkor bizonyos kivételek ellenőriztelenül szállnának ki a programból. Egy ősosztály továbbadása azonban mindenlegelőtt legális, hiszen az őstípusú szűrőn minden specializáció „fennakad”.

*Megjegyzés:* Ellenőrzött kivételt egyelőre csak mesterségesen tudunk előidézni – de később majd természetes formájukban is találkozunk velük, egyebek között az állománykezelésben.

### Feladat – Ellenőrzött kivétel továbbadása

Idézzünk elő egy main által hívott metódusban egy ellenőrzött, `IOException` típusú kivételt. Mit ír ki a program a konzolra, ha nem kezeljük a hibát?

### Forráskód

```
// EllenorzottDobas.java
import java.io.*;

public class EllenorzottDobas {
 static void metodusA() throws IOException {
 throw new IOException("Számot kérek!");
 }

 public static void main (String args[]) throws IOException {
 metodusA();
 }
}
```

### A program futása

```
java.io.IOException: Számot kérek!
 at EllenorzottDobas.metodusA(EllenorzottDobas.java:6)
 at EllenorzottDobas.main(EllenorzottDobas.java:10)
Exception in thread "main"
```

### A program elemzése

Ha bármelyik metódusfejből elhagynánk a kivétel továbbpasszolását, akkor a fordító hibát jelezne. `metodusA` továbbadja a kivételt `main`-nek, s végül a rendszer konzolra írja a kivétel üzenetét. A kivételnek tehát szabad utat adunk a programból való kiszálláshoz.

## 4.4. A kivétel elkapása, kezelése

A kivételek elkaphatók (elfoghatók) és kezelhetők; erre a `try-catch-finally` szerkezetet használjuk; az általános forma a következő:

```
try { // try blokk
 <utasítások>
}
catch (<Kivételosztály1> <obj1>) { // catch blokk
 <utasítások>
}
...
catch (<KivételosztályN> <objn>) { // catch blokk
 <utasítások>
}
finally { // finally blokk
 <utasítások>
}
```

Pontosan egy `try` blokk van, nulla vagy több `catch` blokk és legfeljebb egy `finally` blokk; egy `catch` vagy `finally` blokknak azonban mindenkorábban lennie kell. A `try-catch-finally` szerkezet egy metódusban bárhol elhelyezhető: a blokkban egymás után több is megadható, valamint része lehet szelekciójának vagy iterációnak. A `try-catch-finally` blokkok egymásba ágyazhatók, bár ettől a program olvashatósága erősen romlik.

### **try blokk**

A `try` blokk tartalmazza a program normális logikáját tükröző utasításokat. Általában a `try` blokk futása során keletkeznek azok a kivételek, amelyeket el kell fognunk. A blokk működésének esetei:

- ha a `try` blokk **normálisan lefut**, akkor végrehajtódik a `finally` blokk, (ha van), majd az azt követő utasításokra kerül a vezérlés.
- ha a `try` blokkot a `return` utasítással **elhagyjuk**, a `finally` akkor is végrehajtódik, még a metódusból való kiugrás előtt.
- ha a `try` blokk **végrehajtása közben kivétel keletkezik**, akkor a vezérlés a kivétel típusától függően valamely `catch` blokk végrehajtásával folytatódik. Végül mindenkorábban végrehajtódik a `finally` blokk.

### **catch blokk**

Minden `catch` blokk egy-egy kivételkezelőt definiál. A blokk fejében paraméterként pontosan egy formális kivételobjektum van megadva (pl. `Exception ex`), az osztálya csak a `Throwable` utódja lehet. Az aktuálisan érkező kivételobjektum értékadás szerint kompatibilis kell, hogy legyen ezzel a paraméterrel. **A blokk kezeli az érkező kivételobjektumot,**

**majd a blokk végén az objektum megszűnik.** A kivételobjektum manipulálható a blokkban.

Legfeljebb egy `catch` blokk hajtódiik végre. A `catch` blokkokat olyan sorrendben kell megadni, hogy a paraméterek osztálya egyre általánosabb legyen! Ha egy űskivétel előbb szerepelne, mint az utód, akkor az űs elkapná az utódkivételt is, és ezzel elérhetetlenné tenné az utódot definiáló `catch` blokkot. A fordítóprogram meg is követeli a helyes sorrendet.

Ha a kivétel egyik `catch` blokkra sem húzható rá, akkor a kivétel továbbadódik a hívó metódusnak. Ha ellenőrzött kivételről van szó, akkor a metódus `throws` kitételében definiálni kell a kivétel továbbadását. Ha a `try-catch-finally` sikeresen kezelte a kivételeket, akkor a blokk utáni utasítással folytatódik a program.

### **finally blokk**

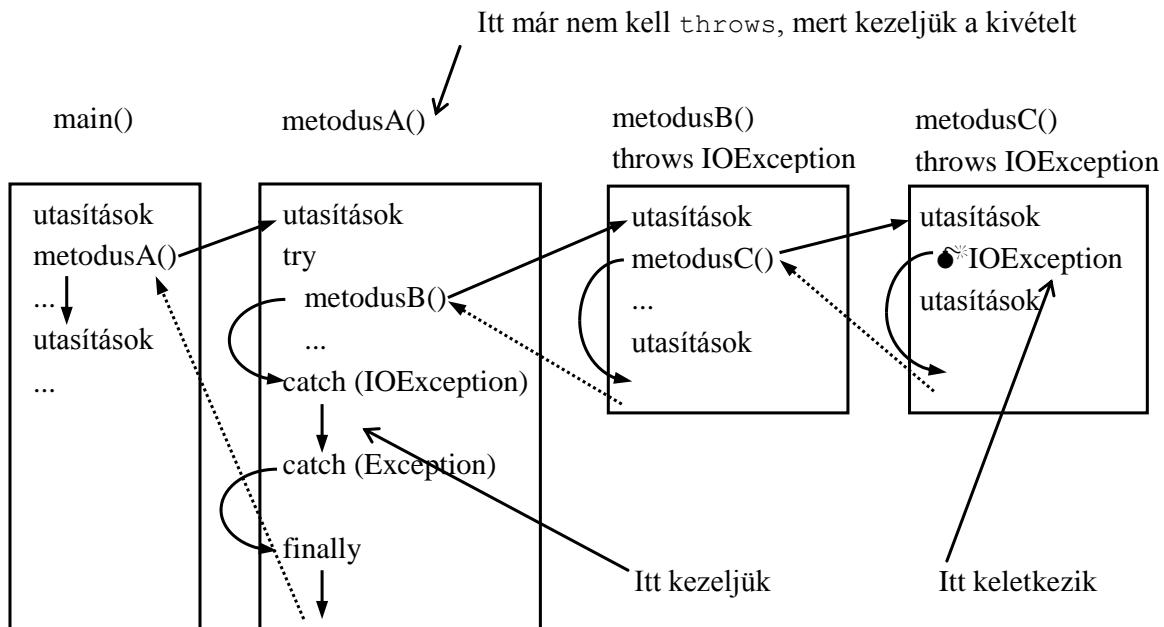
Ha van `finally` blokk, akkor **az minden körülmények között végrehajtódiik**, minden, hogy a végrehajtás normális volt-e, kiugrottunk-e a `try` blokkból, illetve hogy kezeltük-e a keletkezett hibát vagy sem. A `finally` blokkban szokás elvégezni a mindenkorban szükséges befejező tevékenységeket.

## **Kivételek a try blokkon kívül**

Kivételek a metódus bármely pontján keletkezhetnek, a teljes `try-catch-finally` blokkon kívül, a `catch` vagy a `finally` blokkokban is. A `try` blokkon kívül keletkező ellenőrzött kivételeket és azokat, amelyeket a `try` blokk nem kezel, tovább kell adni!

## **A kivétel életútja**

A 4.4. ábrán végigkísérjük egy `IOException` típusú kivétel útját, ahogy a kivételt elkapjuk. A `metodusC`-ben keletkezik egy `IOException` (például azért, mert olvasni akartunk egy állományt a lemezen, de az nincs ott). Sem a `metodusC`, sem a `metodusB` nem törödik a kivételel, ezt a metódusfejben a `throws` mutatja (a kivétel persze már megtörtént, és vállalunk kell a következményeit!). Végül `metodusA` megkapja a kivételt és ő kezeli. Itt a `metodusB` meghívása utáni utasítások nem hajtódnak végre; a kivételt az első olyan `catch` blokk kezeli, amelyikre a kivétel osztálya ráhúzható. Az első `catch` blokk tehát már kezeli, a kivételobjektum megszűnik, és a vezérlés a `finally` blokkra, majd az azutáni utasításokra kerül. `metodusA` tehát normálisan lefut, és a vezérlés visszakerül a `main`-hez. A `main` futása már zökkenőmentes.



4.4. ábra. A kivétel elkapásának útja

**Feladat – Kezelés**

Olvassunk be egy egész számot a konzolról. Csak a `Console.readLine()` metódust használhatjuk. Ne hagyjuk, hogy a program futási hibával elszálljon!

**Forráskód**

```
import extra.*;

public class Kezeles {
 public static void main (String args[]) {
 int szam;
 boolean ok = false;
 do {
 try {
 String str = Console.readLine("Egész szám: ");
 // NumberFormatException lehetséges:
 szam = Integer.parseInt(str);
 // Ide csak sikeres esetén kerül a vezérlés:
 ok = true;
 System.out.println("OK");
 }
 catch (NumberFormatException ex) {
 System.out.println("Még egyszer! ");
 }
 } while (!ok);
 }
}
```

### A program futása

```

Egész szám: G6
Még egyszer!
Egész szám: 12.6
Még egyszer!
Egész szám: 51
OK

```

### A program elemzése

A beolvasást a `main` metódusban végezzük el, és ott mindenki kezeljük is az esetleges kivételt. A kezelés mindenkor abból áll, hogy kiírjuk a felhasználónak: „Még egyszer!”. A kezelést egy `do` ciklusba tesszük, s azt mindaddig végrehajtunk, amíg végül már nem keletkezik kivétel-objektum, és végrehajtóhat az `ok=true` utasítás.

### Kivételek kezelése az inicializálókban

**Inicializáló kifejezések és blokkok nem ejthetnek ellenőrzött kivételt.** Azért nem, mert a rendszer automatikusan hívja meg az inicializáló egységeket. Az inicializálóknak nincs a programon belül hívójuk, ezért nem is kezelhetjük őket. Az ellenőrzött kivételeket még a kifejezés kiértékelése alatt, illetve a blokk lefutása előtt el kell fogni, illetve kezelní kell.

## 4.5. Saját kivételek használata

Gyakori eset, hogy mi magunk készítünk saját kivételosztályt, s az abból készült objektumokat szükség esetén beejünk a rendszerbe. A következő feladatban ilyen magunk által deklarált kivételosztályt használunk.

### Feladat – Saját kivétel alkalmazása

Írunk egy olyan `readSzam(tol,ig)` függvényt, amely bekér egy egész számot valamilyen megadott zárt tartományból (`tol` lehet nagyobb is, mint `ig`)! Addig kérünk számot, ameddig az végre a megadott tartományba esik. Ha a szám kívül esik rajta, akkor a metódus ejtsen egy saját (vagyis nem API-beli) `OutOfRangeException` kivételt és az abba foglalt információs szöveg közölje a megkövetelt határokat!

### Forráskód

```

// SajatKivetel.java
import extra.*;
// Saját kivételosztály:
class OutOfRangeException extends RuntimeException {
 OutOfRangeException(String s) {
 super(s);
 }
}

```

```
public class SajatKivetel {
 static int readSzam(int tol, int ig) {
 if (tol > ig) {
 int seged=tol; tol=ig; ig=segед;
 }

 int szam;
 while (true) {
 try {
 // NumberFormatException keletkezhet:
 szam = Integer.parseInt(Console.readLine("Szám: ")); //1
 if (szam < tol || szam > ig) //2
 throw new OutOfRangeException(tol+"-től "+ig+"-ig!");
 return szam;
 } // try
 catch (NumberFormatException ex) {
 System.out.println("Illegális karakter!");
 }
 catch (OutOfRangeException ex) {
 System.out.println(ex.getMessage());
 }
 } // while
 }

 public static void main (String args[]) {
 int lottoSzam = readSzam(1,90);
 System.out.println("OK");
 }
}
```

### A program futása

```
Szám: y99
Illegális karakter!
Szám: 91
1-tól 90-ig!
Szám: 4
OK
```

### A program elemzése

Az `OutOfRangeException` osztály csak jelölő szerepet játszik, hogy felismerjük, ha el szeretnénk fogni a kivételek objektumot. Mindössze a konstruktorát kell megírnunk: abban az információs szöveget továbbadjuk az ős `RuntimeException` konstruktorának.

A `main`-ben meghívjuk a `readSzam()` metódust. Ha ott //1-nél nem számot ütnek be, akkor egy `NumberFormatException` keletkezik (a `RuntimeException` leszármazottja); ezt a kivételt elkapja a `try` utáni első `catch` ág, és kiírja, hogy `Illegális karakter!` Ha számot ütöttek be, akkor még mindig lehet baj: ha a szám nem a `tol-ig` tartományba esik, akkor a metódus a rendszerbe ejt egy egyenesen erre a hibára létrehozott osztályból való (`OutOfRangeException`) kivételt (//2). Ezt a kivételt a második `catch` blokkja minden által el is kapja, és kon-

zolra írja a kivételbe foglalt szöveget. A végtelen ciklus gondoskodik arról, hogy ha nem jó a beírás, akkor azt meg kelljen ismételni. A ciklusból csak akkor léphetünk ki, ha sikerül a `try` blokk return utasításáig jutni, az pedig csak akkor megy, ha a beírással a rendszerben nem támad kivétel.

A két kezelt kivétel független egymástól, és mindenkor a `RuntimeException` leszármazottja; ezért mindegy, hogy milyen sorrendben kezeljük őket.

## Tesztkérdések

4.1. Jelölje be az összes igaz állítást!

- a) Kivételnek nevezünk minden olyan objektumot, amelynek osztálya a `Throwable` osztály leszármazottja.
- b) Egy szintaktikai hiba is kelthet kivételt.
- c) Az `Error` leszármazottjai olyan hibák, amelyek mindenkor leállítják a program futását.
- d) A `RuntimeException` osztály közvetlenül csupán konstruktorokat definiál.

4.2. Jelölje be az összes igaz állítást!

- a) A kivétel mindenkor elérhető a kiszállás útján.
- b) Programmal tetszőleges kivételt elő lehet idézni.
- c) Az ellenőrzött kivételeket a programozónak kezelnie kell!
- d) Az ellenőrzött kivételek a `RuntimeException` leszármazottai.

4.3. Mely kivételosztály a `RuntimeException` leszármazottja? Jelölje be az összes igaz állítást!

- a) `ArithmeticException`
- b) `FileNotFoundException`
- c) `LinkageError`
- d) `NumberFormatException`

4.4. Egy metódushoz ellenőrzött kivétel érkezhet. Szintaktikailag milyen helyes megoldások lehetségesek? Jelölje be az összes igaz állítást!

- a) A kivételt a metódusban kezeljük.
- b) A kivételt a metódusban nem kezeljük, hanem továbbadjuk a hívó metódusnak (`throws`).
- c) A kivételt a metódusban nem kezeljük, és nem is adjuk tovább.
- d) A kivételt a metódusban nem kezeljük, de továbbadjuk egy leszármazottját.

4.5. Jelölje be az egyetlen igaz állítást! Ha egy metódus nem kezel egy hozzá érkező ellenőrzött kivételt, akkor a metódusfej `throws` záradékában kell, hogy szerepeljen

- a) a kivétel pontos osztálya!
- b) a kivétel osztálya vagy annak egy öse!
- c) a kivétel osztálya vagy annak egy leszármazottja!
- d) az `Exception` osztály!

- 4.6. Mi igaz a try-catch-finally szerkezetre? Jelölje be az összes igaz állítást!
- a) Pontosan egy try blokk van.
  - b) Mindig van catch blokk.
  - c) Mindig van finally blokk.
  - d) Mindig van vagy egy catch blokk vagy egy finally blokk.

## Feladatok

- 4.1. (A) Keltsen `NullPointerException` kivételt egy `main` által hívott metódusban úgy, hogy a program futási hibával leálljon! Kétféleképpen oldja meg a feladatot:
- a) természetes hibával, a lehető legegyszerűbb módon!
  - b) mesterséges módon, vagyis a metódusban ejtsen kivételt!  
*(NullPointerException.java)*
- 4.2. (B) Próbálja elérni, hogy a `main` metódushoz érkező minden nullosztásos hibára magyar legyen a konzolra kiírt üzenet! *(MagyarNull.java)*
- 4.3. (A) Fogja el egy metódusban az esetlegesen keletkező `ArrayIndexOutOfBoundsException` kivételt. A program írja ki az "Ejnye-bejnye, indexelj már rendesen!" üzenetet, majd fússon tovább! *(Indexeles.java)*
- 4.4. (C) Egy metódus segítségével kérjen be konzolról egy dátumot (év, hó, nap)! Ha a beütött dátum érvényes, akkor a metódus adjon vissza egy érvényes dátumobjektumot; egyébként adja vissza a `null` objektumot. Hagyjuk abba a bevitelt, ha valamelyik értéknél nullát ütnek be! Érvényes dátum:
- év csak 1900 és 2010 között lehet;
  - hó értéke 1 és 12 közötti;
  - nap értéke pedig összhangban van a hónappal és azzal hogy az év szökőév-e.
- Először az évet kérje be egészen addig, amíg jó értéket nem ütnek be! Ezután kérje be a hónapot, majd a napot! Ha az érték nem megfelelő, akkor írasson ki a hibának megfelelő információs szöveget! Kivételkezeléssel oldja meg a feladatot!
- (DatumEllen.java)*

# II.



## I. OBJEKTUMORIENTÁLT TECHNIKÁK

1. Csomagolás, projektkezelés
2. Örökítés
3. Interfészek, belső osztályok
4. Kivételkezelés

## II. GRAFIKUS FELHASZNÁLÓI INTERFÉSZ

5. A felhasználói interfész felépítése
6. Elrendezésmenedzserek
7. Eseményvezérelt programozás
8. Swing-komponensek
9. Grafika, képek
10. Alacsony szintű események
11. Belső eseménykezelés, komponensgyártás
12. Applet

## III. ÁLLOMÁNYKEZELÉS

13. Állományok, bejegyzések
14. Folyamok
15. Közvetlen hozzáférésű állomány

## IV. VEGYES TECHNOLÓGIÁK

16. Rekurzió
17. Többszálú programozás
18. Nyomtatás
19. Hasznos osztályok

## V. ADATSZERKEZETEK, KOLLEKCIÓK

20. Klasszikus adatszerkezetek
21. Kölcsönös keretrendszer

## FELADATOK

## FÜGGELÉK

- A tesztkérdések megoldása  
Irodalomjegyzék  
Tárgymutató



## 5. A felhasználói interfész felépítése

---

A fejezet pontjai:

1. Komponensek és azok tulajdonosi hierarchiája
  2. AWT és Swing osztályhierarchia
  3. Swing mintaprogram
  4. Jellemzők
  5. Pont, méret, téglalap
  6. Koordinátarendszer
  7. Szín, betű
  8. Az absztrakt JComponent osztály
  9. Container osztály
  10. java.awt.Window osztály
  11. JFrame osztály
- 

Könyvünkben a programok egészen eddig algoritmusvezérelt módon, konzolos környezetben kommunikáltak a felhasználóval. Manapság azonban szinte minden program eseményvezérelt, és grafikus a felhasználói interfésze. A grafikus felhasználó felület (Graphical User Interface, GUI) programozására a Java kétféle osztálygyűjteményt (keretrendszert) ad, az AWT-t és a Swinget. Az AWT a régebbi rendszer, azt már nem fejlesztik tovább. A Swing az AWT-re épülő újabb osztálygyűjtemény. A Java tervezői mindenkorban arra törekedtek, hogy a felhasználói interfész platformfüggetlen legyen. A két osztálygyűjtemény jellemzői:

- ◆ **AWT** (Abstract Window Toolkit, absztrakt ablakozó eszköztár): Ez az osztálygyűjtemény felhasználja az operációs rendszerek saját, natív (gépi kódú) GUI komponenseit. A natív (másképp: peer) komponensek meghívásával a Java az éppen használt operációs rendszerre jellemző küllemet ad a grafikus felhasználói felületnek. A platformfüggetlenséget az adja, hogy a rendszerhívások révén mindenkorban a pillanatnyi platform ún. nehézsúlyú (heavyweight) komponensei futnak. Az AWT-osztályok a `java.awt` csomagban találhatók.
- ◆ **Swing**: A Java fejlesztői időközben rájöttek, hogy a peer komponensek hívása mégsem ideális, mivel a különböző platformokon más és más képernyőelemek jelentek meg. Ezért megírták az AWT-komponensek pehelysúlyú (lightweight, swing) változatát, és ki

is bővítették a gyűjteményt. A pehelysúlyú komponensek már nem használják az operációs rendszer natív elemeit – a Swing grafikusan rajzolja meg a komponenseket. A Swing az AWT-re épül, s intenzíven használja annak nem látható elemeit. A kirajzolás miatt a Swing-komponensek megjelenése sokkal lassúbb, mint a natívaké; a hardverek fejlődésével azonban ez nem sokáig lesz majd nehézség. A képernyőn megjelenő Swing-komponensosztályok azonosítója J betűvel kezdődik – például az AWT-ben deklarált `TextField`-nek a `JTextField` a swingbeli párja. A Swing-osztályok a `javax.swing` csomagban kaptak helyet.

A Swing alkotói a pehelysúlyú komponensek megalkotásában felhasználták a már meglevő AWT-komponenseket; a pehelysúlyú komponensek pókhálószerűen vannak rátelepítve ezekre a komponensekre. A Swing osztályhierarchiáját nem sikerült olyan kristálytisztán kialakítani, mint az AWT-ét. Az AWT azonban már elavult, taníthatóságán kívül nem sok előnye maradt (bár az appletekben még használják). A Swingbe rengeteg csábító dolog van beépítve, a programozónak könnyű rájuk szoknia. Ilyen többek között az ikonok használata a címkéken és a nyomógombokon, valamint a grafikus elemeket gyorsabban megjelenítő kettős pufferelési technika. Nem utolsó szempont a szabványos dialógusablakok megjelenése sem.

A felhasználói interfész komponensei a képernyőn látható objektumok: a keret, a nyomógomb vagy a beviteli mező. A komponensek programozásához láthatatlan, nem komponens osztályokat is felhasználunk, olyanokat, mint a pont, a téglalap, a szín és a betűosztályok. Ebben a fejezetben nagy vonalakban áttekintjük egyrészt az AWT nem komponens osztályait, másrészt a Swing komponenseit, és öröklési, tulajdonosi hierarchiájukat. Az ismereteket egy egyszerű mintaprogrammal támasztjuk alá.

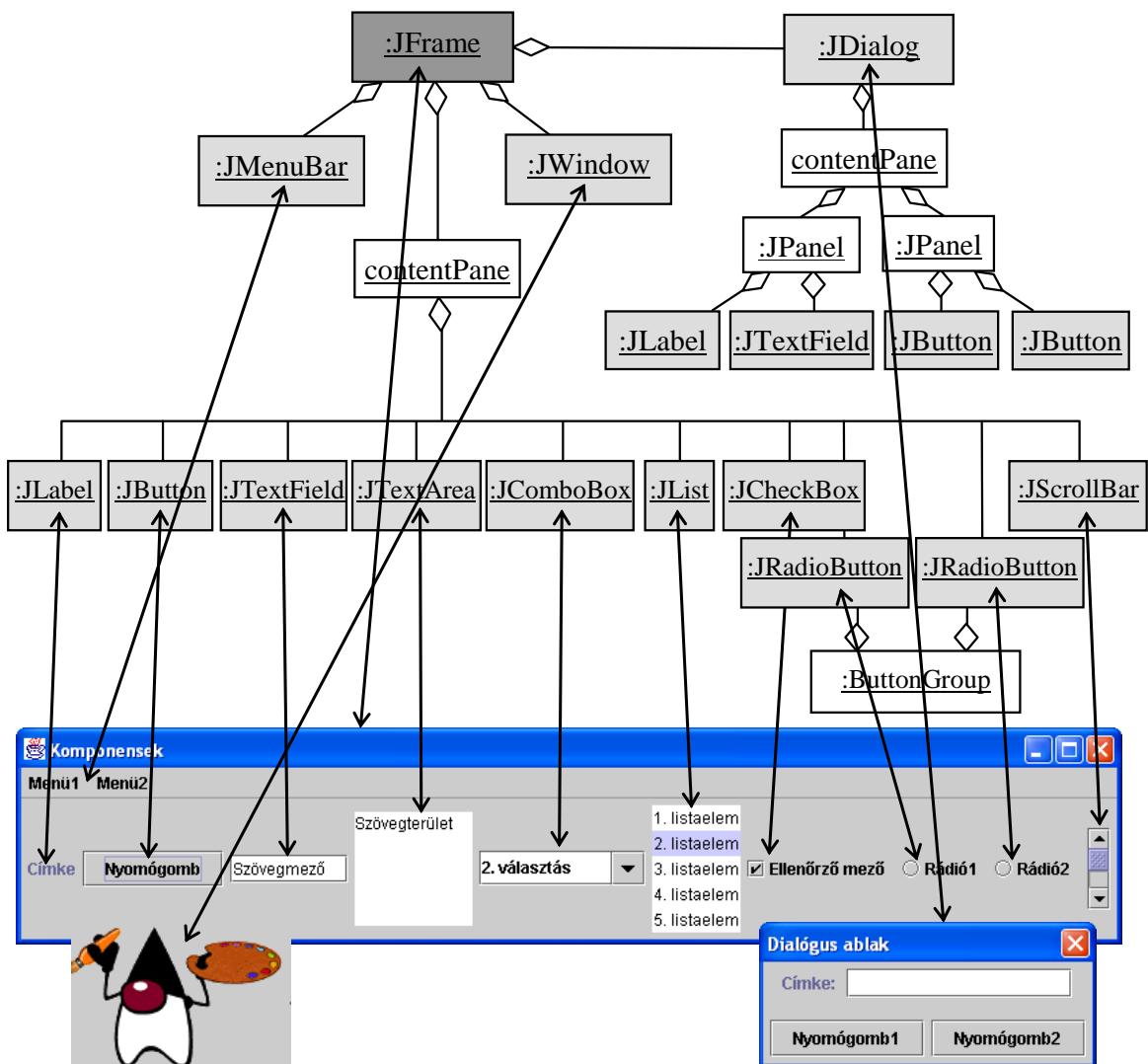
**Osztályok megadása:** Az egyes osztályokhoz csak a fontosabb, **publikus** deklarációkat adjuk majd meg a következő csoportokban: **mezők** (adatok), **jellemzők** (beállítható és lekérdezhető adatok), **konstruktorok** és **metódusok**. A `public` módosítót sehol nem írjuk ki.

## 5.1. Komponensek és tulajdonosi hierarchiájuk

Egy Java alkalmazás grafikus felhasználói interfésze (Graphical User Interface, GUI) komponensekből áll. A **komponensek téglalap alakú képernyőfelületek**, amelyeknek meghatározott tulajdonságaik vannak (elhelyezés, méret, szín, láthatóság stb.), és előre megadott szabályok szerint reagálnak különböző eseményekre (egér-, billentyűesemény stb.). A komponens osztálya minden a `java.awt.Component` leszármazottja. A Swing-komponenseknek közös ősük a `javax.swing.JComponent`.

Vannak olyan komponensek, amelyekre újabb komponensek tehetők, ezeket **konténer-komponenseknek** nevezzük – ilyen például az alkalmazás kerete és a dialógusablak. A **vezérlőkomponensek** nem bonthatók tovább – például a nyomógomb és a szövegmező. A

**felhasználói interfész komponenseinek jól meghatározott tulajdonosi (tartalmazási) hierarchiájuk van.**



5.1. ábra. A komponensek tulajdonosi hierarchiája

## Swing-komponensek

A 5.1. ábra egy „szedett-vedett” Java alkalmazás felhasználói interfészét mutatja: felül a komponensek objektumdiagramja látható, alul a képernyőn való megjelenésük. Az összeállítás tartalmazza a könyvben tárgyalt összes komponenst. A komponenshierarchia tetején egy keret

(`JFrame`) van; ez az alkalmazás fő tulajdonosa. Szürkével jelöltük az alsó képen látható elemeket.

A következő komponensek a keret közvetlen felügyelete alá tartoznak:

- ◆ A keret menüsora (`JMenuBar`); ezt most nem bontjuk tovább.
- ◆ A keret tartalompanelje (`contentPane`). A komponensek nem tehetők be közvetlenül a keretbe, csak a keret által felügyelt tartalompanelbe. A `contentPane JPanel` osztályú. A keret tartalompaneljén tíz vezérlőkomponens van: címke (`JLabel`), nyomógomb (`JButton`), szövegmező (`JTextField`), szövegterület (`JTextArea`) stb. Az utolsó vezérlő egy görgetősáv (`JScrollbar`). A vezérlők nem hagyhatják el tulajdonosuk területét.
- ◆ Egy egyszerű keret nélküli ablak (`JWindow`). A kapcsolat tartalmazási kapcsolat (ha a keretet becsukjuk, megszűnik az ablak is), az ablak azonban „elkészálhat” tulajdonosától.
- ◆ Egy dialógusablak (`JDialog`) – felhasználói bevitelre készült keretes ablak. A dialógusablakban közvetve négy vezérlőkomponens található: egy `JLabel`, egy `JTextField`, és két `JButton` – ezeket egy-egy láthatatlan panel (`JPanel`) fogja össze.

A komponensosztályokat a következő pontban röviden ismertetjük.

Az 5.1 ábrán látható felhasználói interfész a forráskód mellékletben megtalálható `Komponens-Hierarchia.java` program futásának eredménye. Futtassa a programot, és figyelje meg, hogyan viselkednek a különféle komponensek!

*Megjegyzések:*

- A komponensek tulajdonosi hierarchiája (objektumdiagram tartalmazási kapcsolatokkal) nem tévesztendő össze az öröklési hierarchiával (osztályhierarchia-diagrammal)!
- A tulajdonos (owner) más elnevezése szülő (parent), a birtokolt elemé pedig gyerek (child).

A felhasználói interfész felépítésében a **tulajdonosi (szülő-gyerek) viszony** kétféle lehet:

- **Konténer és komponense:** A keret, a dialógusablak és a panel mind konténer. A konténer olyan komponens, amely más komponenseket tartalmaz, összefogja és felügyeli őket. A konténer elemei fizikailag sosem kerülhetnek a konténeren kívülre. A konténer a neki szóló üzeneteket általában továbbadja a komponenseinek is: áthelyezésekor komponensei is vele mennek, átméretezésekor komponensei is megváltoztatják méretüket. Egy konténer megszűnése maga után vonja összes komponensének megszűnését. A konténerbe az `add(Component)` metódussal tehetünk be komponenst. Az 5.1. ábrán a keret komponense a menüsor (`JMenuBar`), a `contentPane` (tartalompanel), és a tartalompanel tíz vezérlőkomponense.

- **Ablak és ablaka:** Egy alkalmazás legfelső szintjén egy (esetleg több) keret áll. A keretek, ha több is van belőlük, különálló életet élnek, s minden az alkalmazás felügyelete alá tartoznak. A keretnek lehet(nek) ablaka(i), de kerete nem. A dialógusablaknak, illetve a közönséges ablaknak (a kerettel ellentétben) minden van tulajdonosa. Az ablak elhelyezkedése és mérete független a szülőablakétől. Egy ablak megszűnése maga után vonja az összes általa birtokolt ablak megszűnését. Az 5.1. ábrán a keretnek két ablaka van: egy `JWindow` és egy `JDialog`.

A felhasználói interfész tulajdonosi hierarchiájának felépítéséről a programozó gondoskodik. **Az a komponens, amely nincs rajta a tulajdonosi hierarchián, nem látható és eseményekre sem reagálhat.** A szülőkomponens megszűnése maga után vonja a gyerekkomponens megszűnését.

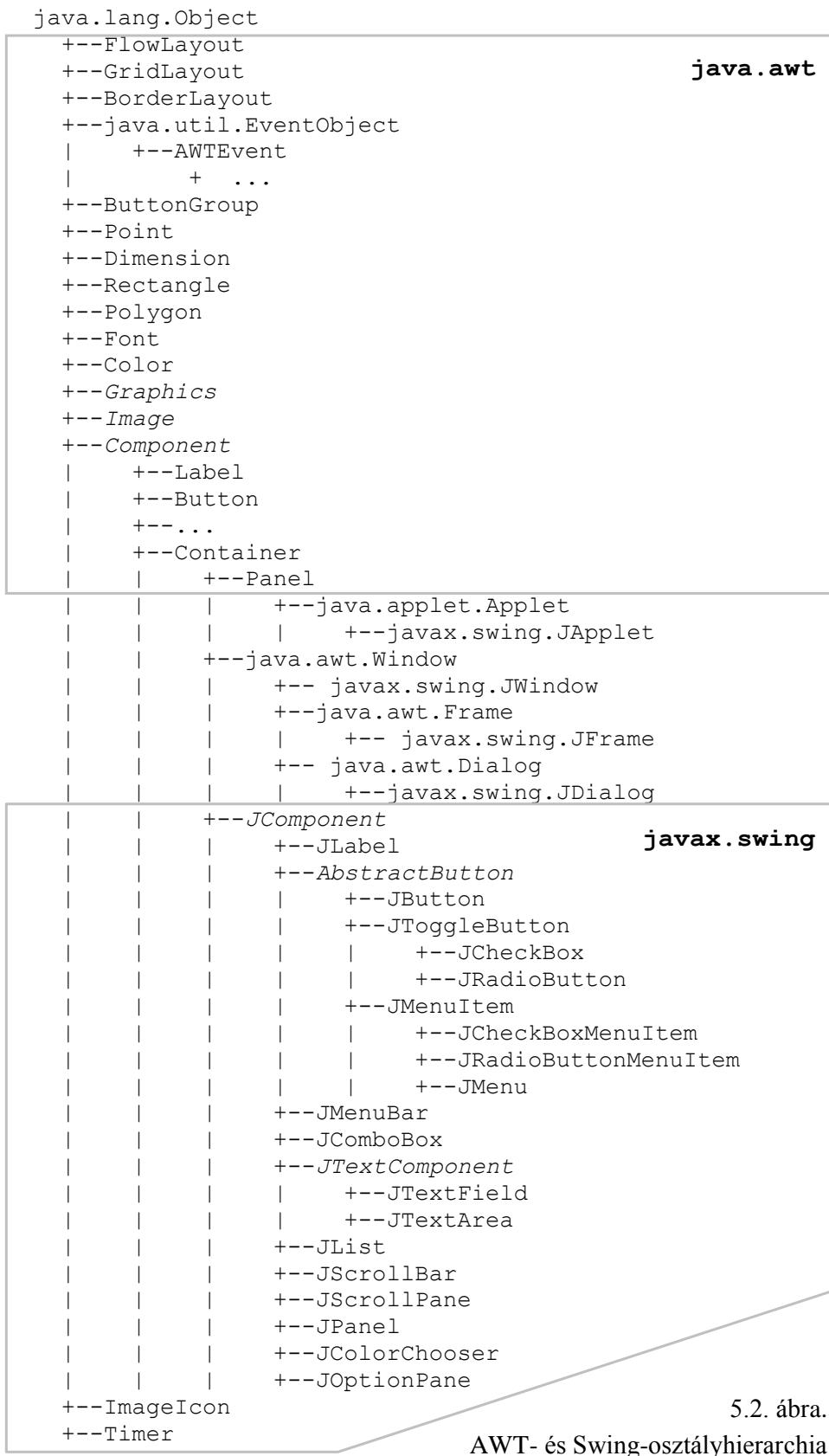
## 5.2. AWT- és Swing-osztályhierarchia

Most nagy vonalakban áttekintjük, milyen osztályokat használhatunk a felhasználói interfész elkészítésére. Az API ide vonatkozó osztályait a `java.awt` és a `javax.swing` csomag és alcsomagjaik tartalmazzák:

- ◆ `java.awt`: AWT-komponensek, rajzolás. Láthatatlan objektumok osztályai.
- ◆ `java.awt.event`: Az AWT- és Swing-komponenseken keletkező események kezelése.
- ◆ `java.awt.print`: Nyomtatás.
- ◆ `javax.swing`: Swing-komponensek.
- ◆ `javax.swing.event`: A Swing-komponenseken keletkező további események kezelése.
- ◆ `javax.swing.border`: A komponenseknek szegélyt (keretet) adó osztályok.
- ◆ `javax.swing.fileChooser`: Állomány kiválasztása lemezről.

E könyv csak a leggyakrabban használt csomagokat és osztályokat tárgyalja, és az osztályoknak csak a legfontosabb funkcióit. Célunk az AWT-ben és a Swingben való eligazodás, valamint egyszerű felhasználói interfések összeállítása, azok működésének megértése. Az 5.2. ábrán az ide vonatkozó, közvetlenül a `java.awt` és a `javax.swing` csomagban deklarált osztályok hierarchiája látható. Az összefüggő csomagrészket bekereteztük. Azokat az osztályokat, amelyek nem a bekeretezett csomagban vannak, csomagnevükkel minősítettük. Az ábra nem tartalmazza az eseményosztályokat; azok lényegében a `java.awt.event` csomagba kerültek, de a Swingnek is vannak saját, külön eseményei. Az eseményekkel a 7., az Eseményvezérelt programozás című fejezet foglalkozik majd.

Tekintsük át nagy vonalakban az 5.2. ábra fontosabb osztályait!



5.2. ábra.

AWT- és Swing-osztályhierarchia

## Vezérlőkomponensek

- ◆ **Component/JComponent:** A képernyön megjelenő AWT-/Swing-komponensek absztrakt őse. A Component osztályban deklarálták többek között a következő adatokat: `location` (bal felső sarok), `size` (méret), `background` (háttérszín), `foreground` (előtérszín), `font` (betű), `visible` (láthatóság) stb. Ezért minden komponensnek (az osztály bármely leszármazottjából létrehozott példánynak) van bal felső sarka, mérete stb. A `JComponent`-ben további adatokat deklaráltak, például `border` (szegély).
- ◆ **JLabel** (címke): Információ megjelenítésére használatos komponens. Az ábrán két címke szerepel, egy a keretben, egy a dialógusablakban.
- ◆ **JButton** (nyomógomb): A rajta való egérkattintás egy eseményt (akciót) indíthat el. Az ábrán három nyomógomb van, egy a keretben, kettő a dialógusablakban.
- ◆ **JCheckBox** (jelölőmező): Ha a jelölőmezőn vagy a hozzáartozó szövegen kattintunk, akkor az elem bejelölődik, s ha be volt bejelölve, akkor a bejelölés megszűnik. A jelölőmezők négyzet alakúak, és hagyományosan egymástól független bejelölésekre használják őket.
- ◆ **JRadioButton** (rádiógomb): Hasonló a jelölőmezőhöz, csak kör alakú. A rádiogombokat hagyományosan egymástól függő bejelölésekre használják. Ilyenkor a gombokat gombcsoporthoz kell foglalni, s attól az egyik gomb bejelölése kiugrasztja a benyomott gombot (a rádió gombjaihoz hasonlóan).
- ◆ **JComboBox** (kombinált lista): Beviteli mező és lista kombinációja. A mező szélén egy lefelé mutató nyíl jelzi, hogy választani is lehet a felkínált szövegekből.
- ◆ **JTextComponent:** A szöveges komponensek közös őse.
- ◆ **JTextField** (szövegmmező): Adatbeviteli mező. Szöveget lehet vele átadni a programnak.
- ◆ **JTextArea** (szövegterület): Többsoros, szerkeszthető szövegterület.
- ◆ **JList** (lista): A lista elemei nem szerkeszthető szövegsorok; az elemek kiválaszthatók (egyszerre egy vagy több). A listától meg lehet tudni, hogy éppen mely elemei vannak kiválasztva.
- ◆ **JScrollBar** (górgetősáv): A górgetősáv egy értéktartományt jelenít meg. A sáv két végén egy-egy nyíl, rajta pedig egy csúszka található. A csúszka pillanatnyi helyzete mutatja az értéktartomány aktuális értékét.
- ◆ **JScrollPane:** Görgetőpanel. Az erre helyezett komponenseknek görgetősávjuk lesz.
- ◆ **JMenuBar:** Menüsor; a menühierarchia tartóeleme. A menüsorba kibomló menüket (`JMenu`) lehetünk. A menü elemei a menütételek (`JMenuItem`); kiválasztásuk egy-egy eseményt (akciót) indíthat el. A menük egymásba ágyazhatók. A keretnek általában van menüsora.
- ◆ **JMenu:** Menü. A menühierarchia elemei.
- ◆ **JMenuItem:** Menütétel
- ◆ **JRadioButtonMenuItem:** Rádió-menütétel
- ◆ **JCheckBoxMenuItem:** Jelölő-menütétel

### Ablakok, konténerek

- ◆ **Container:** A konténerek őse. Egy konténerbe az `add()` metódussal lehet komponenseket tenni, a `remove()` metódussal meg ki lehet venni őket. Egy konténernek van elrendezésmenedzsere, jellemző fontja, színei stb.; ezek a tulajdonságok általában a gyerekeire is átruházódnak.
- ◆ **JFrame:** Keret. Van címe (title) és menüje. A keret az alkalmazás legfelső szintű ablaka, vagyis nem lehet senki birtokában (keretnek tehát nem lehet kerete). Az alkalmazás legfelső szintjén minden keret áll. A keret konténer, és a tartalompanelen keresztül komponensek tehetők bele.
- ◆ **JDialog:** A dialógusablak (dialógusdoboz) hasonlít a kerethez, de van tulajdonosa. A dialógust, ahogy neve is mutatja, a felhasználóval való párbeszédre találták ki. A dialógusablak általában modális (rajta kívül az alkalmazás minden komponense érzéketlen) és nem méretezhető.
- ◆ **Window/JWindow** (ablak): Egyszerű, keret nélküli ablak (egy téglalap a képernyőn). Az ablaknak nincs sem kerete, sem címe, sem menüje. A `Window` osztály fontos metódusa a `show()` – előtérbe teszi az ablakot – és a `pack()` – munkára utasítja az elrendezésmenedzserét. A `JWindow` a `Window` swinges változata – a szoftverek indító képeit szokta például „tartani”.
- ◆ **JPanel:** Komponensek összefogására használatos. A panelbe tett elemek nem hagyhatják el a panel területét.
- ◆ **JColorChooser:** Szabványos színválasztó dialógusablak.
- ◆ **JOptionPane:** Szabványos dialógusablak-gyűjtemény üzenetek megjelenítésére, adatok bevitelére stb.

### Nem látható komponensek

- ◆ **Color:** minden komponenshez tartozik egy háttérszín (background color) és egy előtérszín (szöveg vagy rajzsín, foreground color). Ez az osztály tárolja a szín kikeverését (összeállítását).
- ◆ **Font:** minden komponenshez tartozik egy font (betű) objektum, ez határozza meg a komponensre írt szöveg típusát, stílusát és méretét (például Times Roman, Bold, 12 pt). Ha nem adjuk meg, akkor a font automatikusan a tulajdonoskomponens fontja lesz.
- ◆ **FlowLayout, GridLayout, BorderLayout:** Elrendezésmenedzserek. Implementálják a `LayoutManager` interfész. minden konténernek (a `Container` osztály valamennyi leszármazottjának) van elrendezésmenedzsere: az rendezi el bizonyos szabályok szerint a konténer komponenseit. Ez azért fontos, mert a felhasználó bármikor átméretezheti a konténert (például a keretet), és ekkor a benne levő komponensek is elmozdulnak, illetve átméreteződnek. A `LayoutManager` interfész tartalmazza azokat az absztrakt deklarációkat, amelyeket egy elrendezésmenedzsernek mindenféle implementálnia

kell. A `FlowLayout`, a `GridLayout` és a `BorderLayout` osztály konkrét elrendezés-menедzser: sorfolytonosan, rácsosan, illetve határ mentén rendezik el a komponenseket.

- ◆ **Graphics**: Absztrakt osztály; leszármazottai fizikai képernyőfelületet reprezentálnak. minden komponensnek van egy platformfüggő és a rendszer által felügyelt `Graphics` objektuma; ezen rajzolódik ki a komponens. A programozó rajzolhat is a komponensekre, de csak a rendszer adta `Graphics` osztályú objektum révén.
- ◆ **Image**: Absztrakt osztály. Leszármazottjai képet tároló platformfüggő objektumok.
- ◆ **ImageIcon**: Rögzített méretű képet tároló osztály. Az `Icon` interfést implementálja.
- ◆ **Point, Dimension, Rectangle**: Nem komponens osztályok, vagyis a képernyőn nem láthatók – a komponens helyzetének és méretének megadásánál használhatók.
- ◆ **Polygon**: Sokszög azonosítására szolgáló osztály, a sokszög pontjait tárolja.
- ◆ **ButtonGroup** (gombcsoport): Nem komponens, a gombok összefogására szolgál. A csoportba szervezett rádiógombok közül minden csak egy lehet benyomva.

Az itt felsorolt komponensekről részletesen a 8., a Komponensek című fejezetben lesz szó.

**Eseményvezérelt program futása:** Egy keret létrehozásával a program eseményvezéreltté válik. Ez azt jelenti, hogy elkezd futni egy ún. AWT-programszál, s az állandóan figyeli a felhasználó által keltett eseményeket (pl. egérmozgatás, billentyűleütés). A `main` metódusnak, vagyis a program fő szálának lefutása után a program nem fejeződik be – az AWT-programszál tovább fut. Amíg az alkalmazásnak van érvényes ablaka, addig fut a `Frame` osztályban deklarált eseményfeldolgozó és szétosztó ciklus. A keletkezett eseményektől és a konkrét programtól függően sorban végrehajtódnak az eseménykezelő metódusok.

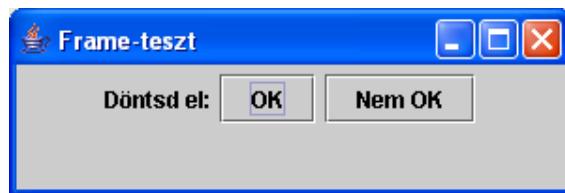
Az eseménykezelő metódusokat az osztályokban kell megírni. Bizonyos eseménykezelő metódusok már gyárilag benne vannak a komponensekben, ezért lehet például átméretezni, elmozdítani a keretet.

Az 5. és 6. fejezetben csak felületek összeállításával foglalkozunk, a felhasználói eseményeket egyelőre nem figyeljük. Eseményvezérlésről majd a 7. fejezetben lesz szó.

### 5.3. Swing mintaprogram

#### Feladat – Frame-teszt

Készítsük el az itt látható keretet! A keret bal felső sarka a képernyő (100,50) pozícióján legyen, mérete 300\*100, címe: Frame-teszt. A kereten legyen egy címke a Dönts el: szöveggel, továbbá legyen két nyomógomb, OK és Nem OK felirattal! A programnak egyelőre nem kell reagálnia semmilyen eseményre.



Három megoldást is készítünk:

- ◆ 1. megoldás: Keret összerakása kívülről: Először az osztálykönyvtár által kínált `JFrame` osztályból létrehozunk egy példányt, majd a keret tulajdonságait kívülről, üzenetekkel fogjuk beállítani, és az elemeket is kívülről fogjuk beledobálni. Ez egy „egyszer használatos” megoldás, az összehasonlítás kedvéért. Nem követendő példa.
- ◆ 2. megoldás: Keret összerakása belülről: Ebben a megoldásban a `JFrame` osztályból egy saját `SpecFrame` osztályt fogunk származtatni, s a beállításokat belülről fogjuk elvégezni, nem kívülről küldött üzenetekkel. Ez utóbbi megoldás sokkal szébb, hiszen a keret maga lesz felelős a külleméért.
- ◆ 3. megoldás: A konténer komponenseit nem deklaráljuk: Ez a megoldás minden összes technikai részletekben tér el az előzőtől.

#### 1. megoldás – Keret összerakása kívülről

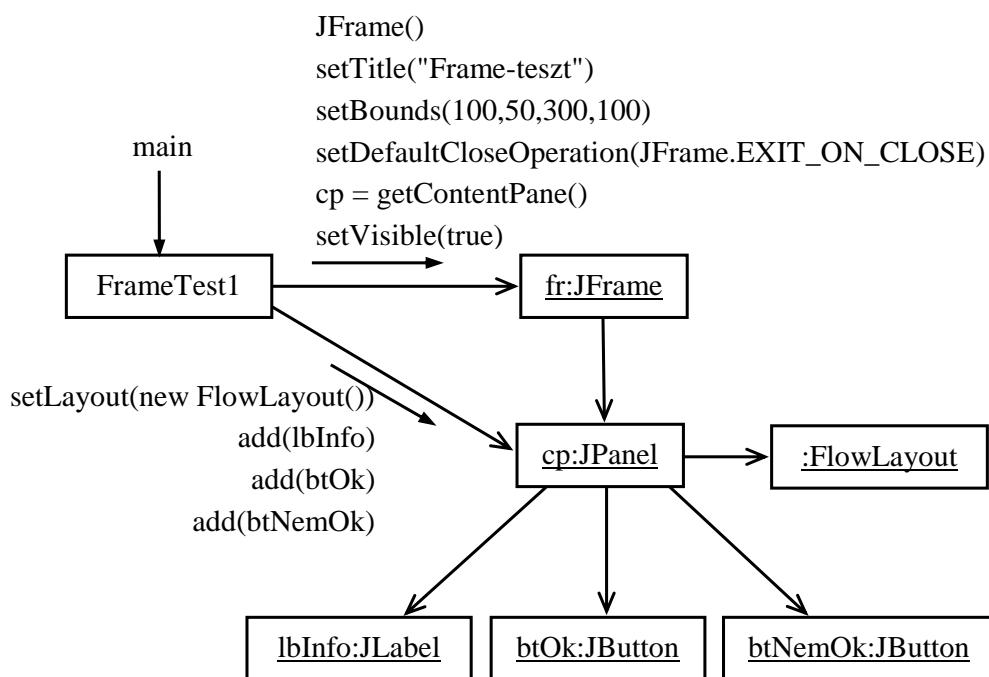
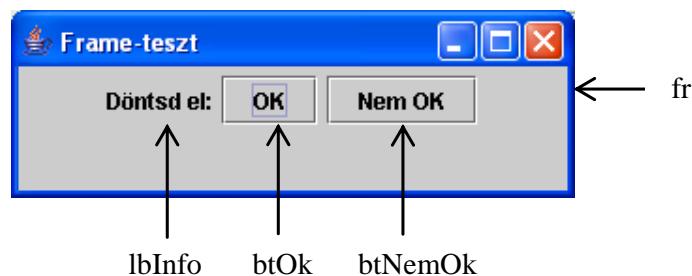
Ez a megoldás bevezető jellegű – a keretet nem ajánlatos így összerakni!

Nevezük el a programban szereplő objektumokat, és készítsük el az együttműködési diagramot (5.3. ábra)! A `main` metódust tartalmazó fő osztály a `FrameTest1`, Ő hozza létre az alkalmazás fő keretét: `fr:JFrame`. Ebbe a keretbe tesszük bele a tartalompanelen keresztül az `lbInfo` címkét, valamint a `btOK` és a `btNemOK` gombot.

A `main` metódus a következő üzeneteket küldi a keretobjektumnak:

- ◆ `JFrame()`: Létrehozza őt.
- ◆ `setTitle("Frame teszt")`: Beállítja a keret címét.
- ◆ `setBounds(100, 50, 300, 100)`: A keret bal felső sarkának koordinátája a képernyőn (100,50), a keret mérete 300\*100 pont lesz.

- ◆ `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)`: Ezzel a metódussal azt határozzuk meg, hogy mi történjék a keret becsukásakor. Az `EXIT_ON_CLOSE` azt jelenti, hogy a keret becsukására fejeződjék be a program futása.
- ◆ `cp = getContentPane()`: Elkéri a kerettől a tartalompanelt. Bár a tartalompanel JPanel osztályú, a metódus Container-ként adja át.
- ◆ `setVisible(true)`: Láthatóvá teszi a keretet (alapértelmezésben a keret nem látható).



5.3. ábra. FrameTest1 képernyőképe és együttműködési diagramja

A keretnek van kész tartalompanelje, `cp` arra mutat. A komponenseket a tartalompanelbe kell tenni. A tartalompanelnek küldött üzenetek:

- ◆ `setLayout(new FlowLayout())`: Sorfolytonos elrendezésmenedzsert rendel a tartalompanelhez. Azért van szükség rá, mert a tartalompanelnek alapértelmezés szerint határ menti az elrendezésmenedzsere, s az nem jó nekünk (lásd következő fejezet).
- ◆ `add(lbInfo), add(btOk), add(btNemOk)`: Ráteszi a tartalompanelre az előzőleg létrehozott `lbInfo` címkét, valamint a `btOk` és a `btNemOk` gombot. A panelen tehát három komponens lesz.

A tartalompanel elterpeszkedik a keretben, és sorfolytonosan kerülnek rá a vezérlők.

### Forráskód – FrameTest1

A forráskódban importálni kell az `awt` és a `swing` csomagot. Ebben a megoldásban az utasítások egyetlen osztály egyetlen `main` metódusában vannak:

```
import javax.swing.*;
import java.awt.*;

public class FrameTest1 {
 public static void main (String args[]) {
 // Komponensek deklarálása:
 JFrame fr;
 JLabel lbInfo;
 JButton btOk, btNemOk;

 // A csupasz keret létrehozása:
 fr = new JFrame();

 // Cím, pozíció és méret megadása:
 fr.setTitle("Frame-teszt");
 fr.setBounds(100,50,300,100);
 fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

 // Tartalompanel kikérése:
 Container cp = fr.getContentPane();

 // Tartalompanel elrendezésmenedzserének beállítása:
 cp.setLayout(new FlowLayout());

 // Komponensek létrehozása:
 lbInfo = new JLabel("Dönts el:");
 btOk = new JButton("OK");
 btNemOk = new JButton("Nem OK");
 // Komponensek beillesztése a tartalompanelbe:
 cp.add(lbInfo);
 cp.add(btOk);
 cp.add(btNemOk);

 // A keret láthatóvá tétele:
 fr.setVisible(true);
 } // main
} // FrameTest1
```

## 2. megoldás – Keret összerakása belülről

Ebben a megoldásban egy saját, SpecFrame keretosztályt definiálunk. A FrameTest2 osztály main metódusának minden összes részlete a keretet hozza létre. A keretet nem kell deklarálnunk, hiszen azt a létrehozás után nem szólítjuk meg. Figyelje meg, hogy most az üzeneteket belső metódushívások váltják fel (az fr minősítés elmarad)! A main-ből most nem csupasz keretet, hanem egy komponensekkel teli keretet hozunk létre, az összerakást a SpecFrame konstruktora végzi el.

### Forráskód – FrameTest2

```
import javax.swing.*;
import java.awt.*;

class SpecFrame extends JFrame {
 // Komponensek deklaráció:
 JLabel lbInfo;
 JButton btOk, btNemOk;

 // Konstruktur:
 public SpecFrame() {
 // Cím, pozíció és méret megadása:
 setTitle("Frame-teszt");
 setBounds(100,50,300,100);
 setDefaultCloseOperation(EXIT_ON_CLOSE);

 // Tartalompanel kikérésé:
 Container cp = getContentPane();

 // Tartalompanel elrendezésmenedzserének beállítása:
 cp.setLayout(new FlowLayout());

 // Komponensek létrehozása:
 lbInfo = new JLabel("Dönts el:");
 btOk = new JButton("OK");
 btNemOk = new JButton("Nem OK");

 // Komponensek beillesztése a tartalompanelbe:
 cp.add(lbInfo);
 cp.add(btOk);
 cp.add(btNemOk);

 // A keret láthatóvá tétele:
 setVisible(true);
 }
}

public class FrameTest2 {
 public static void main (String args[]) {
 // A komponensekkel tele keret létrehozása:
 new SpecFrame();
 } // main
} // FrameTest2
```

### 3. megoldás – A konténer komponenseit nem deklaráljuk, pakolunk

Ebben a megoldásban a komponenseket létrehozás után azonnal bedobjuk a konténerbe, így a deklarációk feleslegessé válnak. Csak azokat a komponenseket kell deklárnunk, amelyekre később hivatkozni szeretnénk. A `Window` osztályban deklárt `pack()` metódus optimális méretűre alakítja az ablakot, és az elrendezésmenedzser szabályai szerint elrendezi az ablak komponenseit. A `pack()` felülbírálja a keret méretét, vagyis a `setBounds` metódus két paraméterét feleslegesen adtuk meg. A `Component`-ben deklárt `setVisible(true)` helyett most a `Window` osztály `show()` metódusával tesszük láthatóvá a keretet. A `show()` előtérbe helyezi az ablakot.

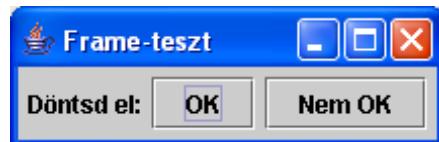
#### Forráskód – FrameTest3

```
import javax.swing.*;
import java.awt.*;

class SpecFrame extends JFrame {
 public SpecFrame() {
 setTitle("Frame-teszt");
 setBounds(100, 50, 300, 100);

 Container cp = getContentPane();
 cp.setLayout(new FlowLayout());
 cp.add(new JLabel("Dönts el:"));
 cp.add(new JButton("OK"));
 cp.add(new JButton("Nem OK"));
 pack();
 show();
 }
}

public class FrameTest3 {
 public static void main (String args[]) {
 new SpecFrame();
 } // main
} // FrameTest3
```



#### Elnevezési konvenció

**Elnevezési konvenció:** a komponenspéldányok neve két-három olyan betűvel kezdődik, amely jellemző az osztályukra: Például:

```
 JButton btOk, btMegse; // bt - button
 JLabel lbInfo; // lb - label
 JTextField tfNev; // tf - textField
 JFrame fr; // fr - frame
 JDialog dlgNevjegy; // dlg - dialog
```

A konvenció betartása nagyon ajánlatos, mert sokkal olvashatóbb lesz tőle a program! A konvenció Charles Simonyi nevéhez fűződik (Hungarian Prefix Notation).

## 5.4. Jellemzők

Az objektum beállítható és lekérdezhető tulajdonságát **jellemzőnek** (property) nevezzük. A beállítást a `set` metódus végzi el, a lekérdezést a `get`, vagy ha boolean típusú jellemzőről van szó, akkor az `is` metódus. **A beállító és lekérdező metódusok neve a jellemző nevétől és típusából adódik, a következő szabály szerint:**

Ha a jellemző típusa és neve:

```
PropType propName;
```

akkor a megfelelő `set` és `get` metódus:

```
void setPropName(PropType propName)
PropType getPropName()
```

Ha a jellemző típusa `boolean`:

```
boolean propName;
```

akkor a megfelelő `set` és `is` metódus:

```
void setPropName(boolean propName)
boolean isPropName()
```

A jellemzőknek a 4GL technikával való programozásban lesz kiemelkedő szerepük; ott vizuálisan állítható a jellemzők értéke.

Például:

| Jellemző                     | Beállító metódus                                | Lekérdező metódus                 |
|------------------------------|-------------------------------------------------|-----------------------------------|
| <code>int columns</code>     | <code>void setColumns(int columns)</code>       | <code>int getColumns()</code>     |
| <code>boolean visible</code> | <code>void setVisible(boolean visible)</code>   | <code>boolean isVisible()</code>  |
| <code>String text</code>     | <code>void setText(String text)</code>          | <code>String getText()</code>     |
| <code>Image iconImage</code> | <code>void setIconImage(Image iconImage)</code> | <code>Image getIconImage()</code> |

A könyvben az osztályok ismertetésekor a **Jellemzők** címszó alatt szereplő jellemzőknek nem adjuk meg explicit módon a beállító és lekérdező metódusait, azok maguktól értetődően léteznek.

## 5.5. Pont, méret, téglalap

Vannak olyan alaposztályok, amelyek maguk nem komponensek ugyan, de elengedhetetlenek a komponensek programozásában. Ilyen többek között a `Point`, a `Dimension` és a `Rectangle` osztály. Ezek az osztályok nem a `Component`-ből származnak, tehát nem láthatók a képernyőn.

## Point osztály

Csomag: java.awt Deklaráció: public class Point

Közvetlenős: java.awt.geom.Point2D

Fontosabb implementált interfések: Serializable

A **Point** osztály megjegyzi egy pont x és y koordinátáját.

## Konstruktörök, metódusok

- Point(int x, int y) // létrehozás
- double getX() // x-koordináta
- double getY() // y-koordináta
- void setLocation(int x, int y) // új hely
- void translate(int x, int y) // eltolás

## Dimension osztály

Csomag: java.awt Deklaráció: public class Dimension

Közvetlenős: java.awt.geom.Dimension2D

Fontosabb implementált interfések: Serializable

A **Dimension** osztály megjegyzi egy téglalap méretét (szélességét és magasságát).

## Konstruktörök, metódusok

- Dimension(int width, int height) // létrehozás
- double getWidth() // szélesség
- double getHeight() // magasság
- Dimension getSize() // méret
- void setSize(double width, double height) // új méret

## Rectangle osztály

Csomag: java.awt Deklaráció: public class Rectangle

Közvetlenős: java.awt.geom.Rectangle2D

Fontosabb implementált interfések: Shape, Serializable

A **Rectangle** osztály megjegyzi egy téglalap

- helyzetét (location): bal felső sarkának x és y koordinátáját, és
- méretét (dimension): szélességét (width) és magasságát (height).

## Konstruktörök, metódusok

- Rectangle(int x, int y, int width, int height)
- double getX() // bal felső sarok x-koordinátája
- double getY() // bal felső sarok y-koordinátája

```

► double getWidth() // szélesség
► double getHeight() // magasság
► Point getLocation() // bal felső sarok
► Dimension getSize() // méret
► void setLocation(int x, int y) // helyzetbeállítás
► void setSize(int width, int height) // méretbeállítás
► boolean contains(Point p) // ponttartalmazás vizsgálata
► boolean contains(Rectangle r) // téglalap-tartalmazás vizsgálata
► void grow(int h, int v) // méretnövelés
► void translate(int x, int y) // eltolás
► void add(Point p) // ponttal való kiterjesztés
► Rectangle intersection(Rectangle r) // metszet – ha üres: (0,0,0,0)
► Rectangle union(Rectangle r) // egyesítés

```

A Rectangle osztályban igen sok más metódus is van; a környezet automatikusan felkínálja őket, illetve kikereshetők a dokumentációból.

A pontok és téglalapok lehetőségeit egy példán keresztül tesszük érhetővé:

### Feladat – Pontok, téglalapok (PontTegla)

Adva van két téglalap: az egyik téglalap bal felső sarka a (100,100) pont, mérete 50\*30; a másik téglalap bal felső sarka a (120,80) pont, mérete 20\*60.

- Határozzuk meg a téglalapok közös részét!
- Vizsgáljuk meg, hogy a (130,110) pont benne van-e a közös részben!
- Határozzuk meg azt a legkisebb téglalapot, amely tartalmazza a téglalapokat és egy megadott pontsorozat összes pontját!

### Forráskód

```

import java.awt.*;

public class PontTegla {

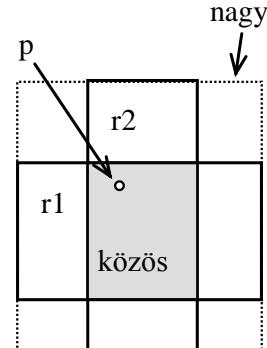
 public static void main (String args[]) {
 // r1 bal felső sarka (100,100), mérete 50x30:
 Rectangle r1 = new Rectangle(100,100,50,30);

 // r2 bal felső sarka (120,80), mérete 20x60:
 Rectangle r2 = new Rectangle();
 r2.setLocation(120,80);
 r2.setSize(20,60);

 // r1 és r2 közös részének meghatározása:
 Rectangle kozos = r1.intersection(r2);
 System.out.println("Közös: "+kozos);

 // A p pont koordinátái: (130,110):
 Point p = new Point(130,110);
 }
}

```



```

// Benne van a pont?
if (kozos.contains(p))
 System.out.println(p+" benne van");
else
 System.out.println(p+" nincs benne");

// Pontsorozat megadása:
Point[] pontok = {new Point(50,80),new Point(15,70),
 new Point(30,95),new Point(120,200)};

// r1 és r2 téglalapok egyesítése, majd a pontok hozzáadása:
Rectangle nagy = r1.union(r2);
for (int i=0; i<pontok.length; i++)
 nagy.add(pontok[i]);
System.out.println("Nagy: "+nagy);
System.out.println("Nagy mérete: "+nagy.getSize());
}
}

```

**A program futása**

```

Kozos: java.awt.Rectangle[x=120,y=100,width=20,height=30]
java.awt.Point[x=130,y=110] benne van
Nagy: java.awt.Rectangle[x=15,y=70,width=135,height=130]
Nagy merete: java.awt.Dimension[width=135,height=130]

```

*Megjegyzések:*

- Természetesen szebbé tehetnénk a kiírást, ha nem a megfelelő `toString()` metódusokat használnánk, hanem a magunk ízlése szerint formálnánk meg az adatokat.

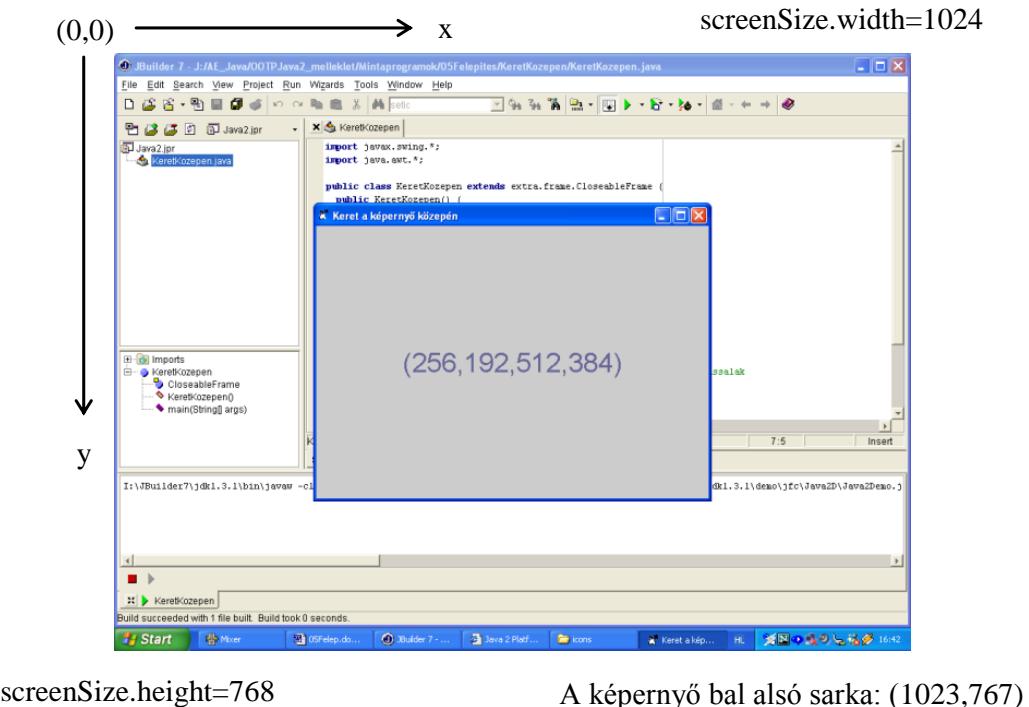
A pontot, a méretet és a téglalapot a (látható) komponensek paraméterezésekor fogjuk majd használni.

**5.6. Koordinárendszer**

Nézzük meg először a grafikus felület koordinárendszerét! A képernyő koordinátái az 5.4. ábra szerint értendők. Ezek szerint:

A **képernyő** bal felső sarka a  $(0, 0)$  pont,  $x$  a vízszintes,  $y$  pedig a függőleges tengely. Ehhez az abszolút koordinárendszerhez képest adjuk meg az ablakok bal felső sarkának koordinátáit. A konténerbe (ablakba, panelbe) tett komponensek bal felső sarkának koordinátáit a konténer bal felső sarkához rögzített koordinárendszerben adjuk meg, vagyis ezek a szülőhöz képest megadott relatív koordináták. A futó programból lekérdezhető a képernyő mérete.

Az ábrán levő keret a képernyő közepén van. Bal felső sarkának koordinátái:  $x=256$ ,  $y=192$ , mérete:  $512*384$ .



A képernyő bal alsó sarka: (1023,767)

5.4. ábra. Koordináták

## Toolkit osztály

Csomag: `java.awt` Deklaráció: `public abstract class Toolkit`

Közvetlen ős: `java.lang.Object`

Fontosabb implementált interfések: -

**A Toolkit osztály** sok alapvető, környezetfüggő metódust tartalmaz a képernyőkezeléshez, nyomtatáshoz stb.; megmondja például a képernyő méretét és felbontását. Az aktuális Toolkit példányt a statikus `getToolkit` metódussal el kell kérni a rendszertől.

## Metódusok

- |                                                   |                                                  |
|---------------------------------------------------|--------------------------------------------------|
| ► <code>static Toolkit getToolkit()</code>        | <code>// Alapértelmezett Toolkit elkérése</code> |
| ► <code>Dimension getScreenSize()</code>          | <code>// Képernyő mérete pontokban</code>        |
| ► <code>int getScreenResolution()</code>          | <code>// Képernyő felbontása (pont/inch)</code>  |
| ► <code>Image createImage(String filename)</code> | <code>// Kép betöltése és létrehozása</code>     |
| ► <code>void beep()</code>                        | <code>// Beépített hangszóró megszólal</code>    |

A képernyő mérete a következőképpen kérdezhető le:

```
Dimension screenSize=Toolkit.getDefaultToolkit().getScreenSize();
```

**Feladat – Keret középen**

Készítsünk olyan keretet, amelynek magassága és szélessége is fele a teljes képernyőnek, és a képernyő közepén helyezkedik el (5.4. ábra)! Írjuk a keretbe a maga (x,y,szelesség,magasság) adatait! Cseréljük ki a keret ikonját az integető hercegre (dukewave.gif)!

**Forráskód**

```

import javax.swing.*;
import java.awt.*;

public class KeretKozepen extends JFrame {
 public KeretKozepen() {
 Toolkit tk = Toolkit.getDefaultToolkit();
 setIconImage(tk.createImage("icons/dukewave.gif"));
 Dimension screenSize = tk.getScreenSize();

 int width = screenSize.width/2;
 int height = screenSize.height/2;
 int x = (screenSize.width-width)/2;
 int y = (screenSize.height-height)/2;
 setBounds(x,y,width,height);
 setTitle("Keret a képernyő közepén");
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container cp = getContentPane();
 JLabel lbInfo = new JLabel((""+x+","+y+","
 +width+","+height+""),JLabel.CENTER);
 lbInfo.setFont(new Font("Dialog",Font.PLAIN,35));
 cp.add(lbInfo);

 show();
 }

 public static void main (String args[]) {
 new KeretKozepen();
 }
}

```

Az x, y, width és height adatot a program elején számítjuk ki. Ha a felhasználó elmozdítja vagy átméretezi az ablakot, akkor ezek az adatok már nem lesznek érvényesek. A mindenkor valós adatok kiírásához figyelni kellene az ablakeseményeket.

## 5.7. Szín, betű

### Color osztály

Csomag: java.awt Deklaráció: public class Color

Közvetlen ős: java.lang.Object

Fontosabb implementált interfészek: Serializable

A **Color osztály** egy színt reprezentál. A Java a színeket RGB színként kezeli, vagyis minden színt három alapszínból: pirosból, zöldből és kékből kever ki. Az RGB e három szín mozaikszava: Red, Green, Blue. Amikor egy színt kikeverünk, megadjuk, hogy melyik alapszínból mennyi szerepeljen benne; minden színből egy 0 és 255 közötti értéket adhatunk meg. A három alapszín maximális bekeverésével a fehér színt kapjuk, s ha egyik színből sem keverünk semmit, akkor a nagy feketeszíget kapjuk.

A Java `Color` objektuma egy RGB színt tárol. A szín összetétele az objektum születése után nem változtatható meg. A `Color` osztály 13 statikus konstans színobjektumot definiál (`WHITE, RED,...`).

Minden komponensnek van háttérszíne és előtérszíne (betűszíne). A komponens színeit a `setBackground(Color)` és `setForeground(Color)` metódusokkal lehet beállítani (lásd következő pont, `JComponent` osztály). Ha egy komponensnek nem adjuk meg külön a színét, akkor az a szülő komponens színét „örökli”.

Példaként nézzünk néhány színkeverést: Fehér: (255,255,255); Fekete: (0,0,0); Piros: (255,0,0); Zöld: (0,255,0); Kék: (0,0,255); Világosszürke: (200,200,200); Középkék: (130,130,180).

Minél több van az egyes alapszínekből, annál világosabb hatást érünk el. A (0,0,150) például sötétebb kék, mint a (0,0,200).

## Konstansok

Statikus színobjektumok: `BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY, MAGENTA, ORANGE, PINK, RED, WHITE` és `YELLOW`. Például:

- ▶ `final static Color WHITE = new Color(255, 255, 255);`
- ▶ `...`

## Konstruktörök, metódusok

- ▶ `Color(int r, int g, int b)`  
Létrehozza a színobjektumot a megadott `r` (red), `g` (green) és `b` (blue), 0 és 255 közötti alapsínértékekkel.
- ▶ `int getRed()`
- ▶ `int getGreen()`
- ▶ `int getBlue()`

Visszaadja a megfelelő alapsín-összetevő értékét.

### Megjegyzések:

- A `Color` osztálynak sok egyéb tulajdonsága is van; a `java.awt.color` csomag pedig további lehetőségeket kínál. A színekkel e könyv nem foglalkozik mélyebben.
- A JDK előző verzióiban a színkonstansokat kisbetűvel írták. A kisbetűs változatok még élnek, vagyis szintaktikailag helyesek.

## SystemColor osztály

Csomag: java.awt Deklaráció: public class SystemColor

Közvetlen ős: java.awt.Color

Fontosabb implementált interfések: Serializable

A **SystemColor osztály** a rendszerre jellemző színkonstansokat definiál. A rendszerszínek használatával programunk színeit a számítógépen futó többi programéhoz igazíthatjuk.

### Konstansok

- static SystemColor desktop // Asztal háttérszíne
- static SystemColor info, infoText // Help színei
- static SystemColor text, textText // Szövegkomponensek színei
- static SystemColor control, controlText // Vezérlők háttér- és betűszíne
- static SystemColor scrollBar // Görgető színe

A következő két utasítással például kiírjuk a görgetősáv szabványos színének RGB összetevőit:

```
Color c = SystemColor.scrollbar;
System.out.println(c.getRed()+" , "+c.getGreen()+" , "+c.getBlue());
// 212, 208, 200
```

## Font osztály

Csomag: java.awt Deklaráció: public class Font

Közvetlen ős: java.lang.Object

Fontosabb implementált interfések: Serializable

A **Font osztály** egy objektuma valamelyen fontot (betűfajtát) reprezentál. A font tulajdon-ságai:

- **fontName**: A font neve. Például: Arial, Arial Black, SansSerif, Times New Roman.
- **style**: A font stílusa. Lehetséges fontstílusok: PLAIN (szimpla), BOLD (kövér), ITALIC (dölt), vagy BOLD+ITALIC (kövér dölt).
- **size**: A betű nagysága pontokban.

Ha a program nem találja a megadott fontot a számítógépen, akkor egy másik fontot jelen-tet meg. Az AWT 5 logikai fontnevet definiál; azokat a program „jó érzékkel” leképezi valamelyen hasonló fontra a pillanatnyi környezetben:

SansSerif, Serif, Monospaced, Dialog, DialogInput

A létrehozott fontobjektum tulajdonságait nem lehet megváltoztatni.

Minden komponensnek van valamelyen fontja, s azt a `setFont(Font)` metódussal lehet beállítani (lásd következő pont, JComponent osztály). Ha egy komponensnek nem adjuk meg külön a fontját, akkor az a szülőkomponens fontját „örökli”.

## Mezők

- ▶ static int BOLD, ITALIC, PLAIN

## Konstruktur

- ▶ Font(String name, int style, int size)

## Metódusok (csak lekérdezés)

- ▶ String getFontName()
- ▶ int getSize()
- ▶ int getStyle()
- ▶ boolean isBold()
- ▶ boolean isItalic()
- ▶ boolean isPlain()

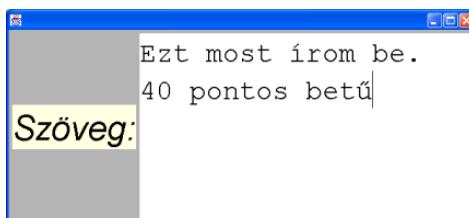
*Megjegyzések:*

- A Font osztálynak rengeteg egyéb tulajdonsága van, és a java.awt.font csomag további lehetőségeket is kínál. A fontokkal e könyv nem foglalkozik mélyebben.
- A mindenkorai környezet érvényes fontnevei a következőképpen kérdezhetők le:  

```
String[] fontNevek = java.awt.GraphicsEnvironment.
 getLocalGraphicsEnvironment().getAvailableFontFamilyNames();
```

## Feladat – Szín, font

Tegyünk egy középszürke keretre egy "Szöveg" címkét 50 pontos, dőlt Dialog betűvel! A címke háttér- és szövegszíne egyezzen meg a rendszer súgó (help) színeivel! A címke mellé tegyünk egy szövegterületet; annak 40 pontos, egyenközű (monospaced) legyen a betűje!



## Forráskód

```
import java.awt.*;
import javax.swing.*;

public class SzinFont extends JFrame {
 Container cp = getContentPane();

 public SzinFont() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);

 // A keret bal felső sarka (100,50):
 setLocation(100,50);

 // A cp panel elrendezése sorfolgytonos, színe középszürke:
 cp.setLayout(new FlowLayout());
 cp.setBackground(new Color(180,180,180));
 }
}
```

```

// Címke hozzáadása a tartalompanelhez:
JLabel lb;
cp.add(lb = new JLabel("Szöveg:"));
lb.setFont(new Font("Dialog", Font.ITALIC, 50)); //2
lb.setBackground(SystemColor.info); //3
lb.setForeground(SystemColor.infoText);
lb.setOpaque(true);

// Szövegterület hozzáadása a tartalompanelhez:
JTextArea ta;
cp.add(ta = new JTextArea(5, 20));
ta.setFont(new Font("Monospaced", Font.PLAIN, 40));

// Az ablak elrendezése és láthatóvá tétele:
pack();
show();
}

public static void main (String args[]) {
 new SzinFont();
}
}

```

### A program elemzése

- ◆ //1: Középszürkére állítjuk a keret háttérszínét – új színt keverünk, egy kicsit sötétebbet a keret eredeti színénél, a Color.LIGHT\_GRAY-nél. A keret háttérszínének beállítása hatással van az összes gyerek háttérszínére. Ezért a címke szintén szürke; a JTextArea azonban alapértelmezés szerint fehér.
- ◆ //2: Beállítjuk a címke fontját.
- ◆ //3: A SystemColor osztály info és infoText objektuma pontosan olyan színt ad meg, mint a rendszer súgójának háttérszíne, illetve betűszíne. A háttérszínt azonban csak akkor lehet látni, ha a címkét átlátszatlanná teszük (opaque==true).

## 5.8. Az absztrakt JComponent osztály

Csomag: javax.swing Deklaráció: public abstract class JComponent

Közvetlen ős: java.awt.Container

Fontosabb implementált interfések: Serializable

A **JComponent absztrakt osztály**, a képernyőn megjelenő Swing-komponensek közös ōse. A JComponent-ben definált jellemzők, metódusok tehát a felhasználói interfész bár-mely Swing-komponensére is értendők. minden komponensnek van többek között helyzete (bal felső sarka), mérete, háttér- és előtérszíne, fontja, egérkurzora és eszköztippje.

A `JComponent` egyik őse a `Component`, az AWT-komponensek közös őse. A `JComponent` a tulajdonságok nagy részét a `Component` osztálytól örököli; a `Component`-ben definiált tulajdonságok tehát már az AWT-komponensekben is megvannak.

A `JComponent`-nek rengeteg tulajdonsága és metódusa van, indulásképpen most csak néhányat sorolunk fel közülük; a komponensek további deklarációit a megfelelő fejezetekben tárgyaljuk.

*Megjegyzés:* Mivel a `JComponent` osztály közvetlen őse a `Container`, a péhelysúlyú komponensek elvileg minden konténerek. De mégsem ez a helyzet: a vezérlőkbe nem lehetünk be más vezérlőket.

### Mezők

- ▶ `static final float BOTTOM_ALIGNMENT`
- ▶ `static final float CENTER_ALIGNMENT`
- ▶ `static final float LEFT_ALIGNMENT`
- ▶ `static final float RIGHT_ALIGNMENT`
- ▶ `static final float TOP_ALIGNMENT`

Igazítási konstansok az `alignmentX` és `alignmentY` jellemző beállításához. Például:  
`komponens.setAlignment(JComponent.TOP_ALIGNMENT);`

### Jellemzők

A jellemzők a megfelelő `set` metódusokkal beállíthatók; a `get` metódusokkal – a boolean típusú jellemzők az `is` metódusokkal – lekérdezhetők.

- ▶ `Color background`
- ▶ `Color foreground`

A komponens háttérszíne és betűszíne. Alapértelmezésben a szülő színei.

- ▶ `boolean opaque`

Átlátszatlan-e. Ha értéke `true`, akkor átlátszatlan, ha `false`, akkor átlátszó. Ez utóbbi esetben a háttérszín nem érvényesül, és láthatjuk az alatta levő komponensem. Alapértelmezés: `false`. A háttérszín megadásának tehát csak akkor van hatása, ha az `opaque` érték két `true`-ra állítjuk.

- ▶ `Font font`

A komponens fontja. Alapértelmezésben a szülő fontja.

- ▶ `Cursor cursor`

A komponens egérkurzora. Ez az objektum határozza meg azt, hogy milyen lesz az egérkurzor, ha a komponens fölé visszük. Alapértelmezésben az egérkurzor a szülő egérkurzora. Komponens kurzorának megváltoztatása például:

```
komponens.setCursor(new Cursor(Cursor.WAIT_CURSOR)); // Homokóra
```

► **Border border**

A komponens szegélye. minden komponensnek lehet külön szegélye; szegélyeket a `javax.swing.border` csomagban találhatunk, a `BorderFactory` osztály statikus metódusai pedig kész szegélyeket adnak. A következő, komponens objektumnak például bemart szegélye lesz, s azt kétféleképpen is megadhatjuk:

```
komponens.setBorder(new EtchedBorder());
komponens.setBorder(BorderFactory.createEtchedBorder());
```

További szegélyek: `BevelBorder` (ferdén levágott szegély), `RaisedBevelBorder` (kiemelt, ferdén levágott szegély), `TitledBorder` (címes szegély) stb.

► **Dimension maximumSize**  
► **Dimension minimumSize**  
► **Dimension preferredSize**

Megadja a komponens maximális, minimális, illetve előnyös méretét. A komponens nem lehet nagyobb a maximális méretnél és kisebb a minimális méretnél. Az elrendezésmenedzser figyelembe veszi a komponens előnyös méretét.

► **float alignmentX**  
► **float alignmentY**

A vízszintes, illetve függőleges igazítás mértéke az x, illetve az y tengely mentén, más komponensekhez viszonyítva. Ez egy 0 és 1 közötti szám: 0 a kezdőponthoz, 1 a végponthoz, 0.5 a középponthoz való igazítást jelenti.

► **String toolTipText**

Eszköztipp. Ha beállítjuk a komponens eszköztipp szövegét, akkor a komponens tippet adhat: ha a felhasználó egy kicsit elidőz az egérrel a komponens felett, akkor megjelenik az eszköztipp szövege. Eszköztipp hozzáadása a komponenshez például: `komponens.setToolTipText("Ez egy komponens");`

► **boolean visible**

Láthatóság. Ha értéke `true`, akkor a komponens látható, feltéve, hogy a szülője is látható. Ha `false` az értéke, akkor a komponens létezik ugyan, de gyerekeivel együtt láthatatlan. Alapértelmezésben `true`.

► **boolean enabled**

Eseményfogadó képesség. Ha értéke `true`, akkor a komponens fogadhat eseményeket, egyébként érzéketlen a külső, felhasználói eseményekre. Alapértelmezésben `true`.

► **boolean requestFocusEnabled**

Ha értéke `true`, akkor a komponens fókusza hozható, egyébként nem. Alapértelmezésben `true`.

## Konstruktörök

Csak az utódosztályokból lehet példányt létrehozni.

### Helyzet, méret (metódusok)

- ▶ `int getX()`
- ▶ `int getY()`
- ▶ `int getWidth()`
- ▶ `int getHeight()`
- ▶ `Rectangle getBounds()`
- ▶ `Point getLocation()`
- ▶ `Point getLocationOnScreen()`
- ▶ `void setBounds(int x, int y, int width, int height)`
- ▶ `void setLocation(int x, int y)`
- ▶ `void setSize(int width, int height)`

A komponens bal felső sarkának (`x`, `y`) és méretének (`width`, `height`) állítása és lekérdezése különböző módokon: `location`: helyzet, `size`: méret, `width`: szélesség, `height`: magasság, `bounds`: határok. A koordináták relatívak a szülő komponenshez képest, kivéve a `getLocationOnScreen`-t: az a képernyő abszolút koordinátarendszereben adja meg a komponens helyzetét.

- ▶ `boolean contains(int x, int y)`
  - ▶ `boolean contains(Point p)`
- `true`, ha a komponens tartalmazza a pontot (relatív koordináta).

### Láthatóság, érvényesség (metódusok)

- ▶ `boolean isDisplayable()`  
Megmondja, hogy a komponens megjeleníthető-e. Egy komponens megjeleníthető, ha az eleme egy olyan tulajdonosi hierarchiának, amelyben megjeleníthető ablak a gyökér elem. Egy ablak megjeleníthető, ha arra alkalmazták a `pack` metódust vagy láthatóvá tették (`show` vagy `setVisible` metódussal). Egy komponens megjeleníthetetlen lesz, ha kiveszik a tulajdonosi hierarchiából vagy (`dispose` metódussal) felszabadították a gyökérablakot, s az emiatt megjeleníthetetlenné vált.
- ▶ `void validate()`  
A komponens érvényesítése, szükség szerint újrarájzolása. Egy komponens érvénytelen, ha újrarájzolásra szorulna, de az még nem történt meg.

### Fókusz, eseményfogadás (metódusok)

- ▶ `boolean hasFocus()`  
`true`, ha éppen ez a komponens van fókuszban. Az alkalmazásban legfeljebb egy komponens lehet fókuszban. A fókuszban levő komponens általában kiemelten látszik. Csak a fókuszban levő komponens fogadhat billentyűeseményeket.
- ▶ `void requestFocus()`  
Fókuszba helyezi a komponenst. Csak akkor van hatása, ha a komponens látható.
- ▶ `void transferFocus()`  
Továbbadja a fókuszt a következő komponensnek. Csak akkor van hatása, ha a komponens látható.

### Szülő, állapot (metódusok)

► `Container getParent()`

Visszaadja a komponens tulajdonosát (szülőjét). Ha nincs szülője (például a keretnek), akkor null az értéke.

► `String toString()`

Visszaadja a komponens jellemzőit, például:

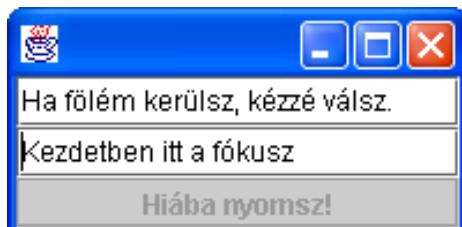
```
SpecFrame[frame0,100,50,112x50,layout=java.awt.BorderLayout,
 resizable,title=Proba]
```

► `void list()`

Konzolra írja a komponens és összes gyerekének jellemzőit. Jól használható hibakeresésre.

### Feladat – Komponensteszt (ComponentTest)

Tegyünk egy 200x100-as keretbe egymás alá két szövegmezőt és egy nyomógombot! A fókusz a második szövegmezőn legyen! A nyomógomb nem fogadhat eseményt! Ha az egérkurzor az első szövegterület fölött kerül, változzon át kézkurzorrá! Listázzuk ki konzolra a tulajdonosi hierarchiát!



### Forráskód

```
import java.awt.*;
import javax.swing.*;

public class ComponentTest extends JFrame {
 Container cp = getContentPane();
 JTextField tf1, tf2;
 JButton bt;

 public ComponentTest() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);

 setSize(200,100);
 cp.setLayout(new GridLayout(3,1)); // Rácsos elrendezés

 // 1. szövegmező létrehozása, egérkurzorának beállítása:
 cp.add(tf1 = new JTextField("Ha fölém kerülsz, kézzé válsz."));
 tf1.setCursor(new Cursor(Cursor.HAND_CURSOR));

 // 2. szövegmező létrehozása:
 cp.add(tf2 = new JTextField("Kezdetben itt a fókusz"));
 }
}
```

```

// A nyomógomb létrehozása és érzéketlenné változtatása:
cp.add(bt = new JButton("Hiába nyomsz!"));
bt.setEnabled(false);
show();

// Fókuszálás. Csak a show() után van hatása:
tf2.requestFocus();
}

public static void main (String args[]) {
 JFrame fr = new ComponentTest();
 fr.list(); // Kilistázzuk a keret tartalmát:
}
}

```

### Kiírás a konzolra (list hatása)

```

javax.swing.JPanel[,null.contentPane,0,0,192x66,layout=...
javax.swing.JTextField[,0,0,192x22,layout=...
javax.swing.JTextField[,0,22,192x22,layout=...
javax.swing.JButton[,0,44,192x22,disabled,layout=...

```

*Megjegyzés:* Az osztály deklarációiban nem minden tesszük ki a `private` módosítót. Azokban az osztályokban, amelyeket más osztály is használ, a deklarációkat ajánlatos priváttá tenni.

## 5.9. Container osztály

Csomag: `java.awt` Deklaráció: `public class Container`

Közvetlen ős: `java.awt.Component`

Fontosabb implementált interfések: -

A **Container osztály** a konténerkomponensek közös őse – az itt megadott metódusok tehát a felhasználói interfész bármely konténerére alkalmazhatók. A konténerbe gyerekkomponenseket lehet betenni, vezérlőket és újabb konténereket is. A gyerekkomponensek elrendezése a konténer elrendezésmenedzsereinek a feladata.

Csak a legfontosabb tulajdonságokat, metódusokat soroljuk fel; a konténer egyéb képességeit a megfelelő fejezetekben tárgyaljuk. A leggyakrabban használt „igazi” konténer a `JPanel`, a `JFrame` és a `JDialog`. A `contentPane` is konténer `JPanel`.

### Jellemzők

- ▶ `LayoutManager layoutMgr`

A konténer elrendezésmenedzsere. Alapértelmezésben `null`.

### Konstruktur

- `Container()`  
Általában csak az utódosztályokból szokás példányt létrehozni.

### Komponens hozzáadása, kivétele

- `Component add(Component comp)`
- `Component add(Component comp, int index)`  
Komponens hozzáadása a konténerhez. Ha nem adunk meg indexet, akkor a komponens utolsóként adódik a konténerhez. Ha indexet is megadunk, akkor a komponens a megadott indexű helyre szúródik be. Ha az index nagyobb, mint a legutolsó elem indexe, akkor `IllegalArgumentException` kivétel keletkezik. A visszatérési érték a betett komponens (azt azonban csak ritkán szokás használni).
- `void remove(Component comp)`
- `void remove(int index)`
- `void removeAll()`  
Komponens törlése a konténerből. Sorrendben: a megadott komponens törlése; az index sorszámú komponens törlése; az összes komponens törlése.

### Gyerekkomponensek

- `int getComponentCount()`
- `Component[] getComponents()`
- `Component getComponent(int n)`  
Sorban: visszaadja a gyerekkomponensek aktuális számát; az összes gyerekkomponenst, valamint az n. gyerekkomponenst. A komponensek tárolási sorrendje egyben a képernyőn való megjelenés és a bejárhatóság sorrendje is.
- `Component getComponentAt(int x, int y)`
- `Component getComponentAt(Point p)`  
Visszaadja azt a legfelső komponenst, amely tartalmazza a megadott pontot.
- `boolean isAncestorOf(Component comp)`  
A visszaadott érték `true`, ha a konténer tartalmazza a megadott komponenst (közvetve vagy közvetlenül).

### Elrendezés

- `Dimension getMaximumSize()`
- `Dimension getMinimumSize()`
- `Dimension getPreferredSize()`  
Megadja a konténer maximális, minimális, illetve előnyös méretét. A konténer nem lehet nagyobb a maximális méretnél és kisebb a minimális méretnél. Az elrendezés menedzser a komponensek előnyös méretei alapján kiszámolja a konténer előnyös méretét, és ennek megfelelően végzi az elrendezést.

## 5.10. java.awt.Window osztály

Csomag: java.awt Deklaráció: public class Window

Közvetlen ős: java.awt.Container

Fontosabb implementált interfések: -

A **java.awt.Window** osztály az összes AWT- és Swing-ablakkomponens közös őse.

Belőle származnak például a következő osztályok:

JFrame, JDialog, JWindow

Mindezen osztályokban megvannak tehát az alábbi tulajdonságok.

### Metódusok

- ▶ void pack()  
Elrendezi az ablakot a gyerekkomponensek előnyös méretei szerint; felülbírálja a közvetlenül megadott méreteket.
- ▶ void show()  
▶ boolean isShowing()  
A show láthatóvá teszi és előtérbe hozza a képernyőn az ablakot és annak gyerekeit. Az isShowing igaz, ha az ablak éppen előtérben van.
- ▶ public void setLocationRelativeTo(Component c)  
A c komponenshez képest helyezi el az ablakot a képernyőn. Ha az ablak nem látható vagy c==null, akkor a képernyő közepére teszi.
- ▶ void toBack()  
▶ void toFront()  
Háttérbe (előtérbe) küldi az ablakot. Ő lesz a tulajdonos legfelső (legfelső) ablaka, és ennek megfelelően átrendezi a többi látható ablakot.
- ▶ void hide()  
Elrejti az ablakot az összes alkomponensével és a felügyelete alatt levő ablakkal együtt.
- ▶ void dispose()  
Felszabadítja az ablaknak és összes gyerekének natív képernyő-erőforrását. Az ablak megjeleníthetetlen lesz (isDisplayable()==false), de továbbra is foglalja majd a memóriát.
- ▶ void addWindowListener(WindowListener l)  
Ablakesemény-felfigyelő hozzáadása az ablakhoz. Az eseményfigyelésről a 7. fejezetben lesz szó.
- ▶ Component getFocusOwner()  
Ha az ablak aktív, akkor visszaadja a fókuszból levő gyerekkomponenst (ha nincs ilyen, akkor önmagát). Ha az ablak nem aktív, akkor null a visszaadott érték.

- ▶ `Window getOwner()`  
Visszaadja az ablak tulajdonosát.
- ▶ `Window[] getOwnedWindows()`  
Visszaadja az ablak által felügyelt (birtokolt) ablakokat.

**Aktív ablak:** Az operációs rendszerben pontosan egy alkalmazás aktív, egy alkalmazásban pedig pontosan egy ablak aktív. Egy ablak rákattintással aktívvá tehető. A keret és a dialógusablak kerete ilyenkor megkülönböztetett színű lesz. A keret kivételével minden ablaknak van tulajdonosa (owner), s az is ablakleszármazott. A tulajdonosi hierarchia tetején azonban mindig egy keret áll.

*Megjegyzés:* A `JWindow` és `JDialog` osztályról részletesen a 8., a Swing-komponensek című fejezetben lesz szó.

## 5.11. JFrame osztály

Csomag: `javax.swing` Deklaráció: `public class JFrame`

Közvetlen ős: `java.awt.Frame`

Fontosabb implementált interfészek: `WindowConstants`

A keret (frame, osztálya `JFrame`) olyan legfelső szintű ablak, amelynek nincsen tulajdonosa. A keretnek van kerete, ikonja, címe és menüsora. Szegélyénél fogva áthelyezhető, átméretezhető, mozgatható, ikonná tehető stb. A keretet ikonná lehet változtatni (leküldeni a tálcára), ekkor csak egy kis kép reprezentálja. A `JFrame` az egyetlen olyan Swing-komponens, amely natív komponense az operációs rendszernek.

A keretbe nem lehet közvetlenül komponenseket tenni; arra a tartalompanelje (content pane) szolgál, abba kell beletenni a komponenseket. A tartalompanel elrendezésmenedzse-re alapértelmezés szerint határ menti (`BorderLayout`).

A keretnek nem küldünk a konténerre jellemző üzeneteket, amilyen az `add`, a `setLayout` stb., mert a komponenseit mindig a tartalompanel „tartja”. A keretnek ilyen üzenetek küldhetők: `setSize`, `setBounds`, `setTitle`, `pack`, `show` stb.

### Események

Az eseményekről részletesen a 7. fejezetben lesz szó.

### Jellemzők

- ▶ `String title`  
A keret címe.

- ▶ `Image iconImage`  
Az ikon a keret bal felső sarkában.
- ▶ `MenuBar menuBar`  
A keret menüsora.
- ▶ `boolean resizable`  
A keret méretezhető-e; ha `true`, akkor átméretezhető, egyébként nem.
- ▶ `int state`  
A keret állapota: normál (`NORMAL`) vagy ikonná változtatott (`ICONIFIED`). Alapértelmezés: `NORMAL`.

### Mezők

- ▶ `static final int NORMAL`
- ▶ `static final int ICONIFIED`  
A keret állapotának lehetséges értékei: Normális vagy ikonná változtatott (ikonizált).
- ▶ `static final int DO NOTHING ON CLOSE`
- ▶ `static final int HIDE ON CLOSE`
- ▶ `static final int DISPOSE ON CLOSE`
- ▶ `static final int EXIT ON CLOSE`  
Azt adják meg, hogy mi történik a keret becsukásakor. Ezek a mezők valójában a `javax.swing.WindowConstants` interfészben vannak deklarálva, de a keretből is elérhetők. A `setDefaultCloseOperation` metódus használja őket.

### Konstruktörök

- ▶ `JFrame(String title)`
- ▶ `JFrame()`  
Keret létrehozása. `title` a keret címe. Alapértelmezés: nincs címe.

### Metódusok

- ▶ `Container getContentPane()`  
Visszaadja a keret tartalompaneljét. A komponenseket erre kell rátenni!
- ▶ `void setDefaultCloseOperation(int operation)`  
Ezzel a metódussal azt adjuk meg, hogy mi történjen, ha a keretet megpróbálják becsukni. Lehetséges paraméterek:
  - `DO NOTHING ON CLOSE`: Nem történik semmi. A keret tovább „él” és mutatkozik.
  - `HIDE ON CLOSE`: A keret tovább „él”, de láthatatlanná válik.
  - `DISPOSE ON CLOSE`: A keret „meghal”, de a program tovább fut (akkor is, ha az utolsó keretet csukjuk be).
  - `EXIT ON CLOSE`: A program futása befejeződik.

► static Frame[] getFrames()

Osztálymetódus; visszaadja a programban létrehozott összes keretet – a már meg nem jeleníthetőket is. A metódus a `Frame` osztályban van deklarálva, emiatt a visszaadott keretek `Frame` osztályúak.

### Feladat – Két keret

Hozzunk létre két keretet:

- az egyik mérete 500\*200, legyen a képernyő közepén, címe "Első", és nem méretezhető át;
- a másik bal felső sarka a (0,0) pont, mérete 200\*200, címe "Második", átméretezhető, és ikon állapotú !

Írjuk ki konzolra az alkalmazásban szereplő keretek adatait!

Futtassa a programot! Próbálja átméretezni az első keretet! Nagyítsa ki a második, ikonná változtatott keretet!

### Forráskód

```
import javax.swing.JFrame;
import java.awt.Frame;

public class KetKeret {
 public static void main(String[] args) {
 JFrame fr = new JFrame("Első");
 fr.setSize(500,200); // mérete 500*200
 fr.setLocationRelativeTo(null); // a képernyő közepén lesz
 fr.setResizable(false);
 fr.show();

 fr = new JFrame("Második");
 fr.setBounds(0,0,200,200);
 fr.setState(JFrame.ICONIFIED);
 fr.show();

 System.out.println("Keretek az alkalmazásban");
 Frame[] frames = Frame.getFrames();
 for (int i=0; i<frames.length; i++)
 System.out.println(frames[i]);
 }
}
```

### A program futása

|                                                                                         |
|-----------------------------------------------------------------------------------------|
| Keretek az alkalmazásban                                                                |
| java.awt.Frame[frame0,100,100,500x200,layout=java.awt.BorderLayout,title=Első]          |
| java.awt.Frame[frame1,0,0,200x200,layout=java.awt.BorderLayout,resizable,title=Második] |

## Tesztkérdések

- 5.1. Mi igaz egy konténerkomponensre? Jelölje be az összes jó választ!
- a) A konténerkomponens a Component osztály leszármazottja.
  - b) A konténerkomponens a Container osztály leszármazottja.
  - c) A konténerkomponens minden ablak.
  - d) A konténer felügyeli a gyerekkomponenseit.
- 5.2. Mely állítások igazak? Jelölje be az összes jó választ!
- a) A JFrame, a JTextField és a JComboBox a java.awt csomagban van deklarálva.
  - b) Amit láthatóvá lehet tenni a képernyőn, az minden a Component osztály leszármazottja.
  - c) A java.awt csomag minden osztálya a Component osztályból származik.
  - d) A JButton osztály a JLabel leszármazottja.
- 5.3. Mely osztály közvetlen vagy közvetett őse a ButtonGroup-nak? Jelölje be az egyetlen jó választ!
- a) JFrame
  - b) JCheckBox
  - c) JComponent
  - d) Object
- 5.4. Mely osztályok nem leszármazottai a JComponent osztálynak? Jelölje be az összes jó választ!
- a) Polygon
  - b) Color
  - c) JCheckBox
  - d) Font
- 5.5. A következő Swing-komponensek közül melyik konténer? Jelölje be az összes jó választ!
- a) JButton
  - b) JPanel
  - c) JDialog
  - d) JList
- 5.6. Mely állítások igazak? Jelölje be az összes jó választ!
- a) A JButton közvetlenül az Object-ból származik.
  - b) A JTextField közvetlen őse a JComponent.
  - c) A JPanel a JFrame-ből származik.
  - d) A JLabel közvetlen őse a JComponent.
- 5.7. A Color osztályt felhasználva milyen összetevőkkel állíthat elő új színeket? Jelölje be az egyetlen jó választ!
- a) HSC (highlight, saturation, color)

- b) RGB (red, green, blue)
- c) COM (cyan, orange, magenta)
- d) BCC (brightness, color, contrast)

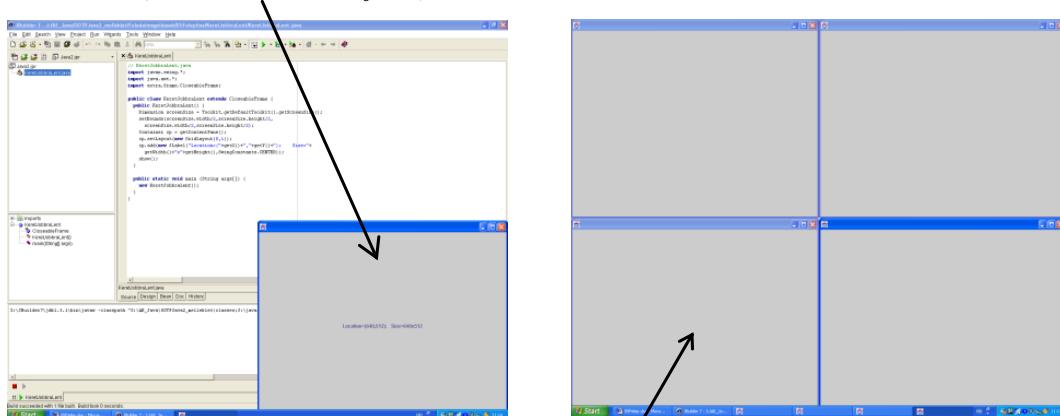
## Feladatok

5.1. (A) Tegyen a keretbe egy címkét és egy gombot! Jelenítse meg az ablakot! (*CimkeGomb.java*)

5.2. (A) Tegyen a keretbe legalább öt, tetszőleges komponenst! Hozzon létre három ilyen keretet! A kereteket tetszőleges sorrendben be lehet csukni. (*HaromKeret.java*)

Tipp: Az `extra.frame.CloseableFrame` osztály egy olyan keret, mely becsukáskor figyeli, hogy ő volt-e az utolsó keret az alkalmazásban. Ha ő volt az utolsó, akkor a program futása befejeződik.

5.3. (A) Tegyen egy keretet a képernyő jobb alsó egynegyedébe, és írja rá a keret helyzetét és méretét! (*KeretJobbraLent.java*)



5.4. (B) Tegyen 4 keretet a képernyőre, mindegyik foglalja le a képernyő egynegyedét! A kereteket egyenként csukja be! (*NegyKeret.java*)

5.5. (B) Hozzon létre egy keretet, és írja rá sötétzöld színű, 32 pontos vastag, dőlt Times New Roman betűvel, hogy "Ide nézz!"! A keret bal felső sarka a (100,50) pont, alapsíne fehér legyen! A keret mérete akkora legyen, hogy beleférjen a szöveg! (*IdeNezz.java*)



## 6. Elrendezésmenedzserek

---

A fejezet pontjai:

1. Az elrendezésmenedzserek tulajdonságai
  2. FlowLayout – sorfolytonos elrendezés
  3. GridLayout – rácsos elrendezés
  4. BorderLayout – határ menti elrendezés
  5. JPanel, az összefogó konténer
- 

A felhasználói interfész megtervezésében figyelembe kell vennünk, hogy a programot a felhasználók többféle platformon, más-más felbontású képernyőn is futtatni szeretnék. Ha egy ablak és a benne levő komponensek koordinátáit és méreteit a programozó adja meg, akkor előfordulhat, hogy a megszokotttól eltérő képernyőfelbontásban a komponensek furcsa helyen, szerencsétlen elrendezésben fognak megjelenni, vagy nem is lesznek láthatók. Ezt a jelenséget úgy lehet elkerülni, ha a program a mindenkor környezetben és helyzetben folyamatosan újraszámolja a komponensek elhelyezését és méretét. Az elrendezésmenedzserek arra valók, hogy a számlagatás súlyos terhét levegyék a vállunkról: minden konténernek van elrendezésmenedzsere, s az szükség esetén (például az ablak első megjelenésekor vagy átméretezésekor) kiszámolja a tartalmazott komponensek koordinátáit és méretét, mégpedig optimális módon. Többféle elrendezésmenedzser létezik: az egyik sorfolytonos helyezi el az elemeket és nem változtat a méretükön, a másik egyenlő méretű rácselemeket készít, a harmadik égtájak szerint helyezi el a komponenseket stb.

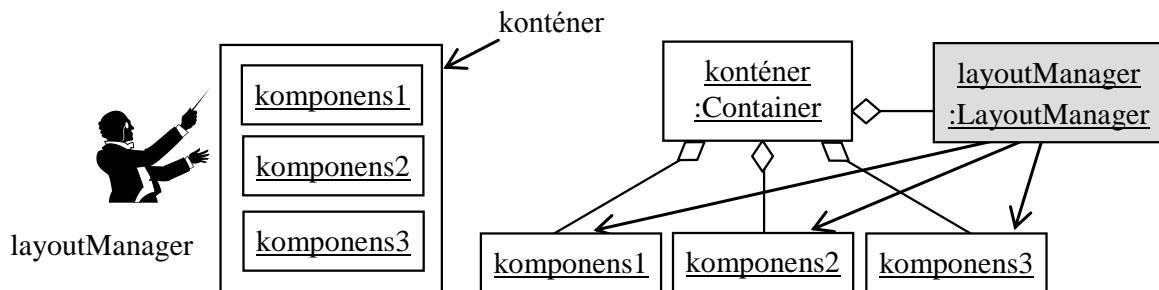
Ebben a fejezetben a `java.awt` csomagban felkínált öt elrendezésmenedzser közül háromat fogunk részletesen tárgyalni: a sorfolytonos (`FlowLayout`), a rácsos (`GridLayout`) és a határmenti (`BorderLayout`) elrendezést.

### 6.1. Az elrendezésmenedzserek tulajdonságai

A felhasználói interfész minden konténerének van alapértelmezés szerinti elrendezésmenedzsere (6.1. ábra), és az a benne megadott algoritmus alapján szükség szerint automatikusan elrendezi a konténer komponenseit, szabályozza a helyzetüket és méretüket. A

programozó az alapértelmezés szerinti elrendezésmenedzsert másra cserélheti. Három lehetősége van: vagy az előredefiniált osztályok közül választ (például `FlowLayout`, `GridLayout`, `BorderLayout`), vagy teljesen egyéni rendezőt készít (ez kevésbé elterjedt megoldás), vagy nem használ elrendezésmenedzsert (`=null`). minden elrendezésmenedzser az osztályában megadott algoritmus szerint dolgozik.

Az elrendezést az ablak `pack` utasítása végzi el, figyelembe véve a gyerekkomponensek előnyös méretét – felülbírálja a közvetlenül megadott méreteket. A `pack` elhagyása esetén nincs elrendezés.



6.1. ábra. Az elrendezésmenedzser automatikusan rendezi a konténer elemeit

Igényes programban egy ablak átméretezése általában maga után vonja az ablak gyerekeinek átméretezését és áthelyezését. Ennek egyéni beprogramozása óriási erőfeszítéssel jár, és sérülékenyebbé is teheti a programot. Nagyon ajánlatos tehát az API által felkínált elrendezésmenedzsereket használni!

A `java.awt` csomagban a következő elrendezésmenedzsereket definiálták:

- ◆ **FlowLayout** (sorfolytos elrendezés): A konténer komponensei sorfolytosan helyezkednek el a kitölthető területen. Ha egy elem nem fér ki a többivel egy sorba, akkor egy új sor elejére kerül. Az elemek mérete nem változik. Ez a `JPanel` és az `Applet` alapértelmezés szerinti elrendezésmenedzsere.
- ◆ **GridLayout** (rácsos elrendezés): A konténerben egy megadott sor- és oszlopszámu rácsot definiálunk. A rácselemei egyenlő méretű téglalapok, és minden komponens egy-egy ilyen téglalapot foglal el.
- ◆ **BorderLayout** (határ menti elrendezés): A konténer komponensei a 4 + 1 égtáj szerint helyezkedhetnek el (észak, dél, kelet, nyugat és közép). Ily módon a konténerben legfeljebb 5 komponens látható – az ugyanarra az égtájra helyezett komponensek takarják egymást. A középső komponens mindenkitől a maradék helyet. Ez a `Window` és leszármazottainak (`JFrame`, `JDialog`), valamint a `JFrame` tartalompaneljének (content pane) alapértelmezés szerinti elrendezésmenedzsere.

- ◆ **CardLayout** (kártyás elrendezés): A komponensek egymás felett helyezkednek el, s kitöltik a teljes teret. Mindig csak a legfelső komponens látszik. Programból megadható, hogy éppen melyik komponens legyen legfelül. Ily módon megtehetjük, hogy egy felületet a „kulisszák mögött” készítünk el – ha az hosszú időbe telik.
- ◆ **GridBagLayout** (rácsos-csomagos elrendezés): A rácsos elrendezés továbbfejlesztett változata. A konténeren egy megadott sor- és oszlopszámú rácsot definiálunk. A komponensek a rácson kijelölt valahány sor és oszlop alkotta téglalapot foglalnak le.

A Container osztály definiál egy elrendezésmenedzsert (`LayoutManager layoutMgr`). A konténer létrehozásakor a rendszer egy alapértelmezés szerinti elrendezésmenedzsert rendel a konténerhez, és a konténerbe tett komponenseket automatikusan a menedzser felügyelete alá vonja. A programozó a `setLayout(LayoutManager)` üzenettel felülbírálhatja a menedzsert, és a `setLayout(null)` hívásával meg is szüntetheti. Ez utóbbi esetben nincs automatikus elrendezés, ezért minden elemnek meg kell adni a helyzetét és a méretét.

Az elrendezésmenedzser megadása után a programozónak nincs semmi dolga a komponensek elrendezésével, a menedzser már automatikusan végzi feladatát.

*Megjegyzés:* Vannak még további elrendezésmenedzserek is, mint például a

- `javax.swingBoxLayout`, vagy a Borland által szállított
- `com.borland.jbcl.layout.VerticalFlowLayout` menedzser.

## LayoutManager interfész

Csomag: `java.awt` Deklaráció: `public interface LayoutManager`

Közvetlen ōs: -

Minden elrendezésmenedzser implementálja a `LayoutManager` interfészt; az a következő metódusokat tartalmazza:

- ▶ `void addLayoutComponent(String name, Component comp)`  
▶ `void removeLayoutComponent(Component comp)`  
    comp bevonása az elrendezésmenedzser felügyelete alá, és kivonása a felügyelet alól.
- ▶ `Dimension minimumLayoutSize(Container parent)`  
    Kiszámítja és visszaadja azt a legkisebb konténerméretet, amekkorán még elérnék a komponensek. A rendszer ennél kisebb területen nem hajlandó megjeleníteni a konténert.
- ▶ `Dimension preferredLayoutSize(Container parent)`  
    Megadja, hogy mekkora lenne a megadott komponensekkel feltöltött konténer optimális mérete. Ekkora lesz a konténer, ha a rendszerre bízzuk a méret meghatározását.
- ▶ `void layoutContainer(Container parent)`  
    Elvégzi az elrendezést.

Ha a programozó új elrendezésmenedzsert ír vagy megváltoztat egy már meglévőt, akkor megírja az interfész megfelelő metódusait. A könyvben nem készítünk elrendezésmenedzsert.

### A Container osztály elrendezésmenedzserrel kapcsolatos metódusai

- ▶ `Component add(Component comp)`
- ▶ `Component add(Component comp, int index)`
- ▶ `void add(Component comp, Object constraints)`
- ▶ `void add(Component comp, Object constraints, int index)`

Komponens hozzáadása a konténer végéhez vagy az `index` sorszámú pozícióba. A komponens automatikusan hozzáadódik a konténer elrendezésmenedzseréhez is. A komponensek aktuális elrendezése függ az elrendezésmenedzser osztályától, a komponensek tárolási sorrendjétől, valamint a `constraints`-ben esetlegesen megadott megszorítástól. A megadható megszorítás az aktuális elrendezésmenedzsertől függ (lásd ott).

- ▶ `LayoutManager getLayout()`  
Visszaadja a konténer aktuális elrendezésmenedzserét.
- ▶ `void setLayout(LayoutManager mgr)`  
A konténer elrendezésmenedzsere ezután `mgr` lesz.
- ▶ `void validate()`  
Szükség esetén újrarajzolja a komponenst.

## 6.2. FlowLayout – sorfolytonos elrendezés

Csomag: `java.awt` Deklaráció: `public class FlowLayout`

Közvetlen ős: `java.lang.Object`

Fontosabb implementált interfések: `LayoutManager`, `Serializable`

A `FlowLayout` elrendezésmenedzser balról jobbra haladva sorfolytonosan helyezi el a konténerbe tett komponenseket – ha egy komponens nem fér el az aktuális sorban, akkor a következő sorba kerül. Egy elem mérete az előnyös méret (`preferredSize`) alapján határozódik meg. Az elemek mérete nem változik az ablak átméretezésekor. A sorok igazodhatnak balra, jobbra vagy középre. Megadható, hogy a komponensek között mekkora legyen a pontban mért vízszintes, illetve függőleges távolság. A konténer elemei természetesen újabb konténerek is lehetnek, a maguk külön elrendezésmenedzserével.

A panel és az applet alapértelmezés szerinti elrendezésmenedzsere a `FlowLayout`.

*Megjegyzés:* A `JFrame` tartalompanelje `JPanel`-leszármazott, elrendezésmenedzsere azonban `BorderLayout`.

### Mezők

- ▶ `static int CENTER, LEFT, RIGHT`

Ezek a konstansok az igazítás megadásához szükségesek. A sorok komponensei a konténerben középre, balra, illetve jobbra igazodnak majd.

## Konstruktorok

- `FlowLayout(int align, int hgap, int vgap)`
- `FlowLayout(int align)`
- `FlowLayout()`

Sorfolytonos elrendezésmenedzser létrehozása. A paraméterek jelentése:

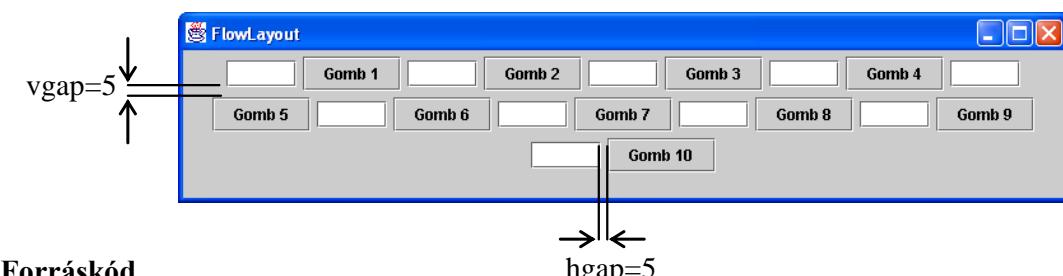
- `align`: a sorok igazítása: `LEFT`, `RIGHT` vagy `CENTER`. Alapértelmezés: `CENTER`
- `hgap`: a vízszintes köz (horizontal gap). A komponensek között és a határ mentén is van köz. Alapértelmezés: 5 pont
- `vgap`: a függőleges köz (vertical gap). A komponensek között és a határ mentén is van köz. Alapértelmezés: 5 pont

### Feladat – FlowLayoutTest

Tegyük az alkalmazás 700\*100-as méretű keretébe sorfolytonosan 10 szövegmező (`JTextField`) és nyomógomb (`JButton`) párost, a sorokat középre igazítsa! A szövegmezők 5 oszloppal helyet foglaljanak; a gombokon a "Gomb" felirat és a gomb sorszáma szerepeljen!

A futó programban interaktív módon méretezzük át a keretet, és közben figyeljük meg, hogyan változik az elemek elrendezése!

*Megjegyzés: A program 1280\*1024-es felbontású képernyőn futott, amikor ezek a képek készültek. Más felbontású képernyőn a komponensek esetleg másképp helyezkednek majd el!*



### Forráskód

```

import java.awt.*;
import javax.swing.*;

public class FlowLayoutTest extends JFrame {
 private Container cp = getContentPane();

 public FlowLayoutTest() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 setTitle("FlowLayout");
 LayoutManager lm = new FlowLayout(); //1
 cp.setLayout(lm); //2
 for (int i=1; i<=10; i++) {
 cp.add(new JTextField(5));
 cp.add(new JButton("Gomb "+i));
 }
 setSize(700,150);
 show();
 }
}

```

```
public static void main (String args[]) {
 new FlowLayoutTest();
}
```

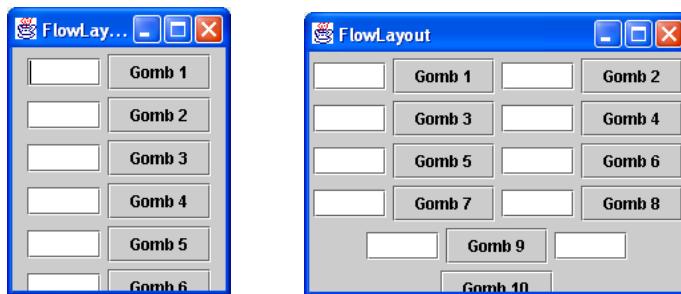
### A program elemzése

//1-ben létrehoztunk egy `FlowLayout` elrendezésmenedzsert alapértelmezés szerinti tulajdon-ságokkal (a sorok középre igazodnak, a komponensek között 5-5 pont a vízszintes és a függő-leges távolság). //2-ben ezt az elrendezésmenedzsert hozzárendeltük a `cP` tartalompanelhez.

A sorok balra igazítására //1 helyett a következő utasítást kellett volna kiadnunk:

```
LayoutManager lm = new FlowLayout(FlowLayout.LEFT);
```

Figyelje meg, hogy a Gomb 10 több helyet foglal el, mint a többi (a gomb felirata egy karakterrel hosszabb)! Az ablak futás közbeni átméretezése ilyen eredményekre vezethet például:



## 6.3. GridLayout – rácsos elrendezés

Csomag: `java.awt` Deklaráció: `public class GridLayout`

Közvetlen ős: `java.lang.Object`

Fontosabb implementált interfések: `LayoutManager`, `Serializable`

**A GridLayout elrendezésmenedzser** egy rácson helyezi el a konténer elemeit. A konténe-ren megadott sor- és oszlopszámú rácsot definiálunk. A rács cellái egyenlő méretű téglalapok, és a komponensek minden egy-egy ilyen téglalapot foglalnak el. A cellák feltöltésekor a rendszer nem enged lyukat. A rácscellában levő komponens kitölti a teljes cellát. A komponensek között megadható valamekkora – pontban mért – vízszintes, illetve függőleges állandó távolság. A konténer elemei természetesen újabb konténerek is lehetnek a maguk külön elrendezésmenedzserével.

### Konstruktörök

- ▶ `GridLayout(int rows, int cols, int hgap, int vgap)`
- ▶ `GridLayout(int rows, int cols)`
- ▶ `GridLayout()`

Rácsos elrendezésmenedzser létrehozása. A paraméterek jelentése:

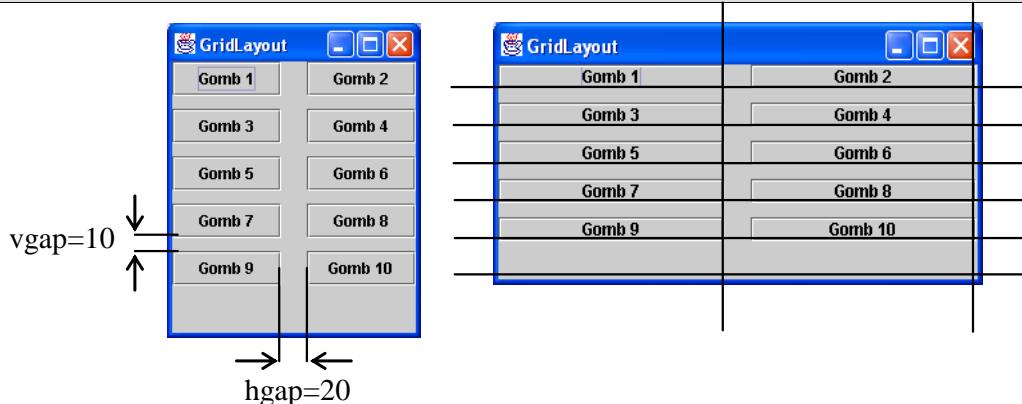
- `rows, cols`: A sorok és oszlopok száma. Ha `rows` vagy `cols` értéke 0, akkor a sorok, illetve oszlopok száma rugalmasan bővül (abban az irányban akárhány elem betehető). A két érték egyszerre nem lehet 0. Ha nem adunk meg paramétert, akkor egy egysoros bővülő rács keletkezik (`rows==1, cols==0`). Ha a konténerbe betett komponensek száma nagyobb lesz a rácszellák számánál, akkor az elrendezésmenedzser felülbírálja az oszlopszámot.
- `hgap, vgap`: vízszintes köz (horizontal gap), illetve függőleges köz (vertical gap). A határ mentén nincs köz, csak a komponensek között. Alapértelmezés: 0.

A rácsos elrendezésben a komponensek sorfolytonosan számoznak, nullától kezdődve. Ha tehát egy komponenst nem utolsó elemként akarunk betenni a konténerbe, akkor eszerint kell kiszámolnunk az elem indexét.

**Megjegyzés:** A rácsos elrendezésmenedzser nem engedi meg, hogy a rácsra kihagyásokkal tegyünk fel komponenseket. A komponenseket csak sorfolytonosan lehet a konténerbe tenni!

### Feladat – GridLayoutTest

Tegyük az alkalmazás keretére képzeletben egy 6\*2-es rácsot, és tegyük az első 10 rácshelyre egy-egy sorszámozott nyomógombot! A komponensek közötti vízszintes távolság legyen 20 pont, a függőleges távolság pedig legyen 10 pont! A futó programban interaktív módon méretezzük át a keretet!



### Forráskód

```
import java.awt.*;
import javax.swing.*;

public class GridLayoutTest extends JFrame {
 public GridLayoutTest() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 setTitle("GridLayout");
 getContentPane().setLayout(new GridLayout(6,2,20,10)); //1
 for (int i=1; i<=10; i++) {
 getContentPane().add(new JButton("Gomb "+i));
 }
 pack();
 show();
 }
}
```

```
public static void main (String args[]) {
 new GridLayoutTest();
}
```

### A program elemzése

//1-ben létrehozunk egy GridLayout elrendezésmenedzsert. A rács sorainak száma 6, oszlopainak száma 2; a vízszintes köz 20 pont, a függőleges köz pedig 10 pont. A menedzsert azon nyomban átadjuk a tartalompanelnek. Ezután a konténerbe teszünk 10 sorszámozott gombot. A pack() eljárás a konténer getPreferredSize() metódusa alapján beállítja a keret méretét.

Figyelje meg, hogy a keret alsó része üres! Ennek így is kellett lennie, mert két cella kitöltetlen maradt.

## 6.4. BorderLayout – határ menti elrendezés

Csomag: java.awt Deklaráció: public class BorderLayout

Közvetlen ős: java.lang.Object

Fontosabb implementált interfések: LayoutManager, Serializable

**A BorderLayout elrendezésmenedzser** a konténer határai mentén rendezi el a komponenseket, 4 + 1 égtáj szerint. A konténer elemeit öt helyre tehetjük:

- **North:** Észak, a konténer felső része. A komponens függőleges irányban akkora helyet foglal le, amekkorára szüksége van; West-East (bal-jobb) irányban kitölти a konténert.
- **South:** Dél, a konténer alsó része. A komponens függőleges irányban akkora helyet foglal le, amekkorára szüksége van; West-East irányban kitölти a konténert.
- **West:** Nyugat, a konténer bal oldala. A komponens West-East irányban akkora helyet foglal le, amekkorára szüksége van; North-South (fent-lent) irányban kitölти a felső és az alsó elem közötti helyet.
- **East:** Kelet, a konténer jobb oldala. A komponens West-East irányban akkora helyet foglal le, amekkorára szüksége van; North-South irányban kitölти a felső és az alsó elem közötti helyet.
- **Center:** Közép, a konténer középső része. Az elem minden irányban kitölти a négy másik elem által hagyott helyet.

Ha egy égtájra több elemet teszünk, akkor azok takarják egymást. A középső elem kitölти a teljes teret.

A Window-nak és leszármazottainak (JFrame, JDialog), valamint a JFrame tartalompaneljének (content pane) alapértelmezés szerint BorderLayout az elrendezésmenedzsere.

## Konstruktörök

- ▶ BorderLayout (int hgap, int vgap)
- ▶ BorderLayout ()

Határmenti elrendezésmenedzser létrehozása. A paraméterek jelentése:

- hgap, vgap: vízszintes, illetve függőleges köz; a határ mentén nincs köz, csak a komponensek között. Alapértelmezés: 0.

## A Container osztály ide vonatkozó hozzáadó metódusai

- ▶ void add(Component comp, constraints object)
- ▶ Component add(Component comp)

A constraints (megszorítás) a következő szövegek egyike lehet: "North", "South", "West", "East" vagy "Center". Alapértelmezés: "Center". A komponens a megfelelő égtájra kerül.

### Feladat – BorderLayoutTest

Tegyük a 400\*200-as méretű keretbe 5 nyomógombot a képen látható módon! A vízszintes köz 2, függőleges köz 1 legyen! A futó programban interaktív módon mértezzük át a keretet!



## Forráskód

```

import java.awt.*;
import javax.swing.*;

public class BorderLayoutTest extends JFrame {
 Container cp = getContentPane();

 public BorderLayoutTest() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 setTitle("BorderLayout");
 cp.setLayout(new BorderLayout(2,1)); //1
 cp.add(new JButton("North - Észak - Felső"), "North");
 cp.add(new JButton("South - Dél - Alsó"), "South");
 cp.add(new JButton("West - Nyugat - Bal"), "West");
 cp.add(new JButton("East - Kelet - Jobb"), "East");
 cp.add(new JButton("Center - Középső"), "Center");
 }
}

```

```

 setSize(400,200);
 show();
 }

 public static void main (String args[]) {
 new BorderLayoutTest();
 }
}

```

### A program elemzése

//1-ben létrehozunk egy `BorderLayout` elrendezésmenedzsert; a vízszintes köz 2 pont, a függőleges köz 1 pont lesz. Ezután a konténerbe teszünk 5 gombot, mindegyiket más égtájra – a gombok felirata mutatja, hogy melyikre.

## 6.5. JPanel, az összefogó konténer

Csomag: `javax.swing` Deklaráció: `public class JPanel`

Közvetlen ős: `javax.swing.JComponent`

Fontosabb implementált interfészek: `ImageObserver`, `Serializable`

A `JPanel` egy konténer, s minden összefogja a benne levő elemeket, ne engedje őket széthullani. A panel lehet láthatatlan, de színnel és szegéllyel láthatóvá is lehet tenni. A panel alapértelmezésben dupla pufferelésű, ami nagyban meggyorsítja a rajzolást (a háttérben is rendezi a képet). A panelnek saját elrendezésmenedzsere van, alapértelmezésben a `FlowLayout`.

### Konstruktorkok

- `JPanel(LayoutManager layout)`
- `JPanel()`

Panel létrehozása a megadott elrendezésmenedzserrel. Alapértelmezés: `FlowLayout`.

A panelnek nincs számunkra fontos metódusa.

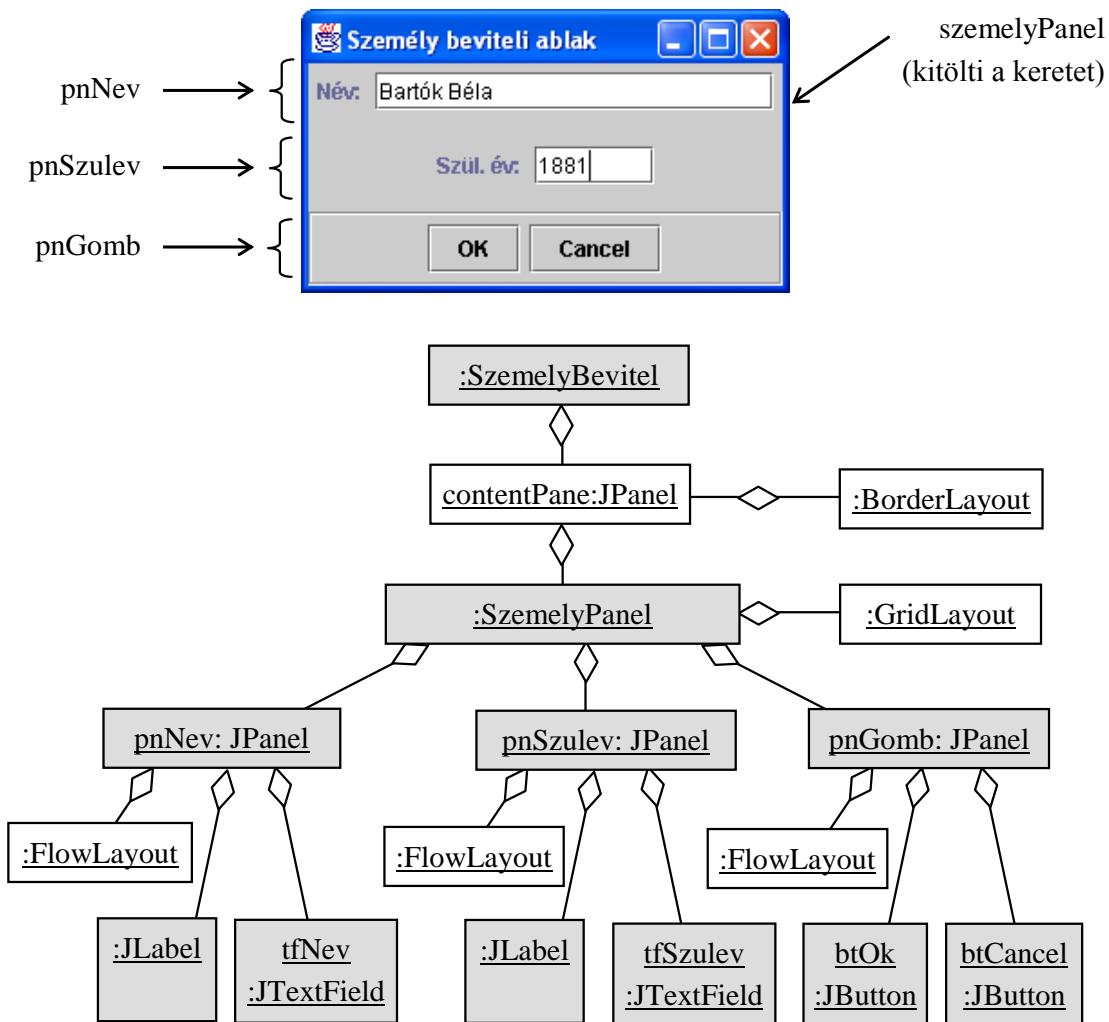
### Feladat – Személy bevitele

Készítsük el a 6.2. ábrán látható keretet! A Név beviti mező 20 oszlopos, a Születési év mező 6 oszlopos legyen! A két címkézett beviti mező és a gombok egy-egy sort alkossanak; a sorok egymás alatt jelenjenek meg!

### A program terve

A feladat megoldására elkészítünk egy külön `SzemelyPanel` osztályt, s annak egy példányát tesszük bele a `contentPane`-be. A `SzemelyPanel`-t függőlegesen három egyenlő részre osztjuk: egy 3\*1-es rácsot készítünk, s a rácszellákba újabb panelek kerülnek: `pnNev`,

pnSzulev és pnGomb összefogják majd a komponenspárokat. Mindegyik kis panel sorfolytonos elrendezésű lesz, s a bennük levő komponensek alapértelmezésben középre igazodnak majd. A keret felosztása és a tulajdonosi hierarchia a 6.2. ábrán látható.



6.2. ábra. A keret felosztása és tulajdonosi hierarchiája

*Megjegyzések:*

- A tulajdonosi hierarchia diagramján csak a magyarázat kedvéért tüntettük fel az elrendezésmenedzsereket. Létezésük természetes, az ablak tervén nem szokás őket szerepeltetni.
- A 3\*1-es rács (grid) helyett létrehozhatnánk 3\*2-es rácsot is, panel nélkül. Csakhogy azon a címke túl sok helyet foglalna el a beviteli mező rovására, és az ablak nem is lenne szép.

**Forráskód**

```

import java.awt.*;
import javax.swing.*;

class SzemelyPanel extends JPanel {
 JTextField tfNev; //1
 JTextField tfSzulev;
 JButton btOk, btCancel;

 public SzemelyPanel() {
 setLayout(new GridLayout(3,1)); //2

 JPanel pnNev = new JPanel(); //3
 pnNev.add(new JLabel("Név: "));
 pnNev.add(tfNev = new JTextField(20));
 add(pnNev);

 JPanel pnSzulev = new JPanel();
 pnSzulev.add(new JLabel("Szül. év: "));
 pnSzulev.add(tfSzulev = new JTextField(6));
 add(pnSzulev);

 JPanel pnGomb = new JPanel();
 pnGomb.setBorder(BorderFactory.createEtchedBorder());
 pnGomb.add(btOk = new JButton("OK"));
 pnGomb.add(btCancel = new JButton("Cancel"));
 add(pnGomb);
 }
}

public class SzemelyBevitel extends JFrame {
 public SzemelyBevitel() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 setTitle("Személy beviteli ablak");
 getContentPane().add(new SzemelyPanel());
 pack();
 show();
 }
 public static void main (String args[]) {
 new SzemelyBevitel();
 }
}

```

**A program elemzése**

//1-ben osztályszinten deklaráljuk a fontosabb komponenseket. Azokat a komponenseket szokás itt deklarálni, amelyekre az osztály különböző metódusaiban hivatkozni szeretnénk. Mivel ennek a programnak a komponensek létrehozásán túl nincs más feladata, azért ezek a deklarációk tulajdonképpen feleslegesek. Egy program általában véve azért hoz létre beviteli mezőt vagy gombot, hogy állapotukat szükség szerint lekérdezze vagy beállítsa. A címkeket rendszerint nem deklaráljuk, mert őket nem szokás megszólítani.

//2-ben 3\*1-es rácsra állítjuk be a panel elrendezésmenedzsérét, majd //3-tól kezdve sorban elkészítjük a három panelt. Először a pnNev panelt hozzuk létre, majd betesszük a névhez tartozó címkét és a szövegmezőt. A szövegmező referenciáját meg is jegyezzük, a címkéét nem. Végül a pnNev-et hozzáadjuk a személypanelhez. A pnNev-et természetesen gyerekeinek hozzáadása előtt is betehettük volna a keretbe:

```
 JPanel pnNev = new JPanel(); //3
 add(pnNev);
 pnNev.add(new JLabel("Név: "));
 pnNev.add(tfNev = new JTextField(20));
```

**Ügyelünk kell arra, hogy a komponensek tulajdonosi hierarchiája helyesen legyen felépítve!** Csak az a komponens látható a képernyőn és vehet részt az események vérkerin-gésében, amely része a tulajdonosi hierarchiának!

## Tesztkérdések

6.1. Jelölje be az összes igaz állítást!

- a) A LayoutManager interfész, nem lehet belőle példányt létrehozni.
- b) Egy konténer alapértelmezés szerinti elrendezésmenedzsere más elrendezésmenedzsere cserélhető.
- c) Az elrendezésmenedzserek az a feladata, hogy valamilyen logika szerint elrendezze a konténer gyerekeit.
- d) Egy konténernek több elrendezésmenedzsere is lehet.

6.2. Jelölje be az összes igaz állítást!

- a) Határ menti elrendezésben a beszúrandó komponens mindenkorábban a konténer következő határvilágkörére kerül: először felülre, aztán jobbra, alulra, balra, középre, majd újra felülre stb.
- b) Sorfolytonos elrendezésben a beszúrandó komponens a paraméterben megadott sorba kerül, következő elemként.
- c) Rácsos elrendezés esetében a komponensek egyenlő területű helyet foglalnak el.
- d) Kártyás elrendezésben egyszerre egy kis rész látható mindegyik komponensből.

6.3. Jelölje be az összes igaz állítást!

- a) Alapértelmezésben minden konténernek FlowLayout az elrendezésmenedzsere.
- b) A BorderLayout menedzsere segítségével egy konténerbe be lehet tenni tíz komponenset úgy, hogy minden komponens látható legyen.
- c) A FlowLayout menedzsere akárhány komponensem képes elhelyezni a konténerben, ha van megfelelő hely az elemeknek.
- d) A GridLayout menedzsere által felügyelt elemek minden egyforma méretű helyet foglalnak le.

6.4. Melyek a helyes paraméterezésű konstruktörök? Jelölje be az összes jó választ!

- a) FlowLayout(int align, int hgap, int vgap)
- b) GridLayout(int rows, int cols, Color color)

- c) `BorderLayout()`  
d) `BorderLayout(int south, int north, int west, int east)`
- 6.5. Jelölje be az összes igaz állítást!
- A `JFrame`-nek `FlowLayout` az alapértelmezés szerinti elrendezésmenedzsere.
  - A `JPanel`-nek `BorderLayout` az alapértelmezés szerinti elrendezésmenedzsere.
  - A `JDialog`-nak `BorderLayout` az alapértelmezés szerinti elrendezésmenedzsere.
  - A `JFrame` tartalompaneljének (content pane) `FlowLayout` az alapértelmezés szerinti elrendezésmenedzsere.
- 6.6. Jelölje be az összes igaz állítást!
- `FlowLayout` elrendezésben az `add(Component)` az előző komponens alá teszi a következő komponenst.
  - `GridLayout` elrendezésben az `add(Component)` a következő sorba teszi a soron következő komponenst.
  - `BorderLayout` elrendezésben az `add(Component, "East")` a szegély bal oldalára teszi a soron következő komponenst.
  - `BorderLayout` elrendezésben az `add(Component)` középre teszi a soron következő komponenst.

## Feladatok

- 6.1. (A) Tegyen az alkalmazás keretébe 6 nyomógombot "A",..., "F" felirattal. Álljon minden egyik nyomógomb után egy "Elintézve" feliratú jelölőmező. A gombok színe narancssárga, a jelölőmezőké `SystemColor.control` legyen! A vezérlőket sorfolytonosan helyezze el, a sorokat balra igazítva! Az ablak induláskor éppen akkora legyen, hogy a komponensek éppen elférjenek benne! Az alkalmazás címe: "Gombok, jelölőnégyzetek". (*GombokJelolok.java*)



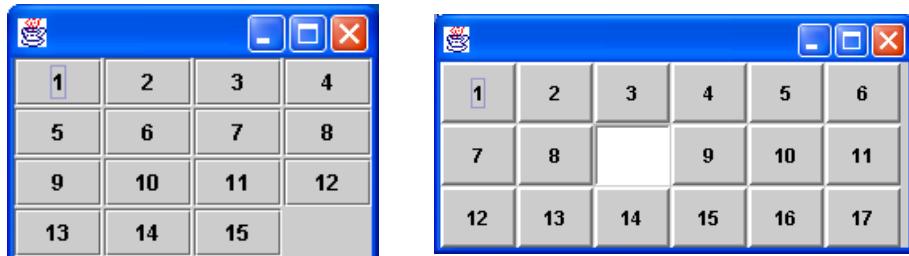
- 6.2. (A) Tegyen a keretbe egy szövegterületet! A keret alján legyen kétsoros gombsor az angol ABC betűivel! (*AlulGombok.java*)



- 6.3. (A) Tegyen a keretre két egyenlő nagyságú szövegterületet! (*KetSzoveg.java*)  
 a) A szövegterületek egymás alatt legyenek!  
 b) A szövegterületek egymás mellett legyenek!
- 6.4. (A) Kérjen be konzolról négy szöveget! Jelenítsen meg egy keretet a 200,100 pontban, és tegyen a keret négy szélére (felülre, alulra, balra, jobbra) egy-egy nyomógombot, s azokra az előzőleg konzolról bekért szövegek kerüljenek feliratként! (*NegyGomb.java*)



- 6.5. Készítse el a Tili-toli játék felületét!  
 a) (A) A játék 4\*4-es legyen, s a lyuk kezdetben a rács utolsó helyére jusson! (*TiliToli\_Felulet1.java*)  
 b) (B) A játék 3\*6-os legyen, s a lyuk kezdetben a 2. sor 3. helyére jusson! Egy-egy gomb mérete most 50\*50 pont legyen! A keretnek lehessen paraméterül átadni a méretet és a lyuk pozícióját! (*TiliToli\_Felulet2.java*)



- 6.6. (B) Tegyen a keretbe egymás mellé két egyforma komponenst: felül egy felirat legyen (Bal, illetve Jobb), a többi rész pedig szövegterület. (*BalJobb.java*)



6.7. (C) Készítse el a következő felületet (*IrogepFelulet.java*):



## 7. Eseményvezérelt programozás

---

A fejezet pontjai:

1. Mintaprogram
  2. Eseményosztályok
  3. Alacsony és magas szintű események
  4. Eseménydelegációs modell
  5. A felhasználói felület tervezése
  6. Eseményadapterek
- 

A grafikus felhasználói interfész (felület) eseményvezérelt, vagyis a programot a futás közben keletkezett események vezérlik. Események keletkezhetnek a programon kívül, például egy billentyű leütésével vagy az egér elmozdításával, és a program maga is kelthet eseményeket. Az esemény egy objektum, s mindenkor egy forrásobjektumhoz kapcsolódik: az esemény azon az objektumon keletkezik. Ez a keletkezett esemény a Java eseménydelegációs modellje szerint eljut a megfelelő figyelőobjektumokhoz, s azok felgyózzák a hozzájuk érkező eseményeket. Vannak alacsony szintű, elemi események, és vannak magas szintű, összetett, logikai események. A Swing használja az AWT-eseménymodellt és a `java.awt.event` csomag eseményosztályait; a `javax.swing.event` csomag újabb eseménytípusokat is definiál.

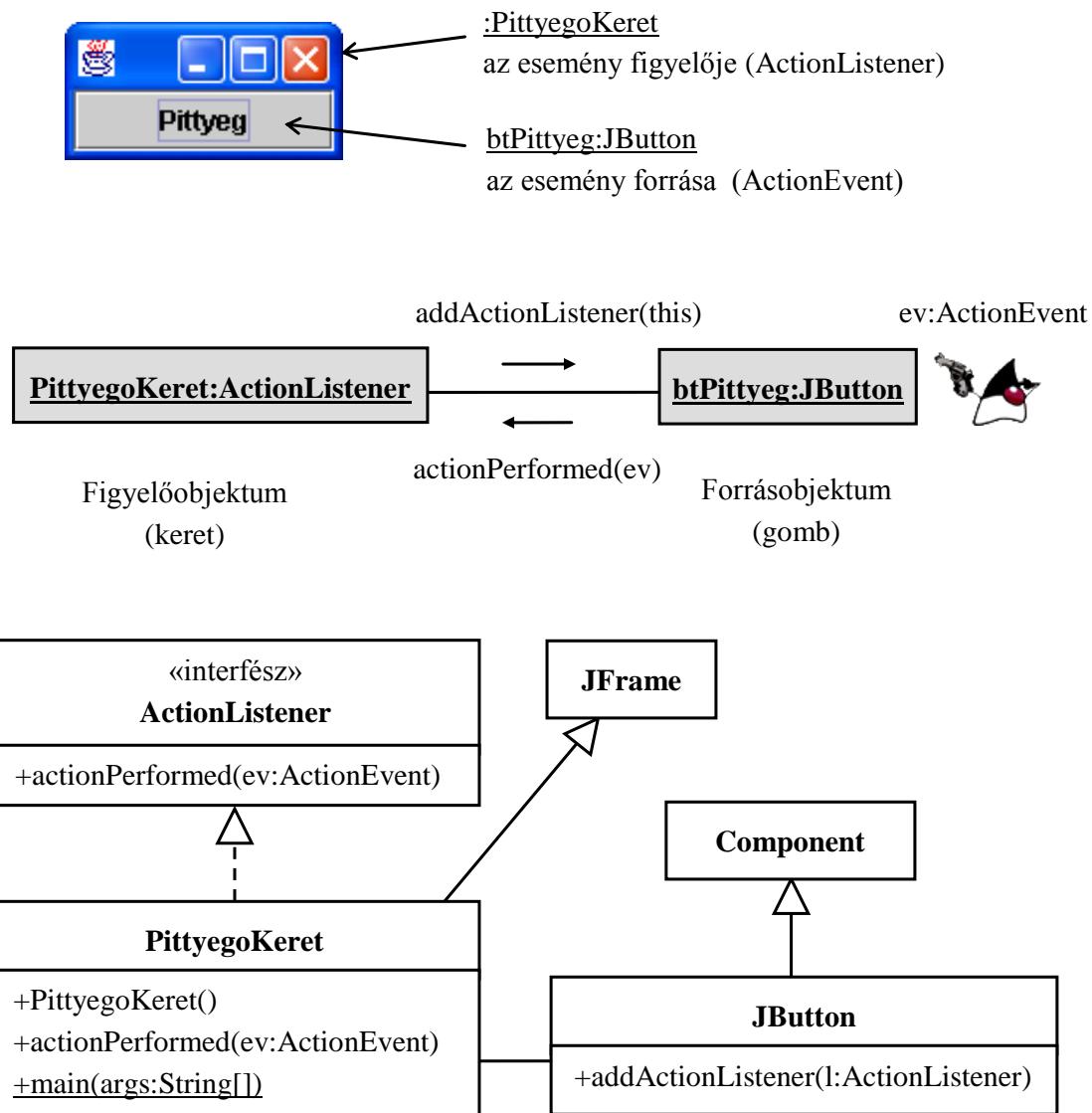
A fejezet az eseményvezérelt program működésének alapelveit és a könyvben használt eseményeket ismerteti. A különféle Swing-komponensek magas szintű eseményeit a 8. fejezet, az alacsony szintű eseményeket a 10. fejezet tárgyalja részletesebben.

### 7.1. Mintaprogram

**Feladat – Pittyegő keret**

Tegyük a keretbe egy "Pittyeg" feliratú gombot. Ha lenyomják a gombot, akkor a számítógép pittyegjen egyet!

Tisztában kell majd lennünk néhány angol szó jelentésével: **event**: esemény; **action**: akció; **source**: forrás; **listener**: figyelő; **perform**: teljesít, végrehajt.



7.1. ábra. A Pittyegő képe, együttműködési és osztálydiagramja

A 7.1. ábra mutatja a keret és a gomb képét, valamint az alkalmazás együttműködési diagramját és osztálydiagramját. A keret osztálya **PittyegoKeret**, a **JFrame** osztály leszármazottja. A **btPittyeg** nyomógomb **JButton** osztályú. Amikor a gombot lenyomják, akkor egy akció- esemény (`ev:ActionEvent`) keletkezik – az esemény forrása a **btPittyeg**. A gombon akkor is keletkezhet esemény, ha a gombot senki sem figyeli – ez a gombba be van programozva. A keletkezett eseményt a forrásobjektumot figyelő (hallgató) objektum feldolgozhatja. A **btPittyeg** forrásobjektumot itt a kerettel figyeltetjük. A keretet a programozónak kell a forrásobjektumhoz „tapasztani” a forrásobjektum `addActionListener` metódusával. Amikor

a nyomógombot lenyomják, a kerethez automatikusan eljut az esemény: meghívódik a keret `actionPerformed` metódusa; ennek az esemény (`ev:ActionEvent`) a paramétere. A figyelő osztályának implementálnia kell az `ActionListener` interfészt; abban ez az `actionPerformed` az egyetlen metódus. Az esemény a forrásobjektumból eljut tehát a figyezőhöz, s az majd lekezeli (kezeli). A keret és a gomb közötti kapcsolat kétirányú.

### Forráskód

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*; //1

public class PittyegoKeret extends JFrame
 implements ActionListener { //2
 private JButton btPittyeg = new JButton("Pittyeg");

 public PittyegoKeret() {
 setDefaultCloseOperation(EXIT_ON_CLOSE)
 getContentPane().add(btPittyeg); //3
 btPittyeg.addActionListener(this); //4
 pack();
 show();
 }

 public void actionPerformed (ActionEvent ev) { //5
 Toolkit.getDefaultToolkit().beep(); //6
 }

 public static void main (String args[]) {
 new PittyegoKeret();
 }
}

```

### A program elemzése

Az `ActionEvent` eseménnyel kapcsolatos osztályokat és interfészket a `java.awt.event` csomag definiálja (//1). Ahhoz, hogy a keret figyelje és kezelje a nyomógombon keletkezett `ActionEvent` eseményt, a következőket tesszük:

- ◆ A nyomógombot felvesszük a tulajdonosi hierarchiára, mert különben nem látszana és nem is keletkezne rajta esemény (//3).
- ◆ A keret osztályában implementáljuk az `ActionListener` interfészt (//2), és ki is fejtjük annak egyetlen, `actionPerformed (ActionEvent ev)` eseménykezelő metódusát (//5). A kezelés abból áll, hogy a számítógéppel egy rövid hangot hallatunk, vagyis megpittyegtetjük: a `java.awt.Toolkit` osztályban definiált `beep()` metódus egy rövid hangot ad (//6). A beep csak akkor szól, ha van a gépben hangkártya!
- ◆ A keretet felfűzzük a `btPittyeg` forrásobjektum figyelőláncára (//4). Az `addActionListener` metódust a `JButton` osztály deklarálja.

*Megjegyzések:*

- Egy forrásobjektumnak több figyelőobjektuma is lehet. Az `addActionListener` metódussal egymás után több figyelőobjektum is a forrásobjektumhoz „tapasztható”.
- A különböző eseményfajták figyeléséhez a programnak más-más figyelőinterfészt kell implementálnia. A `KeyEvent` eseményt például a `KeyListener` figyeli.
- Hiába tesszük figyelésre alkalmassá a keretet, ha nem füzzük fel a forrás figyelőláncára!

## 7.2. Eseményosztályok

Az **esemény** a vele összefüggő információkat (esemény forrása, típusa, időpontja...) magába foglaló **objektum**. Mindig valamilyen **forrásobjektumon** (általában komponensen) keletkezik; a forrásobjektum az esemény egy tulajdonsága.

Az alacsony szintű (például billentyű- és egér-) események az operációs rendszer szintjén keletkeznek. Az operációs rendszer az eseményeket ún. **eseménysorba** (event queue) állítja. Az alkalmazás innen veszi ki a maga eseményeit, és továbbítja őket komponenseinek. Az eseményt először a **forrásobjektum kapja meg** feldolgozásra, s **az adja tovább a figyelőinek**. A forrásobjektumhoz tehát nem a „légből” érkezik az esemény, az alkalmazás szempontjából azonban mégis ott keletkezik. Ezért szokás azt mondani, hogy az esemény a **forrásobjektumon keletkezik, az eseményt a forrásobjektum kelti**.

Az események minden sorban, egymás után keletkeznek, soha nem egyszerre.

Felhasználó által keltett **esemény csak akkor keletkezhet egy komponensen, ha az eleme az alkalmazás komponenshierarchiájának és látható is**.

Az eseményosztályok öröklési hierarchiáját a 7.2. ábra mutatja. Az AWT-eseménymodellt még az AWT-ben dolgozták ki; később azután a Swingben kibővítették: további eseménytípusokat definiáltak. Az AWT-eseményeknek a `java.util.EventObject` a közös ősük – és sok-Swing eseménynek is. Az AWT-események a `java.awt.event` csomagba kerültek, a Swing-események a `javax.swing.event` csomagba. Az AWT-eseményeknek közös ősük a `java.awt.AWTEvent` absztrakt osztály.

Különbséget teszünk alacsony szintű és magas szintű események között. Az alacsony szintű események a `ComponentEvent` utódai, s forrásuk csak komponens lehet. minden más esemény magas szintű, logikai esemény, s nem szükségképpen komponens a forrása.

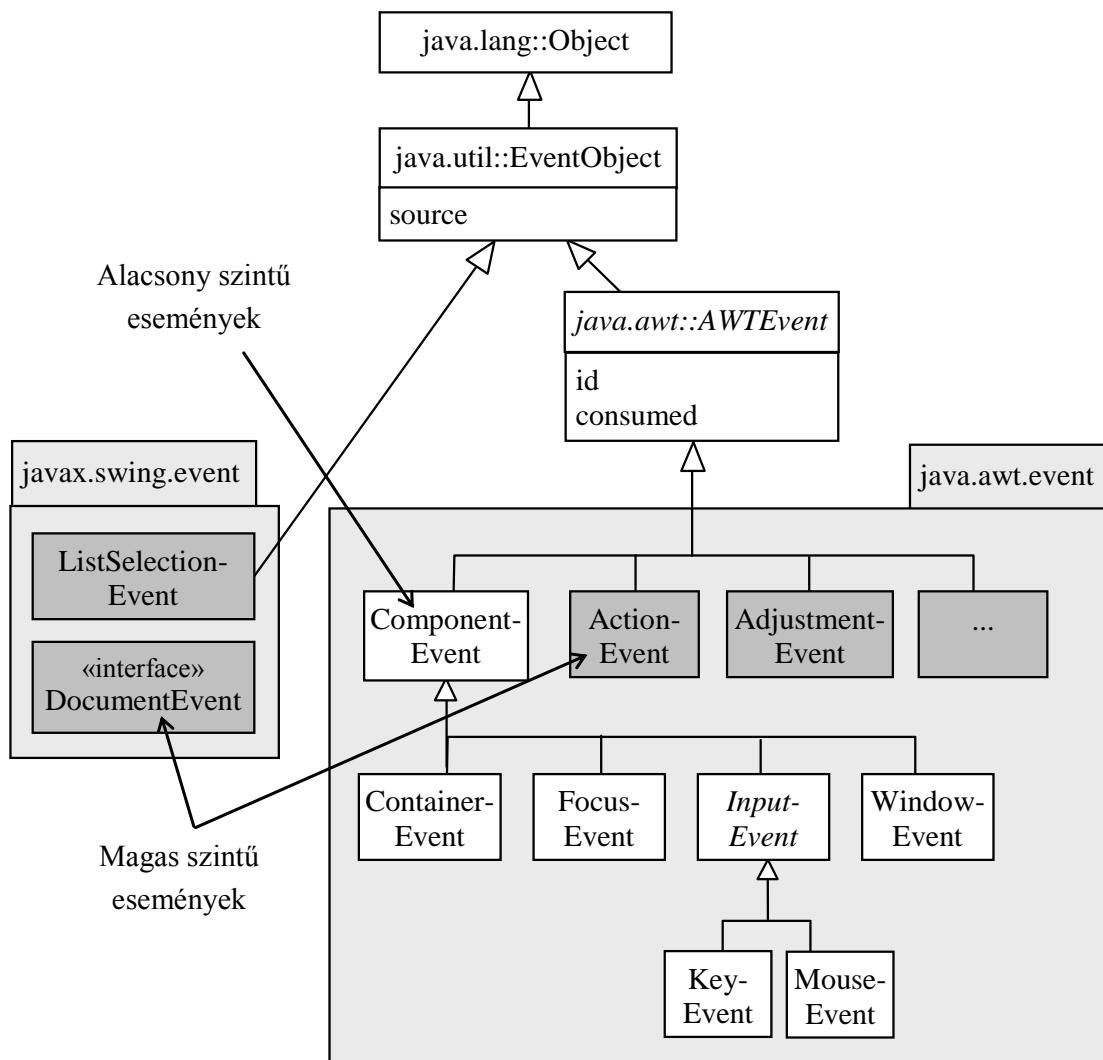
Egy eseményosztály deklarálja az eseményre jellemző adatokat. Lássuk most az `EventObject` és az `AWTEvent` osztály fontosabb adatait és metódusait!

## EventObject osztály

Az események többségének közös őse. Mindössze az esemény forrásobjektumát deklarálja.

### Mező, metódus

- ▶ `protected Object source`  
Forrásobjektum, az esemény keletkezésének helye.
- ▶ `Object getSource()`  
Visszaadja a forrásobjektumot.



7.2. ábra. Az eseményosztályok öröklési hierarchiája

## AWTEvent osztály

Az **AWTEvent** absztrakt eseményosztály, minden AWT-esemény őse. Egy-egy AWT-eseményosztály többféle eseményt testesíthet meg (egy `MouseEvent` lehet például `MOUSE_CLICKED`, `MOUSE_PRESSED` stb.); az esemény azonosítóját (`id`) az `AWTEvent` deklarálja. Az `AWTEvent` leszármazottjai statikus adatokkal adják meg a lehetséges azonosítókat (például a `MOUSE_CLICKED` konstanst a `MouseEvent` deklarálja).

### Mezők

► `protected int id`

Az esemény azonosítója. Az `id` lehetséges értékeit a megfelelő leszármazott eseményosztály deklarálja, `int` típusú osztályadatokként. Például:

- A `KeyEvent` esemény lehetséges azonosítói:

`KEY_TYPED=400, KEY_PRESSED=401, KEY_RELEASED=402`

- A `MouseEvent` esemény lehetséges azonosítói:

`MOUSE_CLICKED=500, MOUSE_PRESSED=501` stb.

- Az `ActionEvent` esemény egyetlen lehetséges azonosítója:

`ACTION_PERFORMED=1001`

► `protected boolean consumed`

Azt mutatja, hogy az esemény már elpusztult-e (érvénytelen) vagy még él. Ha `false` az értéke, akkor az esemény még él; ha `true`, akkor már nem lehet feldolgozni. Az esemény elpusztításával meg lehet akadályozni, hogy az eseményt a többi figyelője is feldolgozza. Alapértelmezés: `false`

► `static final int RESERVED_ID_MAX`

Az API eddig az értéig foglalja az azonosító lehetséges értékeit. A programozó a maga eseményosztályaiban csak ennél nagyobb értékeket adhat meg.

► `static final long FOCUS_EVENT_MASK = ...;`

► `static final long KEY_EVENT_MASK = ...;`

► `...`

Eseménymaszkok különböző eseménycsoportok megadásához, azonosításához.

### Metódusok

► `int getID()`

Visszaadja az esemény azonosítóját.

► `protected void consume()`

► `protected boolean isConsumed()`

Az előbbi elpusztítja az eseményt, az utóbbi megmondja, hogy az esemény elpusztult-e.

Példaként a 7.3. ábra két konkrét eseményobjektumot mutat be:

- ◆ a bal oldali esemény osztálya `ActionEvent`. Az eseményt a `btPittyeg:JButton` objektum keltezte (ő a forrás); az esemény azonosítója `ActionEvent.ACTION_PERFORMED`; az esemény még él (`consumed==false`); az akcióparancs szövege "Ok".
- ◆ a jobb oldali esemény osztálya `MouseEvent`. Úgy keletkezett, hogy a `btPittyeg` nyomógombon egyszer kattintottak az egérrel. A forrásobjektum a `btPittyeg`; az egéresemény azonosítója `MouseEvent.MOUSE_CLICKED`; az esemény érvényes (`consumed==false`); az egér a gomb (20,10) relatív koordinátájú pontján állt az esemény történtekor; a kattintások száma egy volt (`clickCount==1`).

Látható, hogy az első három adat (source, id és consume) minden eseményben megvan (ezeket az `EventObject` és az `AWTEvent` osztály deklarálja), a többi adat már más: az egéresemény az egér pozíóját (`x, y`) és a kattintások számát (`clickCount`) jegyzi meg, az akcioesemény az akcióparancsot (`actionCommand`).

| <u>:ActionEvent</u>                                                                                                                | <u>:MouseEvent</u>                                                                                                                      |
|------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| <pre>source = btPittyeg  {:JButton} id = ActionEvent.ACTION_PERFORMED consumed = false</pre> <hr/> <pre>actionCommand = "Ok"</pre> | <pre>source = btPittyeg  {:JButton} id = MouseEvent.MOUSE_CLICKED consumed = false</pre> <hr/> <pre>x = 20, y = 10 clickCount = 1</pre> |

7.3. ábra. Egy egéresemény- és egy akcioesemény-objektum

### 7.3. Alacsony és magas szintű események

Az **alacsony szintű esemény** az operációs rendszer szintjén keletkező elemi esemény; forrása csak komponens lehet (a forrás osztálya a `Component` leszármazottja). minden alacsony szintű esemény AWT-esemény, éspedig a `ComponentEvent` utódja:

`Object-EventObject-AWTEvent-ComponentEvent-...`

Az alacsony szintű események a következők:

- **ComponentEvent:** Komponensesemény. Lehetséges forrása: bármely Component, illetve JComponent. Azonosítói:  
COMPONENT\_MOVED: Megváltozott a komponens helyzete.  
COMPONENT\_RESIZED: Megváltozott a komponens mérete.  
COMPONENT\_SHOWN: A komponens előtérbe került.  
COMPONENT\_HIDDEN: A komponens eltűnt.
- **ContainerEvent:** Konténeresemény. Lehetséges forrása: bármely Container. Azonosítói:  
COMPONENT\_ADDED: Egy komponenst hozzáadtak a konténerhez.  
COMPONENT\_REMOVED: Egy komponenst kivettek a konténerből.
- **FocusEvent:** Fókuszesemény. Lehetséges forrása: bármely Component. Azonosítói:  
FOCUS\_GAINED: A komponens fókuszba került.  
FOCUS\_LOST: A komponens elvesztette a fókuszt.
- **WindowEvent:** Ablakesemény. Lehetséges forrása: bármely Window. Azonosítói:  
WINDOW\_OPENED: Az ablakot kinyitották.  
WINDOW\_CLOSING: Az ablakot be akarják csukni.  
WINDOW\_CLOSED: Az ablakot becsukták.  
WINDOW\_ICONIFIED: Az ablakot ikonná változtatták.  
WINDOW\_DEICONIFIED: Az ablak ikonállapotból normál állapotba került.  
WINDOW\_ACTIVATED: Az ablak aktív lett.  
WINDOW\_DEACTIVATED: Az ablak inaktív lett.
- **KeyEvent:** Billentyűzetesemény. Lehetséges forrása: bármely Component, ha az éppen fókuszban van. Azonosítói:  
KEY\_TYPED: Begépeltek egy karaktert (lenyomták és felengedték a billentyűt).  
KEY\_PRESSED: Lenyomtak egy billentyűt.  
KEY\_RELEASED: Felengedtek egy billentyűt.
- **MouseEvent:** Egéresemény. Lehetséges forrása: bármely Component, ha felette van az egérkursor. Azonosítói:  
MOUSE\_CLICKED: Lenyomták az egyik egérgombot, majd felengedték.  
MOUSE\_PRESSED: Lenyomták az egyik egérgombot.  
MOUSE\_RELEASED: Felengedték az egyik egérgombot.  
MOUSE\_MOVED: Elmozdult az egér (megváltozott a koordinátája).  
MOUSE\_ENTERED: Az egér a komponens fölé került.  
MOUSE\_EXITED: Az egér elhagyta a komponens területét.  
MOUSE\_DRAGGED: Elmozdították az egeret, s közben lenyomva tartották a gombját.

A program futása során elköpesztően sok alacsony szintű esemény keletkezhet. Amikor mozgatjuk az egeret, akkor folyamatosan, egészen pici időközönként keletkezik egy-egy MOUSE\_MOVED esemény; a forrása minden nyomás-felengedésre három esemény keletkezik: MOUSE\_PRESSED, MOUSE\_RELEASED és MOUSE\_CLICKED. Ha az egeret nem engedjük fel mozgatás közben, akkor a rendszer folyamatosan ontja a MOUSE\_DRAGGED eseményeket. Az is elképzelhető persze, hogy eközben billentyűket is nyomogatunk, ekkor KEY\_PRESSED, KEY\_RELEASED és KEY\_TYPED események áradatával is számolhatunk. Ha közben leütöttük az Alt-F4 billentyűkombinációt, akkor az alkalmazás még egy WINDOW\_CLOSING, majd egy WINDOW\_CLOSED eseményt is bedob a rendszerbe, mindenkor az éppen aktív ablak lesz a forrása.

A konkrét alacsony szintű eseményekkel a 10. fejezet foglalkozik részletesen.

**Magas szintű esemény** az, ami nem alacsony szintű. Ez általában logikai esemény (a program állítja össze), forrása nem feltétlenül komponens.

A könyvben tárgyalt magas szintű események a következők:

- **ActionEvent**: Akcióesemény (elindítottak egy akciót). Forrásai: AbstractButton leszármazottai: JButton, JToggleButton, JCheckBox, JRadioButton, JMenuItem, JCheckBoxMenuItem, JRadioButtonMenuItem, valamint JComboBox, JTextField és Timer. Azonosítója: ACTION\_PERFORMED: Az akció lezajlott.
- **AdjustmentEvent**: Igazítási esemény (megváltozott a görgetősáv aktuális értéke). Forrása: JScrollPane. Azonosítója: ADJUSTMENT\_VALUE\_CHANGED: Megváltozott a görgetősáv helyzetével összefüggő érték.
- **ListSelectionEvent**: Listakiválasztás-esemény (a kiválasztott elem megváltozott). Forrása: JList. Azonosítója: nincs (nem AWTEvent).
- **DocumentEvent**: Dokumentumesemény (a dokumentum megváltozott). A DocumentEvent interfész, az implementált esemény nem az EventObject leszármazottja. Azonosítója nincs. Forrása általában egy komponens „mögötti” dokumentum (pl. a JTextField vagy a JTextArea dokumentuma).

A magas szintű eseményeket beleprogramozták a megfelelő komponensekbe. A JButton például, ha kattintanak rajta, összeállít egy ActionEvent eseményt, s annak ő maga lesz a forrása. Amikor egy JScrollPane komponensben megváltozik a kijelzett érték, akkor a komponens egy AdjustmentEvent osztályú eseményobjektumot hoz létre, s a görgetősáv lesz a forrás. A konkrét magas szintű eseményekkel a 8. fejezet foglalkozik részletesen.

## 7.4. Eseménydelegációs modell

### Forrásobjektum

Egy eseménynek minden van tulajdonosa, illetve forrása – az esemény keletkezésének helye. Egy forrásobjektum – osztályától függően – meghatározott típusú eseményeket kelthet.

Például:

- ◆ Bármely komponens (a Component leszármazottja) MouseEvent eseményt generál, ha elhalad felette az egér és FocusEvent eseményt kelt, ha éppen fókuszba kerül. Ha a komponens fókuszban van, akkor egy billentyű lenyomásakor KeyEvent eseményt hoz létre. Ha az alkalmazás egyetlen komponense sincs fókuszban, akkor nem keletkezik billentyűesemény.
- ◆ A JButton osztály példánya, ha éppen fókuszban van és lenyomják a szóközbillentyűt, akkor a KeyEvent esemény mellett egy ActionEvent típusú eseményt is kelt.

### Figyelőobjektum, figyelőlánc

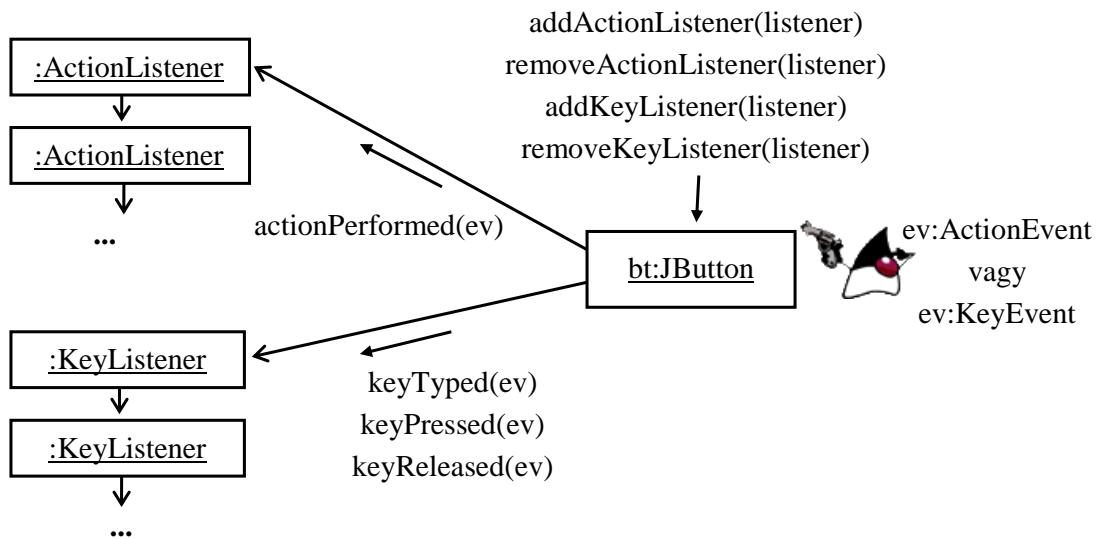
Az eseményekre csak akkor reagálhatunk, ha figyeljük őket. minden forrásobjektumhoz ki kell jelölnünk ún. **figyelőobjektumokat** – ezekben kezelhetjük (le) a forrásobjektumon keletkezett eseményeket. Egy GUI alapú alkalmazásban vannak „gyárilag” beépített figyelők – a program ezért reagál eleve, a programozó beavatkozása nélkül is bizonyos eseményekre (például a keret elmozgatására vagy átméretezésére). Az alkalmazás egyedi figyelőit azonban a programozónak kell a megfelelő forrásokhoz rendelnie.

**A figyelőket a forrásobjektum figyelőláncára fel kell fűzni.** minden forrásobjektumnak – az objektum osztályától függően – többféle figyelőlánc is lehet: a különböző típusú figyelők más-más figyelőláncra kerülnek. Például:

A 7.4. ábrán látható, hogy a nyomógombnak két figyelőláncra van:

- ◆ az egyiken vannak az ActionEvent eseményt figyelő ActionListener objektumok. Az objektumoknak implementálniuk kell az ActionListener interfész. A figyelőt az addActionListener metódussal lehet a gomb akciófigyelő láncára felfűzni.
- ◆ a másikon vannak a KeyEvent eseményt figyelő KeyListener objektumok. Ezeknek az objektumoknak implementálniuk kell a KeyListener interfész. A figyelőt az addKeyListener metódussal lehet a gomb billentyűzetfigyelő láncára felfűzni. A KeyListener interfészben három metódus is van, mindhárat implementálni kell!

**Megjegyzés:** minden komponenshez megvannak a megfelelő alacsony szintű eseményeket számon tartó figyelőláncok – lehet persze, hogy üresek. Események mindenkorban keletkeznek, legfeljebb nem figyel rájuk senki.



7.4. ábra. A JButton akció- és billentyűzetfigyelő lánca

A következő két táblázat összefoglalja a könyvben tárgyalt alacsony és magas szintű események típusát, a hozzájuk tartozó lehetséges források osztályát, az eseményt figyelő interfész, a forrásobjektum felfüző metódusát és az interfészben deklarált eseménykezelő metódusokat.

**Figyelje meg az azonosítóképzési szabályokat!** Ha a keletkezett esemény **xxx**, akkor az eseménynek **XXXEvent** az osztálya, **addXXXListener** a forrásobjektum felfüző metódusa és **XXXListener** a figyelőinterfész.

Az esemény figyelőit a forráshoz tartozó figyelőláncra fel kell fűzni. Egy figyelőláncra csak azonos osztályú események figyelői fűzhetők fel, ezért minden forrásobjektumnak annyi figyelőláncra van, ahány különböző típusú eseményt kelthet. A felfüző (és lefűző) metódus a forrás osztályában szerepel:

- ▶ void addXXXListener(XXXListener listener);
- ▶ void removeXXXListener(XXXListener listener);

Figyelő hozzáadása a figyelőlánchoz, illetve törlése a figyelőláncból (a megfelelő metódusok a forrásobjektum osztályában vannak deklarálva):

Az **xxx** eseményt csak az a komponens fogadhatja és kezelheti, amely implementálta az **XXXListener** interfészét és tagja a forrás figyelőláncának. A figyelőinterfész implementálása:

```

class Figyelő implements XXXListener {
 // XXXListener eseménykezelő metódusainak kifejtése
}

```

**Alacsony szintű események:**

| Eseménytípus   | Forrás    | Figyelő interfész    | Felfűző metódus         | Eseménykezelő metódusok                                                                                                       |
|----------------|-----------|----------------------|-------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| ComponentEvent | Component | ComponentListener    | addComponent-Listener   | componentResized<br>componentMoved<br>componentShown<br>componentHidden                                                       |
| ContainerEvent | Container | ContainerListener    | addContainer-Listener   | componentAdded<br>componentRemoved                                                                                            |
| FocusEvent     | Component | FocusListener        | addFocusListener        | focusGained<br>focusLost                                                                                                      |
| KeyEvent       | Component | KeyListener          | addKeyListener          | keyTyped<br>keyPressed<br>keyReleased                                                                                         |
| MouseEvent     | Component | MouseListener        | addMouseListener        | mousePressed<br>mouseReleased<br>mouseEntered<br>mouseExited<br>mouseClicked                                                  |
| MouseEvent     | Component | MouseMotion-Listener | addMouseMotion-Listener | mouseDragged<br>mouseMoved                                                                                                    |
| WindowEvent    | Window    | WindowListener       | addWindow-Listener      | windowOpened<br>windowClosing<br>windowClosed<br>windowIconified<br>windowDeiconified<br>windowActivated<br>windowDeactivated |

**Magas szintű események:**

| Eseménytípus        | Forrás                                         | Figyelő interfész      | Felfűző metódus           | Eseménykezelő metódusok                       |
|---------------------|------------------------------------------------|------------------------|---------------------------|-----------------------------------------------|
| ActionEvent         | Abstr.Button utódai<br>JComboBox<br>JTextField | ActionListener         | addActionListener         | actionPerformed                               |
| AdjustmentEvent     | JScrollBar                                     | AdjustmentListener     | addAdjustment-Listener    | adjustmentValueChanged                        |
| ListSelection-Event | JList                                          | ListSelection-Listener | addListSelection-Listener | valueChanged                                  |
| DocumentEvent       | Document                                       | DocumentListener       | addDocument-Listener      | insertUpdate<br>removeUpdate<br>changedUpdate |

Megfigyelhetjük, hogy az AWT-eseményeket kezelő metódusok nevében benne van az esemény azonosítója. Például a `KeyEvent` eseményosztályon belül háromféle esemény van: `KEY_TYPED`, `KEY_PRESSED` és `KEY_RELEASED`; ennek megfelelően a `KeyListener` három eseménykezelő metódust definiál: `keyTyped`, `keyPressed` és `keyReleased`.

Egy eseményosztályhoz általában egyetlen felfűző metódus tartozik, vagyis egy forrásobjektumnak eseménytípusonként csak egy figyelőláncra van. Ebből a szempontból a `MouseEvent` kivétel, az az eseményfajták nagy száma miatt két felfűző metódust – és ennek megfelelően két interfész (MouseListener és MouseMotionListener) – definiál.

Az események keletkezésének sorrendje és a keletkezett esemény osztálya a program futásakor derül ki. Az esemény az eseménydelegációs modell által meghatározott úton eljut bizonyos figyelőobjektumokhoz.

#### Az eseménydelegációs modell elemei:

- **Esemény:** Objektum; állapota magába foglalja az eseményre jellemző adatokat, és a forrásobjektum referenciáját. Az `AWTEvent` tartalmazza az esemény azonosítóját.
- **Eseményforrás (forrásobjektum):** Objektum; rajta keletkezik az esemény. A forrásobjektumhoz figyelőláncok tartoznak.
- **Eseményfigyelő:** Az eseményt figyelő és kezelő objektum. Egy objektum csak akkor figyelhet (hallgathat) egy eseményt, ha fel van fűzve a forrásobjektum megfelelő figyelőláncára (hallgatóláncára), és osztálya implementálja a figyelőinterfészt. A figyelésnek nincs semmilyen további feltétele.

**Az eseményt először a forrásobjektum kapja meg feldolgozásra. A forrásobjektum az eseményt továbbadja a figyelőinek, hogy azok is feldolgozhassák.**

A különféle eseményekkel és azok kezelésével a következő fejezetek foglalkoznak részletesen.

## 7.5. A felhasználói felület tervezése

Ebben a kötetben sok képernyőt (grafikus felhasználói interfész) fogunk készíteni. A következő módszer segítséget ad az interfész tervezéséhez (a szerző ajánlása).

#### Ajánlás a grafikus felhasználói interfész tervezéséhez:

Logikailag és nagyság szerint bontsuk fel a grafikus felhasználói interfész képernyőrészekre (keretekre, ablakokra, panelekre stb.)! **Tervezzük meg a feladat megoldásához szükséges összes képernyőrészét és azok tulajdonosi hierarchiáját!** Nyilakkal jelezük, hogy melyik akcióra melyik ablak nyílik meg!

Minden képernyőrészre végezzük el a következőket:

- **Rajzoljuk meg a képernyőrészt**, és tegyük bele a szükséges komponenseket (vezérlőket)!
- **Határozzuk meg az eseményforrásokat!** Tüntessük fel a képen azonosítójukat és/vagy osztályukat! Zárójelben adjuk meg a keletkezett esemény osztályát!
- **Határozzuk meg az eseményfigyelőket!** Tüntessük fel a képen azonosítójukat és/vagy osztályukat! Zárójelben adjuk meg a figyelőinterfészt, esetleg a figyelt objektum azonosítóját!

Csak annyi információt tegyünk a tervre, hogy a kép kódolható legyen és még át is lehessen tekinteni!

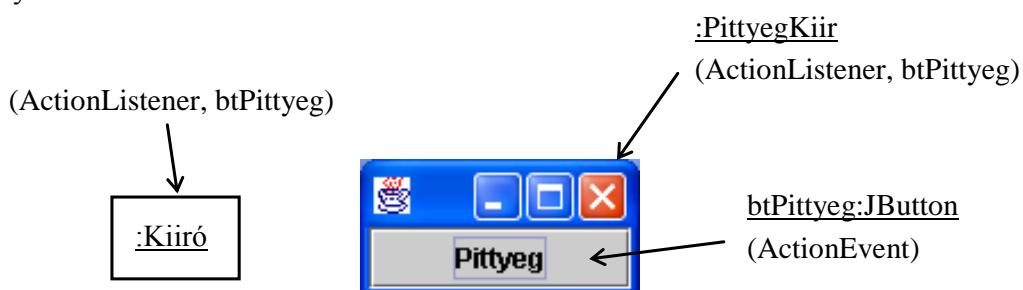
Most nézzünk néhány kisebb, egyablakos feladatot, melyben a fenti módszert követjük!

*Megjegyzés:* Nagyobb feladatok a kötet utolsó fejezetében, valamint az elektronikus melléklet *Esettanulmányok* könyvtárában találhatók.

#### Feladat – Pittyeg, kiír

A 7.1. pont mintaprogramjában szereplő nyomógombot most ne csak a keret figyelje, hanem egy másik, nem komponens objektum is, s az minden lenyomáskor írja ki a konzolra a "Lenyomták a gombot." szöveget!

Tervezzük meg először a programban szereplő objektumokat, eseményforrásokat és figyelőket!  
A feladatban van egy láthatatlan objektum is, s az is figyeli a nyomógomb ActionEvent eseményét:



#### Forráskód

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class Kiíró implements ActionListener { //1
 public void actionPerformed (ActionEvent ev) { //2
 System.out.println("Lenyomták a gombot.");
 }
}

```

```

public class PittyegKiir extends JFrame
 implements ActionListener { //3
private JButton btPittyeg;

public PittyegKiir() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 getContentPane().add(btPittyeg = new JButton("Pittyeg"));
 btPittyeg.addActionListener(this); //4
 btPittyeg.addActionListener(new Kiro()); //5
 pack();
 show();
}

public void actionPerformed (ActionEvent ev) {
 Toolkit.getDefaultToolkit().beep();
}
public static void main (String args[]) {
 new PittyegKiir();
}
}

```

### A program elemzése

A nyomógomb a forrásobjektum, ActionEvent keletkezik rajta. A Kiro osztályt felkészítjük az esemény fogadására: //1-ben implementáltuk az ActionListener interfészt, //2-ben pedig az interfésznek megfelelő eseménykezelő metódusban konzolra írjuk a "Lenyomták a gombot." szöveget. A PittyegKiir osztály már az eredeti programban is fel volt készítve az akcióesemény kezelésére. A gombhoz hozzápasztjuk a két figyelőt: //4-ben magát a keretet, //5-ben pedig a Kiro osztály egy példányát. Az már az alkalmazás dolga, hogy az eseményt eljuttassa a figyelőkhöz.

#### Feladat – Nyomásszámító

Tegyük két nyomógombot a keretbe, indulásképpen mindeneknek "0" legyen a felirata! Ha lenyomnak egy gombot, akkor növeljük meg eggyel a rajta levő szám értékét!

A két gomb egy-egy eseményforrás lesz, s a keret fogja figyelni őket. A keretnek majd el kell döntenie, hogy melyik forrástól érkezett az esemény, hiszen minden eseményt a forrásgomb címkéjét kell megváltoztatnia:



**Forráskód**

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class NyomasSzamlalo extends JFrame
 implements ActionListener {
 private JButton btBal, btJobb;
 private int nBal=0, nJobb=0;

 public NyomasSzamlalo() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 Container cp = getContentPane();
 cp.setLayout(new GridLayout(1,2)); //1
 cp.add(btBal = new JButton(""+nBal)); //2
 cp.add(btJobb = new JButton(""+nJobb)); //3
 btBal.addActionListener(this); //4
 btJobb.addActionListener(this); //5
 pack();
 show();
 }

 public void actionPerformed (ActionEvent ev) { //6
 if (ev.getSource()==btBal) //6
 btBal.setText(""+++nBal); //7
 else
 btJobb.setText(""+++nJobb); //8
 }

 public static void main (String args[]) {
 new NyomasSzamlalo();
 }
}

```

**A program elemzése**

A keret elrendezésmenedzsere egy 1 soros, 2 oszlopos rács lesz (//1). A rácspontokra rátesszük a két gombot (//2 és //3); kezdetben a "0" felirat szerepel majd mindenkorban. Az előző feladatban két objektum figyelt egyetlen forrást, most egy objektum (a keret) figyel majd két forrást (a két gombot). //4-ben az egyik gomb, //5-ben pedig a másik gomb figyelőláncára fűzzük fel a kereket (this). A keretnek implementálnia kell az ActionListener interfészt; hozzá két helyről érkezhet ActionEvent esemény. Aszerint, hogy az esemény melyik gombtól érkezett (//6), a bal gomb (//7) vagy a jobb gomb (//8) címkéjén levő számot növeljük meg egyel.

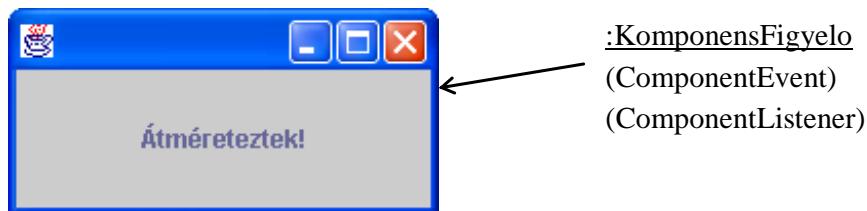
**Megjegyzések:**

- Meglehetünk volna az nBal és nJobb globális (osztály szintű) nyomásszámítás nélkül is: elég lett volna leolvasni a megfelelő gomb címkéjét és visszaírni rá az eggyel növelt értéket.
- A feladat a JButton kiterjesztésével is megoldható úgy, hogy a gomb maga számlálja, hányszor nyomták le (*NyomasSzamlalo2.java*).

### Feladat – Komponensfigyelő

Hozzunk létre egy keretet! Ha ezt a keretet átméretezik vagy elmozdítják, akkor jelenjen meg rajta az "Átméreteztek!", illetve "Elmozdítottak!" felirat.

A keret most magát fogja figyelni, vagyis a forrás és a figyelőobjektum azonos. A keretben levő címke csak az információ kiírására szolgál:



### Forráskód

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class KomponensFigyelo extends JFrame
 implements ComponentListener {
 private JLabel lbInfo;

 public KomponensFigyelo () {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 getContentPane().add(lbInfo =
 new JLabel(" ",JLabel.CENTER));
 addComponentListener(this); //1
 pack(); //2
 show();
 }

 public void componentResized(ComponentEvent ev) { //3
 lbInfo.setText("Átméreteztek!");
 }

 public void componentMoved(ComponentEvent ev) { //4
 lbInfo.setText("Elmozdítottak!");
 }

 public void componentShown(ComponentEvent ev) { //5
 }

 public void componentHidden(ComponentEvent ev) { //6
 }

 public static void main (String args[]) {
 new KomponensFigyelo();
 }
}

```

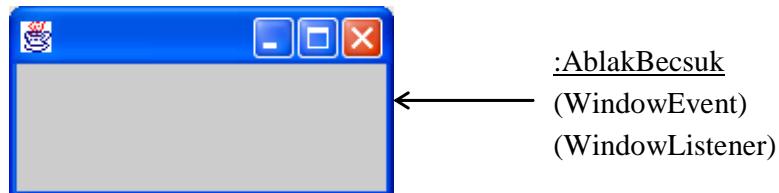
### A program elemzése

//1-ben a keretbe teszünk egy címkét. A figyelendő esemény most a ComponentEvent. A keret magát fogja figyelni, vagyis azonos a forrás- és a figyelőobjektum. A felfűző metódus az addComponentListener (//2). A ComponentListener interfészben négy eseménykezelő metódust definiáltak: componentResized (átméretezték), componentMoved (elmozgatták), componentShown (látható lett) és componentHidden (láthatatlan lett). A négyből csak kettőbe írunk kódot, mivel a feladat szerint az eseményre csak akkor kell reagálnunk, ha a komponenst átméretezték vagy elmozgatták. Ilyenkor ráírjuk a keretben levő címkére a megfelelő szöveget (//3 és //4). //5 és //6 üres metódusok – az interfész összes metódusát ki kell fejtenünk, akkor is, ha a megfelelő eseményre nem akarunk reagálni.

*Megjegyzés:* A JLabel szövege kezdetben nem üres, hanem egy szóköz. Ha üres lenne, akkor a pack() úgy értelmezné a helyzetet, hogy nincs mit kiírni, és teljesen összenyomná a keretet.

#### Feladat – Ablak becsukása

Készítsünk egy keretet! Ha a keretet be akarják csukni, akkor csukódjon be úgy, hogy a program rövid hangot hallat és befejezi futását!



#### Forráskód

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class AblakBecsuk extends JFrame
 implements WindowListener { //1
 public AblakBecsuk () {
 setBounds(100,100,200,100);
 addWindowListener(this); //2
 show();
 }

 public void windowOpened(WindowEvent ev) {} //3
 public void windowClosing(WindowEvent ev) { //4
 Toolkit.getDefaultToolkit().beep();
 System.exit(0);
 }

 public void windowClosed(WindowEvent ev) {} //5
 public void windowIconified(WindowEvent ev) {} //6
 public void windowDeiconified(WindowEvent ev) {} //7
 public void windowActivated(WindowEvent ev) {} //8
 public void windowDeactivated(WindowEvent ev) {} //9
}

```

```

 public static void main (String args[]) {
 new AblakBecsuk();
 }
 }
}

```

### A program elemzése

A kereten ablakesemények (WindowEvent) keletkezhetnek. Ezeket az eseményeket a keret maga fogja figyelni, ezért //1-ben implementálja a WindowListener interfészt. //2-ben a keretet önmagával figyeltetjük. //3-tól //9-ig találhatók az eseménykezelő metódusok; közülük csupán egynek a blokkjába írunk kódot. Csak a WINDOW\_CLOSING eseményre akarunk figyelni; ez az esemény akkor történik meg, amikor a keret jobb felső sarkában található becsukóikonra rákattintanak. A program ezt az eseményt a windowClosing metódusban kezeli: hangadás után a System.exit meghívásával megszakíthatjuk a program futását.

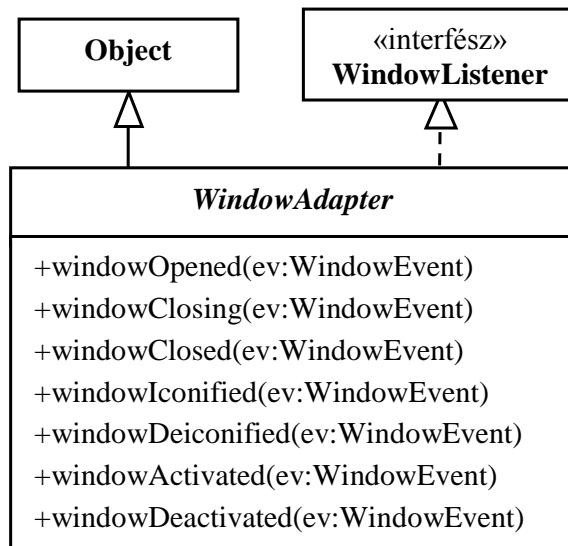
## 7.6. Eseményadaptek

Egy interfész implementálásakor ki kell fejtenünk a benne megadott összes metódust, még akkor is, ha nem akarjuk minden működtetni. Többször megesik, hogy az öt-hét eseményfajta közül csak egyet vagy kettőt akarunk kezelni, ezért a metódusok nagy része üresen marad. A felesleges kódolási munka elkerülésére a Java ún. adapterosztályokat definiál. Egy adapterosztály üres metódusokkal kifejt a megfelelő interfész összes metódusát. Az adapterosztály utódjában már elég csak a megfelelő metódust felülírni.

**Eseményadapternek** nevezzük azt az absztrakt osztályt, amely üres metódusokkal implementálja a figyelőinterfészt. A Java egy-egy eseményadaptert implementált a több metódust tartalmazó figyelőkhöz.

A 7.5. ábra példaként a WindowAdapter osztályt mutatja. Az eseményadaptek a következők:

| Figyelőinterfész    | Adapterosztály     | Metódusok száma |
|---------------------|--------------------|-----------------|
| ComponentListener   | ComponentAdapter   | 4               |
| ContainerListener   | ContainerAdapter   | 2               |
| FocusListener       | FocusAdapter       | 2               |
| KeyListener         | KeyAdapter         | 3               |
| MouseListener       | MouseAdapter       | 5               |
| MouseMotionListener | MouseMotionAdapter | 2               |
| WindowListener      | WindowAdapter      | 7               |



7.5. ábra. WindowAdapter osztály

Jó lenne, ha az eseménykezelő metódusok osztálya örökölhetné az eseményadaptert. De mivel a Javában nincs kettős öröklés, azért egy osztály nem lehet egyszerre adapter és komponens is. Kénytelenek vagyunk tehát társítási kapcsolatot létesíteni az eseményadapterrel.

### Feladat – Adapterek

Az előző pont Ablak becsukása programjában a hét ablakesemény-kezelő metódusból hatot üresen hagytunk – minden össze a `windowClosing` metódusba írtunk programkódot. Használjuk most a `WindowAdapter` osztályt az esemény kezeléséhez!

Három megoldást is adunk. Mindháromban a `WindowAdapter` osztályból származtatunk majd egy osztályt, s az csak egyetlen metódust fog kifejteni: a `windowClosing`-ot. A különbség annyi lesz, hogy a szóban forgó adapterosztály az első megoldásban külső osztály (`WindowFigyelo`), a második megoldásban belső osztály (`Adapter1.WindowFigyelo`), a harmadikban pedig névtelen belső osztály lesz.

### 1. megoldás: Esemény kezelése külső eseményadapter-osztályban

#### Forráskód

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

```

```

class WindowFigyelo extends WindowAdapter { //1
 public void windowClosing(WindowEvent ev) { //2
 System.exit(0);
 }
}

public class Adapter1 extends JFrame {
 // Esetleges változók. A WindowFigyelo-ból nem érhetők el.

 public Adapter1() {
 setBounds(300,300,300,200);
 addWindowListener(new WindowFigyelo()); //3
 show();
 }

 public static void main (String args[]) {
 new Adapter1();
 }
}

```

### A program elemzése

A `WindowAdapter` osztály kiterjesztésével létrehozunk egy `WindowFigyelo` osztályt (//1), és kifejtjük azt a metódusát, amelyben eseményt akarunk kezelní (//2). //3-ban a keret figyelőláncához hozzáadjuk az éppen létrehozott `WindowFigyelo` objektumot.

Ebben a megoldásban a `WindowFigyelo` objektum nem érné el az `Adapter1` keret esetleges példányváltozóit, ezért azokra nem is lehetne hivatkozni az esemény kezelésében. Az eseménykezelés feladatát nem érdemes egy másik osztálynak továbbadni, ha annak ismernie kell a forrásobjektum adatait.

## 2. megoldás: Esemény kezelése belső eseményadapter-osztályban

### Forráskód

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Adapter2 extends JFrame {
 // Esetleges változók. A WindowFigyelo-ból elérhetők.

 class WindowFigyelo extends WindowAdapter { //1
 public void windowClosing(WindowEvent ev) {
 System.exit(0);
 }
 }

 public Adapter2() {
 setBounds(300,300,300,200);
 addWindowListener(new WindowFigyelo());
 show();
 }
}

```

```

 public static void main (String args[]) {
 new Adapter2();
 }
}

```

### A program elemzése

A WindowFigyelo osztály szerepe ugyanaz, mint az előző feladatban, de ezúttal belső osztály (//1).

Ebben a megoldásban a kezelőmetódus ismerné az Adapter2 keret esetleges változóit, vagyis dolgozhatna velük. Az eseményeket kezelő osztályokat általában ajánlatos belső osztályként megírni.

## 3. megoldás: Esemény kezelése névtelen eseményadapter-osztályban

### Forráskód

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class Adapter3 extends JFrame {
 // Esetleges változók. A névtelen WindowAdapter-ból elérhetők.

 public Adapter3() {
 setBounds(300,300,300,200);

 // Névtelen osztály hozzáadása a figyelőlánchoz:
 addWindowListener(new WindowAdapter() { //1
 public void windowClosing(WindowEvent e) {
 System.exit(0);
 }
 });
 show();
 }

 public static void main (String args[]) {
 new Adapter3();
 }
}

```

### A program elemzése

Ebben a megoldásban az ablakfigyelő osztálynak nem adtunk nevet; a létrehozásával egy időben implementáltuk és azonnal fel is fűztük a figyelőláncre (//1). Ezt a megoldást akkor szokás használni, ha a figyelőosztály tényleg nagyon kicsi, és nem rontja az áttekinthetőséget.

## Tesztkérdések

- 7.1. Mely állítások igazak? Jelölje be az összes jó választ!
- a) minden alacsony szintű esemény AWT-esemény.
  - b) minden esemény tudja, hogy melyik objektum az ő forrása.
  - c) két billentyű egyszerre való leütésével elérhető, hogy a két esemény pontosan egy időben keletkezzék.
  - d) Az eseményt a forrásobjektum nem kapja meg feldolgozásra, csak a figyelői.
- 7.2. Mely állítások igazak? Jelölje be az összes jó választ!
- a) Az AWT-esemény azonosítóját az `EventObject` osztály deklarálja.
  - b) Az `AWTEvent` osztály deklarálja, hogy az esemény él-e még.
  - c) Egy eseménynek több forrása is lehet egyszerre.
  - d) Egy komponensnek elvileg egyszerre több figyelőlánca is lehet.
- 7.3. Az alábbiak közül melyek az alacsony szintű események? Jelölje be az összes jó választ!
- a) `WindowEvent`
  - b) `MouseEvent`
  - c) `KeyEvent`
  - d) `ActionEvent`
- 7.4. Az AWT-esemény mely metódusa adja vissza azt az objektumot, amelyen az esemény keletkezett? Jelölje be az egyetlen jó választ!
- a) `getClass()`
  - b) `getId()`
  - c) `getSource()`
  - d) `getActionCommand()`
- 7.5. A következők közül melyek a `WindowListener` interfész metódusazonosítói? Jelölje be az összes jó választ!
- a) `windowIconified`
  - b) `windowHidden`
  - c) `windowPressed`
  - d) `windowClosed`
- 7.6. Mely állítások igazak? Jelölje be az összes jó választ!
- a) A felhasználó csak a látható komponensen kelthet eseményt.
  - b) Az `addMouseListener(obj)` metódus révén az `obj` objektum az esemény forrá-sává válik.
  - c) Egy figyelőt az `addListener()` metódussal bármely figyelőláncra fel lehet fűzni.
  - d) A `KeyEvent` figyelőjének implementálnia kell a `KeyListener` interfészt.
- 7.7. Mely állítások igazak a `WindowAdapter`-re? Jelölje be az összes jó választ!
- a) A `WindowAdapter` interfész.
  - b) A `WindowAdapter` osztály, az `Object` osztály közvetlen utódja.
  - c) A `WindowAdapter` osztály, és implementálja a `WindowListener` interfészt.
  - d) A `WindowAdapter` osztály, és a kiterjesztésében felül kell írni mind a hét metódusát.

## Feladatok

- 7.1. (A) Tegyen a keretre egy gombot! Kezdetben a keretnek legyen az a címe, hogy „Haha”, s ez a cím minden gombnyomáskor bővüljön egy „ha”-val! (*Haha.java*)
- 7.2. (A) Bővítse az előző feladatot úgy, hogy az ablak becsukásakor fejeződjék be a program! (*BecsukHaha.java*)
- 7.3. (A) Ha a keretet ikonná változtatják, akkor íródjon ki rajta az, hogy "n-szer ikonná változtattál! Háromnál többet nem bírok!"! És a negyedszeri ikonná változtatásra tényleg vessünk véget a programnak! (*Ikonizalgalat.java*)
- 7.4. (A) Készítsen egy olyan keretet, amely felett az egérkurzor homokórává válik, elmozdításra pedig hangot ad! (*HomokHang.java*)
- 7.5. (A) Készítsen egy keretet, és tegyen bele négy gombot a következő felirattal: jobbra, balra, fel, le! Ha rákattint egy gombra, akkor az ablak mozduljon el a képernyőn 10 ponttal a megfelelő irányban! (*AblakMozdul.java*)
- 7.6. (A) Gépeljünk be egy szöveget, melyet jelenítsük meg folyamatosan a keret címében! (*CimIras.java*)

## 8. Swing-komponensek

---

A fejezet pontjai:

1. Swing-konstansok – SwingConstants
  2. Címke – JLabel
  3. A gombok öse – AbstractButton
  4. Nyomógomb – JButton
  5. Jelölőmező – JCheckBox
  6. Rádiógomb – JRadioButton, csoportosítás
  7. Kombinált lista – JComboBox
  8. MVC-modell, dokumentumkezelés
  9. A szövegek öse – JTextField
  10. Szövegmező – JTextField
  11. Szövegterület – JTextArea
  12. Lista – JList
  13. Görgetősáv – JScrollPane
  14. Menüsor – JMenuBar...
  15. Ablak – JWindow
  16. Dialógusablak – JDialog, kész dialógusok – JOptionPane
  17. Időzítő – Timer
- 

Ebben a fejezetben a Swing alapkomponenseit fogjuk tárgyalni. Mindegyikről elmondjuk, hogy melyik csomagba tartozik, mi a közvetlen öse és melyek a fontosabb interfészei, röviden leírjuk magát a komponenst, és még a következőket (egy mintát, a fontosabb eseményeket...):

♦ **Mintakomponens a keretben**

Bemutatjuk a komponens képét egy keretbe foglalva. Mindig megadjuk a következő forráskód // utasítások részét:

```
import javax.swing.*;
import java.awt.*;

public class Minta extends JFrame {
 Container cp = getContentPane();
```

```

public Minta() {
 cp.setLayout(new FlowLayout()); // így jobban látszik
 // Utasítások. Komponens betétele a keretbe, például
 // cp.add(new JLabel("Minta"));
 pack();
 show();
}

public static void main (String args[]) {
 new Minta();
}
}

```

- ◆ **Események:** A komponensen keletkező események, a figyelőinterfészek, a felfűző- és eseménykezelő metódusok.
- ◆ **Jellemzők:** A get, set és is metódussal beállítható, illetve lekérdezhető adatok. Csak a leggyakrabban használt jellemzőket soroljuk fel.
- ◆ **Konstruktorok, metódusok.** Csak a leggyakrabban használtakat soroljuk fel.

Végül minden komponenshez megadunk legalább egy mintafeladatot.

! **Figyelem!** Ez a fejezet nagyon terjedelmes. Egy-egy pont tanulmányozása után oldja meg a fejezet végi vonatkozó feladatokat!

## 8.1. Swing-konstansok – SwingConstants

Csomag: javax.swing Deklaráció: public interface SwingConstants

A **SwingConstants** interfész általános konstansokat deklarál. Mindegyik konstans statikus és int típusú. Az interfészt a Swing-komponensek nagy része implementálja.

### Mezők – Igazítások

- static int TOP // Felül
- static int BOTTOM // Alul
- static int LEFT // Bal
- static int RIGHT // Jobb
- static int CENTER // Közép
- static int LEADING // Vezető, megelőző
- static int TRAILING // Végző, követő

### Mezők – Állás

- static int HORIZONTAL // Vízszintes
- static int VERTICAL // Függőleges

### Mezők – Irány

- static int NORTH // Észak, fel
- static int SOUTH // Dél, le

- static int EAST                    // Kelet, jobb
- static int WEST                  // Nyugat, bal

Ha egy osztály implementálja a `SwingConstants` interfést, akkor ezeket a konstansokat el lehet érni az osztály vagy az osztály egy példányának megszólításával is. A következő példában a vastagon szedett azonosítók ugyanazt a konstanst azonosítják:

```
JLabel lbInfo = new JLabel("Információs szöveg");
lbInfo.setHorizontalAlignment(SwingConstants.CENTER);
lbInfo.setHorizontalAlignment(jLabel.CENTER);
lbInfo.setHorizontalAlignment(lbInfo.CENTER);
```

## 8.2. Címke – `JLabel`

Csomag: `javax.swing`      Deklaráció: `public class JLabel`

Közvetlen ős: `javax.swing.JComponent`

Fontosabb implementált interfészek: `SwingConstants`

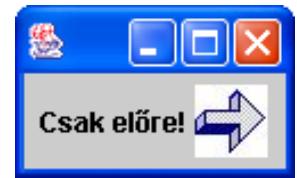
A **címke** (`label`, osztálya `JLabel`) információ kiírására használatos komponens. Magas szintű eseményt nem kelt, és nem kerülhet billentyűzetfókusba. A címke szöveget és/vagy ikont jeleníthet meg. Teljes tartalma a megjelenési területén belül vízszintesen és függőlegesen igazítható, és megadható a címke szövegének helyzete az ikonhoz képest.

### Mintacímke a keretben

```
// Csak szöveg:
cp.add(new JLabel("Csak szöveg"));
```



```
// Szöveg és címke. A szöveg megelőzi az ikont:
ImageIcon iiJobb = new ImageIcon("icons/jobb.jpg");
JLabel lb = new JLabel("Csak előre!", iiJobb, JLabel.CENTER);
lb.setHorizontalTextPosition(JLabel.LEADING);
add(lb);
```



### Események

A `JLabel` komponensen nem keletkezik magas szintű esemény.

### Jellemzők

- `String text`  
A címke szövege. Alapértelmezés: "" (üres).
- `Icon icon`  
A címke ikonja. A Java a gif és jpg kiterjesztésű ikonokat (képeket) ismeri fel. Az `ImageIcon` az `Icon` interfész implementációja – konstruktorában egy abszolút vagy

relatív útvonalat kell megadnunk az ikonhoz (a relatív útvonal a fejlesztőkörnyezetben megadott Working directory). Képekről a 9., a Grafika, rajzolás című fejezetben lesz szó részletesebben. Alapértelmezés: null (nincs ikon).

- ▶ `int horizontalAlignment`  
A címke vízszintes igazítása a megjelenési területen (pl. egy rácselemen) belül. Alapértelmezés: csak ikon esetén: `SwingConstants.RIGHT`, egyébként: `SwingConstants.LEFT`.
- ▶ `int verticalAlignment`  
A címke függőleges igazítása a megjelenési területen belül. Alapértelmezés: `SwingConstants.TOP`.
- ▶ `int horizontalTextPosition`  
A címke szövegének vízszintes helyzete a címke ikonjához képest. LEFT, LEADING: előtte; CENTER: rajta; RIGHT, TRAILING: utána.
- ▶ `int verticalTextPosition`  
A címke szövegének függőleges helyzete a címke ikonjához képest. TOP: kép tetejéhez igazítva; CENTER: a kép közepéhez igazítva; BOTTOM: a kép aljához igazítva.

### Konstruktorkok

- ▶ `JLabel(String text, Icon icon, int horizontalAlignment)`
- ▶ `JLabel(Icon icon, int horizontalAlignment)`
- ▶ `JLabel(String text, int horizontalAlignment)`
- ▶ `JLabel(String text)`
- ▶ `JLabel(Icon icon)`
- ▶ `JLabel()`

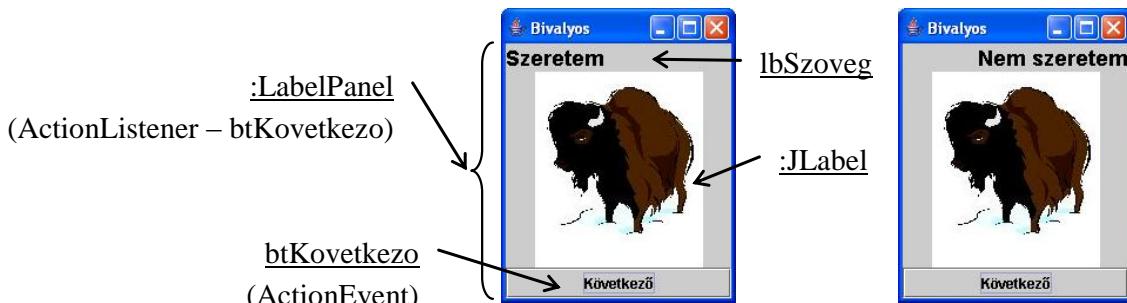
Címke létrehozása függőlegesen középre igazítva. A paraméterek jelentését lásd a jellemzőknél.

#### Feladat – LabelTest

A keretben felül egy címke van 18 pontos betűkkel, középen egy bivaly, alul egy "Következő" feliratú nyomógomb. A címke két állapotban lehet:

- Balra igazítva, "Szeretem" felirattal;
- Jobbra igazítva, "Nem szeretem" felirattal.

A gomb lenyomására a címke minden kerüljön át a másik állapotba!



### Forráskód

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class LabelPanel extends JPanel implements ActionListener {
 private JLabel lbSzoveg;
 private JButton btKovetkezo;

 public LabelPanel() {
 setLayout(new BorderLayout()); //1
 add(lbSzoveg = new JLabel("Szeretem",JLabel.LEFT),"North");
 lbSzoveg.setFont(new Font("Dialog",Font.BOLD,18));

 add(new JLabel(new ImageIcon("icons/bivaly.gif"))); //2
 add(btKovetkezo = new JButton("Következő"),"South");
 btKovetkezo.addActionListener(this);
 }

 public void actionPerformed(ActionEvent e) { //3
 if (lbSzoveg.getHorizontalAlignment()==JLabel.LEFT) {
 lbSzoveg.setHorizontalAlignment(JLabel.RIGHT);
 lbSzoveg.setText("Nem szeretem");
 }
 else {
 lbSzoveg.setHorizontalAlignment(JLabel.LEFT);
 lbSzoveg.setText("Szeretem");
 }
 }
}

class LabelTestFrame extends JFrame {
 public LabelTestFrame() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 setBounds(100,100,200,250);
 setTitle("Bivalyos");
 getContentPane().add(new LabelPanel()); //4
 setVisible(true);
 }
}

public class LabelTest {
 public static void main(String[] args) {
 new LabelTestFrame();
 }
}
```

### A program elemzése

A feladatot egy LabelPanel osztályú panellel oldjuk meg, s rátesszük majd a keret tartalom-paneljére.

- ◆ //1-től kezdve először határ mentire állítjuk a panel elrendezésmenedzserét, mert a címkét északra, a gombot délről, a bivalyt pedig középre szeretnénk majd tenni. Ezután létrehozzuk az `lbSzoveg` címkét a "Szeretem" szöveggel, vízszintesen balra igazítva, és betessük a panel északi részére.
- ◆ //2-től kezdve létrehozzuk a szöveg nélküli, bivalyos címkét; ez határ menti elrendezésben alapértelmezés szerint a panel közepére kerül. A "Következő" gombot a panel déli részére tesszük, majd akciófigyelő láncára felfűzzük a panelt.
- ◆ //3-ban kezeljük a gomb által keltett akciót: ha a szöveg eddig bal oldalon volt, akkor jobbra tesszük, ha jobb oldalon volt, akkor balra. A címke szövegét minden esetben megváltoztatjuk.

### 8.3. A gombok őse – AbstractButton

Csomag: `javax.swing` Deklaráció: `public abstract class AbstractButton`

Közvetlen ős: `javax.swing.JComponent`

Fontosabb implementált interfések: `SwingConstants`

Az `AbstractButton` osztály a gombszerű komponensek közös tulajdonságait tartalmazza. Utódjait a következő osztályhierarchia mutatja:

```
AbstractButton
+--JButton // Nyomógomb
+--JToggleButton // Kétállapotú gomb
| +--JCheckBox // Jelölőmező
| +--JRadioButton // Rádiógomb
+-- JMenuItem // Menütétel
```

A fenti osztályokra tehát érvényesek az itt megadott jellemzők és metódusok. minden gombszerű komponensnek van tehát felirata és/vagy ikonja. Van akcióparancs szövegük is; az a gomb születésekor megegyezik a felirattal. Az akcióparancs megadására akkor lehet szükség, ha két ugyanolyan feliratú gomb lenyomására különböző parancsot szeretnénk végrehajtatni.

**Események** ( minden `AbstractButton`-leszármazottón keletkezhetnek):

| <i>EsemOsztály</i> | <i>Mi történt?</i>                                                  | <i>Figyelőinterfész</i> | <i>Felfűzőmetódus</i> | <i>Eseménykezelők</i> |
|--------------------|---------------------------------------------------------------------|-------------------------|-----------------------|-----------------------|
| ActionEvent        | Kattintottak rajta, vagy leütötték a szóközt, amikor fókuszon volt. | ActionListener          | addActionListener     | actionPerformed       |

#### Jellemzők

- `String text`

A gomb felirata. Alapértelmezés: "" (üres).

- ▶ `Icon icon`  
A gomb ikonja. Alapértelmezés: `null` (nincs ikon).
- ▶ `int mnemonic`  
Emlékeztető karakter `int` formában. Az Alt + karakterbillentyű leütésére lenyomódik a gomb, ha a fókusz a komponens valamely szülőjénél van. A komponens feliratában az első ilyen karakter alá aláhúzás kerül. A mnemonic nem érzékeny arra, hogy kisbetű vagy nagybetű ütünk-e le. Alapértelmezés: 0 (nincs emlékeztető karakter).
- ▶ `boolean selected`  
Ki van választva vagy nincs. Alapértelmezés: `false`
- ▶ `String actionCommand`  
A gombhoz tartozó akcióparancs. Értéke `null` is lehet. Alapértelmezés: `text`

### Metódusok

- ▶ `void addActionListener(ActionListener l)`  
Figyelő hozzáadása a gomb akciófigyelő láncához.
- ▶ `void doClick()`  
A gomb lenyomása programból (mintha a felhasználó nyomná le).

## 8.4. Nyomógomb – JButton

Csomag: `javax.swing`      Deklaráció: `public class JButton`  
 Közvetlen ős: `javax.swing.AbstractButton`  
 Fontosabb implementált interfészek: `SwingConstants`

A **nyomógomb** (`button`, osztálya `JButton`) általában valamilyen akció, parancs elindítására használatos gomb. Lenyomáskor megváltozik a képe, a szegélye vastagabb lesz. A `JButton` őse az `AbstractButton` osztály.

### Mintanyomógomb a keretben

```
cp.add(new JButton("Nyomd meg!",
 new ImageIcon("icons/red-ball.gif")));
```



### Események

`AbstractButton`-tól örökölt események: **ActionEvent**.

### Jellemzők

`AbstractButton`-tól örökölt jellemzők: `text`, `icon`, `mnemonic`, `selected`, `actionCommand`.

### Konstruktörök

- JButton(String text, Icon icon)
- JButton(String text)
- JButton(Icon icon)

A paraméterek jelentése:

- text: A nyomógomb felirata. Alapértelmezés: "" (üres).
- icon: A nyomógomb ikonja. Alapértelmezés: null (nincs ikon).

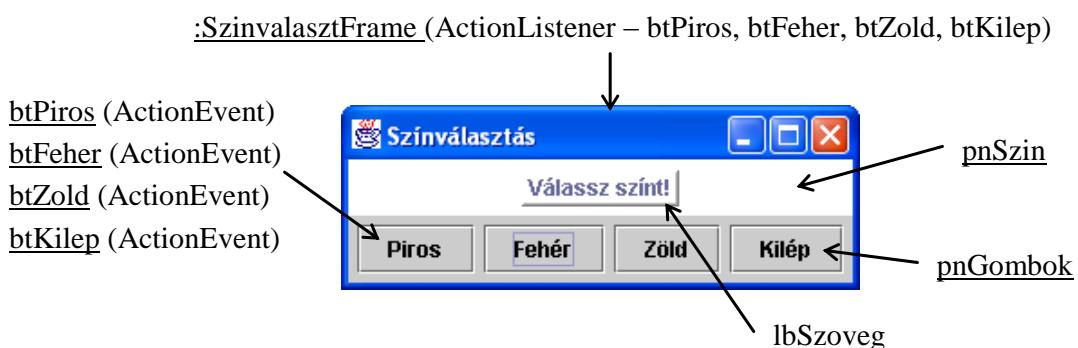
### Feladat – ButtonTest

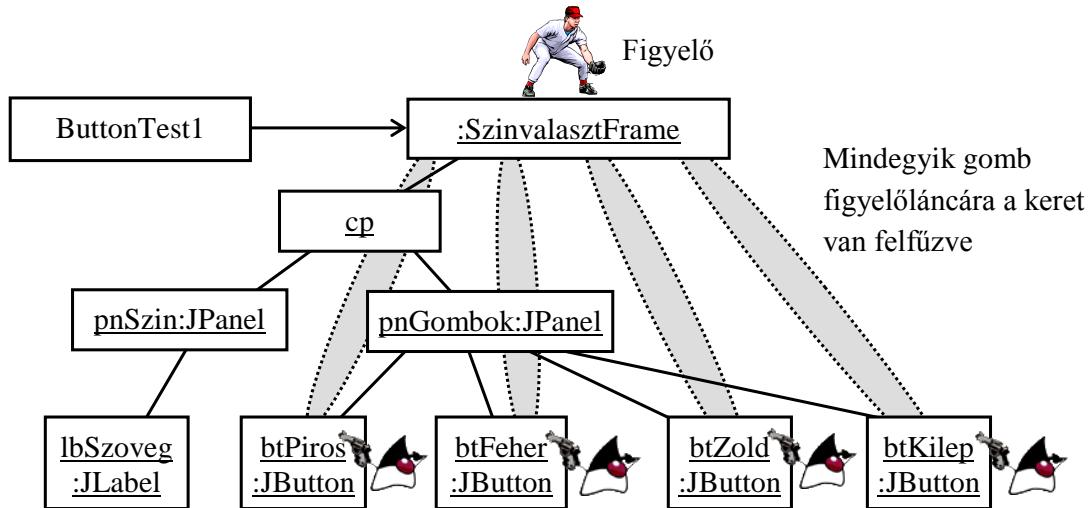
Tegyünk a képernyőre egy keretet; a felső részén egy színes terület legyen, benne a "Válassz színt!" felirattal. A terület színe kezdetben legyen piros, a felirat minden fehér alapon fekete. A keret alsó részére tegyünk négy gombot; három a terület színét állítsa a gomb felirata szerint, a negyedik gomb állítja le a programot. Az ablakot ne is lehessen másképp becsukni.

A színes terület egy panel lesz (pnSzin), ennek fogjuk változtatni a színét. A panelre rátesszük a szöveget (lbSzoveg). A gombokat is összefogjuk a pnGombok panel segítségével, s így együtt a keret déli részére tesszük őket. A gombok lenyomására ActionEvent események keletkeznek, valakinek majd figyelnie és kezelnie kell őket. Két megoldást fogunk megadni: az első megoldásban a keret lesz a figyelő, a másodikban a színét változtató panel.

### 1. megoldás – A keret a figyelő

Ebben a megoldásban mind a négy gombot a keret figyeli (8.1. ábra). Ennek megfelelően a pnSzin panelt a keret fogja átszínezni az akciófigyelő metódusában.





8.1. ábra. ButtonTest1 figyelőláncai

### Forráskód – ButtonTest1

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class SzinvalasztFrame extends JFrame implements ActionListener {
 Container cp = getContentPane();
 JButton btPiros, btFeher, btZold, btKilep;
 JPanel pnSzin;
 JLabel lbSzoveg;

 public SzinvalasztFrame() {
 setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
 setTitle("Szinválasztás");
 cp.add(pnSzin = new JPanel(), "North"); //1
 pnSzin.setBackground(Color.RED);
 lbSzoveg = new JLabel(" Válassz színt! ", JLabel.CENTER); //2
 lbSzoveg.setBackground(Color.WHITE);
 lbSzoveg.setOpaque(true);
 lbSzoveg.setBorder(BorderFactory.createRaisedBevelBorder());
 pnSzin.add(lbSzoveg);

 JPanel pnGombok = new JPanel(); //3
 pnGombok.add(btPiros = new JButton("Piros"));
 pnGombok.add(btFeher = new JButton("Fehér"));
 pnGombok.add(btZold = new JButton("Zöld"));
 pnGombok.add(btKilep = new JButton("Kilép"));
 cp.add(pnGombok, "South");

 btPiros.addActionListener(this);
 btFeher.addActionListener(this);
 }
}

```

```

 btZold.addActionListener(this);
 btKilep.addActionListener(this);
 pack();
 show();
 }

 public void actionPerformed(ActionEvent e) { //5
 if (e.getSource() == btPiros)
 pnSzín.setBackground(Color.RED);
 else if (e.getSource() == btFeher)
 pnSzín.setBackground(Color.WHITE);
 else if (e.getSource() == btZold)
 pnSzín.setBackground(Color.GREEN);
 else if (e.getSource() == btKilep) //6
 System.exit(0);
 }
}

public class ButtonTest1 {
 public static void main(String[] args) {
 new SzinvalasztFrame();
 }
}

```

### A program elemzése

//1-ben létrehozzuk a pnSzín panelt és hozzáadjuk a kerethez, majd a kezdeti színét pirosra állítjuk. //2-ban létrehozzuk a címkét, beállítjuk a színét és a szegélyét, majd rátesszük a panelre. //3-ban elkészítjük a gombpanelt, és rátesszük a négy gombot. A gombpanelt a tartalompanel déli részén helyezzük el.

//4-ben magát a keretet fűzzük fel a négy gomb figyelőláncára, ahogy az a 8.1. ábrán látszik. Négy láncról van tehát szó, mindenre csak a keret van felfűzve.

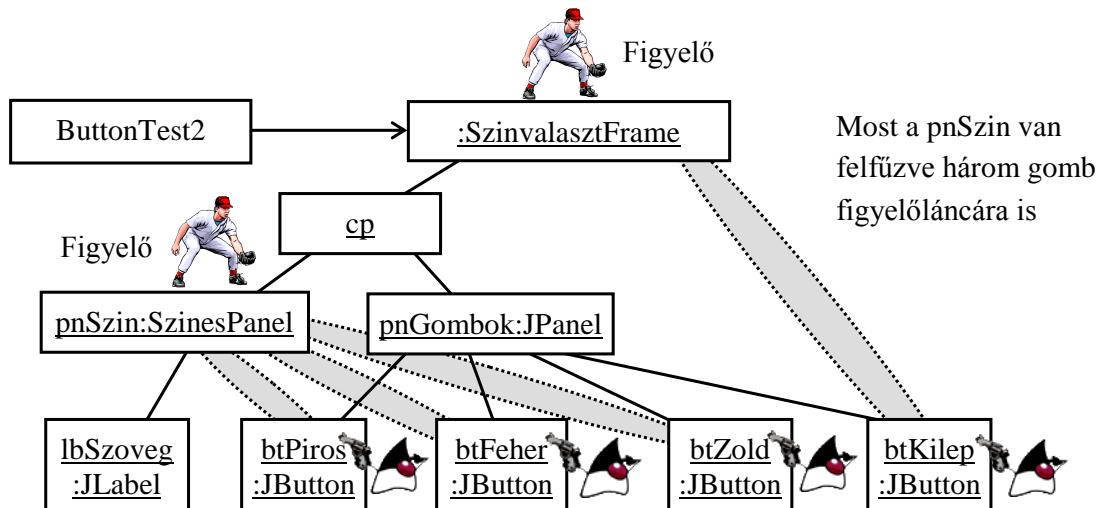
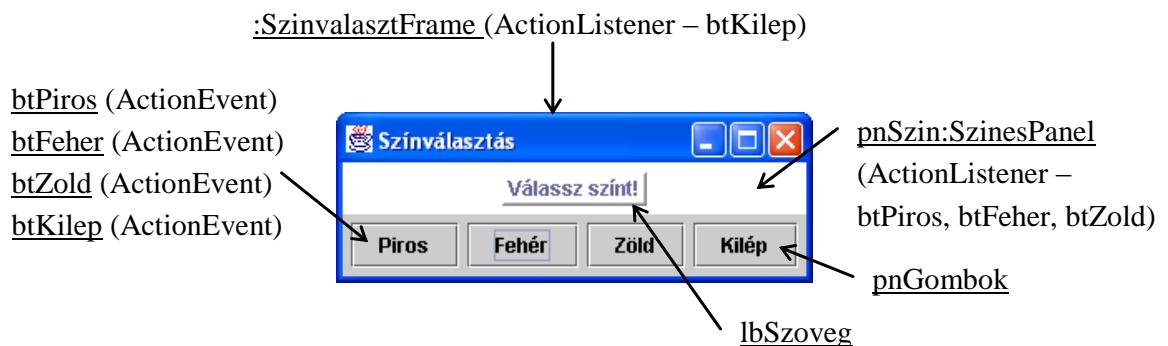
//5 a keret egyetlen eseménykezelő metódusa. Az osztály fejében megadtuk, hogy a keret az `ActionEvent` figyelője: `implements ActionListener`. Ezután a rendszer gondoskodik róla, hogy ha az akciófigyelő lánc forrásán egy esemény keletkezik, akkor az mindenkiéppen eljusszon az itt megadott `actionPerformed` metódushoz. A metódushoz tehát egy `ActionEvent` fog érkezni; a programozó feladata, hogy megadja a a tényleges teendőket. Az `actionPerformed` metódus megvizsgálja, hogy melyik gombtól jött az esemény, és szerint kéri a pnSzín panelt háttérszínének megváltoztatására. Ha a forrás a `btPiros`, akkor a panel háttérszíne piros lesz; ha a forrás a `btFeher`, akkor fehér stb., végül ha a forrás a `btKilep`, akkor befejeződik a futás.

//6-ban feleslegesen vizsgáljuk meg, hogy az esemény forrása a `btKilep` nyomógomb-e. Általában mégis ezt tesszük, mert a kód így könnyebben bővíthető.

❖ Ha a keretet nem fűzzük fel a gombok figyelőláncaira, akkor az esemény nem juthat el a kerethez! //4-ben mindenre csak a négy gomb figyelőláncára felfűztük a keretet.

## 2. megoldás – a SzínesPanel a figyelő

Ebben a megoldásban a színváltó gombokat a színes panel figyeli, a Kilép gombot pedig a keret (8.2. ábra). A JPanel osztályba nincs beépítve az akciófigyelés (ő nem ActionListener) – ezért most egy saját, SzínesPanel osztályból kell elkészítenünk a pnSzin panelt.



8.2. ábra. ButtonTest2 figyelőláncai

### Forráskód – ButtonTest2

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
class SzinesPanel extends JPanel implements ActionListener {
 public void actionPerformed(ActionEvent e) {
 if (e.getActionCommand().equals("Piros")) //1
 setBackground(Color.RED);
 else if (e.getActionCommand().equals("Fehér"))
 setBackground(Color.WHITE);
 else if (e.getActionCommand().equals("Zöld"))
 setBackground(Color.GREEN);
 }
}

class SzinvalasztFrame extends JFrame implements ActionListener {
 Container cp = getContentPane();
 JButton btPiros, btFehér, btZold, btKilep;
 SzinesPanel pnSzin;
 JLabel lbSzoveg;

 public SzinvalasztFrame() {
 setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
 setTitle("Színválasztás");
 cp.add(pnSzin = new SzinesPanel(), "North");
 pnSzin.setBackground(Color.RED);
 lbSzoveg = new JLabel(" Válassz színt! ", JLabel.CENTER);
 lbSzoveg.setBackground(Color.WHITE);
 lbSzoveg.setOpaque(true);
 lbSzoveg.setBorder(BorderFactory.createRaisedBevelBorder());
 pnSzin.add(lbSzoveg);
 JPanel pnGombok = new JPanel();
 pnGombok.add(btPiros = new JButton("Piros"));
 pnGombok.add(btFehér = new JButton("Fehér"));
 pnGombok.add(btZold = new JButton("Zöld"));
 pnGombok.add(btKilep = new JButton("Kilép"));
 cp.add(pnGombok, "South");

 btPiros.addActionListener(pnSzin);
 btFehér.addActionListener(pnSzin);
 btZold.addActionListener(pnSzin);
 btKilep.addActionListener(this);
 pack();
 show();
 }

 public void actionPerformed(ActionEvent e) {
 System.exit(0);
 }
}

public class ButtonTest2 {
 public static void main(String[] args) {
 new SzinvalasztFrame();
 }
}
```

### A program elemzése

Ahhoz, hogy a színes panel figyelő lehessen, implementálnia kell az `ActionListener` interfész – ez azt jelenti, hogy egy egyedi, `SzinesPanel` osztályt kell gyártanunk a `JPanel`-ből. Ennek a példánya azonban nem azonosíthatja a forrásobjektumokat, mert csak a keret ismeri a gombokat, a `SzinesPanel`-példány nem. Az eseménybe azonban nemcsak a forrásobjektum azonosítója van belefoglalva, hanem a gomb akcióparancsa is, s az alapértelmezésben meggyezik a felirattal. A `getActionCommand()` metódus tehát világossá teszi, hogy melyik gombon keletkezett az esemény. Ha a panelhez "Piros" parancs érkezik (/1), akkor pirosra festi a maga háttérszínét, a "Zöld" parancsra zöld színüre stb.

A `SzinesPanel` osztályt belső osztályként is deklarálhatnánk, és akkor már hivatkozhatnánk a keret által deklarált adatokra is.

## 8.5. Jelölőmező – JCheckBox

Csomag: `javax.swing` Deklaráció: `public class JCheckBox`

Közvetlen ős: `javax.swing.JToggleButton`

Fontosabb implementált interfészek: `SwingConstants`

A **jelölőmező** (check box, osztálya `JCheckBox`) egy kétállapotú gomb; egy mezőbe (ez egy négyzet és egy szöveg együtt) való kattintással vihető át egyik állapotból a másikba. A felhasználó egy logikai igaz vagy hamis állapotot jelölhet vele. Az üres mező a hamis értéknek felel meg, a (pipával vagy x-szel) kitöltött mező az igaz értéknek. A négyzeten vagy a mellette levő feliraton való kattintás az ellenkezőjére változtatja a jelölőmező értékét.

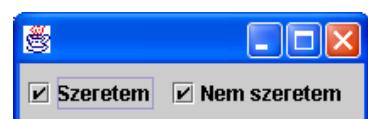
A jelölőmezőket (s bármiféle gombokat) a `ButtonGroup` segítségével csoportba lehet foglalni – a négyzet ekkor sem változik körré (a csoportosítást lásd a rádiogombnál). A jelölőmezőket azonban nem szokás csoportosítani. A jelölőmezőket megállapodás szerint **egymástól független** jelölésre szokás használni.

A `JCheckBox` őse az `AbstractButton` és a `JToggleButton` osztály.

A `JToggleButton` osztály a kétállapotú gombok közös őse, s önmagában is példányosítható. Vele most nem foglalkozunk.

### Minta jelölőmezők a keretben

```
cp.add(new JCheckBox("Szeretem", true));
cp.add(new JCheckBox("Nem szeretem", true));
```



### Események

AbstractButton-tól örökolt események: **ActionEvent**.

### Jellemzők

AbstractButton-tól örökolt jellemzők: `text`, `icon`, `mnemonic`, `selected`, `actionCommand`.

### Konstruktörök

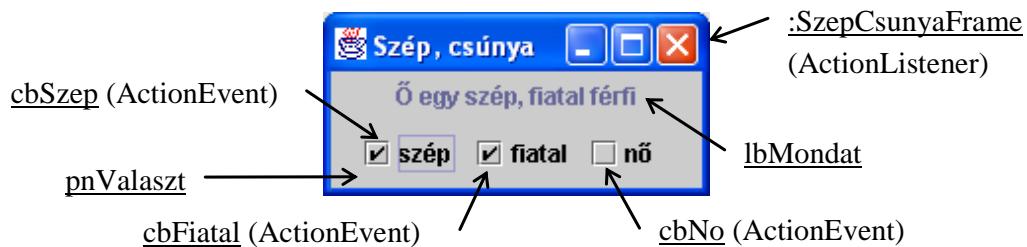
- ▶ `JCheckBox(String text, Icon icon, boolean selected)`
- ▶ `JCheckBox(String text, Icon icon)`
- ▶ `JCheckBox(String text, boolean selected)`
- ▶ `JCheckBox(String text)`
- ▶ `JCheckBox(Icon icon, boolean selected)`
- ▶ `JCheckBox(Icon icon)`
- ▶ `JCheckBox()`

Létrehozza a jelölőmezőt. A paraméterek jelentése:

- `text`: A jelölőmező felirata. Alapértelmezés: üres.
- `icon`: A jelölőmező ikonja. Alapértelmezés: `null` (nincs ikon).
- `selected`: A jelölőmező bejelölése. Alapértelmezés: `false` (nincs bejelölve).

### Feladat – CheckBoxTest

A keret alján 3 darab jelölőmező látható a következő szöveggel: szép, fiatal és nő. Bármelyik mezőt be lehet jelölni. A bejelölés alapján írjuk ki a keret felső részébe a képzeletbeli személy leírását!



### Forráskód

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class SzepCsunyaFrame extends JFrame implements ActionListener {
 Container cp = getContentPane();
 JLabel lbMondat;
 JCheckBox cbSzep, cbFiatal, cbNo;

```

```

public SzepCsunyaFrame() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 setLocation(100,100);
 setTitle("Szép, csúnya");
 cp.add(lbMondat =
 new JLabel(" ",JLabel.CENTER), "North"); //1

 JPanel pnValaszt = new JPanel(); //2
 pnValaszt.add(cbszep = new JCheckBox("szép",true));
 pnValaszt.add(cbfatal = new JCheckBox("fiatal",true));
 pnValaszt.add(cbno = new JCheckBox("nő"));
 cbSzep.addActionListener(this);
 cbFatial.addActionListener(this);
 cbNo.addActionListener(this);
 cp.add(pnValaszt,"South");
 pack();
 show();
 kiertekel();
}

void kiertekel() { //3
 String szepe = cbSzep.isSelected() ? "szép":"csúnya";
 String fiatale = cbFatial.isSelected() ? "fiatal":"öreg";
 String noe = cbNo.isSelected() ? "nő":"férfi";
 lbMondat.setText("Ó egy "+szepe+", "+fiatale+" "+noe);
}

public void actionPerformed(ActionEvent ev) { //4
 kiertekel();
}
}

public class CheckBoxTest {
 public static void main (String args[]) {
 new SzepCsunyaFrame();
 }
}

```

### A program elemzése

- ◆ //1-ben a keret középére tesszük a középre igazított címkét; az fogja majd megjeleníteni a mondatot.
- ◆ //2-től létrehozzuk a jelölőmezőket tartalmazó választópanelt (ennek alapértelmezés szerint sorfolytonos az elrendezésmenedzsere). Az első két jelölőmezőt bekattintjuk (szép és fiatal), a harmadik jelöletlen marad (nem nő). A keretet felfűzzük a három jelölőmező hallgatóláncára, és a keret déli részére helyezzük a panelt.
- ◆ //3-ban készítünk egy metódust: az fogja a jelölőmezők alapján összeállítani a mondatot. Az egyszerűség kedvéért minden megvizsgáljuk az összes mező állapotát. Amint megváltozott valamelyiké (keletkezett egy ActionEvent esemény), végignézzük a jelölőmezőket, és aszerint, hogy ki vannak-e választva (isSelected), a mondat részévé tesszük a nekik megfelelő szöveget.

## 8.6. Rádiogomb – JRadioButton, csoportosítás

Csomag: javax.swing Deklaráció: public class JRadioButton

Közvetlen ős: javax.swing.JToggleButton

Fontosabb implementált interfések: SwingConstants

A **rádiogomb** (radio button, osztálya JRadioButton) a jelölőmezőhöz hasonló kétállapotú gomb, de a négyzet helyett itt kört jelölünk be. A felhasználó egy logikai igaz vagy hamis állapotot jelölhet vele.

A rádiogombokat (és bármiféle más gombokat is) a ButtonGroup segítségével csoportba lehet foglalni. A rádiogombokat csoportosítani szokás. A rádiogombokat megállapodás szerint **egymástól függő** jelölésekre szokás használni.

A JRadioButton őse az AbstractButton és a JToggleButton osztály.

### Minta rádiogombok a keretben

```
cp.add(new JRadioButton("Szeretem", true));
cp.add(new JRadioButton("Nem szeretem"));
```



### Események

AbstractButton-tól örökolt események: **ActionEvent**.

### Jellemzők

AbstractButton-tól örökolt jellemzők: text, icon, mnemonic, selected, actionPerformed.

### Konstruktörök

- ▶ JRadioButton(String text, Icon icon, boolean selected)
- ▶ JRadioButton(String text, Icon icon)
- ▶ JRadioButton(String text, boolean selected)
- ▶ JRadioButton(String text)
- ▶ JRadioButton(Icon icon, boolean selected)
- ▶ JRadioButton(Icon icon)
- ▶ JRadioButton()

Létrehozza a rádiogombot. A paraméterek jelentése:

- text: A rádiogomb felirata. Alapértelmezés: "" (üres).
- icon: A rádiogomb ikonja. Alapértelmezés: null (nincs ikon).
- selected: A rádiogomb bejelölése. Alapértelmezés: false (nincs bejelölve).

## Gombok csoportosítása – ButtonGroup osztály

Csomag: javax.swing Deklaráció: public class ButtonGroup

Közvetlen ős: java.lang.Object

Fontosabb implementált interfészek: –

A ButtonGroup segítségével a gombok logikailag csoportosíthatók. A csoportba foglalt gombok közül egyszerre csak egy gomb jelölhető be. Ha kiválasztunk egy ilyen gombot, akkor a csoport többi gombja minden kiválasztatlan lesz. A gombcsoportok elemeit hagyományosan **rádiógomboknak** nevezzük, mert a rádió gombjai is így viselkednek.

Egy gombcsoportot úgy hozunk létre, hogy az egy csoportba teendő AbstractButton-utódokat hozzákötjük egy ButtonGroup objektumhoz: az felügyeli az így összefogott gombok ki- és bekapsolását.

A ButtonGroup az Object közvetlen leszármazottja; csak vezérlő szerepet játszik. A ButtonGroup nem komponens, nem keletkezik rajta esemény.

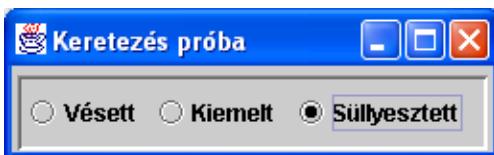
### Konstruktur, metódusok

- ▶ `ButtonGroup()`  
Létrehoz egy gombcsoportot.
- ▶ `void add(AbstractButton b)`  
Beletesz a gombcsoportba egy gombot.
- ▶ `int getButtonCount()`  
Megmondja, hány gomb van a csoportban.

*Megjegyzés:* Gombcsoport esetén hiába választunk ki egy gombot, ha aztán a gombcsoport egy másik elemét is kiválasztjuk – a második „kiugrasztja” az elsőt!

### Feladat – RadioButtonTest

A keretben 3 jelölőmező látszik, mindenhol egy-egy szegélyfajtát jelöl. Vegyük körül a gombok paneljét a kiválasztott szegéllyel! Egyszerre persze csak egy szegélyt lehet választani.



### Forráskód

```
import javax.swing.*;
import java.awt.*;
import java.awt.event;
```

```
class KeretezesFrame extends JFrame implements ActionListener {
 Container cp = getContentPane();
 ButtonGroup bg = new ButtonGroup(); //1
 JRadioButton rbVesett = new JRadioButton("Vésett"); //2
 JRadioButton rbKiemelt = new JRadioButton("Kiemelt");
 JRadioButton rbSullyesztett =
 new JRadioButton("Súllyesztett");
 JPanel pnValaszt;

 public KeretezesFrame() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 setTitle("Keretezés próba");

 cp.setLayout(new FlowLayout()); //Így jobban látszik a keret
 pnValaszt = new JPanel();

 bg.add(rbVesett); //3
 pnValaszt.add(rbVesett);
 rbVesett.addActionListener(this);

 bg.add(rbKiemelt);
 pnValaszt.add(rbKiemelt);
 rbKiemelt.addActionListener(this);

 bg.add(rbSullyesztett);
 pnValaszt.add(rbSullyesztett);
 rbSullyesztett.addActionListener(this);

 cp.add(pnValaszt);
 pack();
 show();
 }

 public void actionPerformed(ActionEvent ev) { //4
 JRadioButton rb = (JRadioButton)ev.getSource();
 javax.swing.border.Border border = null;
 if (rb.getText().equals("Vésett"))
 border = BorderFactory.createEtchedBorder();
 else if (rb.getText().equals("Kiemelt"))
 border = BorderFactory.createRaisedBevelBorder();
 else if (rb.getText().equals("Súllyesztett"))
 border = BorderFactory.createLoweredBevelBorder();
 pnValaszt.setBorder(border);
 }
}

public class RadioButtonTest {
 public static void main (String args[]) {
 new KeretezesFrame();
 }
}
```

### A program elemzése

- ◆ //1: Létrehozunk egy üres gombcsoportot – ebbe tesszük majd a rádiógombokat.
- ◆ //2: Létrehozzuk a három rádiógombot a megfelelő feliratokkal.
- ◆ //3: Sorban mindegyik rádiógombbal elvégezzük a következő három műveletet:
  - hozzáadjuk a gombcsoportot;
  - hozzáadjuk a panelhez. A gombcsoport csak logikai csoportosítás; a képernyőn a panel fogja össze a gombokat!
  - felfüzzük a keretet a gomb figyelőláncára. Mindhárom gombot figyelni kell!
- ◆ //4: A felhasználó bejelölte a gombcsoport valamelyik elemét, mégpedig `rb`-t, ami a kép szerint éppen az `rbSullyesztett`. A gombok paneljét a rádiógomb szövegének megfelelő szegéllyel vesszük körül.

Ha készítenénk egy  `addButton` metódust, akkor egyszerűbb lenne a gombokat inicializálni:

```
void addButton(AbstractButton button, JPanel pn, ButtonGroup bg) {
 bg.add(button);
 pn.add(button);
 button.addActionListener(this);
}
```

Így egyszerre intézhetnénk el a gombok minden „nyúgét”:

```
addButton(rbVesett, pnValaszt, bg);
addButton(rbKiemelt, pnValaszt, bg);
addButton(rbSullyesztett, pnValaszt, bg);
```

## 8.7. Kombinált lista – JComboBox

Csomag: `javax.swing` Deklaráció: `public class JComboBox`

Közvetlen ős: `javax.swing.JComponent`

Fontosabb implementált interfészek: –

**A kombinált lista** vagy kombilista (combined list, osztálya `JComboBox`) szerkeszthető szövegmező és legördülő lista kombinációja. A mező jobb oldalán látható nyilacska mutatja, hogy a listából választani lehet. Ha a nyilacsrá vagy a mezőre kattintunk, akkor legördülnek a lista objektumainak szöveges reprezentációi (`toString`). Rákattinthatunk valamelyikre vagy lenyomhatjuk az Enter-t, és márás kiválasztottuk az elemet. A szövegek között a le-fel billentyűvel is mozoghatunk. A mező tartalma a kiválasztott szöveg lesz.

### Minta választómező a keretben

```
JComboBox cb = new JComboBox();
cp.add(cb);
cb.addItem("Erika");
cb.addItem("Margó");
cb.addItem("Mari");
```



### Események

| <i>EsemOsztály</i> | <i>Mi történt?</i>                                                          | <i>Figyelőinterfész</i> | <i>Felfűzőmetódus</i> | <i>Eseménykezelő metódus</i> |
|--------------------|-----------------------------------------------------------------------------|-------------------------|-----------------------|------------------------------|
| ActionEvent        | Kattintottak rajta, vagy leütötték a szóközt/Enter-t amikor fókuszban volt. | ActionListener          | addActionListener     | actionPerformed              |

### Jellemzők

- ▶ `boolean editable`  
Ha értéke `true`, akkor a mező szerkeszthető, egyébként nem. Alapértelmezés: `false`.
- ▶ `int maximumRowCount`  
A gördítősáv nélkül legördülő sorok maximális száma. Ha az elemszám nagyobb ennél, akkor automatikusan megjelenik a függőleges gördítősáv. Alapértelmezés: 8.
- ▶ `boolean popupVisible`  
ha `true` az értéke, akkor a lista első megjelenésekor nyomban le is gördül. Csak a komponens megjelenítése után lehet `true`-ra állítani. Alapértelmezés: `false`.
- ▶ `Object selectedItem`  
A kiválasztott objektum.
- ▶ `int selectedIndex`  
A kiválasztott objektum pozíciója. Alapértelmezés: 0.

### Konstruktur

- ▶ `JComboBox()`
- ▶ `JComboBox(Object[] items)`
- ▶ `JComboBox(Vector items)`

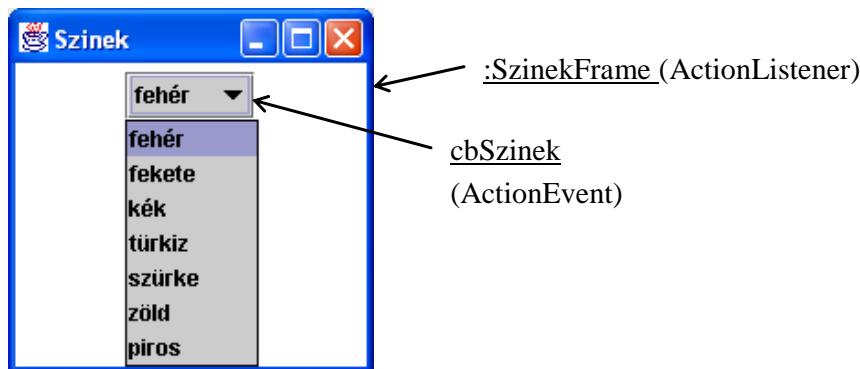
Létrehoz egy kombinált listát üresen, illetve az `items` elemekkel.

### Metódusok

- ▶ `void addItem(Object anObject)`
- ▶ `void insertItemAt(Object anObject, int index)`  
Elem hozzáadása a lista végéhez, illetve beszúrása a megadott pozícióba. A listában az elemek szöveges reprezentációja jelenik meg. Az elemek indexelése 0-ról indul.
- ▶ `void removeItem(Object anObject)`
- ▶ `void removeItemAt(int anIndex)`
- ▶ `void removeAllItems()`  
A megadott elem törlése (ha több egyforma van, akkor az első); a megadott pozíción levő szöveg törlése, illetve az összes választószöveg törlése.
- ▶ `void addActionListener(ActionListener l)`  
Felfűzi a figyelőt a komponens akciófigyelő láncára.

**Feladat – ComboBoxTest**

A keret felső részére tegyünk egy különböző színeket (fehéret, pirosat stb.) felkínáló kombinált listát! A keret háttérszíne mindenkor a kiválasztott szín legyen!

**Forráskód**

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class SzinekFrame extends JFrame implements ActionListener {
 Color[] colors = {Color.WHITE, Color.BLACK, Color.BLUE, //1
 Color.CYAN, Color.GRAY, Color.GREEN, Color.RED};
 String[] colorStrings = {"fehér", "fekete", "kék",
 "türkiz", "szürke", "zöld", "piros"};
 Container cp = getContentPane();
 JComboBox cbSzinek;

 public SzinekFrame() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 setTitle("Szinek");
 setBounds(300, 100, 200, 200);
 cp.setLayout(new FlowLayout());
 cp.add(cbSzinek = new JComboBox(colorStrings)); //2
 cbSzinek.addActionListener(this);
 show();
 }

 public void actionPerformed(ActionEvent ev) { //3
 cp.setBackground(colors[cbSzinek.getSelectedIndex()]);
 }
}

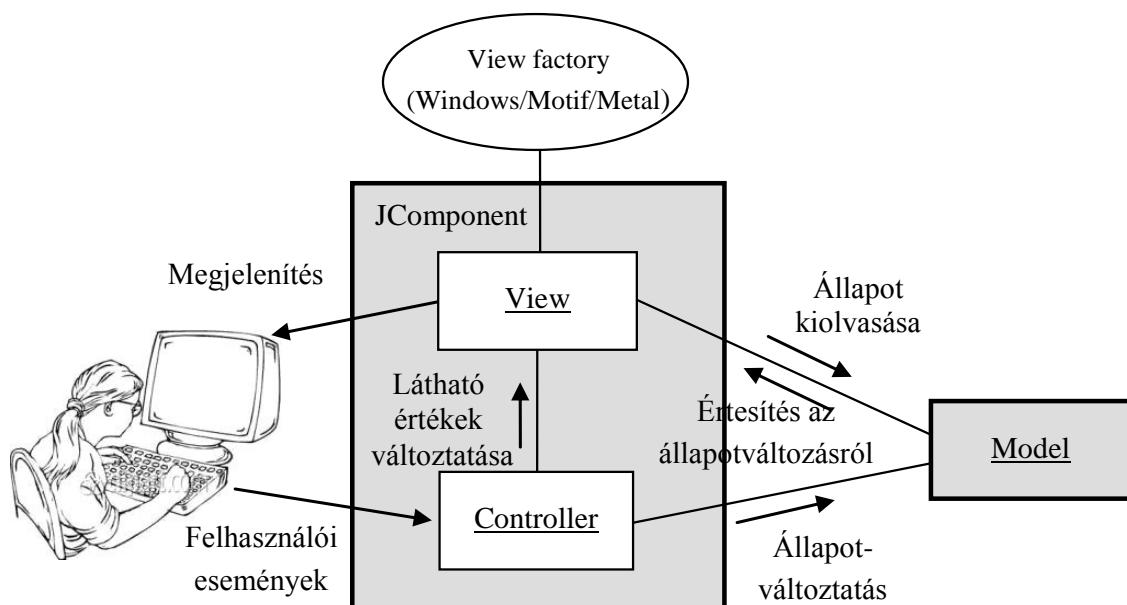
public class ComboBoxTest {
 public static void main (String args[]) {
 new SzinekFrame();
 }
}

```

### A program elemzése

- ◆ //1-től kezdve két tömböt deklarálunk: `colors` a kiválasztandó színek konstansát, `colorStrings` pedig sorban a színek magyar nevét tartalmazza.
- ◆ //2-ben A `cbSzinek` választómezőbe indulásképpen a `colorStrings` tömb elemei kerülnek. A választómezőt ráteszük a tartalompanelre.
- ◆ //3-ban kezeljük az `ActionEvent` eseményt: kiolvassuk a kiválasztott mező indexét, és a `colors` tömb ilyen indexű színével beszínezzük a tartalompanelt.

## 8.8. MVC-modell, dokumentumkezelés



8.3. ábra. MVC-modell

A Swing komponenseket az MVC (model, view, controller) architektúra alapján tervezték meg. A három összetevő jelentése a következő:

- ◆ **Model** (modell): a komponens adatai, állapota. A modell dolga a komponens adatainak tárolása. Egy modellen így több nézet is osztozhhat.
- ◆ **View** (nézet): a komponens megjelenése a képernyőn. Így könnyebb kicserálni a komponensek küllemét („look and feel”-jét). A speciális küllemet bőrnek (skin) is szokás nevezni. A JBuilder környezetben például a *Tools/IDE Options.../Look and feel* pontban választani lehet a különböző operációs rendszereknek megfelelő bőrből:

Windows, Motif (Solaris) vagy Metal (a Sun által kitalált Java-bőr). Bőrt programból is lehet váltani.

- ◆ **Controller** (vezérlő): a komponens viselkedése. A vezérlő szabályozza a külvilág eseményeire való reagálást.

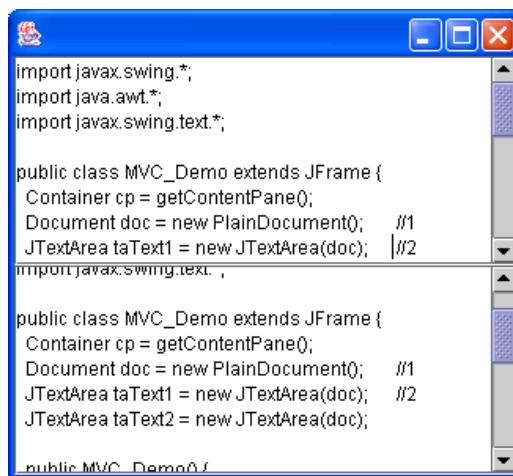
Az MVC-architektúra működési elvét a 8.3. ábra mutatja. A vezérlő (Controller) kezeli a felhasználói eseményeket, például az egérkattintást vagy a billentyűlétfést. A történtek alapján eldönti, hogy megváltoztatja-e a modellt és/vagy a nézetet. A modell értesítést küld a nézetnek, ha megváltozott volna. Az értesítés hatására a nézet kiolvassa a modellből az adatokat.

A Javában a nézetet (view) és a viselkedést (controller) összeépítették: a komponensbe beépítették az eseménykezelést.

A legtöbb Swing-komponenst az MVC-modell alapján készítették. Ez azt jelenti, hogy a komponens mögött mindenkor „lapul” egy modellobjektum. A modell sokszor a háttérben dolgozik, és a programozó nem is vesz róla tudomást. Némelyik komponensben azonban bizonyos tulajdonságok csak a modellen keresztül érhetők el. A szövegterület (JTextArea) komponens mögött például egy PlainDocument dokumentum áll: azon keletkezik a szövegváltozáshoz kapcsolódó esemény. Előfordulhat, hogy két szövegterületnek ugyanaz a dokumentuma; lássunk erre egy példát:

### Feladat – MVC-demo

Tegyünk rá a keretre két szövegterületet, és kössük mindenkorhoz a dokumentummodellhez!



```
import javax.swing.*;
import java.awt.*;
import javax.swing.text.*;

public class MVC_Demo extends JFrame {
 Container cp = getContentPane();
 Document doc = new PlainDocument(); //1
 JTextArea taText1 = new JTextArea(doc); //2
 import javax.swing.text.;

 public class MVC_Demo extends JFrame {
 Container cp = getContentPane();
 Document doc = new PlainDocument(); //1
 JTextArea taText1 = new JTextArea(doc); //2
 JTextArea taText2 = new JTextArea(doc);

 public MVC_Demo() {

```

A doc:PlainDocument modell tárolja a dokumentum adatait (//1). A keretre ráteszünk két szövegterületet (//2), s mindenkorhoz a doc dokumentumot „kötjük”. A két szövegterület tehát ugyanazt a dokumentumot jeleníti meg. Figyelje meg, hogy amit az egyik területen írunk, az a másikon is megjelenik! A két terület egymástól függetlenül görgethető, és külön kurzoruk is van.

**Forráskód**

```

import javax.swing.*;
import java.awt.*;
import javax.swing.text.*;

public class MVC_Demo extends JFrame {
 Container cp = getContentPane();
 Document doc = new PlainDocument(); //1
 JTextArea taText1 = new JTextArea(doc); //2
 JTextArea taText2 = new JTextArea(doc);

 public MVC_Demo() {
 setBounds(100,100,500,300);
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 cp.setLayout(new GridLayout(2,1));
 cp.add(new JScrollPane(taText1));
 cp.add(new JScrollPane(taText2));
 show();
 }

 public static void main(String[] args) {
 new MVC_Demo();
 }
}

```

A `JTextArea` modellje a `Document` interfész implementálja. A Javában rengeteg modellinterfész definiáltak, adatbeállító és kiolvasó metódusokkal. Példaként lássunk néhányat:

| Modellinterfész     | Metódusai (néhány)                                             | Mi használja?                               | Alapértelm. osztály                     |
|---------------------|----------------------------------------------------------------|---------------------------------------------|-----------------------------------------|
| Document            | addDocumentListener<br>getLength<br>getText                    | (JTextComponent)<br>JTextField<br>JTextArea | (AbstractDocument)<br>PlainDocument     |
| ListModel           | addListDataListener<br>getElementAt<br>getSize                 | JList                                       | (AbstractListModel)<br>DefaultListModel |
| ListSelection-Model | addListSelectionListener<br>clearSelection<br>getSelectionMode | JList                                       | DefaultListSelection-Model              |

Mindegyik modellinterfészhez készült egy-egy alapértelmezés szerinti modellosztály. A `Document` interfész például az `AbstractDocument` osztály implementálja, s annak a `PlainDocument` az egyik konkrét leszármazottja – a `JTextArea` alapértelmezés szerinti modellje. A modell kicserélhető. A modellekben események keletkezhetnek, s ennek megfelelően vannak figyelőláncok is. A dokumentumon (`Document`) például `DocumentEvent` esemény keletkezik, ha megváltozik a szöveg.

A programozó a komponenssel van közvetlen kapcsolatban; a modell adatait sokszor csak közvetett kiolvasással kaphatja meg. Szerencsére bizonyos adatok eléréséhez a komponensben csatolófelületet (interfész) adnak; ilyenkor a kérést a komponens továbbítja az adatmodell-jéhez. A következő két utasítás például egyenértékű:

```
int hossz = taText1.getDocument().getLength();
int hossz = taText1.getText().length();
```

A dokumentumesemények a modellen keletkeznek, az eseményfigyelőt a modellre kell tehát felfűzni:

```
taText1.getDocument().addDocumentListener(this);
```

## 8.9. A szövegek űse – JTextComponent

Csomag: javax.swing.text Deklaráció: public abstract class JTextComponent

Közvetlen űs: javax.swing.JComponent

Fontosabb implementált interfések: –

A JTextComponent absztrakt osztály a szövegkomponensek közös tulajdonságait tartalmazza. A következő osztályhierarchián néhány utódja látható:

```
JTextComponent
 +--JTextField // Szövegmező
 | +--JPasswordField // Jelszómező (ezt nem tárgyaljuk)
 +--JTextArea // Szövegterület
```

A fenti osztályokban megvannak tehát a JTextComponent jellemzői és metódusai. Ezek szerint minden szövegkomponensnek van dokumentummodellje, kurzora stb. A JTextComponent-nek csak beszúró üzemmódja van; a felülíró üzemmódot programban kell megírni.

A JTextComponent-nek PlainDocument az adatmodellje (az AbstractDocument utódja). Az adatmodell dokumentumeseményt (DocumentEvent) bocsát ki a szöveg megváltozásakor. A DocumentEvent eseménykezelői:

- insertUpdate: beszúrtak egy karaktert;
- removeUpdate: töröltek egy karaktert;
- changedUpdate: megváltozott a stílus (PlainDocument-en nem keletkezik ilyen esemény).

### Események (az adatmodellen)

| Esemény       | Mi történt?            | Figyelőinterfész | Felfűzőmetódus      | Eseménykezelők                                |
|---------------|------------------------|------------------|---------------------|-----------------------------------------------|
| DocumentEvent | Megváltozott a szöveg. | DocumentListener | addDocumentListener | insertUpdate<br>removeUpdate<br>changedUpdate |

A DocumentEvent interfész (metódusai a getDocument, getType stb.); a dokumentumon keletkezett konkrét esemény implementálja. A getDocument az esemény forrását mondja meg, a getType a típusát (INSERT, REMOVE, CHANGED). A DocumentEvent nem leszármazottja az EventObject-nek.

## Jellemzők

- ▶ Document document
  - A komponens adatmodellje. Alapértelmezés: PlainDocument
  - ▶ String text
  - A komponens aktuális szövege (az adatmodelltől kéri el). Alapértelmezés: "" (üres)
  - ▶ Caret caret
  - ▶ int caretPosition
  - ▶ Color caretColor
  - A dokumentum szöveges kurzora (nem egérkurzor!); a kurzor pozíciója és színe. A kurzor mindenkor két betű között áll, és pozícióján az előtte levő betűk száma értendő.
  - ▶ boolean editable
  - Megadja, hogy a komponens szerkeszthető-e. Ha nem szerkeszthető, akkor a billentyűleütésnek nincs hatása. Alapértelmezés: true
  - ▶ int selectionStart
  - ▶ int selectionEnd
  - ▶ Color selectionColor
  - ▶ Color selectedTextColor
  - A kiválasztott szöveg jellemzői: első és utolsó pozíció (0-tól számítva), a szöveg háttér- és betűszíne.

## AbstractDocument osztály

Csomag: javax.swing.text Deklaráció: public abstract class AbstractDocument  
Közvetlen ős: java.lang.Object  
Fontosabb implementált interfések: Document

## Metódusok

- ▶ void addDocumentListener(DocumentListener l)  
Objektum felfűzése a figyelőláncra.
  - ▶ String getText(int offset, int length)  
Visszaadja a length hosszúságú részláncot az offset pozíciótól kevésbé.
  - ▶ int getLength()  
Visszaadja a szöveg hosszát.

Az `AbstractDocument` egyik leszármazottja, a `PlainDocument` a `JTextField` és a `JTextArea` modellje. A dokumentumkezelésre a `JTextArea` komponens bemutatásakor láthatunk példát.

## 8.10. Szövegmező – JTextField

Csomag: `javax.swing` Deklaráció: `public class JTextField`

Közvetlen ős: `javax.swing.text.JTextComponent`

Fontosabb implementált interfések: `SwingConstants`

**A szövegmező** vagy beviteli mező (text field, osztálya `JTextField`) olyan egysoros komponens, amelybe a felhasználó szöveget írhat.

A szöveg szerkesztéséhez használhatók a megszokott szerkesztőműveletek: navigálás, beszúrás, törlés, blokkműveletek stb. Bármilyen hosszú szöveg beírható; ha a szöveg hosszabb a mezőnél, akkor görgethetővé válik. Az oszlopok számát létrehozáskor lehet megadni, egyébként a kezdeti szöveg méretéhez igazodik. Az aktuális szöveg lekérdezhető, illetve beállítható a programból. A szöveg a megjelenítési területen belül vízszintesen igazítható balra, jobbra vagy középre.

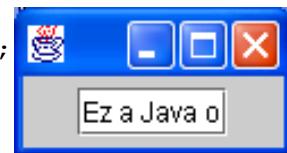
Szövegmező szélessége (`getWidth`) = oszlopszám (`getColumns`) \* oszlopszélesség (`getColumnWidth`); az oszlopszélesség alapértelmezésben az m betű szélessége.

A `JTextField` őse az absztrakt `JTextComponent` osztály. A szövegmezőnek van egy `Document` típusú adatmodellje.

Az adatmodell lényeges elemei elérhetők a komponensen keresztül, ezért az adatmodellt itt nem tárgyaljuk. A `Document` adatmodellre és a `DocumentEvent` eseményre a `JTextArea` komponens ismertetésekor látunk majd példát.

### Minta szövegmező a keretben

```
cp.add(new JTextField("Ez a Java olyan jó", 10));
```



### Események

A `JTextComponent`-től örökolt események: `DocumentEvent` (adatmodellen).

*EsemOsztály* *Mi történt?*

*Figyelőinterfész*

*Felfűzőmetódus*

*Eseménykez. met.*

| <i>EsemOsztály</i> | <i>Mi történt?</i>                           | <i>Figyelőinterfész</i> | <i>Felfűzőmetódus</i> | <i>Eseménykez. met.</i> |
|--------------------|----------------------------------------------|-------------------------|-----------------------|-------------------------|
| ActionEvent        | Leütötték az Enter-t, amikor fókuszból volt. | ActionListener          | addActionListener     | actionPerformed         |

### Jellemzők

A JTextField-től örökölték: document, text, caret, caretPosition, caretColor, editable, selectionStart, selectionEnd, selectionColor, selectedTextColor

- ▶ int columns  
A mező látható oszlopainak száma. Alapértelmezés: text hossza.
- ▶ int horizontalAlignment  
A szöveg vízszintes igazítása a megjelenítési területen: LEFT, RIGHT, CENTER. Alapértelmezés: LEFT.

### Konstruktörök

- ▶ JTextField(Document document, String text, int columns)
- ▶ JTextField(String text, int columns)
- ▶ JTextField(String text)
- ▶ JTextField(int columns)
- ▶ JTextField()

Létrehoz egy szövegmező-objektumot. A paraméterek jelentését lásd a jellemzőknél.

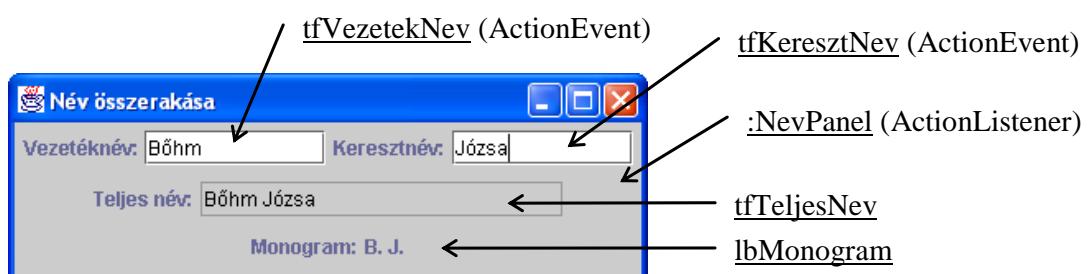
### Metódusok

- ▶ void addActionListener(ActionListener l)  
Elfűzi a figyelőt a szövegmező akciófigyelő láncára.

#### Feladat – TextFieldTest

Készítsünk egy névösszerakó alkalmazást az ábrán látható módon! A keret felső sorában beírható a vezeték- és keresztnév. Ha bármelyik szövegmezőn leütjük az Entert, akkor megjelenik

- a középső sorban egy nem szerkeszthető szövegmezőben a teljes név,
- az alsó sorban pedig a monogram egy címkén.



### Forráskód

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
class NevPanel extends JPanel implements ActionListener { //1
 JTextField tfVezetekNev, tfKeresztNev, tfTeljesNev;
 JLabel lbMonogram;
 public NevPanel() {
 setLayout(new GridLayout(3,1));

 JPanel pn = new JPanel(); //2
 pn.add(new JLabel("Vezetéknév:"));
 pn.add(tfVezetekNev = new JTextField(10));
 pn.add(new JLabel("Keresztnév:"));
 pn.add(tfKeresztNev = new JTextField(10));
 add(pn);

 pn = new JPanel();
 pn.add(new JLabel("Teljes név:"));
 pn.add(tfTeljesNev = new JTextField(20));
 tfTeljesNev.setEditable(false);
 add(pn);

 pn = new JPanel();
 pn.add(new JLabel("Monogram:"));
 pn.add(lbMonogram = new JLabel(""));
 add(pn);

 tfVezetekNev.addActionListener(this); //3
 tfKeresztNev.addActionListener(this);
 }

 public void actionPerformed(ActionEvent ev) { //4
 String vez = tfVezetekNev.getText();
 String ker = tfKeresztNev.getText();
 tfTeljesNev.setText(vez+" "+ker);
 try {
 lbMonogram.setText(vez.charAt(0)+"."+ker.charAt(0)+".");
 }
 catch (StringIndexOutOfBoundsException ex) {
 lbMonogram.setText("");
 }
 }
}

public class TextFieldTest extends JFrame {
 public TextFieldTest() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 setTitle("Név összerakása");
 setLocation(300,200);
 getContentPane().add(new NevPanel());
 pack();
 show();
 }

 public static void main (String[] args) {
 new TextFieldTest();
 } // main
} // TextFieldTest
```

### A program elemzése

- ◆ //1: A NevPanel lesz az ActionEvent események figyelője, benne van meg az actionPerformed eseménykezelő metódus.
- ◆ //2: Összeállítjuk a NevPanel tulajdonosi hierarchiát. A szövegmezők elől egy-egy címkét teszünk. A szövegmezőket és a monogramot deklaráljuk, mert az eseménykezelőben hivatkozni fogunk rájuk.
- ◆ //3: Csak a két nevet fogjuk figyelni.
- ◆ //4: Kivesszük a tfVezetekNev és a tfKeresztNev objektumból a szöveget. Összadjuk a két nevet, s azt értékül adjuk a tfTeljesNev-nek; összeállítjuk a monogramot is, az lesz az lbMonogram értéke. Ha nem ütik be valamelyik nevet, akkor a charAt metódus kivételt ejthet. Ezt a kivételt elkapjuk és üresnek vesszük a monogramot.

## 8.11. Szövegterület – JTextArea

Csomag: javax.swing Deklaráció: public class JTextArea

Közvetlen ős: javax.swing.JTextComponent

Fontosabb implementált interfészek: –

**A szövegterület** (text area, osztálya JTextArea) többsoros beviteli mező. A JTextArea szövege egyetlen String, a benne levő \n karakter soremelésként jelenik meg a szövegterületen. A szövegterülethez szöveget lehet hozzáadni (append), szöveget lehet beleszűrni (insert), illetve szöveget lehet benne cserélni és kitörölni belőle (replaceRange).

A szövegterületnek nincs görgetősávja. Ha a szöveget görgetni akarjuk, előbb rá kell tenni egy JScrollPane (görgetőlap) komponensre, s azt kell majd rátenni a tartalompanelre:

```
JTextArea ta = new JTextArea();
cp.add(new JScrollPane(ta));
```

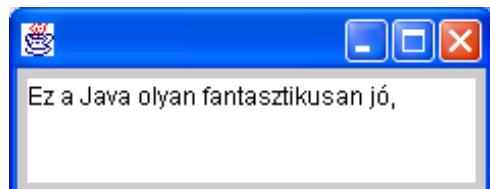
A görgetősávok automatikusan, de csak szükség esetén jelennek meg.

A JTextArea őse az absztrakt JTextComponent osztály. A szövegmezőnek van egy Document adatmodellje, osztálya az AbstractDocument utódja.

Ha rossz pozíciót adunk meg, akkor az osztály metódusai BadLocationException kivételt ejthetnek.

### Minta szövegterület a keretben

```
cp.add(new JTextArea(
 "Ez a Java olyan "+
 "fantasztikusan jó, ", 5, 20));
```



## Események

A `JTextComponent`-től örökolt események: `DocumentEvent` (adatmodellen).

A komponensnek nincs saját, magas szintű eseménye.

## Jellemzők

A `JTextComponent`-től örököltek: `document`, `text`, `caret`, `caretPosition`, `caretColor`, `editable`, `selectionStart`, `selectionEnd`, `selectionColor`, `selectedTextColor`

- ▶ `int columns`
- ▶ `int rows`

A szövegterület látható oszlopainak száma (szélesség), és sorainak száma (magasság).

Alapértelmezés: 0, 0

- ▶ `int tabSize`

Ennyi oszlopot ugrik a kurzor a TAB billentyű lenyomására. Alapértelmezés: 8.

- ▶ `boolean lineWrap`

Sortörés. Ha `true` az értéke és a sor nem fér be a szövegterület látható részébe, akkor a sor automatikusan megtörök. Alapértelmezés: `false`.

- ▶ `boolean wrapStyleWord`

Csak akkor van hatása, ha `lineWrap=true`. Ha értéke `true`, akkor nem töri ketté a szavakat. Alapértelmezés: `false`.

## Konstruktörök

- ▶ `JTextArea(Document document, String text, int rows, int columns)`
- ▶ `JTextArea(String text, int rows, int columns)`
- ▶ `JTextArea(int rows, int columns)`
- ▶ `JTextArea(String text)`
- ▶ `JTextArea(Document document)`
- ▶ `JTextArea()`

Létrehoz egy szövegterület objektumot. A paraméterek jelentését lásd a jellemzőknél.

## Metódusok (ha `document==null`, nem történik semmi)

- ▶ `void insert(String str, int pos)`  
Beszúrja az `str` szöveget a `pos` pozíciótól kezdve.
- ▶ `void append(String str)`  
`str` hozzáadása a szöveg végéhez.
- ▶ `void replaceRange(String str, int start, int end)`  
Kicseréli a `start` és az `end-1` pozíció közti szöveget `str`-re. Szövegrészt törölni üres karakterláncjal való kicseréléssel lehet (vagy az adatmodellen keresztül).
- ▶ `int getLineCount()`  
Visszaadja a szöveg sorainak számát.

- ▶ int getLineStartOffset(int line)
- ▶ int getLineEndOffset(int line)
 

Visszaadja a szöveg line sorszámú sorának első és utolsó utáni pozícióját.
- ▶ int getLineOfOffset(int offset)
 

Visszaadja azt a sorszámot, amelyben ez a pozíció van.

### Feladat – TextAreaTest

Szerkesszünk meg egy önéletrajzot! Az önéletrajz egy 6 soros, 30 oszlopnyi, szükség esetén görgethető szövegterületen van. A szövegterület felett áll az önéletrajz írójának neve, alatta pedig egy „Nyugtáz” gomb. Tegyünk a szövegterületre egy kezdőszöveget, és állítsuk a kurzort a szöveg végére!

Amikor elkezdjük szerkeszteni a szöveget, a szövegterület feletti név válton dőlt betűsre. A „Nyugtáz” gomb lenyomására nyugtázzuk a szöveget; ekkor a név egyenesedjék ki. De mi helyt megváltozik a szöveg, a név megint dőlt betűvel íródjon ki. A nyugtázással a szövegterület ne veszítse el a fókuszt!



### Forráskód

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.event.*;

class Oneletrajz extends JPanel implements
 ActionListener, DocumentListener {
 JLabel lbNev;
 JTextArea taCV; // Curriculum Vitae = Önéletrajz (latin)
 JButton btNyugtaz;
 Font fntItalic = new Font("Dialog", Font.ITALIC+Font.BOLD, 12);
 Font fntNormal = new Font("Dialog", Font.PLAIN+Font.BOLD, 12);

 public Oneletrajz(String nev) { //1
 setLayout(new BorderLayout());
 add(lbNev = new JLabel(nev), "North");
 lbNev.setFont(fntNormal);
 }
}

```

```

taCV = new JTextArea(6,30);
taCV.setText("Önéletrajz\n\nNév: " + nev + "\n\n");
add(new JScrollPane(taCV));
add(btNyugtaz = new JButton("Nyugtáz"), "South");

taCV.getDocument().addDocumentListener(this); //2
btNyugtaz.addActionListener(this);
taCV.setCaretPosition(taCV.getText().length()); //3
}

public void actionPerformed(ActionEvent ev) { //4
 lbNev.setFont(fntNormal);
 taCV.requestFocus();
}

public void insertUpdate(DocumentEvent e) { //5
 lbNev.setFont(fntItalic); // beszúrtak egy karaktert
}

public void removeUpdate(DocumentEvent e) { //6
 lbNev.setFont(fntItalic); // töröltek egy karaktert
}

public void changedUpdate(DocumentEvent e) { //7
}
}

public class TextAreaTest JFrame {
 public TextAreaTest() {
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 setTitle("Önéletrajz");
 setLocation(300,200);
 getContentPane().add(new Oneletrajz("Dudás Mari"));
 pack();
 show();
 }

 public static void main (String args[]) {
 new TextAreaTest();
 } // main
} // TextAreaTest

```

### A program elemzése

- ◆ //1: Az OneletRajz osztály konstruktőrben összállítjuk a keretbe helyezendő panelt. Az önéletrajzhoz tartozó nevet – az a felső címke szövege lesz – paraméterben adjuk meg. Középre tesszük a taCV szövegterületet egy görgetőlapra. Kezdőszövegként beletesszük többek között a nevet is. Délre tesszük a nyugtázógombot.
- ◆ //2: A taCV-tól szövegváltozási eseményt várunk, a nyomógombtól akcióeseményt. A DocumentEvent nem a komponenstől jön, hanem az adatmodelljétől – arra kell tehát felfűzni a figyelőt: `getDocument().addDocumentListener(this)`. A terven nem jelöljük, hogy az esemény az adatmodelltől ered.
- ◆ //3: A kurzort a szöveg utolsó karaktere után állítjuk.

- ◆ //4: Ha megnyomják a Nyugtáz gombot, akkor kiegynenesítjük a címke fontját. Mivel a fókusz a gombra került, visszavisszük a szövegterületre.
- ◆ //5-7: Ezek a DocumentEvent kezelőmetódusai. Ha a szövegbe beszúrtak vagy onnan töröltek egy karaktert, akkor elfordítjük a nevet. A changedUpdate a stíluskarakterek változására vonatkozik; a JTextArea-ban nincs ilyen.

## 8.12. Lista – JList

Csomag: javax.swing Deklaráció: public class JList

Közvetlen ős: javax.swing.JComponent

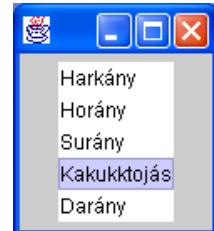
Fontosabb implementált interfések: –

A lista (list, osztálya JList) választható szövegeket tartalmaz. A lista nem gördül le, hanem egy rögzített területen helyezkedik el. A listából egyszerre több elemet is ki lehet választani (hacsak külön meg nem tiltjuk). Ha a lista egy tételen egérrel kattintunk, akkor listakiválasztási esemény (ListSelectionEvent) keletkezik.

A komponensnek van egyListModel adatmodellje. A kiválasztást egy ListSelectionModel végzi, de a kiválasztási funkciók többsége a JList metódusain keresztül is elérhető.

### Minta lista a keretben

```
JList lista;
cp.add(lista=new JList(new String[] {"Harkány",
 "Horány","Surány","Kakukktojás","Darány"}));
lista.setSelectedValue("Kakukktojás",true);
```



A JList-en keresztül a listát csak olvasni lehet; ha a programban módosítani akarjuk a listát, akkor saját magunknak kell elkészítenünk az adatmodellt, és hozzákapcsolnunk a listához:

```
DefaultListModel model = new DefaultListModel();
JList lista = new JList(model);
model.addElement("Hosszúhetény");
int size = model.getSize();
```

### Események

| Esem/Osztály        | Mi történt?                       | Figyelőinterfész       | Felfűzőmetódus            | Eseménykez. met. |
|---------------------|-----------------------------------|------------------------|---------------------------|------------------|
| ListSelection-Event | A kiválasztott elem megváltozott. | ListSelection-Listener | addListSelection-Listener | valueChanged     |

### Jellemzők

- ListModel model

Az osztály adatmodellje. Alapértelmezésben DefaultListModel, lásd később.

► `int selectionMode`

Kiválasztási mód. Lehetőségek (a `ListSelectionModel` konstansai):

- `SINGLE_SELECTION`: Egyszerre csak egy listaelem jelölhető ki.
- `SINGLE_INTERVAL_SELECTION`: Egyetlen összefüggő intervallum jelölhető ki.
- `MULTIPLE_INTERVAL_SELECTION`: Bármi kijelölhető. (Ez az alapértelmezés.)

► `int selectedIndex`

Az első kiválasztott elem indexe. Ha nincs ilyen, akkor értéke -1. Beállításkor a többi kiválasztást kioltja. Csak akkor van hatása, ha a lista látható. Alapértelmezés: -1

► `int visibleRowCount`

A lista látható sorainak száma (az esetleges görgetősávot is beszámítva). Alapértelmezés: 8

► `int fixedCellWidth`

► `int fixedCellHeight`

A lista celláinak (sorainak) szélessége, illetve magassága pontokban.

### Konstruktörök

- `JList(ListModel dataModel)`
- `JList(Object[] listData)`
- `JList(Vector listData)`
- `JList()`

Létrehoz egy listakomponenst a megadott adatmodellel, és megjeleníti a modell elemeinek szöveges reprezentációját. Nincs kiválasztva semmi. Paraméterek:

- `dataModel`: A `JList` adatmodellje. Alapértelmezés: `DefaultListModel`
- `listData`: A `dataModel`-be teendő elemek. Alapértelmezés: üres.

### Metódusok

- `void addListSelectionListener(ListSelectionListener l)`  
Felfűzi a figyelőt a komponens listakiválasztást figyelő láncára.
- `void setListData(Object[] listData)`
- `void setListData(Vector listData)`  
Az adatmodell elemeit a megadottakkal cseréli fel.
- `void setSelectedValue(Object anObject, boolean shouldScroll)`  
Kiválasztja az `anObject` elemet. Ha `shouldScroll` értéke `true`, akkor a lista úgy gördül, hogy a kiválasztott elem mindenképpen látszódjék.
- `Object getSelectedValue()`
- `Object[] getSelectedValues()`  
Az első visszaadja az első kiválasztott értéket (objektumot); ha nincs ilyen, akkor a visszaadott érték `null`. A második visszaadja az összes kiválasztott objektumot.
- `void setSelectedIndices(int[] indices)`
- `int[] getSelectedIndices()`  
Egyszerre több elemet is kiválaszt, illetve visszaadja a kiválasztott elemek indexét.

- ▶ `boolean isSelectedIndex(int index)`
- ▶ `boolean isSelectionEmpty()`

Megmondja, hogy az `index` sorszámú elem ki van-e választva, illetve hogy van-e egyáltalán kiválasztás.

- ▶ `void clearSelection()`  
Töröl minden kiválasztást.
- ▶ `int getFirstVisibleIndex()`
- ▶ `int getLastVisibleIndex()`

Visszaadja az első, illetve az utolsó látható elem indexét.

### **DefaultListModel osztály**

Csomag: `javax.swing` Deklaráció: `public class DefaultListModel`

Közvetlen ős: `javax.swing.AbstractListModel`

Fontosabb implementált interfések: `ListModel`

Az osztály a `JList` alapértelmezés szerinti adatmodellje. A metódusok hasonlítanak a `Vector` metódusaihoz. A `ListModel` interfészben csak néhány van meg közülük.

- ▶ `DefaultListModel()`  
Létrehoz egy lista modellt.
- ▶ `int getSize()`  
Visszaadja a modell elemeinek a számát.
- ▶ `void add(int index, Object elem)`
- ▶ `void addElement(Object elem)`  
Beszűr egy elemet a megadott indexű helyre, illetve a lista végére.
- ▶ `boolean contains(Object elem)`
- ▶ `Object firstElement()`
- ▶ `Object lastElement()`
- ▶ `Object get(int index)`
- ▶ `int indexOf(Object elem)`  
Tartalmazásvizsgálat; első, illetve utolsó elem kikérése; a megadott indexű elem kikérése; a megadott elem indexének kikérése.
- ▶ `void clear()`
- ▶ `boolean removeElement(Object obj)`
- ▶ `Object remove(int index)`  
A lista törlése; az adott (indexű) elem törlése.

### **Egyszerű beviteli dialógusablak**

A következő feladatban nagyon kényelmes szövegbeviteli párbeszédablakot fogunk használni.

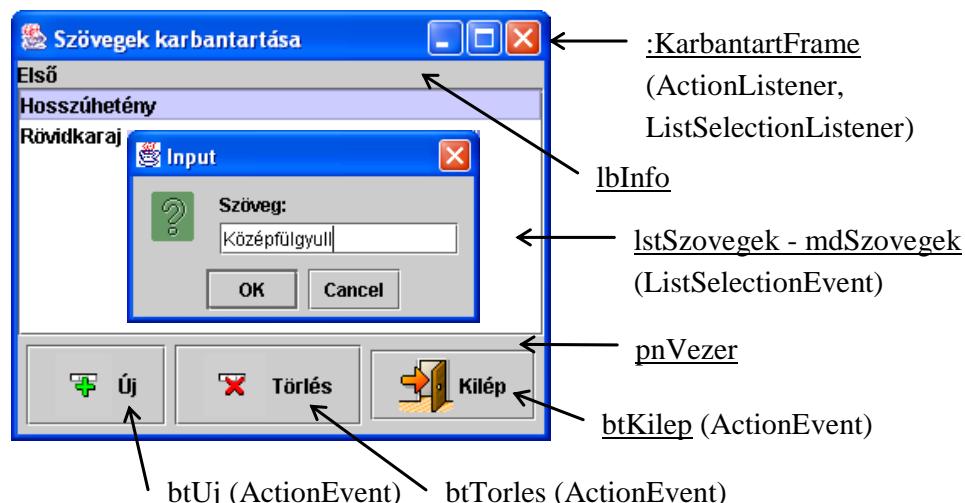
A `JOptionPane` osztály sok statikus metódust kínál fel különböző egyszerű és szabványos fehasználói bevitelre. A következő metódus szöveg bevitelére használható:

- static String showInputDialog(Component parentComp, Object message)
- Egy beviteli dialógust jelenít meg. parentComp a dialógus tulajdonosa, message az ablakban megjelenő információs szöveg. Ha a felhasználó az OK gombot nyomja le, akkor a visszatérési érték a bevitt szöveg, ha a Cancel gombot, akkor null. Például:
- ```
String szoveg = JOptionPane.showInputDialog(this, "Szöveg:");
```

A JOptionPane osztályról részletesebben e fejezet Dialógusablak pontjában lesz szó.

Feladat – Szövegek karbantartása – ListTest

Készítsünk egy olyan alkalmazást, amellyel új szövegeket lehet felvinni, s ki lehet törölni a már meglévőket. Az új szöveget egy kis dialógus segítségével kérjük be. Írjuk ki a felső sorba, ha az első vagy az utolsó szöveg van-e kiválasztva!



A felhasználói interfész terve

Át kell gondolnunk a keret (tartalompanel) felépítését és az eseménykezelést:

- ♦ **A keret felépítése:** A keret alapjában három részre bomlik: felül az információs címke (lbInfo), középen a szövegeket tartalmazó lista (lstSzovegek). A listát egy görgetőlapra tesszük, hogy ha az elemek száma ezt megkívánja, akkor a lista automatikusan görögöljön. A lista adatmodellje az mdSzovegek. Alul van a vezérlősor (pnVezer); arra tesszük egymás után a nyomógombokat. A komponensek (vagy osztályuk) azonosítója a képen látható.
- ♦ **Eseménykezelés:** Végig kell gondolnunk, hol keletkezik figyelemre méltó esemény, és hol kezeljük őket. Mindhárom gomb akciós eseményére reagálnunk kell. A lista kiválasztás-eseményét is figyelnünk kell, hiszen az aktuális választás értékével összhangban módosítanunk kell a felső címkét, valamint a törlés gomb engedélyezését. Mindkét fajta eseményt a keret kezeli, ó tehát egyszerre ActionListener és ListSelectionListener.

Forráskód

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

class KarbantartFrame extends JFrame implements
    ActionListener, ListSelectionListener { //1
    DefaultListModel mdSzovegek;
    JList lstSzovegek;
    JButton btUj, btTorol, btKilep;
    JLabel lbInfo;

    public KarbantartFrame() {
        super("Szövegek karbantartása");
        setLocation(200,200);
        lbInfo = new JLabel(" ");

        mdSzovegek = new DefaultListModel(); //2
        mdSzovegek.addElement("Hosszúhetény");
        mdSzovegek.addElement("Rövidkaraj");
        lstSzovegek = new JList(mdSzovegek); //3
        lstSzovegek.setSelectionMode(
            ListSelectionModel.SINGLE_SELECTION);
        lstSzovegek.setSelectedIndex(0);
        lstSzovegek.addListSelectionListener(this);
        JScrollPane spSzovegek = new JScrollPane(lstSzovegek);

        btUj = new JButton("Új",new ImageIcon("icons/insert.gif"));
        btUj.addActionListener(this);
        btTorol = new JButton("Torles",
            new ImageIcon("icons/delete.gif"));
        btTorol.addActionListener(this);
        btKilep = new JButton("Kilép",
            new ImageIcon("icons/exit.gif"));
        btKilep.addActionListener(this);
        JPanel pnVezer = new JPanel();
        pnVezer.add(btUj);
        pnVezer.add(btTorol);
        pnVezer.add(btKilep);
        Container contentPane = getContentPane();
        contentPane.add(lbInfo,"North");
        contentPane.add(spSzovegek,"Center");
        contentPane.add(pnVezer,"South");
        pack();
        setVisible(true);
        valasztasFigyelo();
    }

    void valasztasFigyelo() { //4
        int index = lstSzovegek.getSelectedIndex();
        lbInfo.setText(" ");
        if (index== -1)
            // Nincs kiválasztott elem, törlés gomb letiltása
            btTorol.setEnabled(false);
    }
}
```

```
    else {
        // Van kiválasztott elem, törlés gomb engedélyezése:
        btTorol.setEnabled(true);
        if (index==0)
            lbInfo.setText("Első");
        else if (index==mdSzovegek.getSize()-1)
            lbInfo.setText("Utolsó");
    }
}

void torles() { //5
    // Kiválasztott szöveg törlése:
    int index = lstSzovegek.getSelectedIndex();
    mdSzovegek.remove(index);
    // A kiválasztás újraértékelése:
    if (mdSzovegek.getSize()!=0) {
        if (index==mdSzovegek.getSize())
            index--;
        lstSzovegek.setSelectedIndex(index);
    }
    valasztasFigyelo();
}

void uj() { //6
    String szoveg = JOptionPane.showInputDialog(this,"Szöveg:");
    if (szoveg==null)
        return;
    if (szoveg.equals(""))
        Toolkit.getDefaultToolkit().beep();
    else {
        mdSzovegek.addElement(szoveg);
        lstSzovegek.setSelectedValue(szoveg,true);
        valasztasFigyelo();
    }
}

public void actionPerformed(ActionEvent ev) { //7
    if (ev.getSource()==btUj)
        uj();
    else if (ev.getSource()==btTorol)
        torles();
    else if (ev.getSource()==btKilep)
        System.exit(0);
}

public void valueChanged(ListSelectionEvent e) {
    valasztasFigyelo();
}
}

public class ListTest {
    public static void main(String[] args) {
        new KarbantartFrame();
    }
}
```

A program elemzése

- ◆ //1: A keret kétféle eseményt figyel, tehát két figyelőt implementálunk.
- ◆ //2: `mdSzovegek` lesz az `lstSzovegek` lista adatmodellje. Létrehozunk egy alapértelmezés szerinti listamodellt, és rögtön bele is teszünk két szöveget.
- ◆ //3: `lstSzovegek` lesz a listakomponens. Őt az `mdSzovegek` adatmodellel inicializáljuk. Ezzel a lista adatmodellje hozzá van kötve a listakomponenshez, s a benne levő adatok automatikusan megjelennek a komponensen. A választási módot (`SelectionMode`) úgy állítjuk be, hogy egyszerre csak egy elemet lehessen kiválasztani, és mindenki ki is választjuk (`setSelectedIndex`) a legelső elemet. A lista figyelemláncára felfüzzük a keretet (`addListSelectionListener`), s végül a listát ráteszszük egy görgetőlapra (`ScrollPane`).
- ◆ //4: A `valasztasFigyelo` feladata: az aktuális kiválasztásnak megfelelően aktualizálja az információs szöveget, és szükség esetén letiltja a törlés gombot (ha nincs mit törölni).
- ◆ //5: A `törles` metódus akkor hívódik meg, ha lenyomták a törlés gombot. Lekérdezzük a kiválasztás indexét, és az adatmodellből kitöröljük az ennyiadik sorszámú elemet. Ha nem lenne kiválasztott elem (`index== -1`), az sem lenne baj, mert ilyenkor a `remove` hatástalan. Mivel pont a kiválasztott elemet töröljük, kijelöljük az új kiválasztást.
- ◆ //6: Új szöveg felviteléhez bekérünk egy szöveget a `JOptionPane.showDialog` segítségével. Ha a felhasználó elvetette a bevitelt (Cancel-t nyomott), akkor a szöveg értéke `null` és nincs mit tennünk. Ha nem vitt be semmit, akkor a visszatérési érték egy üres szöveg – ezt nem tessük a listába. A nem üres szöveget viszont betesszük az adatmodellbe. A szöveg utolsó elemként automatikusan megjelenik a listában.
- ◆ //7: A `ListSelectionEvent` eseményt a `valueChanged` metódus kezeli. Elég csak meghívunk a `valasztasFigyelo` metódust.

8.13. Görgetősáv – JScrollBar

Csomag: `javax.swing` Deklaráció: `public class JScrollBar`

Közvetlen ős: `javax.swing.JComponent`

Fontosabb implementált interfések: –

A **görgetősáv** (scroll bar, osztálya `JScrollBar`) egy függőleges vagy vízszintes sáv; a felhasználó leolvashatja róla vagy beállíthatja vele egy értéktartomány aktuális értékét. A görgetősáv teljes mérete az értéktartományt reprezentálja, a rajta levő csúszka pillanatnyi helyzete pedig az értéktartomány egy értékét. A sáv végi nyílak lenyomásával egy megadott alapértékkel növelhetjük az aktuális értéket; a sáv egyéb részein való kattintás nagyobb ugrást idéz elő. Az értéktartomány csúszka által megjelenített aktuális értéke

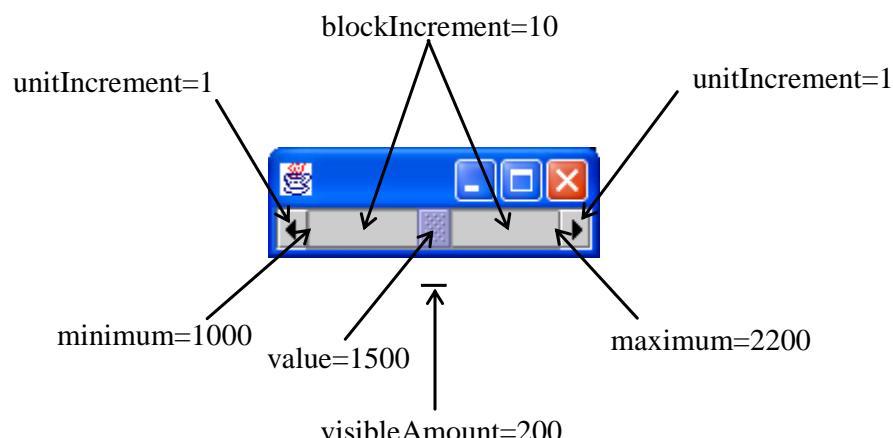
lekérdezhető. Ha a csúszka a görgetősáv elején van, akkor az aktuális érték az értéktartomány alsó határa, ha a végén, akkor a felső határa.

A megadott maximális érték soha nem érhető el, mert abba beleszámít a csúszka által reprezentált érték (lásd jellemzőknél).

A JScrollBar-nak BoundedRangeModel az adatmodellje, de azzal a könyv nem foglalkozik.

Minta görgetősáv a keretben

```
int visibleAmount = 200;
// Paraméterek: orientation, value, visibleAmount, minimum, maximum
cp.add(new JScrollBar(JScrollBar.HORIZONTAL, 1500,
                      visibleAmount, 1000, 2000+visibleAmount));
```



Események

EsemOsztály	Mi történt?	Figyelőinterfész	Felfűzőmetódus	Eseménykez. met.
AdjustmentEvent	Az aktuális érték megváltozott.	AdjustmentListener	addAdjustmentListener	adjustmentValueChanged

Jellemzők

- ▶ `int orientation`
A görgetősáv állása. Lehetséges értékei az Adjustable interfész konstansai: HORIZONTAL: vízszintes, VERTICAL: függőleges. Alapértelmezés: VERTICAL
- ▶ `int minimum, maximum`
Az értéktartomány alsó és felső határa. Alapértelmezés: `minimum=0, maximum=100`
- ▶ `int value`
Az értéktartomány aktuális értéke. Alapértelmezés: `minimum`.

► `int visibleAmount`

Más neve: `extent`. A csúszka arányos mérete a teljes értéktartományhoz képest. Lapozáskor ennyivel változik meg az értéktartomány aktuális értéke. Ha például `extent` a teljes tartománynak (`maximum-minimum`) a fele, akkor a csúszka a teljes sáv méretének a fele lesz. Alapértelmezés: 10.

► `int unitIncrement`

Egységnövekmény, az aktuális érték növelésének alapegysége. Ennyivel növekszik az aktuális érték, ha a sáv végi le vagy fel nyílra kattintunk. Alapértelmezés: 1.

► `int blockIncrement`

Blokknövekmény. Ennyivel növekszik az aktuális érték, ha a görgetősáv egyéb részére kattintunk. Alapértelmezés: `visibleAmount`.

A `value` a csúszka alsó szélénél felel meg, a `maximum` pedig a görgetősáv felső szélénél. Amikor a csúszka nekiütközik a `maximum`-nak, akkor a `value` értéke `maximum-visibleAmount`. Érvényes a következő összefüggés:

```
minimum <= value <= maximum-visibleAmount
```

Ahhoz tehát, hogy `value` tényleg elérhesse a kívánt `maximum` értéket, a görgetősáv maximumát `maximum+visibleAmount`-ra kell állítani!

Konstruktörök

- `JScrollbar(int orientation, int value, int visibleAmount, int minimum, int maximum)`
- `JScrollbar(int orientation)`
- `JScrollbar()`

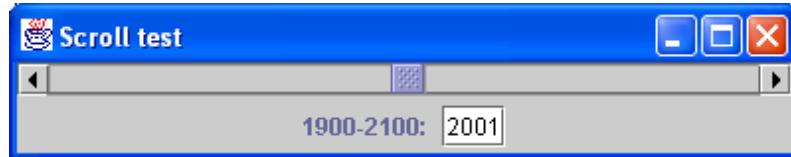
Létrehoz egy görgetősáv komponenst. A paraméterek jelentéseit lásd a jellemzőknél.

Metódusok

- `void addAdjustmentListener(AdjustmentListener l)`
Figyelő hozzáadása a görgetősáv `AdjustmentEvent` figyelőláncához.
- `void setValues(int value, int visibleAmount, int minimum, int maximum)`
Négy érték beállítása egyszerre.

Feladat – Évszám görgetése – ScrollBarTest

Tegyük a keretbe egy szövegmezőt; abban mindenig egy 1900 és 2100 közötti évszám álljon! A szám kezdeti értéke 2001 legyen. A szám értékét beírással, vagy egy görgetősáv segítségével lehessen megváltoztatni. Ha a görgetősáv nyíláról kattintunk, akkor egygel növekedjék az érték, de lehessen a számot tízzel is pörgetni! Ha a szövegmezőbe írt évszámot vagy a görgetősávot megváltoztatjuk, akkor a másik értéke mindenig igazodjon ehhez az új értékhez! Ha rossz évszámot ütnénk be, akkor íródjék vissza az előző jó érték!



Forráskód

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class EvszamBeallit extends JFrame
    implements AdjustmentListener, ActionListener {
    Container cp = getContentPane();
    int minEv=1900, maxEv=2100, aktEv=2001;
    JTextField tfSzam;
    JScrollBar sbSzam;

    public EvszamBeallit() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setBounds(100,100,400,80);
        setTitle("Scroll test");
        // Param: orientation, value, visibleAmount, minimum, maximum:
        cp.add(sbSzam = new JScrollBar(JScrollBar.HORIZONTAL,
            aktEv,5,minEv,maxEv+5),"North");           //1
        sbSzam.setBlockIncrement(10);
        sbSzam.addAdjustmentListener(this);

        JPanel pn;
        cp.add(pn = new JPanel());
        pn.add(new JLabel(minEv+"-"+maxEv+": "));
        pn.add(tfSzam = new JTextField(""+aktEv));
        tfSzam.addActionListener(this);
        show();
    }

    public void adjustmentValueChanged(AdjustmentEvent e) { //2
        tfSzam.setText(Integer.toString(sbSzam.getValue()));
    }

    public void actionPerformed(ActionEvent e) {           //3
        try {
            int szam = Integer.parseInt(tfSzam.getText());
            if (szam<sbSzam.getMinimum() ||
                szam>sbSzam.getMaximum()-sbSzam.getVisibleAmount())
                throw new Exception();
            sbSzam.setValue(szam);
        }
        catch (Exception ex) {
            tfSzam.setText(Integer.toString(sbSzam.getValue()));
        }
    }
}

```

```
public class ScrollBarTest {
    public static void main (String args[]) {
        new EvszamBeallit();
    } // main
} // ScrollBarTest
```

A program elemzése

A görgetősávot //1-ben hozzuk létre: vízszintes állású, az induló érték 2001, a látható tartomány 5 – ezt hozzá is adjuk maxEv-hez, hogy az aktuális értékkel elérhessük a 2100-at. A létrehozás után a blokknövekményt 10-re állítjuk. Ezután a görgetősávhoz és a szövegmezőhöz is hozzápasztjuk a keretet mint hallgatót. Ha a görgetősávot elmozdították (/2), akkor a görgetősáv aktuális értékét beírjuk a szövegmezőbe. Ha a szövegmezőn nyomták le az Entert (/3), akkor a beírt érték alapján beállítjuk a görgetősávot. Ha az érték nem szám vagy nem a helyes tartományba eső szám, akkor a mező értékét visszaállítjuk a görgetősáv aktuális értékére.

8.14. Menüsor – JMenuBar...

Menürendszer segítségével a program funkciói könnyen csoportosíthatók, hierarchikus rendbe szedhetők. A menü használata megkönnyíti a felhasználó tájékozódását. A menü-hierarchia tetején mindig egy **menüsor** (**JMenuBar**) van; ez lebomló **menükből** (**JMenu**) áll. A menü tovább bontható, elemei a következők lehetnek:

- újabb menü (**JMenu**)
- **menütétel** (**JMenuItem**);
- **jelölő-menütétel** (**JCheckBoxMenuItem**);
- **rádió-menütétel** (**JRadioButtonMenuItem**);
- menüelválasztó (szeparátor).

A menütételek nem bonthatók tovább. A menühierarchiát az **add** metódusokkal építhetjük fel, akárcsak a tulajdonosi hierarchiát. A főmenüt (menüsor) a **setJMenuBar** (**JMenuBar mb**) parancssal kell a kerethez rendelni.

A menütételek kiválaszthatók – az **AbstractButton** utódai. Kiválasztásukkal akciót indíthatunk el, éppúgy, mint a többi gombbal. A menüponthoz emlékeztető (mnemonic) karakter és gyorsbillentyű is rendelhető.

A menü közvetlenül a keret tulajdona, nem a tartalompanelé.

Végig fogjuk nézni a **JMenuBar**, **JMenu**, **JMenuItem** és a **JCheckBoxMenuItem** osztályt. A **JRadioButtonMenuItem** a rádiógombhoz hasonlóan működik – a rádió-menütételeket a **ButtonGroup** segítségével lehet csoportba foglalni.

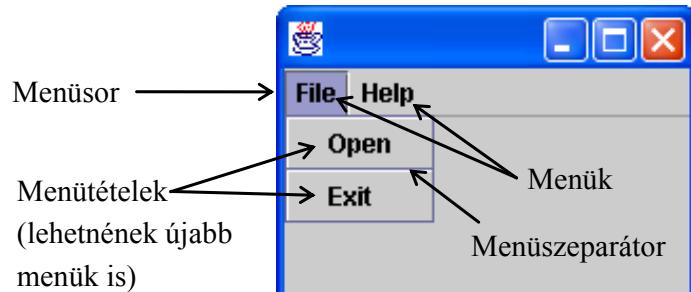
Minta menühierarchia a keretben (teljes program, import nélkül)

```
public class Minta extends JFrame {
    public Minta() {
        setSize(200,150);
        JMenuBar mb;
        JMenu mFile;
        setJMenuBar(mb=new JMenuBar());

        mb.add(mFile=new JMenu("File"));
        mb.add(new JMenu("Help"));

        mFile.add(new JMenuItem("Open"));
        mFile.addSeparator();
        mFile.add(new JMenuItem("Exit"));
        setVisible(true);
    }

    public static void main (String args[]) {
        new Minta();
    }
}
```



A menüosztályok hierarchiája:

Csomag: javax.swing

```
JComponent
+--JMenuBar
+--AbstractButton
|   +--JMenuItem
|   |   +--JCheckBoxMenuItem
|   |   +--JRadioButtonMenuItem
|   |   +--JMenu
```

JMenuBar osztály

Csomag: javax.swing Deklaráció: public class JMenuBar
Közvetlen ős: javax.swing.JComponent

Konstruktorok, metodusok

- ▶ **JMenuBar()**
Létrehozza a főmenüt. Paraméterek nincsenek. A menüsöt a JFrame osztály `setJMenuBar(JMenuBar mb)` metódusával hozzá kell rendelnünk a kerethez.
- ▶ **JMenu add(JMenu m)**
Menü hozzáadása a menüsorhoz. A menüt előzőleg létre kell hozni.

JMenu osztály

Csomag: javax.swing Deklaráció: public class JMenu
 Közvetlen ős: javax.swing.JMenuItem

Konstruktörök, metódusok

- ▶ JMenu(String text)
- ▶ JMenu()

Létrehozza a menüt. A paraméter jelentése:

– text: A menü megjelenő szövege. Alapértelmezés: üres.

- ▶ JMenuItem add(JMenuItem mi)

Menü vagy menütétel hozzáadása az előzetesen létrehozott menühöz.

- ▶ void addSeparator()

Elválasztó sor hozzáadása a menü végéhez.

JMenuItem osztály

Csomag: javax.swing Deklaráció: public class JMenuBar
 Közvetlen ős: javax.swing.AbstractButton

Események: AbstractButton-tól örökolt események: **ActionEvent**

Jellemzők

AbstractButton-tól örökolt jellemzők: text, icon, mnemonic, selected, actionPerformed.

- ▶ boolean enabled

Megadja, hogy a menütéten keletkezhet-e esemény. Ha értéke false, akkor hiába választjuk ki a menütételt, az érzéketlen marad.

- ▶ KeyStroke accelerator

Gyorsbillentyű. Például az Alt-X gyorsbillentyű:

KeyStroke.getKeyStroke(KeyEvent.VK_X, Event.ALT_MASK)

Konstruktörök, metódusok

- ▶ JMenuItem(String text, Icon icon)
- ▶ JMenuItem(String text)
- ▶ JMenuItem(Icon icon)
- ▶ JMenuItem(String text, int mnemonic)
- ▶ JMenuItem()

A paraméterek jelentése:

– text: a menü megjelenő szövege. Alapértelmezés: "" (üres).

– mnemonic: Emlékeztető billentyű (aláhúzottan jelenik meg). Alapértelmezés: nincs.

Az örökolt setMnemonic metódussal is a menütételhez rendelhető.

JCheckBoxMenuItem osztály

Csomag: javax.swing Deklaráció: public class JCheckBoxMenuItem
 Közvetlen ős: javax.swing.JMenuItem

Események: AbstractButton-tól örökölt események: **ActionEvent**

Jellemzők: AbstractButton-tól örökölt jellemzők: text, icon, mnemonic, selected, actionCommand; JMenuItem-tól örökölt jellemzők: enabled, accelerator

Konstruktörök, metódusok

- ▶ JCheckBoxMenuItem(String text, Icon icon)
- ▶ JCheckBoxMenuItem(String text)
- ▶ JCheckBoxMenuItem(Icon icon)

Jelölő-menütétel létrehozása. A paraméterek jelentése:

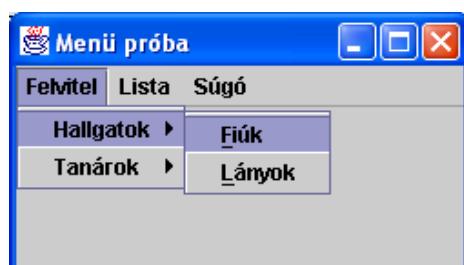
- text: a menü megjelenő szövege. Alapértelmezés: üres.
- icon: a menü ikonja. Alapértelmezés: nincs ikon.
- ▶ void setState(boolean state)
- ▶ boolean getState()

Beállítja, illetve lekérdezi a jelölő-menütétel állapotát.

Feladat – MenuTest

Készítsük el az alább látható menüt! A menüsor elemei: Felvitel, Lista, Súgó. Az érzékeny menüpontok vastag betűvel vannak szedve, mellettük ott áll a menüpont jellege és a végrehajtandó tevékenység. A jelölő-menütétel előtt egy kis pipa szerepel. A többi menüpont egyelőre érzéketlen. A Fiúk és Lányok menütételnek legyen emlékeztető karaktere, a Súgó/Használatnak legyen ikonja!

Felvitel	(JMenu)	
Hallgatók	(JMenu)	
Fiúk	(JMenuItem)	Konzolra írjuk: Fiúk felvitele
<u>Lányok</u>	(JMenuItem)	
Tanárok	(JMenu)	
Lista		
Lista1	(JMenuItem)	Megjelenik egy keret Lista1 címmel
Lista2	(JMenuItem)	Megjelenik egy keret Lista2 címmel
-----	(menüszeprátor)	
✓ Nyomtatóra	(JCheckBoxMenuItem)	Konzolra: Nyomtatóra / Nem nyomt.
Súgó	(JMenu)	
Használat	(JMenuItem)	
Névjegy	(JMenuItem)	



Forráskód

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import extra.frame.CloseableFrame;
class MenuFrame extends CloseableFrame implements ActionListener{
    JMenuBar mb;
    JMenuItem miFelvitelFiuk;
    JMenuItem miLista1, miLista2;
    JCheckBoxMenuItem cmiNyomtatora;
    JMenuItem miHasznalat, miNevjegy;

    public MenuFrame(String title) {
        super(title);
        setBounds(400,200,250,120);

        // Menü
        setJMenuBar(mb = new JMenuBar());
        JMenu mFelvitel = new JMenu("Felvitel");
        mb.add(mFelvitel);
        JMenu mFelvitelHallgatok = new JMenu("Hallgatók");
        mFelvitel.add(mFelvitelHallgatok);
        mFelvitel.add(new JMenu("Tanárok"));
        mFelvitelHallgatok.add(miFelvitelFiuk =
            new JMenuItem("Fiúk",'F'));
        mFelvitelHallgatok.add(new JMenuItem("Lányok",'L'));

        JMenu mLista = new JMenu("Lista");
        mb.add(mLista);
        mLista.add(miLista1 = new JMenuItem("Listá1"));
        mLista.add(miLista2 = new JMenuItem("Listá2"));
        mLista.addSeparator();
        mLista.add(cmiNyomtatora=new JCheckBoxMenuItem("Nyomtatóra"));

        JMenu mSugo;
        mb.add(mSugo = new JMenu("Súgó"));
        mSugo.add(miHasznalat = new JMenuItem("Használat",
            new ImageIcon("icons/help.gif")));
        mSugo.add(miNevjegy = new JMenuItem("Névjegy"));

        miFelvitelFiuk.addActionListener(this);
        miLista1.addActionListener(this);
        miLista2.addActionListener(this);
        cmiNyomtatora.addActionListener(this);
        show();
    }

    public void lista1() {
        JFrame frList = new CloseableFrame("Listá1");    //1
        frList.setLocation(getX()+10,getY()+10);
        frList.setSize(200,100);
        frList.show();
    }
}

```

```

public void lista2() {
    JFrame frList = new CloseableFrame("Lista2"); //2
    frList.setLocation(getX()+50, getY()+10);
    frList.setSize(300,50);
    frList.show();
}

public void actionPerformed(ActionEvent e) {
    if (e.getSource() == miLista1)
        lista1();
    else if (e.getSource() == miLista2)
        lista2();
    else if (e.getSource() == miFelvitelFiuk)
        System.out.println("Fiúk felvitele");
    else if (e.getSource() == cmiNyomtatora) {
        if (cmiNyomtatora.getState())
            System.out.println("Nyomtatóra");
        else
            System.out.println("Nem nyomtatóra");
    }
}
} // MenuFrame

public class MenuTest {
    public static void main (String args[]) {
        new MenuFrame("Menü próba");
    }
} // MenuTest

```

A keret az extra csomag `CloseableFrame` osztályából származik. //1-ben és //2-ben is `CloseableFrame`-eket hozunk létre. A `CloseableFrame` becsukásakor csak akkor fejeződik be a program futása, ha ő volt az utolsó "futó" keret az alkalmazásban.

8.15. Ablak – JWindow

Csomag: `javax.swing` Deklaráció: `public class JWindow`

Közvetlen ős: `java.awt.Window`

Fontosabb implementált interfések: –

A keret nélküli ablak (`JWindow`) olyan legfelső szintű konténer komponens, amelynek nincs sem kerete, sem címe, sem menüje. Mivel nincsen kerete, az ablakot a felhasználó közvetlenül nem mozgathatja és nem is méretezheti át. Az ablaknak kell, hogy legyen egy másik ablak (`Window`-leszármazott) tulajdonosa, s ezt a tulajdonost az ablak létrehozáskor kell megadni. A `JWindow` osztályt csak ritkán példányosítják – általában az alkalmazás képét szokta „tartani” betöltéskor (splash screen).

Az ablaknak van tartalompanelje, abba kell tenni a komponenseket. A tartalompanel elrendezés-menedzsere alapértelmezés szerint határ menti (`BorderLayout`).

Események

EsemOsztály	Mi történt?	Figyelőinterfész	Felfűzőmetódus	Eseménykez. met.
WindowEvent	Először lett látható.	WindowListener	addWindowListener	windowOpened
	Becsukták.			windowClosed

Konstruktorkok, metódusok

- ▶ `JWindow(Frame owner)`
- ▶ `JWindow(Window owner)`

Létrehoz egy láthatatlan ablakot; a tulajdonos a `Frame` vagy a `Window` egy leszármazottjának példánya.

- ▶ `Container getContentPane()`
- Visszadja az ablak tartalompaneljét.

Megjegyzés: Vizsgálja meg a forráskód mellékletben a `WindowTest.java` programot!

8.16. Dialógusablak – JDialog, kész dialógusok – JOptionPane

Csomag: `javax.swing` Deklaráció: `public class JDialog`

Közvetlen ős: `java.awt.Dialog`

Fontosabb implementált interfések: `WindowConstants`

A **dialógusablak** (`dialog` window, osztálya `JDialog`), más néven párbeszédablak olyan ablak, amelyet általában felhasználói adatbevitelre, nyugtázásra szoktak használni. minden dialógusablaknak kell, hogy legyen egy `Frame` vagy `Dialog` típusú tulajdonosa. A dialógusablak csak akkor látható, ha tulajdonosa is látható.

A dialógusablakot **modálissá** lehet tenni; ez azt jelenti, hogy az alkalmazáson belül csak ez az ablak fogad eseményt, rajta kívül megszűnik az élet. Amíg a dialógusablakot be nem zártuk, nem lehet átváltani az alkalmazás másik ablakára, csak a dialógusablak saját gyerekeire. A dialógusablak kényszerítheti a felhasználót a kért adatok beírására. Az operációs rendszer alkalmazásai között természetesen továbbra is lehet változatottni. A dialógusablakot nem lehet ikonná változtatni.

A dialógusablak nem lehet közvetlen tulajdonosa komponenseinek: a komponenseket a tartalompanelbe tenni, akárcsak a keretéit:

```
dialog.getContentPane().add(gyerek);
```

A tartalompanel elrendezés-menedzsere alapértelmezésben határ menti (`BorderLayout`), mint a `JFrame` esetén is.

Események

EsemOsztály	Mi történt?	Figyelőinterfész, adapter	Felfűzőmetódus	Eseménykezelő metódus
WindowEvent	Először lett látható.	WindowListener WindowAdapter	addWindowListener	windowOpened
	Be akarják csukni.			windowClosing
	Becsukták.			windowClosed
	Fókuszba került.			windowActivated
	Elveszítette a fókuszt.			windowDeactivated

A windowOpened és a windowClosed a Window-tól örökölt események

Jellemzők

- ▶ `String title`
Az ablak címe.
- ▶ `boolean modal`
A dialógusablak modalitása. Ha az ablak modális, akkor az alkalmazáson belül csak ő fogad eseményt.
- ▶ `boolean resizable`
Az ablak átméretezhetősége. Ha `true`, akkor a felhasználó megváltoztathatja az ablak méretét, ha `false`, akkor nem.

Konstruktörök

- ▶ `JDialog(Frame owner, String title, boolean modal)`
- ▶ `JDialog(Frame owner, boolean modal)`
- ▶ `JDialog(Frame owner, String title)`
- ▶ `JDialog(Frame owner)`
- ▶ `JDialog(Dialog owner, String title, boolean modal)`
- ▶ `JDialog(Dialog owner, String title)`
- ▶ `JDialog(Dialog owner)`

A paraméterek jelentése:

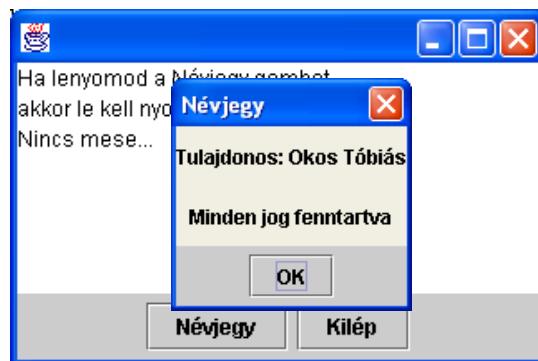
- `owner`: A tulajdonos; csak keret vagy dialógusablak lehet.
- `title`: A dialógusablak címe. Alapértelmezés: üres.
- `modal`: Modalitás. Alapértelmezés: `false`, vagyis nem modális.

Metódusok

- ▶ `void show()`
Az ablak előtérbe kerül.
- ▶ `void hide()`
Az ablak háttérbe kerül.

Feladat – DialogTest

Készítsünk egy dialógusablakot; legyenek rajta a képen látható információk! A Névjegy gomb lenyomására jelenjen meg a Névjegy ablak, és csak az OK gomb lenyomására tűnjön el – amíg az OK-t le nem nyomják, addig az alkalmazás minden más része maradjon érzéketlen. A Névjegy ablak helyzete a főablakhoz képest (60,40) legyen, a háttérszíne drapp, és ne lehessen átméretezni.



Forráskód

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class NevjegyDialog extends JDialog implements ActionListener {
    Container cp = getContentPane();
    JButton btOk;

    NevjegyDialog(JFrame owner) { //1
        super(owner, "Névjegy", true);
        setLocation(getParent().getX() + 60, getParent().getY() + 40);
        setResizable(false);
        cp.setBackground(SystemColor.controlHighlight);
        cp.setLayout(new GridLayout(3, 1));
        cp.add(new JLabel("Tulajdonos: Okos Tóbiás", JLabel.CENTER));
        cp.add(new JLabel("Minden jog fenntartva", JLabel.CENTER));
        JPanel pnOk = new JPanel();
        cp.add(pnOk);
        pnOk.add(btOk = new JButton("OK"));
        btOk.addActionListener(this);
        pack(); //2
        show();
    }

    public void actionPerformed(ActionEvent e) { //3
        dispose();
    }
}

```

```

class TestFrame extends JFrame implements ActionListener {
    Container cp = getContentPane();
    JButton btNevjegy, btKilep;                                //4

    public TestFrame() {
        setDefaultCloseOperation(DO NOTHING ON CLOSE);
        setBounds(300,200,300,200);
        cp.add(new JTextArea("Ha lenyomod a Névjegy gombot, \n"+
            "akkor le kell nyomnod az OK-t.\nNincs mese..."));

        JPanel pn = new JPanel();
        pn.add(btNevjegy = new JButton("Névjegy"));
        pn.add(btKilep = new JButton("Kilép"));
        cp.add(pn,"South");

        btNevjegy.addActionListener(this);
        btKilep.addActionListener(this);
        show();
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == btNevjegy)
            new NevjegyDialog(this);                                //5
        else if (e.getSource() == btKilep)
            System.exit(0);
    }
}

public class DialogTest {
    public static void main (String args[]) {
        new TestFrame();
    } // main
} // DialogTest

```

Valahányszor a felhasználó lenyomja a Névjegy gombot, mindenkor újra összeállítjuk a névjegydialógus objektumot (//5), kilépéskor pedig „megöljük” (//3). Megtehetnénk azt is (és legtöbbször így is teszünk majd), hogy az alkalmazás elején összeállítjuk a dialógust, és mindenkor megmutatjuk, ha a felhasználó lenyomja a Névjegy gombot, az OK megnyomására pedig megint rejtjük. Ehhez a következőket kell változtatnunk a forráskód melléklet: DialogTest2):

- ◆ a keretben deklaráljuk és a keret konstruktőraban létrehozzuk a névjegydialógust (//4 után); **nem a deklarációban hozzuk létre, mert akkor még nincs a this-nek értéke!**;
- ◆ a NevjegyDialog konstruktőréről kivesszük a show metódust (//2);
- ◆ az ablak bezárásakor (//3) nem szabadítjuk fel az ablakot (dispose), hanem csak eltüntetjük (hide);
- ◆ az ablak kinyitásakor létrehozás helyett megmutatjuk: //5-ben new NevjegyDialog(this) helyett nevjegyDialog.show().

A dialógusok ismételt létrehozásával csak a memóriát kíméljük, a felhasználó idejét nem!

JColorChooser osztály

Csomag: javax.swing Deklaráció: public class JColorChooser

Közvetlen ős: javax.swing.JComponent

Fontosabb implementált interfések: –

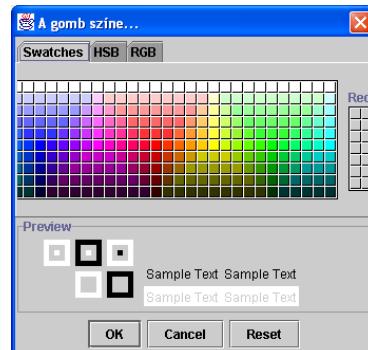
A színválasztó dialógussal (JColorChooser) egy színt lehet kiválasztani. A showDialog statikus metódusa egy dialógust jelenít meg, s azon a felhasználó tetszőleges színt kikeverhet. A dialógusnak egy színobjektum a visszatérési értéke.

- ▶ static Color showDialog(Component component, String title, Color initialColor)

Megjelenik a modális színdialógus. Az Ok vagy a Cancel lenyomására a dialógus eltűnik; Ok-val a kiválasztott szín lesz a visszatérési érték, Cancellel null.

Feladat – Színválasztás – ColorChooserTest

Tegyük a keretbe egy nyomógombot a "Színválasztás" felirattal! A gomb lenyomására jelenjen meg a színdialógus, és azon válasszuk ki, hogy milyen színű lesz majd a gomb!



Forráskód

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ColorChooserTest extends JFrame
    implements ActionListener {
    private JButton btSzin = new JButton("Színválasztás");
    private Color color;

    public ColorChooserTest() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        getContentPane().add(btSzin);
        btSzin.addActionListener(this);
        color = btSzin.getBackground();
        pack();
        show();
    }

    public void actionPerformed(ActionEvent ev) {
        color=JColorChooser.showDialog(this,"A gomb színe...",color);
        if (color != null)
            btSzin.setBackground(color);
    }
}
```

```

public static void main(String[] args) {
    new ColorChooserTest();
}
}

```

JOptionPane osztály

Csomag: javax.swing Deklaráció: public class JOptionPane

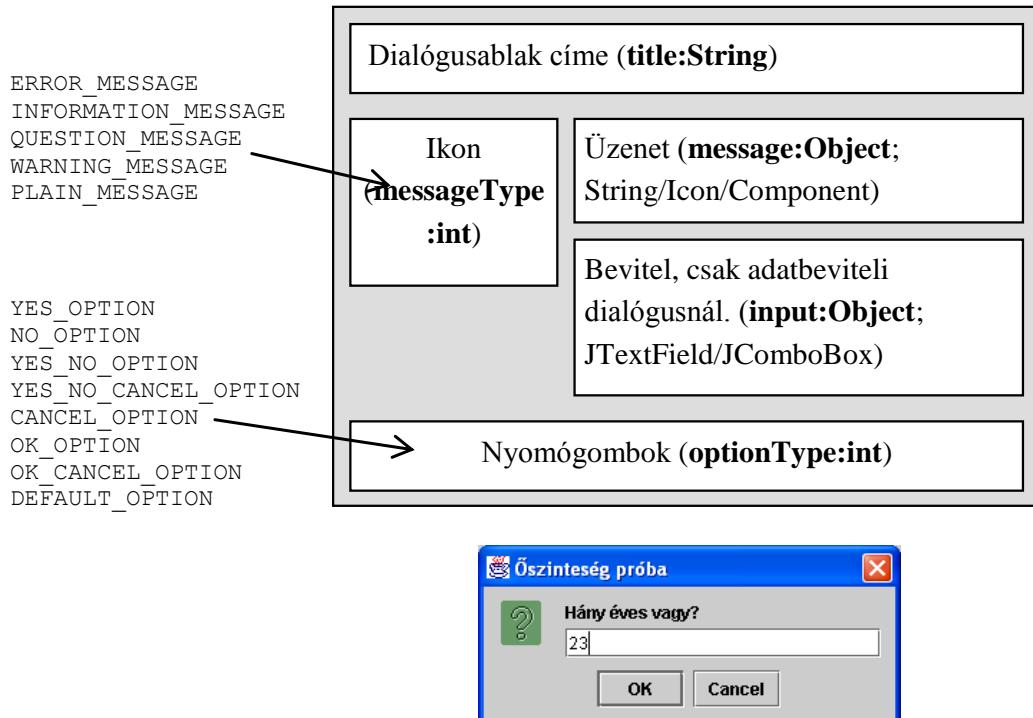
Közvetlen ős: javax.swing.JComponent

Fontosabb implementált interfészek: –

Az opciópanel (JOptionPane) statikus metódusaival szabványos dialógusokat jeleníthetünk meg a felhasználói felületen. A leggyakrabban használt dialógustípusok a következők:

- Információközlő dialógus (showMessageDialog)
- Megerősítő dialógus (showConfirmDialog)
- Adatbekérő dialógus (showInputDialog)

A dialógusok modálisak, és egy nyomógomb lenyomására várnak. A dialógusok elrendezése és paraméterezése szabványos (8.4. ábra).



8.4. ábra. A szabványos dialógusablak elrendezése és paraméterezése

A JOptionPane dialóguskészítő metódusainak paraméterei

- ◆ Component parent: A dialógus szülőablaka; a dialógus a szülőablak közepén jelenik meg. Ha parent értéke null, akkor a képernyő közepén jelenik meg.
 - ◆ Object message: Egy szöveg, ikon vagy egy tetszőleges komponens; a dialógus Üzenet helyén jelenik meg.
 - ◆ String title: A dialógusablak címe.
 - ◆ int messageType: Ez határozza meg a dialógus Ikon helyén megjelenő ikont. Lehetséges értékei: ERROR_MESSAGE, INFORMATION_MESSAGE, QUESTION_MESSAGE, WARNING_MESSAGE, PLAIN_MESSAGE. minden értékhez tartozik egy alapértelmezés szerinti ikon.
 - ◆ Icon icon: A messageType alapértelmezés szerinti ikonja erre cserélődik fel.
 - ◆ int optionType: Ez határozza meg az Opciók helyén megjelenő gombokat és a dialóguskészítő metódus lehetséges visszatérési értékeit. Lehetséges értékek metódusonként mások. Lásd a Metódusoknál: showMessageDialog, showConfirmDialog stb.
 - ◆ Object[] selectionValues
 - ◆ Object initialValue
- Input dialóguson a Bevitel területén egy kombinált listában megjelennek a selectionValues értékek. Kezdetben az initialValue a kiválasztott érték.
- ◆ Object[] options
 - ◆ Object initialValue
- Az általános dialóguskészítőben (showOptionDialog) ezekkel lehet megadni a nyomó gombokat. A gombok az options[i].toString felirattal jelennek meg. Kezdetben az initialValue gomb van kiválasztva.

A dialógus összeállítása

A dialógusablak összeállításakor a következőket kell eldöntenünk, illetve megadnunk:

- ◆ **Dialógus típusa:** Message, Confirm, Input, Option. A JOptionPane megfelelő metódusát kell meghívunk (showMessageDialog, showConfirmDialog...).
- ◆ **Ikon** (messageType) megadása. Egyforma az összes dialógusra.
- ◆ **Üzenet** (message) megadása. Egyforma az összes dialógusra. Formái: String, Icon, komponens, vagy ezek variációi.
- ◆ **Opciók** kiválasztása. Dialógusonként más-más értékeket adhatunk meg. Lásd a megfelelő metódus leírásában.
- ◆ Ha Input dialógusról van szó, akkor a bevitel szövegmező lehet vagy kombinált lista.

Metódusok

Megadjuk a dialógus célját és a különféle kombinációkban megadható lehetséges paramétereket. Válasszunk nyugodtan a környezet által felkínált paraméterekből!

- ▶ static void showMessageDialog(paraméterek)

Megjelenít egy üzenetet. Az Ok lenyomására vár.

Lehetséges paraméterek: parent, message, title, messageType, icon.

Lásd forráskód: //2, 4, 5, 6, 7

- ▶ static int showConfirmDialog(paraméterek)

Megerősítést kér a felhasználótól. A megadott opciók kiválasztására vár.

Lehetséges paraméterek: parent, message, title, optionType, messageType, icon.

optionType lehetséges értékei: YES_NO_OPTION, YES_NO_CANCEL_OPTION, OK_CANCEL_OPTION

Lásd forráskód: //3

- ▶ static String showInputDialog(paraméterek)

Adatot kér be a felhasználótól. Az Ok vagy Cancel lenyomására vár.

Lehetséges paraméterek: parent, message, title, messageType, icon, selectionValues, initialValue.

Lásd forráskód: //1

- ▶ static int showOptionDialog(paraméterek)

Általános dialóguskészítő metódus, az előzőek kombinációja. Az a jó benne, hogy saját feliratú nyomógombokat gyárthatunk vele, de magunknak kell legyártani őket és meg kell adnunk az összes paramétert.

Lehetséges paraméterek: parent, message, title, optionType (=DEFAULT_OPTION), messageType, icon, options, initialValue

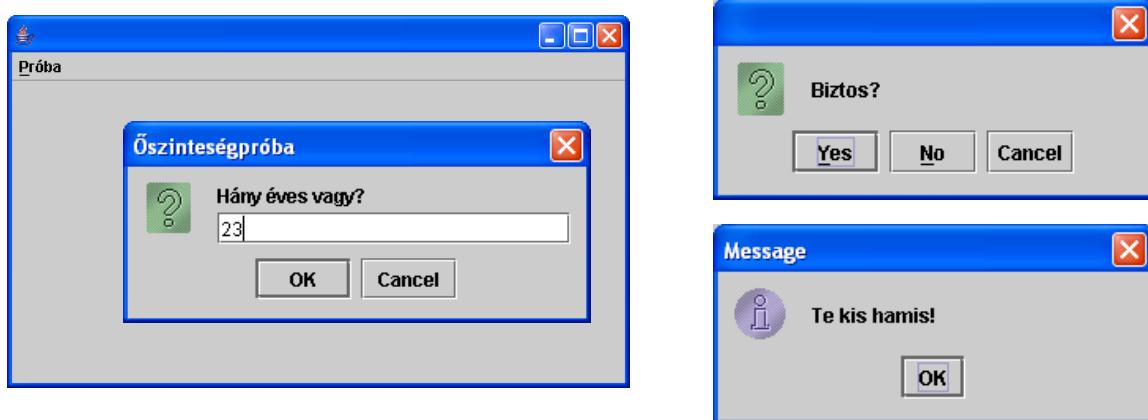
Lásd forráskód: //8

Feladat – Ószinteségpróba

Kérdezzük meg a kedves felhasználót, hogy hány éves! Csak pozitív számot fogadjunk el! Ha nem ír be semmit, írjuk vissza, hogy "Talán megfutamodtál?"! Ha egy ≥ 40 számot ír be, akkor írjuk ki, hogy "Kiálltad a próbát!"; Ha 40-nél kisebb számot ütött be, akkor írjuk ki, hogy "Biztos?". A válasz szerint a következő üzeneteket írjuk ki:

- Igen: "Nem hiszem!"
- Nem: "Gondoltam!"
- Mégse: "Te kis hamis!"

A program bezárása előtt kérdezzük meg a felhasználót: „Biztosan ki akar lépni?”



Forráskód

```

import javax.swing.*;
import java.awt.event.*;

public class OszintesegProba extends JFrame implements
    ActionListener {
    JMenuBar mb = new JMenuBar();
    JMenu mProba = new JMenu("Próba");
    JMenuItem miProba = new JMenuItem("Öszinteség", 'Ó');
    JMenuItem miKilep = new JMenuItem("Kilép", 'K');

    public OszintesegProba() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setBounds(200,50,500,300);
        setJMenuBar(mb);
        mb.add(mProba);
        mProba.setMnemonic('P');
        mProba.add(miProba);
        mProba.add(miKilep);
        miProba.addActionListener(this);
        miKilep.addActionListener(this);
        show();
    }
}

```

Pozitív szám ismételt bekérése, amíg nem ütnek be egyet. Cancel esetén a visszaadott érték -1:

```

double ev() {
    String evStr = "";
    double ev = 0;
    while (ev<=0) {
        try {
            evStr= JOptionPane.showInputDialog(this,
                "Hány éves vagy?", "Öszinteségpróba",
                JOptionPane.QUESTION_MESSAGE);
            if (evStr == null)
                return -1;
            else
                ev=Double.parseDouble(evStr);
        }
        catch (NumberFormatException ex) {
        }
    }
}

```

```

        if (ev<=0)
            JOptionPane.showMessageDialog(this, //2
                "Pozitív számot kérlek!", "", JOptionPane.ERROR_MESSAGE);
    }
    return ev;
}

```

A proba metódus öszinteséget vizsgál:

```

void proba() {
    double ev = ev();
    if (ev == -1) {
        JOptionPane.showMessageDialog(this, "Talán megfutamodtál?");
        return;
    }
    if (ev < 40) {
        int option = JOptionPane.showConfirmDialog(this,
            "Biztos?", "", JOptionPane.YES_NO_CANCEL_OPTION); //3
        if (option == JOptionPane.YES_OPTION)
            JOptionPane.showMessageDialog(this, "Nem hiszem!", //4
                "", JOptionPane.WARNING_MESSAGE);
        else if (option == JOptionPane.NO_OPTION)
            JOptionPane.showMessageDialog(this, "Gondoltam!", //5
                "Lebukás", JOptionPane.INFORMATION_MESSAGE);
        else
            JOptionPane.showMessageDialog(this, //6
                "Te kis hamis!");
    }
    else
        JOptionPane.showMessageDialog(this, //7
            "Kiálltad a próbát!", "", JOptionPane.INFORMATION_MESSAGE);
}

```

A tenyleg metódus a message kérdésre igen/Nem választ vár. true az érték, ha igen. A showOptionDialog paraméterei: parent, message, title, optionType,messageType, icon, options, initialValue:

```

boolean tenyleg(String message) {
    Object[] opciok = {"Igen", "Nem"};
    int valasz = JOptionPane.showOptionDialog(this, //8
        message, "", JOptionPane.DEFAULT_OPTION,
        JOptionPane.WARNING_MESSAGE, null, opciok, opciok[1]);
    return valasz==0;
}
public void actionPerformed(ActionEvent ev) {
    if (ev.getSource()==miProba)
        proba();
    else if (ev.getSource()==miKilep) {
        if (tenyleg("Biztosan ki akar lépni?")) System.exit(0);
    }
}
public static void main (String args[]) {
    new OszintesegProba();
}
}

```

Megjegyzés: Az extra.hu.HuOptionPane osztály az JOptionPane osztály magyar változata. Használja egészseggel!

8.17. Időzítő – Timer

Csomag: javax.swing

Deklaráció: public class Timer

Közvetlen ős: java.lang.Object

Közvetlenül implementált interfések: -

A Timer osztálynak **időzítők** a példányai. Az időzítő megadott időközönként (delay, késleltetés, periódus) egy ActionEvent eseményt bocsát ki és elküldi az őt hallgató objektumoknak. Az időzítőt a stop metódussal lehet leállítani, a start-tal pedig újraindítani. Az időzítőnek van kezdeti késleltetése (initDelay) is: indításkor (start) minden ennyi idő műlva kelti az első eseményt. A periódus és a kezdeti késleltetés menet közben is állítható. Egy időzítőhöz akárhány figyelő kapcsolható, és egy figyelő több időzítőt is figyelhet.

Megjegyzés: A Timer nem komponens, nem a JComponent osztály leszármazottja. Ezért az időzítő „kakukktojás” ebben a fejezetben. Csak azért szerepel itt, mert használatával sok érdekes feladatot lehet megoldani.

Jellemzők

- ▶ int delay
Késleltetés (periódusidő). Ekkora időközönként fog ActionEvent eseményt előállítani.
- ▶ int initialDelay
Kezdeti késleltetés (várakozás). Induláskor, valamint minden egyes start hívásánál ennyi idő telik el az első esemény generálásáig. Alapértelmezés: delay.

Konstruktur, metódusok

- ▶ Timer(int delay, ActionListener listener)
Létrehoz egy Timer objektumot, delay késleltetéssel. Az időzítőnek mindenkorban lesz egy figyelője, a listener.
- ▶ void addActionListener(ActionListener listener)
Hozzáad egy újabb hallgatót az időzítő figyelőláncához.
- ▶ void start()
Elindítja az időzítőt. Ezután az időzítő objektum initialDelay után, delay időközönként eseményeket kelt.

- ▶ `void stop()`
Leállítja az időzítőt. Ettől kezdve nincs eseménykeltés.
- ▶ `void restart()`
Leállítja, majd újraindítja az időzítőt.
- ▶ `boolean isRunning()`
Ha az időzítő éppen fut, vagyis küldözetet eseményeket, akkor a visszaadott érték `true`, egyébként `false`.

Feladat – Stopper (StopperApp)

Tegyük a keretbe két stoppert! Az egyik 1/10 másodpercenként, a másik 2 másodpercenként számoljon! Mindjárt induljon el minden kettő! Az időzítőt a Stop gombbal lehessen állítani és a Start gombbal újra lehessen indítani.

A két stopper hasonló, csak periódusidejük különbözik.

Készítsünk hát egy Stopper osztályt; az egy panelen jelenjen meg és végezze el a fenti feladatokat!

Ezután tegyük a keretbe két ilyen stoppert!



Forráskód

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class Stopper extends JPanel implements ActionListener { //1
    private Timer timer;
    private int ido = 0;
    private JLabel kijelzo;
    private JButton btStart, btStop;

    public Stopper(int delay) {
        setBorder(BorderFactory.createLoweredBevelBorder());
        setLayout(new GridLayout(0,1));
        add(kijelzo = new JLabel("0",JLabel.CENTER));
        add(btStart=new JButton("Start"));
        add(btStop=new JButton("Stop"));
        btStart.addActionListener(this);
        btStop.addActionListener(this);
        timer = new Timer(delay,this); //2
        timer.start();
    }
}

```

```

public void actionPerformed(ActionEvent e) {
    if (e.getSource() == timer) { //3
        ido++;
        kijelzo.setText(""+ido);
    }
    else if (e.getSource() == btStart) { //4
        timer.restart();
    }
    else if (e.getSource() == btStop) { //5
        timer.stop();
    }
}
}

public class StopperApp extends JFrame {

    public StopperApp() {
        setLocation(200,200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        Container cp = getContentPane();
        cp.setLayout(new GridLayout(1,0));
        cp.add(new Stopper(100));
        cp.add(new Stopper(2000));
        pack();
        show();
    }

    public static void main (String args[]) {
        new StopperApp();
    }
}
}

```

A program elemzése

A keretnek figyelnie kell az időzítő által gerjesztett akcióeseményeket (//1). Az időzítőt //2-ben hozzuk létre: az „eseményosztás” a paraméterben megadott milSec ezredmásodperces időközönként zajlik. Az eseménynek a panel (this) lesz az egyetlen megfigyelője. Ameddig az időzítőnek nem küldjük el a start üzenetet, addig nem történik semmi. //4-ben az időzítőt elindítjuk, //5-ben pedig leállítjuk. Amikor az időzítő éppen aktív, akkor a timer mint forrás milSec másodpercenként automatikusan kelt egy-egy ActionEvent eseményt, s azt //3-ban kezeljük: kiírjuk az idő változó eggyel megnövelt értékét.

Figyelje meg, hogy a panelhez három helyrőlérkezhet ActionEvent esemény: a két gombtól és az időzítőtől. E három forrás közül csak a két gomb látható; az időzítő láthatatlan, az a hátterből dobálja az eseményeket.

Tesztkérdések

- 8.1. Mely állítások igazak? Jelölje be az összes jó választ!
- a) A címke a megjelenítési területén csak vízszintesen igazítható. Függőlegesen minden középre igazodik.
 - b) A címkének lehet szövege és ikonja is, de egyszerre csak az egyik.
 - c) Címkén nem keletkezik magas szintű esemény.
 - d) Ha a címkének van ikonja, akkor az minden megelőzi a szöveget.
- 8.2. Mely állítások igazak? Jelölje be az összes jó választ!
- a) A `JCheckBox` osztály az `AbstractButton` osztályból származik.
 - b) A `JComboBox` osztály az `AbstractButton` osztályból származik.
 - c) Az `AbstractButton` leszármazottainak osztályából való komponenseken keletkezhet `ActionEvent` esemény.
 - d) Az `AbstractButton`-ból nem lehet példányt létrehozni.
- 8.3. Mely komponenseken keletkezik `ActionEvent` esemény? Jelölje be az összes jó választ!
- a) `JTextField`
 - b) `JTextArea`
 - c) `JCheckBox`
 - d) `JScrollBar`
- 8.4. Az alábbi osztályok közül melyek származnak **közvetlenül** a `JComponent` osztályból? Jelölje be az összes jó választ!
- a) `JComboBox`
 - b) `JCheckBoxMenuItem`
 - c) `JScrollBar`
 - d) `JTextField`
- 8.5. Mely állítások igazak? Jelölje be az összes jó választ!
- a) A `JFrame` a `JPanel` ōse.
 - b) A `JFrame` egy `Window`.
 - c) A `JFrame` egy `JComponent`.
 - d) A `JPanel` egy `Container`.
- 8.6. Mely állítások igazak? Jelölje be az összes jó választ!
- a) A `JScrollBar` egy görgetősáv; a funkciója egy tartomány aktuális értékének állítása.
 - b) A `JScrollBar` egy görgetőlap; ennek segítségével lehet a szövegterületet görgetni.
 - c) A `JTextField` a `JTextComponent` leszármazottja.
 - d) Az MVC-modellben az adatok elkülönülnek a látványtól.
- 8.7. Mely állítások igazak? Jelölje be az összes jó választ!
- a) A menühierarchiában egy `JMenu` van legfelül.
 - b) A `JMenuItem` az `AbstractButton` leszármazottja.

- c) A menütételnek lehet ikonja.
d) Ha kiválasztják a menütételt, akkor keletkezik rajta `MouseEvent` esemény.
- 8.8. Mely állítások igazak? Jelölje be az összes jó választ!
- A `JDialog` ablakba közvetlenül az `add` metódusral tehetők be a komponensek.
 - A `JDialog` a `Window` utódja.
 - A dialógusablak minden modális.
 - A `JOptionPane.showMessageDialog` metódusban a paraméterezéstől függ a nyomógombok száma.
- 8.9. Mely állítások igazak az időzítőre (`Timer` osztályra)? Jelölje be az összes jó választ!
- A `Timer` osztály implementálja az `ActionListener` interfészt.
 - A `Timer` objektum figyelője lehet bármely olyan objektum, amelynek osztálya az `Object` leszármazottja.
 - A `Timer` figyelőjét csak az `addActionListener` metódussal lehet az időzítőhöz kapcsolni.
 - Az időzítő adott időközönként `ActionEvent` eseményeket kelt, s azokat a figyelő kezeli az `actionPerformed` metódussal.

Feladatok

Minden esetben tervezze meg a felhasználói interfést! Ha szükséges, készítse el a program osztálydiagramját is!

Címke

- 8.1. (A) Készítse el az itt látható keretet (szalad a herceg kávét inni)!
Használja fel a `duke.running.gif` és a `java_logo.gif` ikont! A herceg magától rohan, a rohanás bele van építve az animált ikonba. (*SzaladaHerceg.java*)



Nyomógomb

- 8.2. (A) Tegyen a keretbe egy nyomógombot; azon kezdetben a 0 szám álljon, jelezvén, hogy még egyszer sem volt lenyomva. S minden lenyomás után növekedjék meg egyvel a gombon levő szám! (*HanyszorNyomtak.java*)
- 8.3. (B) Készítsen egy keretet; öt gomb legyen rajta, az 1, 2..., 5 felirattal. A gombok nyomkodására a keret alján sorban jelenjen meg egy címkén a lenyomott gombok sorszáma, a lenyomások sorrendjében! Látni lehessen a nyomkodások egész történetét, illeszformán: 42151113. Ha a történet nem fér ki a sorba, szélesítsük az ablakot! A gombok egy tömb elemei legyenek!

A feladat módosítása: A program paraméterként kapja meg a gombok számát! Ha a felhasználó nem adott meg paramétert, akkor a gombok száma legyen 5! (*PressHistory.java*)

Jelölőmező, rádiógomb

- 8.4. (A) Adjon a keretben módöt a felhasználónak arra, hogy megszabassa, méretezhető legyen-e a keret vagy sem. S ennek megfelelően legyen a keret méretezhető vagy nem méretezhető. (*Meretezhetoseg.java*)
- 8.5. (B) Tegyen a keretbe egy piros, egy kék és egy sárga golyót (az icons könyvtárban megtalálja őket)! Tegye lehetővé, hogy a felhasználó
- a golyókat egymástól függetlenül eltüntesse, illetve újra láthatóvá tegye! (*Golyok1.java*)
 - válasszon, melyik golyót akarja látni (egyszerre minden csak egyet)! (*Golyok2.java*)
- 8.6. (B) Legyen a keret alján 5 jelölőmező, s mindegyik egy-egy nyelvnek feleljen meg (angol, magyar, kínai, német és szlovák). A keret felső részében jelenjen meg
- sorban az aktuálisan kiválasztott összes nyelv! Egyszerre akárhány nyelvet be lehet jelölni. (*Nyelvek1.java*)
 - az aktuálisan kiválasztott nyelv! Egyszerre csak egy nyelvet lehet bejelölni. (*Nyelvek2.java*)



Kombinált lista

- 8.7. (A) Készítsen egy mondatösszeállító alkalmazást! Tegyen a keretbe két kombinált listát: az elsőből egy keresztnévet lehessen választani, a másodikból egy igét. A keretben az aktuális kiválasztástól függően jelenjen meg a következő mondat:
 <KERESZTNÉV>! Menj el szépen <IGE>!
 (*Mondat.java*)

Szövegmező

- 8.8. (A) Tegyen a keretbe egy szövegmezőt, majd egy-egy piros és narancsszínű nyomógombot "Meggy", illetve "Narancs" felirattal! Ha valamelyik gombot megnyomják, akkor kerüljön a felirat a szövegmezőbe! A szövegmező ne lehessen szerkeszteni. A fókusz kezdetben a második gombon legyen! (*MeggyNarancs.java*)

- 8.9. (B) Jelenítsünk meg egy szöveget a keret alsó sorában! A keret felső sora egy eszközsor legyen, s aznal állítani lehessen a szöveg következő paramétereit:

- Szöveg tartalma. Kezdetben: "Mintaszöveg".
- Betűtípus: 3-féle
- Betűméret: 10, 12, 14... 50
- Betűstílus: vastag-e (Bold), dőlt-e (Italic)
- Szöveg színe: black, white, red, magenta vagy orange
- Igazítás: Left, Center, vagy Right

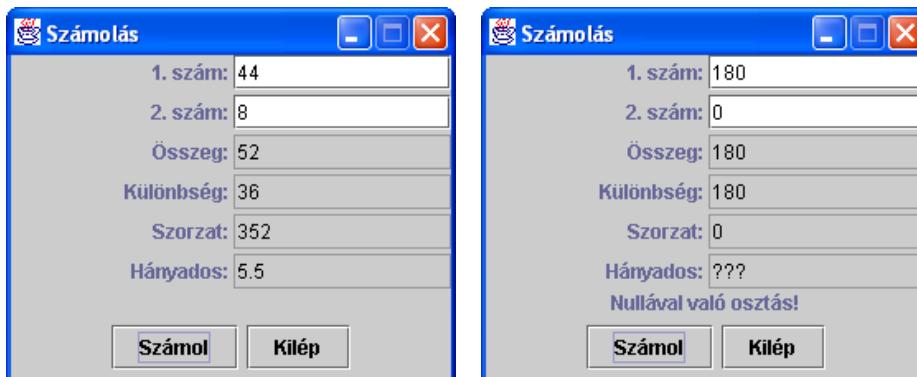
Minden beállítás után változzon meg a szöveg!

(*SzovegStilus.java*)

- 8.10. (A) Készítsen egy programot két szám szorzásának elvégzésére! A szorzandó számokat egy-egy szövegmezőbe lehessen beírni. A szorzat gombnyomásra jelenjen meg egy eredménymezőben! (*Szorzas.java*)

- 8.11. (B) Készítsen olyan programot, amelyben két szám között el lehet végezni bármelyik alapműveletet. A 4 alapművelet közül egy lenyíló listából lehessen választani. Ha a művelet nem végezhető el, akkor az eredménymezőben a "Hiba!" szöveg jelenjen meg! A keretet pontosan a képernyő közepén helyezze el, a képernyő felbontásától függetlenül! (*Muveletek.jpx*)

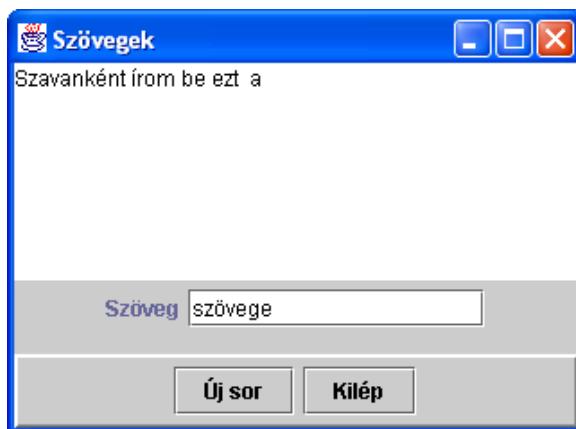
- 8.12. (B) Készítse el a következő számolóablakot. Ha valamelyik szám hibás vagy nullával osztanánk, akkor a program jelenítsen meg egy figyelmeztető szöveget! A számolt értékeket ne lehessen szerkeszteni! (*Szamolo.java*)



Szövegterület

- 8.13. (B) A keret nagy részét egy szövegterület foglalja el; alatta egy beviteli mező legyen, s abba lehessen szöveget beírni. Ha lenyomjuk az Enter-t, akkor a beviteli mező szövege jelenjen meg a szövegterületen, minden az előzőleg beírt szöveg után, szóközzel elválasztva! Ezután a beviteli mező ürüljön ki, hogy újra beírhassunk egy szöveget! Legyen az alsó sorban egy Új sor gomb is: annak a lenyomására legyen soremelés a szövegterületen! S ez ismétlődjön mindenkor, amíg a keret alján levő Kilép gombot meg nem

nyomják – ekkor csukódjék be a keret! A felső szövegterületre ne lehessen írni. A fókusz kezdetben a beviteli mezőn legyen! (*Szovegek.java*)



- 8.14. **(A)** Tegyen egy keretbe egymás alá két szövegterületet! Az alsó szöveg legyen mindenkor ugyanaz, mint a felső! Az alsó szöveget ne lehessen szerkeszteni. (*TukorSzoveg.java*)
- 8.15. **(B)** Az ablak felső részén egy nem szerkeszthető, alsó részén egy szerkeszthető szövegterület legyen. Az alsó területre bekezdéseket írunk be. Amikor leütjük az Entert, akkor a bekezdés jelenjen meg a felső szövegmezőben az előző bekezdés után, és a beviteli szövegterület legyen újra üres! Mindkét szövegterület legyen görgethető, és legyen bennük sortörés (linewrap)! A fókusz kezdetben legyen az alsó szövegterületen! (*Paragrafusok.java*)
- 8.16. **(B)** Készítsen egy írógépprogramot! Az ablak alsó részén billentyűzet legyen, felső részén egy görgethető szövegterület. Ha „leütünk” egy billentyűt, akkor az annak megfelelő betű jelenjen meg a szövegterületen. A klaviatúrán legyenek vezérlőbillentyűk is! (*Irogep.java*)

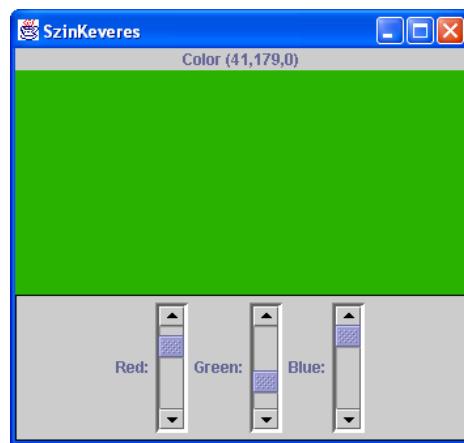
Lista

- 8.17. **(A)** A keretbe egy italokat tartalmazó lista kerüljön, feltöltéséről a programból fogunk gondoskodni. A listából lehessen italokat kiválasztani egy ital kikeveséséhez, egyszerre akárhány italt. A listából kiválasztott italok jelenjenek meg a keret alsó részén, egy szövegmezőben! A szövegmezőt ne lehessen szerkeszteni. (*Italkeveres.java*)
- 8.18. **(B)** A program segítségével olvasmányokat szeretnénk bevinni, s azok egymás után jelenjenek meg az Olvasandók listájában, ahogyan sorban bevittük őket. Ebből a listából szeretnénk olvasmányokat áttenni az Olvasottak listájába. Az áttett olvasmány törlődjék az Olvasandók listájából! (*Olvasmanyok.java*)

- 8.19. (B) Az ablak felső részében legyen egy neveket ábécérendben felsoroló lista. Az ablak alsó részén egy beviteli mezőből újabb nevet lehessen listára felvenni; a listából törölni is lehessen az e célból megjelölt neve(ke)t. (*Nevek.java*)

Görgetősáv

- 8.20. (A) Készítsen egy színkeverő alkalmazást! Az RGB színeket a keret alsó részén levő három görgetősáv segítségével lehessen állítani. A keret felső részének színe mindenkor a kikevert szín legyen! (*SzinKeveres.jpx*)



Menü

- 8.21. (A) Készítse el a számítógépen található valamelyik játékprogram menüjét! Például:

```

Játék
  Új játék
  └──
    • Kezdő
    ○ Haladó
    ○ Mester
  └──
  Kilépés

Súgó
  Tartalom
  Témakörök...
  A nap tippje
  └──
  Névjegy

(JatekMenu.java)

```

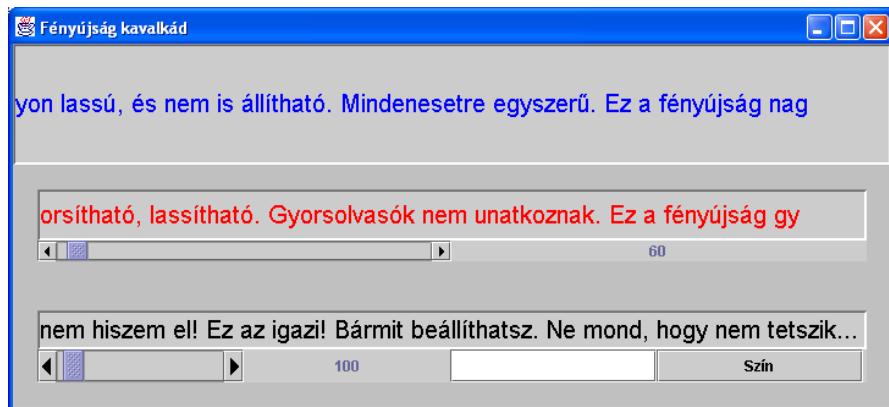
Dialógusablak

- 8.22. (A) A Java beépített lehetőségeit felhasználva jelenítse meg következő dialógusokat!
- a) Hibakijelzés. Cím: nincs; Megjelenő szöveg: "Nem süt a nap!"; Gomb: OK.
 - b) Kérdés. Cím: nincs; Megjelenő kérdés: "Ön a tengerparton van?"; Gomb: Yes, No.
Ha a Yes-t ütik le, akkor
 - c) Információ. Cím: "Csak hogy tudja..."; Megjelenő szöveg: "Én is megyek."; Gomb: OK.
 - d) Kérdés. Cím: "Diszkréció"; Megjelenő kérdés: "Mennyit keres havonta euróban?"; Gomb: OK. Ha 100 000-nél kevesebbet vagy hibás adatot ír be, akkor
 - e) Figyelmeztetés. Cím: "Csalódás"; Megjelenő szöveg: "Mégsem megyek"; Gomb: Rendben. (Magyar feliratú gomb!)

A dialógusok a fő keret közepén jelenjenek meg, az utolsó kettő azonban ne ott, hanem a keret közepén! (*Dialogusok.java*)

Időzítő

- 8.23. (A) Tegyen a keretbe egy ütemesen villogó (megjelenő és eltűnő) feliratot!
- a) Az időzítőt a keret kezelje! (*Villogas1.java*)
 - b) Az időzítőt egy névtelen osztály példánya kezelje! (*Villogas2.java*)
- 8.24. (B) Készítsen olyan játékot, amelyben egy „Kapj el!” feliratú gomb ugrik 0,8 másodpercenként a képernyő egy véletlenszerűen kiválasztott helyére! A cél a gomb megnyomása. Ha ez sikerül, akkor íródjék ki a gombra a „Jaj!” szöveg és egy szám: az, hogy hány ugrálás után sikerült elcsípni a gombot! Ekkor a keretre is íródjék ki címként, hogy „Elkaptalak!”! A játék kerete kezdetben foglalja le a teljes képernyőt! (*UgraloGomb.java*)
- 8.25. (B) Készítsen hirdetőtáblát! A táblán sorban jelenjenek meg az előre eltárolt szövegek, s minden szöveget valamekkora megadott ideig lehessen látni. Közben a szöveg másodpercenként válson színt, s új szöveg megjelenésekor kezdődjön előlről a színváltás! A hirdetési szöveg 30 pontos Arial betűből legyen szedve. A hirdetést a tábla alján levő gomb segítségével lehessen „lelöni”. (*HirdetoTabla.java*)
- 8.26. (C) Tegyen a keretbe egymás alá három fényújságot! A fényújság jobbról balra haladva folyamatosan léptet egy szöveget. A fényújság szövegét, a betűk színét és a sebességet (léptetési időt) inicializáláskor adjuk meg.
- Az első fényújság egy egyszerű fényújság legyen.
 - A második fényújságot egy hozzácsatolt görgetősávval lehessen gyorsítani, és minden íródjék ki, hogy mekkora a pillanatnyi sebesség.
 - A harmadik fényújságnak szövegét, a színét és a sebességét is lehessen állítani. (*Fenyujsagok.jpx*)



9. Grafika, képek

A fejezet pontjai:

1. Rajzolás
 2. Mintaprogram
 3. A Graphics osztály
 4. Sokszög rajzolása – a Polygon osztály
 5. Képek – az absztrakt Image osztály
-

A Javában csak a rendszer által felkínált grafikus felületre, környezetre (graphics context), lehet írni és rajzolni – a programozó nem húzhat, mondjuk, vonalat a képernyő tetszőleges két pontja között. A program minden komponenshez magától felkínál egy `Graphics` osztályú objektumot, s a programozó ezt használhatja valaminek a megjelenítésére. Ebben a fejezetben lényegében a `Graphics` osztály adta megjelenítési lehetőségekről lesz szó. A komponensekre, illetve azok grafikus objektumaira szöveget fogunk írni; egyeneseket, téglalapokat, sokszögeket, ellipsziseket és íveket rajzolunk különböző színekkel, kitöltve vagy kitöltetlenül. Arról is szó lesz, hogyan lehet a felületeken képeket megjeleníteni. A grafikus lehetőségeknek itt csak egy kis részhalmazát mutatjuk be – a programozó sok további osztály, illetve csomag közül választhat, ha látványos, többdimenziós grafikus felületeket, animációt és multimédiás programot szeretne készíteni.

9.1. Rajzolás

A `java.awt.Graphics` absztrakt osztály; grafikus műveletekhez (rajzolás, képmegjelenítés stb.) ad deklarációkat, s azokat külön-külön implementálták a használatos platformokra. minden komponensnek van a `Graphics` valamely leszármazottjából való grafikus objektuma (felülete). Egy komponens grafikus objektumát a rendszer hozza létre, Ő felügyeli és adja át a programnak, hogy az rajzolhasson rá – ez a biztosíték arra, hogy csak az operációs rendszer által elfogadott műveletek játszódhassanak le. A grafikus objektumot többek között az aktuális rajzolószín és a betűtípus jellemzi.

A gr grafikus objektumra például így rajzolhatunk:

```
// Egyenes húzása a (10,20) és a (100,20) pont között:  
gr.drawLine(10,20,100,20);
```

Egy komponensen lényegében kétféleképpen lehet valamit megjeleníteni:

- A JComponent osztály getGraphics() metódusával elkérjük a komponenstől a grafikus felületét. Ez a metódus egy Graphics objektumot ad vissza, s arra rajzolunk. Ilyenkor a rajz csak egyszer jelenik meg, a komponens újraraajzolásakor egyszerűen eltűnik.
- Felülírjuk a JComponent osztály paintComponent(Graphics gr) metódusát. A metódus paraméterben kínálja fel a komponens grafikus objektumát; most arra rajzolhatunk. Az alkalmazás minden olyan helyzetben meghívja ezt a paintComponent() metódust, amelyben a célfelületet frissítenie kell, például a felhasználó előtérbe helyez, elmozdít vagy átméretez egy komponenst.

Rajzoláskor a grafikus objektum ügyel arra, hogy a program ne írhasson rajta kívül eső területekre – az oda eső alakzatok egyszerűen nem rajzolódnak ki. A grafikus objektum továbbadható más metódusnak is, hogy az rajzoljon rá.

A komponens gr:Graphics objektuma lefedi a komponens teljes felületét. A grafikus objektum bal felső sarkának két koordinátája a (0,0); jobb alsó sarkának koordinátáit a komponens mérete határozza meg. minden koordináta pixelben értendő. Ablak keretére nem lehet rajzolni.

Bármely komponensre lehet rajzolni, de általában a JComponent osztály leszármazottjára, a JPanel-re szokás.

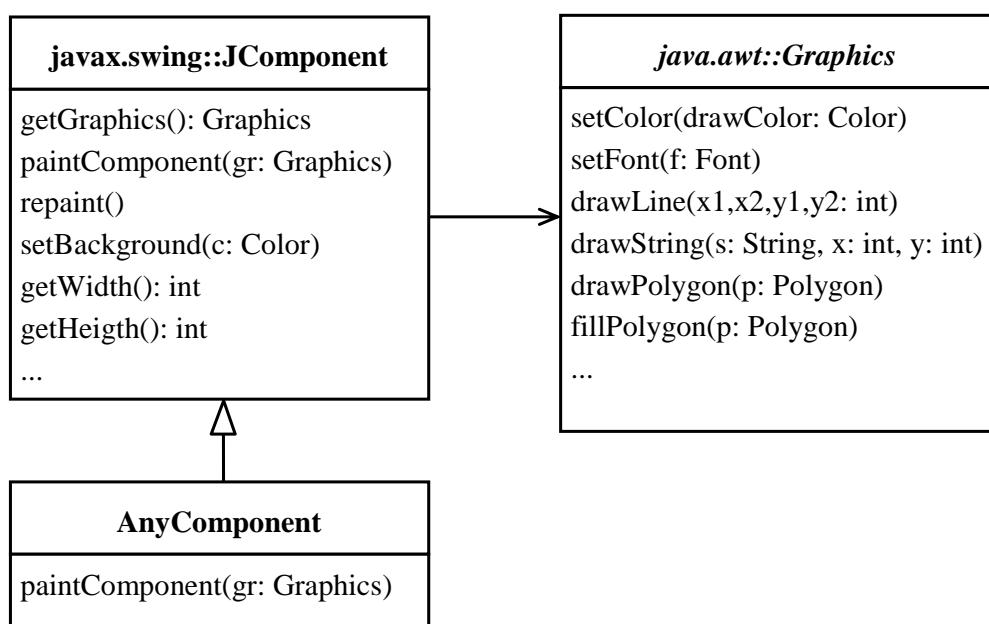
A következő programrészlet elkéri egy tetszőleges komponenstől a grafikus objektumát, majd egy vízszintes egyenest rajzol rá:

```
JComponent comp = new ...; // bármilyen komponens létrehozása  
...  
Graphics gr = comp.getGraphics();  
gr.drawLine(10,20,100,20);
```

Ez a megoldás nem gondoskodik arról, hogy az egyenes újra kirajzolódjék a komponens elmozgatása, átméretezése stb. után. Ahhoz, hogy rajzunk ilyenkor is a komponensen maradjon, a JComponent osztály paintComponent() metódusát kell felülírnunk:

```
protected void paintComponent(Graphics gr) {  
    super.paintComponent(gr);  
    gr.drawLine(10,20,100,20);  
}
```

A 9.1. ábra összefoglalja a rajzoláskor használt legfontosabb metódusokat, és azok helyét az osztályhierarchiában. Látható, hogy rajzolásra minden komponensnek van egy `Graphics` osztályú grafikus objektuma. A rajzoláshoz vagy felülírjuk a komponens `paintComponent()` metódusát – az paraméterben tálalja ezt a grafikus objektumot –, vagy magát a grafikus objektumot kérjük el a `getGraphics()` metódussal.



9.1. ábra. A rajzoláshoz szükséges fontosabb metódusok

JComponent metódusok

- ▶ `Graphics getGraphics()`

Visszaadja a komponens grafikus felületét. Ha a komponens nem jeleníthető meg (`isDisplayable() == false`, mert például az öt tartalmazó ablakot a `dispose` metódussal felszabadították), akkor a metódus `null` értéket ad vissza.

- ▶ `protected void paintComponent(Graphics gr)`

Ebben a metódusban szokás elhelyezni a rajzolóutasításokat. Ez a metódus mindenkor meghívódik, valahányszor újra kell rajzolni a komponenst. A metódus paraméterében adja meg a grafikus felületet. A rajzolás előtt meg kell hívni az előző `paintComponent` metódust (az öst), mert különben ott maradhat az előző kép.

- ▶ `void repaint()`

Újrarájzolja a komponenst. Akkor szokás meghívni, ha az operációs rendszer nem veheti észre az újrarájzolás szükségeségét – például ha a program belső adatai változnak meg, s azokat tükröznie kell a komponens képének.

Megjegyzések:

- A megjeleníthetőség nem tévesztendő össze a láthatósággal! Egy nem látható (`visible == false`), de megjeleníthető komponensre nyugodtan lehet rajzolni.
- Ha nem a `paintComponent` metódussal rajzoltatunk, akkor a rajz csak addig marad a komponensen, ameddig meg nem hívódik a `paintComponent` metódus; az a benne megadott utasításokkal újrarendezze a komponenst!
- A `JFrame` nem leszármaztatja a `JComponent`-nek: nincs is benne `paintComponent` metódus. A keretre nem lehet rajzolni, csak a tartalompanelre és annak gyerekeire!

9.2. Mintaprogram

Feladat – Grafikateszt (GraphTest)

A keret legyen pasztellzöld színű. Rajzolunk a keretbe egy fekete ellipszist, s köréje egy fekete téglalapot! Írjuk az ellipszisbe a "Végre grafika!" szöveget fehér színű, vastag, 40 pontos, Arial betűvel! A szöveget húzzuk alá fehérrel. A keret aljára húzzunk egy vízszintes, vastag, narancsszínű vonalat!



Forráskód

```

import java.awt.*;
import javax.swing.*;

class RajzPanel extends JPanel {

    protected void paintComponent(Graphics gr) {           //1
        super.paintComponent(gr);                         //2
        setBackground(new Color(130,180,160));           //3
        gr.drawOval(40,80,320,100);                      //4
        gr.drawRect(40,80,320,100);                     //5
        gr.setColor(Color.WHITE);                        //6
        gr.setFont(new Font("Arial",Font.BOLD,40));      //7
        gr.drawString("Végre grafika!",60,140);          //8
        gr.drawLine(60,140,340,140);                    //9
        gr.setColor(Color.ORANGE);                       //10
        gr.fillRect(0,200,500,10);                      //11
    }
}

```

```
public class GraphTest extends JFrame {  
    public GraphTest() {  
        setBounds(500,200,400,280);  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        getContentPane().add(new RajzPanel());  
        show();  
    }  
  
    public static void main (String args[]) {  
        new GraphTest();  
    } // main  
} // GraphTest
```

A program elemzése

A programban beállítottuk a keret méretét, és felülírtuk a keretbe tett panel paintComponent metódusát. Elemezzük a metódus sorait:

- ◆ //1: A metódus paramétere egy Graphics típusú objektum. A gr objektumra való írással a panelre írunk, hiszen annak írjuk felül a paintComponent() metódusát. A gr konkrét osztálya a Graphics osztály egy olyan leszármazottja, amelyet az alkalmazás majd felkínál a paintComponent () végrehajtásakor. A gr objektumnak olyan tulajdonságai vannak, mint az aktuális rajzolószín, a betűtípus stb.
- ◆ //2: Meghívjuk a paintComponent metódus ösét; az elvégzi a szükséges előmunkálásokat, például letörli a komponenst.
- ◆ //3: Beállítjuk a panel háttérszínét. Figyelem! Itt nem a gr objektum tulajdonságát állítjuk be, hanem a panelét. A gr objektumnak nincsen háttérszíne, csak rajzolószíne.
- ◆ //4: A gr grafikus felületre rajzolunk egy fekete ellipszist, amely a (40,80) bal felső sarkú és 320*100 méretű téglalapba illeszkedik. A gr alapértelmezés szerinti rajzolószíne a panel rajzolószíne, most tehát a fekete. A rajzolás és kitöltés minden az aktuális rajzolószínnel folyik.
- ◆ //5: Egy fekete téglalapot rajzolunk a gr grafikus felületre, bal felső sarka a (40,80) pont, a mérete 320*100.
- ◆ //6: Átállítjuk gr rajzolószínét. minden további rajz fehér színű lesz.
- ◆ //7: Beállítjuk a gr aktuális betűjét 40 pontos, félkövér Arial típusra. minden további szöveg ilyen betűvel íródik majd ki.
- ◆ //8: Ráírjuk a gr objektumra a (60,140) ponttól kezdve a "Végre grafika!" szöveget. A pont a szöveg alapvonalának kezdő, bal oldali koordinátája.
- ◆ //9: Színtelen fehér színnel egy vonalat húzunk a (60,140) és a (340,140) pontok közé, ezzel húzzuk alá az előzőleg kiírt szöveget.
- ◆ //10: Narancssárgára (orange) állítjuk át a rajzolószínt.
- ◆ //11: Rajzolunk egy kitöltött téglalapot; bal felső sarka a (0,200) pont, a mérete 500*10. A téglalap méretét szándékosan vettük nagyobbra a komponensnél: figyelje meg, hogy ami nem fér rá a gr objektumra, az nem jelenik meg!

9.3. A Graphics osztály

Csomag: java.awt Deklaráció: public abstract class Graphics

Közvetlen ős: java.lang.Object

Fontosabb implementált interfések: -

A Graphics absztrakt osztály, minden platformra külön implementálni kell. minden komponensnek van grafikus objektuma (felülete), s az a Graphics egy leszármazott osztályából való. A komponensek grafikus objektumát a rendszer hozza létre és felügyeli. Egy Graphics objektum állapotát a következő dolgok határozzák meg:

- **Komponens**, amire rajzolunk.
- **Rajzolószín** (draw color): A következő rajz ilyen színű lesz.
- **Betű** (font): A következő írás ilyen betűvel jelenik meg.
- **Kivágási terület** (clipping area): A komponens részét alkotó téglalap alakú terület. minden rajzolási művelet erre a kivágási területre irányul, az ezen a területen kívüli rajzolás hatástalan. A kivágási terület kezdetben a keret nélküli komponens teljes területe. A kivágási terület programból szabályozható.
- **Eltolási pont** (translation origin): Viszonyítási koordináta, az aktuális kirajzolásban minden ez a (0,0) pont. Az eltolási pont révén könnyebb kirajzolni az idomokat és a programot is egyszerűbb módosítani.
- **Rajzolási mód** (paint mode): Kétféle módban rajzolhatunk: felülíró vagy XOR módban. Felülíró módban a kirajzolt alakzat színe független az alatta levő rajzoktól; XOR módban a kirajzolt pont tényleges színe az alatta levő pont színétől függ.

A java.awt.Graphics osztály metódusaiban megadott koordinátáknak a komponens bal felső sarka a viszonyítási pontjuk, illetve az eltolási pont, ha az különbözik a (0,0)-tól. A rajzolt elemeknek a komponens keretére, illetve szegélyére eső része nem jelenik meg. Általában is igaz, hogy a megengedett felületen kívülre való rajzolás nem okoz bajt, csak nem fog megjelenni a rajzolt elem. minden rajzolás a grafikus környezet legutoljára beállított állapota (rajzolószín, betű stb.) szerint folyik. A koordináták és eltolások megadásában negatív értékek is írhatók.

A Graphics osztály jó néhány metódusa absztrakt. A programozó olyan grafikus környezetre rajzol, amely a Graphics osztály egy konkrét platformon implementált leszármazottjából való. Hogy ez a leszármazott pontosan milyen osztályú, azt a programozónak nem kell tudnia – a program ettől lesz platformfüggetlen.

Megjegyzés: A Graphics osztály absztrakt metódusait nem jelöljük; a programozó már mindenéppen implementált metódust hív meg.

Konstruktur

Az osztály absztrakt, nem példányosítható.

Metódusok

- ▶ `void setColor(Color c)`
- ▶ `Color getColor()`

A grafikus felülethez tartozó aktuális rajzolószín beállítása, illetve lekérdezése.

- ▶ `void setFont(Font font)`
- ▶ `Font getFont()`

A grafikus felülethez tartozó aktuális betű beállítása, illetve lekérdezése.

- ▶ `void setClip(int x, int y, int width, int height)`
- ▶ `Rectangle getClipBounds()`

Beállítja, illetve visszaadja az aktuális kivágási területet. Ha a komponensnek nincs szegélye és a kivágási területet nem állítottuk át, akkor ez a metódus a komponens határait adja vissza.

- ▶ `void translate(int x, int y)`

Eltolja a grafikus felület $(0,0)$ pontját a megadott értékekkel. minden további rajzolás az új eltolási ponttól folyik. Vigyázat! Az eltolás az előző eltolási pontot veszi alapul; nem a viszonyítási pont abszolút megadásáról van szó!

- ▶ `void copyArea(int x, int y, int width, int height, int dx, int dy)`
Átmásol egy téglalapot a grafikus felület egy másik részére. A másolandó téglalap bal felső sarka az (x,y) , mérete $(width, height)$. Az eredményül kapott téglalap (dx,dy) távolsággal eltolva jelenik meg.

- ▶ `void setPaintMode()`

Felülíró rajzolási mód megadása. Hatására a következő rajzolóutasítások egyszerűen felülírják a már kirajzolt képet az aktuális színnel.

- ▶ `void setXORMode(Color color)`

Alternatív, XOR rajzolási mód megadása – ettől kezdve így megy minden rajzolás. Tegyük fel, hogy `paintColor` az aktuális rajzolószín! Ha a kirajzolandó pont alatti pont `paintColor` színű, akkor a pont `color` színű lesz; s ha a kirajzolandó pont alatt `color` szín van, akkor a pont maga `paintColor` színű lesz. Egyéb színek esetén megjósolhatatlan a pont kirajzolt színe, de az mindig igaz, hogy újrarajzolással a pont színe az előző lesz. Ha tehát egy alakzatot kétszer kirajzolunk, akkor visszakapjuk az eredeti állapotot.

- ▶ `void drawLine(int x1, int y1, int x2, int y2)`

Egyenes húzása a felületre az aktuális rajzolószínnel az $(x1,y1)$ és az $(x2,y2)$ pontok között (beleértve a pontokat is).

- ▶ `void drawString(String str, int x, int y)`

Szöveg írása a rajzolófelületre. `str` a kiírandó szöveg, (x,y) a szöveg alapvonalanak bal szélénél pontja. A szöveg alapvonala közvetlenül a „nem alálóból” betük alatt húzódik.

- ▶ `void drawRect(int x, int y, int width, int height)`
- ▶ `void fillRect(int x, int y, int width, int height)`

Kirajzol egy téglalapot az aktuális rajzolószínnel. A téglalap bal felső sarka (x, y), mérete $width*height$. A `drawRect` a téglalapnak csak a szegélyét rajzolja meg, a `fillRect` ki is tölti ugyanazzal a színnel.

- ▶ `void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)`
- ▶ `void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)`

Téglalapot rajzol az aktuális rajzolószínnel; a téglalap bal felső sarka (x, y), mérete $width*height$. Az `arcWidth` és `arcHeight` paraméter a téglalap négy sarkán levő kerekítőkörök (ellipszisek) vízszintes, illetve függőleges átmérője.

- ▶ `void draw3DRect(int x, int y, int width, int height, boolean raised)`
- ▶ `void fill3DRect(int x, int y, int width, int height, boolean raised)`

Térhatású téglalapot rajzol. Ha `raised` értéke `true`, akkor a téglalap kiemelkedik, ha `false`, akkor besüllyed.

- ▶ `void clearRect(int x, int y, int width, int height)`
Átfesti a téglalap területét a komponens aktuális háttérszínével.
- ▶ `void drawOval(int x, int y, int width, int height)`
- ▶ `void fillOval(int x, int y, int width, int height)`

Ellipszist rajzol az aktuális rajzolószínnel. Az ellipszist az ($x, y, width, height$) téglalap határolja. A `drawOval` az ellipszisnek csak a keretét rajzolja meg, a `fillOval` ki is tölti azt (a rajzolószínnel).

- ▶ `void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)`
- ▶ `void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)`

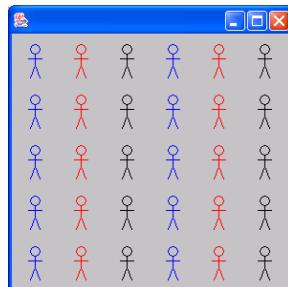
Az ellipszis egy ívének meghúzása, illetve az ellipsziscikk kitöltése. Az ellipszist az ($x, y, width, height$) téglalap határolja. Az ívszakasz kezdetét és terjedelmét egy-egy szöggel (`startAngle`, `arcAngle`) adjuk meg. A szögek fokokban értendők, a 0 fok az ellipszis vízszintes tengelyének jobb oldalát mutatja, a 90 fok a függőleges tengely tetejét stb. Figyelem! A `Math` osztály függvényei radiánban dolgoznak!

Megjegyzés: A kitöltőmetódusok (mint például a `fillRect`) jobboldalt és alul egy egyenes-sel kevesebbet rajzolnak, mint a megfelelő kirajzolómetódusok (például a `drawRect`), vagyis nem húzzák meg a jobb és az alsó határvonalat.

Még számos metódus létezik. A sokszög rajzolásához és a kép megjelenítéséhez szükséges metódusokat külön pontok tárgyalják.

Feladat – Emberkék

Tegyünk a keretbe különböző színű pálcikaemberkéket, egyenletesen elosztva!

**Forráskód**

```

import java.awt.*;
import javax.swing.*;

class Emberke extends JPanel {
    private Color color;

    public Emberke(Color color) { //1
        this.color = color;
    }

    protected void paintComponent(Graphics gr) { //2
        // A (0,0) pont az emberke szíve helye:
        super.paintComponent(gr);
        gr.translate(getWidth()/2,getHeight()/2);
        gr.setColor(color);
        gr.drawOval(-5,-15,10,10); // emberke feje
        gr.drawLine(0,-5,0,7); // törzse, függőleges egyenes
        gr.drawLine(-7,0,+7,0); // két keze
        gr.drawLine(0,7,-5,20); // bal lába
        gr.drawLine(0,7,5,20); // jobb lába
    }
}

class EmberkekPanel extends JPanel {
    public EmberkekPanel() {
        setLayout(new GridLayout(5,0));
        setBackground(Color.LIGHT_GRAY);
        for (int i=0; i<10; i++) { //3
            add(new Emberke(Color.BLUE));
            add(new Emberke(Color.RED));
            add(new Emberke(Color.BLACK));
        }
    }
}

public class Emberkek extends JFrame {
    public Emberkek() {
        setBounds(100,100,300,300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        getContentPane().add(new EmberkekPanel());
        show();
    }
}

```

```
public static void main (String args[]) {
    new Emberkek();
} // main
} // Emberkek
```

A program elemzése

Mindegyik ember külön panelen jelenik meg; színükkel inicializáljuk őket (//1). minden ember „képe” egyforma, s ezt a képet az osztály `paintComponent()` metódusában adjuk meg (//2). Az ember rajzolásában a viszonyítási pont az ember szíve, ez a ráfektetett koordinátarendszer (0,0) pontja. //3-ban sorba betesszük az embereket a keret ráccselláiba, összesen harmincat.

9.4. Sokszög rajzolása – a Polygon osztály

Polygon osztály

Csomag: `java.awt` Deklaráció: `public class Polygon`

Közvetlenős: `java.lang.Object`

Közvetlenül implementált interfések: `Serializable`, `Shape`

A `Polygon` osztály egy sokszöget határoz meg; példányait a `Graphics` osztály sokszögrajzoló metódusai rajzolják ki a grafikus felületre. A csúcsok összekötése a deklarálás sorrendjében halad. A megadott csúcsokkal bizonyos metódusok zárt, mások nyílt sokszöget rajzolnak. A zárt sokszög kitölthető. Az osztály tárolja a sokszög csúcsainak számát és a csúcsok koordinátáit.

Mezők

- ▶ `Rectangle bounds`
A sokszög határoló téglalapja.
- ▶ `int npoints`
A sokszög csúcsainak száma.
- ▶ `int[] xpoints`
▶ `int[] ypoints`
A csúcsok x, illetve y koordinátájának tömbje.

Konstruktörök

- ▶ `Polygon()`
Egy üres sokszöget hoz létre.
- ▶ `Polygon(int[] xpoints, int[] ypoints, int npoints)`
Létrehoz egy `npoints` számú csúcs alkotta sokszöget a megadott csúcsokkal.

Metódusok

- ▶ `void addPoint(int x, int y)`
Újabb csúcs hozzáadása a sokszöghöz.
- ▶ `boolean contains(int x, int y)`
Megadja, hogy az (x,y) pont benne van-e a sokszögben (rajta van-e a sokszög területén).

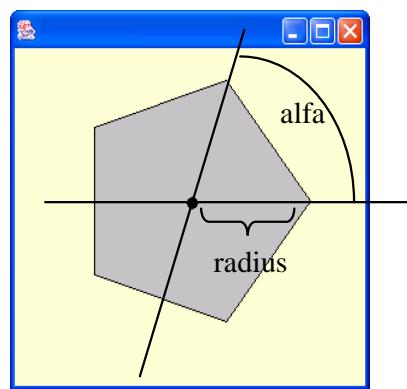
Graphics-metódusok a sokszög megrajzolására

- ▶ `void drawPolygon(int xPoints[], int yPoints[], int nPoints)`
- ▶ `void drawPolygon(Polygon p)`
- ▶ `void drawPolyline(int xPoints[], int yPoints[], int nPoints)`
- ▶ `void fillPolygon(int[] xPoints, int[] yPoints, int nPoints)`
- ▶ `void fillPolygon(Polygon p)`

Sokszög rajzolása a megadott csúcsok alapján. A csúcsok összekötése a koordináták indexének növekvő sorrendjében halad; az oldalszakaszok az aktuális rajzolószínnel rajzolódnak meg. A `drawPolygon` bezárja a sokszöget, vagyis az utolsó pontot összeköti az elsővel. A `drawPolyline` nyitva hagyja a sokszöget. A `fillPolygon` metódus kitölți a sokszöget az aktuális rajzolószínnel.

Feladat – Polygon-teszt

Rajzolunk egy világosszürke szabályos sokszöget egy tojásszínű keret közepére; a sokszög középpontját jelöljük meg! A sokszög szögeinek a számát és a köré írható kör sugarát a program paramétereként adjuk meg! Ha nem jók a paraméterek, akkor a program írja ki, hogyan kell őt használni!



Forráskód

```
import javax.swing.*;
import java.awt.*;

class PolygonPanel extends JPanel {
    private int n;          // a szabályos sokszög szögeinek a száma
    private int radius;     // a sokszög köré írható kör sugara
    private Polygon poly;   // a sokszög
```

```

public PolygonPanel(int n, int radius) {
    this.n = n;
    this.radius = radius;
    int alfa = 360/n; // egy cikk szöge fokban

    poly = new Polygon(); //1
    for (int i=0; i<n; i++) {
        int x=(int)(radius*Math.cos(Math.toRadians(i*alfa)));
        int y=(int)(radius*Math.sin(Math.toRadians(i*alfa)));
        poly.addPoint(x,y); //2
    }
}

protected void paintComponent(Graphics gr) {
    super.paintComponent(gr);
    gr.translate(getWidth()/2,getHeight()/2); //3
    setBackground(new Color(255,255,210));
    gr.setColor(Color.LIGHT_GRAY);
    gr.fillPolygon(poly); //4
    gr.setColor(Color.BLACK);
    gr.drawPolygon(poly); //5
    gr.fillOval(-5,-5,10,10); //6
}
}

public class PolygonTest extends JFrame {

    public PolygonTest(int n, int radius) {
        setBounds(500,200,300,300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        getContentPane().add(new PolygonPanel(n, radius));
        show();
    }

    public static void main (String args[]) {
        try {
            int n = Integer.parseInt(args[0]);
            int radius = Integer.parseInt(args[1]);
            new PolygonTest(n, radius);
        }
        catch (Exception e) {
            JOptionPane.showMessageDialog(null,
                "A program használata: \nPolygonTest n radius." +
                "\n\nPéldául : PolygonTest 5 100",
                "Nincs programparaméter!",
                JOptionPane.WARNING_MESSAGE);
            System.exit(0);
        }
    }
} // PolygonTest

```

A program elemzése

- ◆ //1-ben létrehozunk egy üres sokszöget, s //2-ben sorban hozzáadjuk a csúcsokat. A csúcsok koordinátáit a radius sugáról és az alfa szögből számítjuk ki. Ehhez az alfa szöget radiánra kell konvertálnunk, mert a Math osztály trigonometrikus függvényei radiánban várják a szögértékeket.
- ◆ //3-ban az eltolási pontot a keret közepére tesszük, ettől kezdve ez lesz tehát a (0,0) pont.
- ◆ //4-ben egy szürke kitöltött sokszöget teszünk a keretre, középpontja az eltolási pont lesz. //5-ben feketével kihúzzuk a sokszög széleit.
- ◆ //6-ban berajzoljuk a sokszög közepét – egy 10 pont átmérőjű fekete, kitöltött kört.

9.5. Képek – az absztrakt Image osztály

A Java alkalmazásokkal be lehet töltetni egy állományban tárolt képet és megjeleníttetni egy grafikus objektumon. Az állomány lehet a mi gépünkön, s lehet az Internet valamely URL címmel megadott távoli gépen is. A Java csak a GIF, JPG és PNG képállományokat használhatja erre a célra.

Ide vonatkozó fontosabb deklarációk:

- Image osztály: grafikus képeknek megfelelő absztrakt osztály – implementációja platformfüggetlen;
- Toolkit.getDefaultToolkit().getImage(...): képállományt betöltő metódus;
- Graphics.drawImage(...): képmegjelenítő metódus.

A getImage metódus elkezdi betölteni a képet, és azonnal, még a szükséges műveletek befejezése előtt visszatér. A betöltés egy külön programszálon folytatódik, s az a betöltés befejezésével értesíti (notify) a képhez kapcsolt ImageObserver-t, az pedig újrarendezheti a képet. Az ImageObserver egy interfész; az imageUpdate az egyetlen metódusa. A Component osztály implementálja az ImageObserver interfészt, vagyis komponens is lehet a betöltés figyelője.

Megjegyzés: Az Image osztály nem tévesztendő össze az ImageIcon osztályal! Az ImageIcon-ról a címek és a nyomógombok díszítésének ismertetésekor volt szó. Az ImageIcon is az Object közvetlen leszármazottja, és implementálja az Icon interfészt. Az Icon rögzített méretű kép, egyenesen a komponensek dekorációjához való.

Image osztály

Csomag: java.awt Deklaráció: public abstract class Image
Közvetlen ős: java.lang.Object
Közvetlenül implementált interfések: -

Mezők

- ▶ static int SCALE_AREA_AVERAGING
- ▶ static int SCALE_DEFAULT
- ▶ static int SCALE_FAST
- ▶ static int SCALE_REPLICATE
- ▶ static int SCALE_SMOOTH

Különböző átméretező algoritmusok azonosítói. A képek átméretezésekor (kicsinyítéskor, illetve nagyításkor) ezek az algoritmusok határozzák meg a konkrét képpontokat.

Metódusok

- ▶ int getWidth(ImageObserver observer)
- ▶ int getHeight(ImageObserver observer)

Visszaadja a kép szélességét, illetve magasságát. Ha a képnél még nincs mérete (mert, mondjuk, a kép betöltése még nem fejeződött be), akkor -1 a visszaadott érték. A méretről az observer-t folyamatosan értesíti.

- ▶ Graphics getGraphics()

Visszaadja a képhez tartozó Graphics objektumot, s arra további rajzok tehetők. Csak a képernyőn nem látható (off-screen) képekre hívható meg.

- ▶ Image getScaledInstance(int width, int height, int hints)

Visszaad egy képet, ennek a képnél egy átméretezett (kicsinyített vagy nagyított) másolatát. Az átméretezett kép width*height méretű. hints az egyik átméretező algoritmus (pl. SCALE_DEFAULT). Ha width vagy height közül valamelyik negatív, akkor a rendszer olyan értékkel helyettesíti, hogy a visszaadott kép mérete arányos legyen a kép méretével.

A kép betöltése – Toolkit.createImage

- ▶ Image createImage(String fileName)
- ▶ Image createImage(URL url)

Létrehoz egy Image objektumot, és a megadott állományból beletölt egy pontképet (raszteres képet). A kép formátuma GIF, JPG vagy PNG lehet. A metódus azonnal visszatér. Ha nincs ilyen képállomány, akkor a kép width és height adata -1 lesz.

A kép kirajzolása – Graphics.drawImage

- ▶ boolean drawImage(Image img, int x, int y, ImageObserver observer)
- ▶ boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)

Kirajzol a képből annyit, amennyi már megvan. Paraméterek:

- img: A megjelenítendő kép.
- (x, y): A kép bal felső sarka a grafikus objektumon.

- width, height: A kép új mérete. Alapértelmezés: a kép mérete.
- observer: A kép figyelője. Értesítést kap, ha a kép betöltődött – ennek hatására meghívódik az observer.imageUpdate metódus.

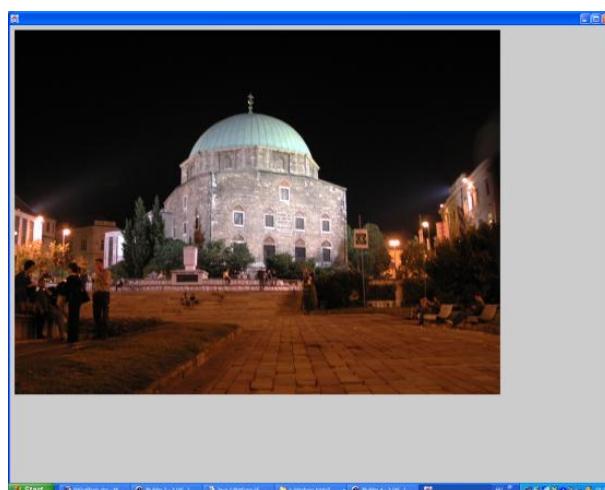
Médiakövető – MediaTracker

A `java.awt.MediaTracker` az `Object` közvetlen utódja. Segítségével egyszerre több kép betöltését követhetjük nyomon. A médiakövető használatát egy példán mutatjuk be: Létrehozunk egy `MediaTracker` példányt (//1), majd hozzáadunk egy azonosítószámmal megjelölt képet (//2). A példában megvárjuk, amíg a 0 azonosítójú kép betöltődik a tárba –addig nem engedjük tovább a programot (//3). Végül a képet kivesszük médiakövető felügylete alól (//4):

```
MediaTracker tr = new MediaTracker(this); //1
tr.addImage(image, 0); //2
try {
    tr.waitForID(0); //3
} catch(InterruptedException e) {
}
finally {
    tr.removeImage(img, 0); //4
}
```

Feladat – Image-teszt (ImageTest)

Jelenítsük meg teljesképernyős keretben az "images" mappában található "BelvTempEste.jpg" képállományt! A kép az állomány méretének megfelelően jelenjen meg, ne törődjünk azzal, ha nem fér bele a keretbe vagy kisebb annál! A keretben a kép bal oldalán és felette hagyjunk ki 10 pontnyi helyet!



Forráskód

```
import java.awt.*;
import javax.swing.*;
```

```

class Kep extends JPanel {
    private Image img;                                         //1

    public Kep() {
        MediaTracker tr = new MediaTracker(this);
        img = Toolkit.getDefaultToolkit().getImage(
            "images/BelvTempEste.jpg");                         //2
        tr.addImage(img,0);
        try {
            tr.waitForID(0);
        }
        catch(InterruptedException e) {
        }
        finally {
            tr.removeImage(img,0);
        }
    }

    protected void paintComponent(Graphics gr) {
        super.paintComponent(gr);
        gr.drawImage(img,10,10,this);                           //3
    }
}

public class ImageTest extends JFrame {
    public ImageTest() {
        setSize(Toolkit.getDefaultToolkit().getScreenSize());
        getContentPane().add(new Kep());                      //4
        show();
    }

    public static void main (String args[]) {
        new ImageTest();
    }
}

```

A program elemzése

- ◆ //1: A Kep osztálynak a kép betöltése és megjelenítése a dolga. A képet az img:Image objektum tárolja.
- ◆ //2: A képet a Kep osztály konstruktorában töltjük be, a Toolkit osztály getImage metódusával. A MediaTracker addig nem engedi tovább a programot, amíg be nem töltődik a teljes kép.
- ◆ //3: A Kep objektum által tartalmazott képet a paintComponent() minden kirajzolja (megjeleníti), ha arra szükség van. A kép méretét az img határozza meg, a komponens mérete nem befolyásolja. A képet a komponens (10,10) pontjától tesszük ki.
- ◆ //4: Beszűrjuk a keretbe a képet betöltő és megjelenítő objektumot, hogy az a felhasználi interfész eleme lehessen, és így valóban meg is jelenhessen.

Feladat – Kép átméretezése

Az "images" mappából tegyük bele egy képet a teljesképernyős keretbe, hogy megnézhessük. A kép a kitölthető helyhez képest a lehető legnagyobb legyen – ez a követelmény akkor teljesüljön, ha a keretet átméretezik! Írjuk rá a képre a megjelentett képállomány nevét!



A képet most nem eredeti nagyságában fogjuk megjeleníteni – az ablak és a kép méretétől és fekvésétől (álló vagy fekvő) függően át kell méretezni (skálázni)! Az `getScaledInstance` metódus visszaad egy méretre igazított új képet – ezt tesszük majd a keretbe.

Forráskód

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class Kep extends JPanel { //1
    private Image img; // eredeti kép
    private Image fittedImg = null; // kicsinyített/nagyított kép
    private double ratio; // arány: szélesség/magasság
    private String fileName; // a képállomány neve
    private MediaTracker tr;

    public Kep(String fileName) {
        // A kép betöltése file-ból a médiakövető segítségével:
        this.fileName = fileName;
        img = Toolkit.getDefaultToolkit().createImage(fileName); //2
        tr = new MediaTracker(this);
        tr.addImage(img, 0);
        try {
            tr.waitForID(0);
        }
        catch (InterruptedException ex) {
        }
        finally {
            tr.removeImage(img, 0);
        }
    }
}

```

```

// Arány (ratio) számítása; csak akkor jó, ha betöltődött a kép:
ratio = 1.0*img.getWidth(this)/img.getHeight(this);
System.out.println("ratio= " + ratio);

addComponentListener(new ComponentAdapter() { //3
    // Átméretezték a panelt:
    public void componentResized(ComponentEvent e) {
        // A kép átméretezése a mediakövető segítségével.
        // Igazítás a komponens szélességéhez vagy magasságához:
        if (getHeight()*ratio>getWidth()) //4
            // A kép szélessége a komponens szélessége lesz:
            fittedImg = img.getScaledInstance(getWidth(), -1,
                Image.SCALE_DEFAULT);
        else
            // A kép magassága a komponens magassága lesz:
            fittedImg = img.getScaledInstance(-1,
                getHeight(), Image.SCALE_DEFAULT);
        tr = new MediaTracker(e.getComponent()); //5
        tr.addImage(fittedImg, 0);
        try {
            tr.waitForID(0);
        }
        catch (InterruptedException ex) {
        }
        finally {
            tr.removeImage(img, 0);
        }
    }
});
}

protected void paintComponent(Graphics gr) { //6
    super.paintComponent(gr);
    // A kép kirajzolása:
    gr.drawImage(fittedImg, 0, 0, this);

    // A képfájl nevének megjelenítése egy szürke téglalapban:
    gr.setColor(Color.LIGHT_GRAY);
    gr.fillRoundRect(0, getHeight()-30, 300, 30, 5, 5);
    gr.setColor(Color.BLACK);
    gr.drawRoundRect(0, getHeight()-30, 300, 30, 5, 5);
    gr.setFont(new Font("TimesRoman", Font.PLAIN, 20));
    gr.drawString(fileName, 10, getHeight()-10);
}
}

public class KepAtmeretezes extends JFrame {
    public KepAtmeretezes() { //7
        Dimension dim = Toolkit.getDefaultToolkit().getScreenSize();
        setBounds(0, 0, dim.width, dim.height-50);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        getContentPane().add(new Kep("images/SanFran.jpg"));
        //getContentPane().add(new Kep("images/auto.jpg"));
        //getContentPane().add(new Kep("images/PosterChild.jpg"));
        show();
    }
}

```

```

public static void main (String args[]) {
    new KepAtmeretezes();
}
}

```

A program elemzése

A program megjegyzései magukért beszélnek, csupán néhány dologra térünk ki:

- ◆ A keret konstruktőrben (//7) csak beállítjuk a keret pontos méretét és létrehozunk egy `Kep` objektumot.
- ◆ A `Kep` osztály (//1) dolga a kép betöltése és szükség esetén a kép átméretezése. Az osztály megjegyzi az eredeti képet (`img`). Az átméretezés minden ebből a képből indul ki, hiszen az átméretezés információvesztéssel jár. A keretbe éppen beillő (`fitted`) képet a `fittedImg` objektum tárolja. A kép a konstruktőrban töltődik be (//2). A programot a kép teljes betöltéséig feltartóztatjuk, mivel az arányszámítás a kép szélességének és hosszúságának ismerete hiánynak helytelen lenne.
- ◆ A komponens átméretezésekor `ComponentEvent` keletkezik; kezelőmetódusában (//3) a képet újra kell méretezni (//4). mindenéppen arányosan nagyítjuk vagy kicsinyítjük a képet; ha fekvő képről van szó, akkor a kép szélességét vesszük alapul, ha álló képről, akkor a magasságát. A kép átméretezését is a médiakötőre bízzuk (//5).
- ◆ A képet //6-ban jelenítjük meg.

Tesztkérdések

9.1. Mely állítások igazak? Jelölje be az összes helyes állítást!

- a) A felhasználói interfész minden komponensének van grafikus objektuma, s annak osztálya a `Graphics` utódja.
- b) A grafikus objektumot a programozó hozza létre.
- c) A grafikus objektum (0,0) pontja alapértelmezésben a komponens közepe.
- d) A grafikus műveletek koordinátái negatív értékűek is lehetnek.

9.2. Adva van az alábbi programrészlet. Melyik utasításról mondható el, hogy szintaktikusan helyes és ha //1-be behelyettesítjük, akkor egy vízszintes egyeneset jelenít meg a `MyComponent` komponensen? Jelölje be az összes jó megoldást!

```

class MyComponent extends JComponent {
    protected void paintComponent(Graphics gr) {
        //1: Vízszintes egyenes rajzolása
    }
}

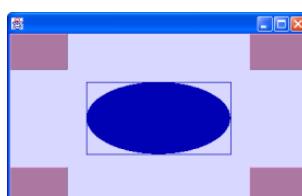

- a) drawLine(5,10,5,100);
- b) drawLine(5,10,50,10);
- c) gr.drawLine(5,10,5,100);
- d) gr.drawLine(5,10,50,10);

```

- 9.3. Jelölje be azokat a pontokat, amelyekben a felsorolt metódusok mind a `JComponent` osztályában vannak deklarálva!
- `getGraphics()`, `setBackground()`, `setColor()`, `getWidth()`
 - `getGraphics()`, `setBackground()`, `getWidth()`, `paintComponent()`
 - `getWidth()`, `repaint()`, `setForeground()`, `drawImage()`
 - `fillPolygon()`, `drawLine()`, `setFont()`
- 9.4. Mi jellemző a `Graphics` objektumra? Jelölje be az összes helyes állítást!
- Háttérszín
 - Rajzolószín
 - Kivágási terület
 - Betű
- 9.5. Mely állítások igazak? Jelölje be az összes helyes állítást!
- A `Polygon` osztály a `JComponent` osztály leszármazottja.
 - A sokszöget kirajzoló metódusokat a `Graphics` osztály deklarálja.
 - A `Graphics` osztály a `JComponent` osztályból származik.
 - Egy Java programban csak `Container`-re lehet rajzolni.
- 9.6. Mely állítások igazak? Jelölje be az összes helyes állítást!
- Egy képállomány betölthető az `Image` osztály egy metódusával.
 - Egy `Image` objektumot megjeleníthetünk a `Graphics` osztály `drawImage` metódusával.
 - Egy Java alkalmazás felismeri a BMP (bitmap) képállományt.
 - A `Toolkit` osztálynak van egy képbetöltő metódusa, s annak a vezérlése még a kép betöltése előtt visszatér.

Feladatok

- 9.1. Tegyen a keretbe egy – a teljes keretet betöltő – pluszjelet!
- (A) A pluszjel fekete színű legyen és 1 pontos vastagságú! (*PluszJelA.java*)
 - (A) A pluszjel zöld színű legyen és 5 pontos vastagságú! (*PluszJelB.java*)
 - (B) A pluszjel színe és vastagsága legyen paraméterezhető! Az adatokat a program elején kérjük be! (*PluszJelC.java*)
- 9.2. (A) Tegyen a világoskék ablak négy sarkába egy-egy 80*50-es háromdimenziós, mélylila kitöltött téglalapot! Rajzoljon az ablak közepébe egy sötétkék 200*100-as kitöltetlen téglalapot és egy azt belülről érintő sötétkék színű kitöltött ellipszist! (*Rajz.java*)



9.3. **(B)** Készítsen egy színes kisautót! A kisautó színét létrehozáskor adjuk meg.

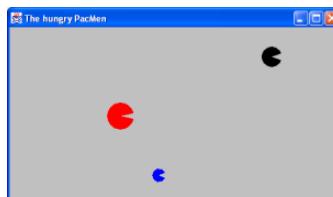
- a) Tegyen egy piros kisautót a keret (100,100) pontjára!
 - b) Tegyen egy kék kisautót a keret közepére!
 - c) Tegyen sok kisautót a keretbe!
- (Autok.jpx)

9.4. **(B)** Rajzoljon az ablakba különféle csigákat! (Csigak.jpx)



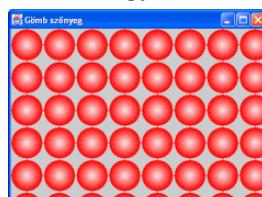
9.5. **(A)** Forgasson egy egyenest a középpontja körül mindaddig, amíg be nem csukják az ablakot! A háttér szürke legyen, az egyenes pedig lila! Időzítővel oldja meg a feladatot! (ForgoEgyenes.jpx)

9.6. **(B)** Készítsen egy Pacman figurát; az tátogjon és mozogjon egyenletesen az ablakban balról jobbra! Tegyen egy ablakba három különböző nagyságú, színű és gyorsaságú Pacman figurát! Ha egy figura elér a pálya végére, akkor álljon meg! (PacMen.jpx)

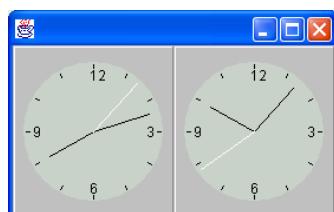


9.7. **(B)** Készítsen egy szőnyeget: díszítésül tegye tele domború (térfelületű) gömböcskékkel! (GombSzonyeg.java)

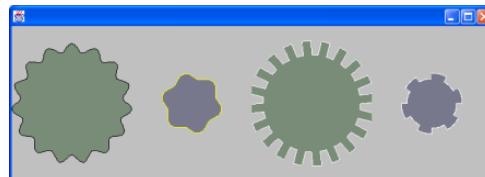
Tipp: A térfelületet úgy lehet elérni, hogy kívülről befelé világosodó körökkel rajzolunk.



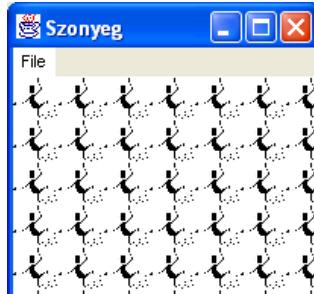
9.8. **(C)** Készítsen olyan órát, melynek három mutatója van, és jár is. Tegyen két ilyen órát egy keretbe (Orak.java):



- 9.9. (C) Tegyen az ablakra hullámos (szinuszos) szélű köröket, valamint fogaskerekeket! Létrehozáskor adjuk meg a kerek sugarát, a kiemelkedő periódusok számát és magasságát, az idomot kitöltő színt és az idom szélénak színét. (*HullamFogas.java*)



- 9.10. (C) Készítsen egy másodfokú függvényt előállító programot! (*Fuggveny.java*)
- 9.11. (B) Készítsen olyan szönyeget, melyen egy kép ismétlődik. Kérje be egy dialógusból a képállomány nevét, majd inicializálja a szönyeget a bekért névvel! A kép szélessége mindenkor 30 pont legyen, magassága pedig arányos az eredeti képpel!
(*KepSzonyeg.java*)



- 9.12. (C) Készítsen egy képeket mutogató demóprogramot! A program a képeket egymás után mutassa be 5-5 másodpercig, felhasználói beavatkozás nélkül!
- A képet nem kell átméretezni. Tegye rá a képet a teljes képernyőre, és azzal ne törődjön, hogy a képből mennyi látszik!
 - A képet méretezze úgy, hogy minden látható legyen a teljes kép!

Lehetőség szerint a képek között legyen olyan kép is, amelyet saját képernyőjéről „lopott el”, és olyan is, amelyet képletapogatóval készített egy papírképről!

Tipp: Képernyőképről a következőképpen készíthet képállományt: a Ctrl+Alt+PrintScreen billentyűkombinációt használva tegye a képet a vágólapra. Ezután vágja bele a képet egy olyan szoftverbe (például Adobe PhotoShop), amellyel JPEG-formátumban is elmentheti!

(*Demo.java*)

10. Alacsony szintű események

A fejezet pontjai:

1. Az alacsony szintű események osztályhierarchiája
 2. Komponensesemény – ComponentEvent
 3. Fókuszesemény – FocusEvent
 4. Billentyűesemény – KeyEvent
 5. Egéresemény – MouseEvent
-

Alacsony szintű esemény csak komponensen keletkezhet, és általában a felhasználó kelti. A programozó az alacsony szintű események közül a billentyű- és egéreseményeket dolgozza fel a leggyakrabban. Az eseményvezérelt programozás alapelveit, közelebbről az alacsony és magas szintű események jellemzőit már tárgyaltuk a 7. fejezetben. A konkrét magas szintű eseményekkel a 8. fejezet foglalkozott; az alacsony szintű eseményeket ebben a fejezetben vesszük sorra.

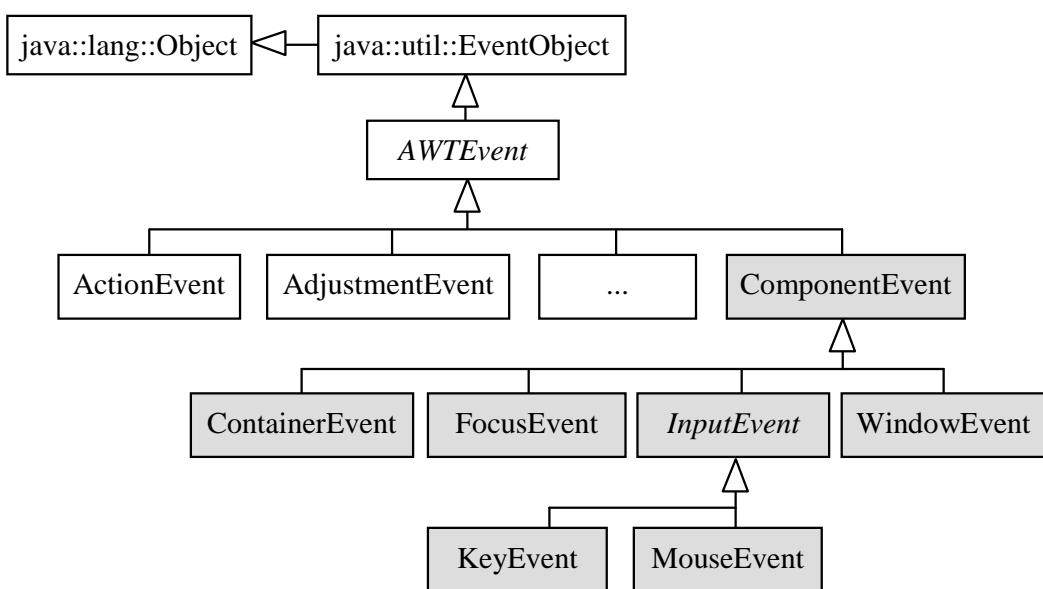
10.1. Az alacsony szintű események osztályhierarchiája

A 10.1. ábra (a 7.2. ábrához hasonlóan) a `java.awt.event` csomag eseményeinek osztályhierarchiáját ábrázolja – az alacsony szintű eseményosztályok szürkén jelennek meg rajta.

Minden alacsony szintű esemény a `ComponentEvent` osztály leszármazottja; a magas szintű eseményekre ez már nem igaz. **Alacsony szintű esemény csak komponensen keletkezhet**, általában a felhasználó beavatkozása révén. Az alacsony szintű események a következő osztályok objektumai lehetnek:

- `ComponentEvent` (komponensesemény): akkor keletkezik, ha a komponens átméreteződik, elmozdul, láthatóvá vagy láthatatlanná válik;
- `ContainerEvent` (konténeresemény): akkor keletkezik, ha a komponens konténerkomponens és éppen megváltozik az összetétele (komponenst tesznek bele vagy vesznek ki belőle);
- `FocusEvent` (fókuszesemény): akkor keletkezik, ha a komponens fókuszbba kerül vagy éppen elveszíti a fókuszt;

- KeyEvent (billentyűesemény): akkor keletkezik, ha a komponens fókuszban van és éppen leütnek egy billentyűt;
- MouseEvent (egéresemény) akkor keletkezik, ha az egérkursor a komponens felett van és éppen megnyomják vagy felengedik az egér gombját; ha az egér éppen beérkezik a komponens területére vagy elhagyja azt; ha kattintanak a komponensen; ha vonszolják vagy egyszerűen csak elmozdítják rajta az egeret;
- WindowEvent (ablakesemény) akkor keletkezik, ha a komponens ablakkomponens és éppen megnyílik vagy be akarják zárni; ha már be is záródott; ha ikonná változtatják vagy újra teljes méretűvé; ha aktiválódik vagy éppen elveszti aktív mivoltát.



10.1. ábra. A `java.awt.event` csomag eseményei

A következő pontokban egyenként megtárgyaljuk az alacsony szintű eseményeket, kivéve az ablakeseményt – az mintaként már szerepelt a 7. fejezetben – és a konténereseményt, mert az ritkábban használatos. Ezek az események ugyanúgy keletkeznek, mint a többiek, és megfigyelni is ugyanúgy kell őket.

Emlékeztetőül megjegyezzük, hogy minden `AWTEvent`-objektumba bele van foglalva a forrás-objektum (`source`) és az esemény azonosítója (`id`).

10.2. Komponensesemény – ComponentEvent

ComponentEvent elvileg bármely komponenstől származhat. Akkor keletkezik, amikor a komponens valamiért elmozdul a szülőhöz képest vagy átméreteződik, előtérbe vagy háttérbe kerül. Az esemény csak az akció (pl. átméretezés) befejeztével áll össze; vagyis az eseménykeletkezés nem folyamatos. Az eseményt megkapja minden olyan ComponentListener és ComponentAdapter, amely az addComponentListener metódussal feljelentkezett a komponens figyelőláncára (a ComponentAdapter osztály üres metódusokkal implementálja a ComponentListener interfészt).

A komponensesemény keletkezése és kezelése:

Eseményosztály	Forrás	Figyelőinterfész	Felfűzőmetódus	Eseménykezelők
ComponentEvent	Component	ComponentListener	addComponentListener	componentMoved componentResized componentShown componentHidden

Most végignézzük a komponenseseménnyel kapcsolatos osztályok, interfészök fontosabb deklarációt:

ComponentEvent

Mezők

- ▶ static final int COMPONENT_MOVED
- ▶ static final int COMPONENT_RESIZED
- ▶ static final int COMPONENT_SHOWN
- ▶ static final int COMPONENT_HIDDEN

Konstansok: a komponensesemény lehetséges azonosítói (`id`).

Konstruktur, metódusok

- ▶ ComponentEvent(Component source, int id)

Paraméterek:

- `source`: az az objektum, amelyen az esemény keletkezett.
- `id`: az esemény azonosítója.

- ▶ Component getComponent()

Mint az AWTEvent.getSource() metódus, csak a visszatérési érték Component.

ComponentListener, ComponentAdapter

- ▶ void componentMoved(ComponentEvent e)
- ▶ void componentResized(ComponentEvent e)

- ▶ void componentShown(ComponentEvent e)
 - ▶ void componentHidden(ComponentEvent e)

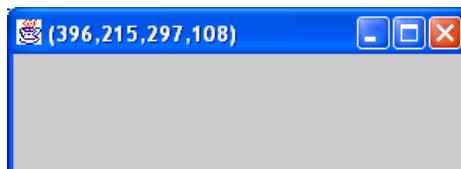
A komponensesemény kezelőmetódusai: `moved`=elmozdult; `resized`=átméreteződött; `shown`=előtérbe került; `hidden`=eltűnt.

Component

- void addComponentListener(ComponentListener l)
 - void removeComponentListener(ComponentListener l)

A komponens hozzáadása a figyelőlánchoz, illetve levétele róla.

Feladat – Komponens teszt (ComponentTest)



Forráskód

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ComponentTest extends JFrame {
    public ComponentTest() {
        setBounds(100,100,200,80);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        addComponentListener(new ComponentManager());           //1
        show();
        setTitle();
    }

    void setTitle() {                                         //2
        setTitle("(" + getX() + "," + getY() + "," +
                  getWidth() + "," + getHeight() + ")");
    }
}

class ComponentManager extends ComponentAdapter {          //3
    public void componentMoved(ComponentEvent e) {
        setTitle();
        if (getX() < 0) System.exit(0);
    }
    public void componentResized(ComponentEvent e) {
        setTitle();
    }
}
```

```
public static void main (String args[]) {  
    new ComponentTest();  
} // main  
}
```

A forráskód elemzése

//1-ben egy a ComponentAdapter-t kiterjesztő belső osztályt adunk meg a keret figyelőjéül.
//2-ben beállítjuk a címet: az minden a keret aktuális (x,y,szél,mag) adata lesz. A belső osztályban (//3) elegendő a componentMoved és a componentResized kezelőmetódusokat kifejteni – csak ezekre az eseményekre szeretnénk reagálni:

- ◆ componentMoved: Akkor kerül ide a vezérlés, ha a keretet elmozgatják – ekkor beállítjuk a címet. Ha a keret bal széle a képernyőn kívülre esik, leállítjuk a programot.
- ◆ componentResized: Akkor kerül ide a vezérlés, ha a keretet átméretezik. Ilyenkor csak a címet állítjuk be.

10.3. Fókuszesemény – FocusEvent

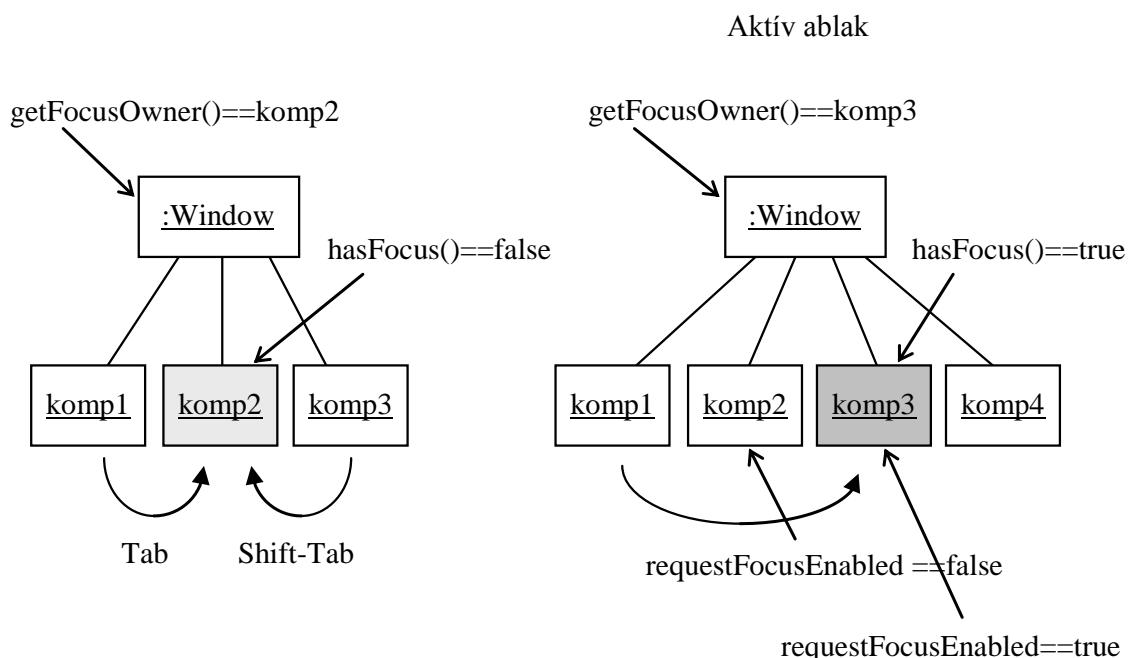
A fókusz- és billentyűesemények megértéséhez tudnunk kell, mit jelent az, hogy egy ablak aktív, hogy egy komponens fókuszból van, s azt is, hogyan lehet egy komponenst fókuszból hozni és a fókuszlánc elemeit bejárni.

Aktív ablak: Egy alkalmazás felhasználói interfészéhez tartozó ablakokból (főként keretekből és dialógusokból) áll, s azok közül legfeljebb egy lehet aktív. Ha az alkalmazás maga nem aktív, akkor egyetlen ablaka sem lehet aktív. Az aktív ablak látványban is különbözik a többiből – például más színű a kerete.

Fókuszlánc, fókusz (billentyűfókusz): minden ablaknak (Window-leszármazottnak) külön fókuszlánca van, s az az ablakból és a közvetlenül vagy közvetve bele tett komponensekből áll. A fókuszláncnak legfeljebb egy **fókuszbirtokosa** (focus owner) lehet. A fókuszbirtokos akkor kapja meg ténylegesen a **fókuszt** (has focus: nála van a fókusz), ha az ablak aktívvá válik. Nem aktív ablaknak egyetlen eleme sem lehet fókuszból. A fokusztárolás az ablak fókuszmenedzserének a feladata. A program bármely ablaktól megkérdezheti, hogy éppen melyik elemén van a fókusz (getFocusOwner), s egy komponenstől is megkérdezheti, hogy éppen fókuszból van-e (hasFocus).

Az aktuális billentyűeseményeket minden az éppen fókuszból levő komponens kapja – ezért szokás a fókuszt billentyűfókusznak is nevezni. Az egész alkalmazásban, sőt az operációs rendszer összes alkalmazásában is **csak egy komponens lehet fókuszból**. A billentyűleütéseket minden az éppen fókuszból levő komponens fogadja, ő lesz a keletkezett esemény forrása.

A fókuszban levő komponens rendszerint látványban is különbözik a többitől: ha ez a komponens szövegmező vagy szövegterület, akkor rajta villog a kurzor; ha nyomógomb, akkor jobban kiemelkedik, mint a nem fókuszban levő; a fókuszban levő komponensek némelyike sötétebb, élesebb színű stb. Az ablakok fókuszláncát a 10.2. ábra mutatja.



10.2. ábra. Ablakok fókuszláncai

Fokuszálás, bejárhatóság, bejárási sorrend

A fókusz áthelyezhető egyik komponensről a másikra – ezt a műveletet **fokuszálásnak** nevezzük. Csak az a komponens kerülhet fókuszba, amelyik látható (`visible`), engedélyezve van (`enabled`) és engedélyezve van a fókuszba hozása is (`requestFocusEnabled`).

A program és a felhasználó is fokuszálhat. A felhasználó azonban csak az ún. **fokuszálható** komponenseket állíthatja fókuszba. A fokuszálhatóság a `focusable` jellemzővel szabályozható. A komponensek többsége alapértelmezésben fokuszálható (a címke és a panel azonban nem). A `focusable` tulajdonság csak akkor lehet `true`, ha a `requestFocusEnabled` tulajdonság `true`.

A felhasználó kétféleképpen változtathatja meg a fókuszt:

- közvetlenül, egérkattintással;
- a Tab és a Shift-Tab billentyűvel oda-vissza, illetve körbe bejárhatja a fókuszlánc komponenseit.

Programból a `transferFocus()` metódussal járhatók be a komponensek, de csak egy irányban; átlépve a be nem járható komponenseket. A `requestFocus()` metódussal a nem fokuszálható komponenseket is fókuszba lehet hozni.

Az ablak fókuszláncának **bejárási sorrendjét** elsősorban a komponenshierarchy határozza meg. Egy konténerben a bejárási sorrendet

- Swingben a komponensek bal felső sarkának (x,y) koordinátái határozzák meg: a bejárás balról jobbra halad, majd felülről lefelé;
- AWT-ben a konténerbeli sorrend adja meg.

Az ablak létrehozásakor **az első olyan gyerek kerül fókuszba**, amelyik látható, engedélyezve van és fokuszálható. Ha a gyerekek között nincs fokuszálható komponens, akkor az első látható, engedélyezett komponens kerül a fókuszba. Ha egyáltalán nincs látható és engedélyezett gyerek, akkor az ablak maga lesz a fókusztulajdonos. Például ha az ablak egy címkét és egy szövegmezőt tartalmaz, akkor kezdetben a szövegmező lesz fókuszban; de ha csak címkét tartalmaz, akkor maga az ablak.

A fókuszesemény tartós vagy ideiglenes lehet. A **fókuszesemény** akkor **tartós**, ha attól véglegesen megváltozik a fókuszlánc fókuszbirtokosa. A **fókuszesemény ideiglenes** (temporary), ha a fókuszlánc fókuszbirtokosa nem változik meg – például ha ablakot váltunk (akkor a fóokuszt elhagyása ideiglenes, mert ha visszaváltunk az ablakra, akkor újra a fókusztulajdonos kerül fókuszba), vagy ha görgetünk (a görgetés után az eredeti komponens kapja vissza a fókuszt).

Fókuszesemény

Fókuszeseményt bármely komponens kelthet, ha megkapja vagy elveszíti a billentyű-fókuszt. Az eseményt megkapja minden olyan `FocusListener`, illetve `FocusAdapter`, amely az `addFocusListener` metódussal feljelentkezett a komponens figyelőláncára (a `FocusAdapter` osztály üres metódusokkal implementálja a `FocusListener` interfészét).

A fókuszesemény keletkezése és kezelése:

Esemény	Mi történt?	Figyelőinterfész	Felfűzőmetódus	Eseménykezelők
FocusEvent	Az elem fókuszba került, vagy elvesztette a fókuszt.	FocusListener	addFocusListener	focusGained focusLost

Most végignézzük a fókuszeseménnyel kapcsolatos osztályok nevezetesebb deklarációit:

FocusEvent

Mezők

- ▶ static final int FOCUS_GAINED
- ▶ static final int FOCUS_LOST

Konstansok: a fókuszesemény lehetséges azonosítói (`id`).

Konstruktörök, metódusok

- ▶ FocusEvent(Component source, int id)
- ▶ FocusEvent(Component source, int id, boolean temporary)

Paraméterek:

- `source`: az a forráskomponens, amelyen az esemény keletkezett.
 - `id`: az esemény típusának azonosítója.
 - `temporary`: a fókuszesemény ideiglenessége/tartóssága. Alapértelmezés: `false` (tartós).
 - ▶ `boolean isTemporary()`
- Visszaadja, hogy a fókusz megszerzése, illetve elvesztése ideiglenes-e (`true`) vagy végleges (`false`).

FocusListener, FocusAdapter

- ▶ public void focusGained(FocusEvent e)
- ▶ public void focusLost(FocusEvent e)

A fókuszesemény kezelőmetódusai: `gained=megszerezte; lost=elveszítette`.

Component/JComponent

- ▶ void addFocusListener(FocusListener l)
- A komponens hozzáadása a fókuszfigyelő lánchoz, illetve levétele róla.
- ▶ void setRequestFocusEnabled(boolean flag)
 - ▶ boolean isRequestFocusEnabled()
- A `requestFocusEnabled` jellemző értéke `true`, ha a komponens fókuszba hozható a `requestFocus` metódus meghívásával.
- ▶ setNextFocusableComponent(Component aComponent)
- Megváltoztatja a bejárási sorrendet: kijelöli a komponens után veendő komponenst. Alapértelmezésben a bejárás az (x,y) koordináták szerint halad.
- ▶ void requestFocus()
- A komponenst fókuszba hozza, feltéve, hogy az `isRequestFocusEnabled()` visszatérési értéke `true` és a komponens még nincs fókuszban.
- ▶ boolean hasFocus()
- Visszaadja, hogy a komponens fókuszban van-e.

► `void transferFocus()`

Átteszi a fókuszt a konténer következő fokuszálható komponensére.

Window

► `Component getFocusOwner()`

Ha az ablak aktív, akkor visszaadja a fókuszban levő gyerekkomponenst. Ha nincs ilyen komponens, akkor a visszaadott érték null.

Feladat – Fókuszteszít

Tegyük a keretre három szövegmezőt! Az éppen fókuszban levő mező háttérszíne legyen halványsárga, a többié halványszürke!

Hozzunk létre két ilyen keretet, és tegyük felváltva aktívvá az ablakokat!



Forráskód

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import extra.frame.CloseableFrame;

class FocusTestPanel extends JPanel
    implements FocusListener { //1
private JTextField tf1, tf2, tf3;

public FocusTestPanel() {
    add(tf1 = new JTextField(10));
    add(tf2 = new JTextField(10));
    add(tf3 = new JTextField(10));

    tf1.setBackground(SystemColor.LIGHT_GRAY);
    tf2.setBackground(SystemColor.LIGHT_GRAY);
    tf3.setBackground(SystemColor.LIGHT_GRAY);

    tf1.addFocusListener(this); //2
    tf2.addFocusListener(this);
    tf3.addFocusListener(this);
}

public void focusGained(FocusEvent e) { //3
    e.getComponent().setBackground(SystemColor.info);
}
public void focusLost(FocusEvent e) { //4
    e.getComponent().setBackground(SystemColor.LIGHT_GRAY);
}
}

```

```

public class FocusTest extends CloseableFrame { //5
    public FocusTest(int x, int y) {
        getContentPane().add(new FocusTestPanel());
        setLocation(x,y);
        pack();
        show();
    }
    public static void main (String args[]) { //6
        new FocusTest(100,100);
        new FocusTest(500,100);
    } // main
} // FocusTest

```

A program elemzése

A panel fogja hallgatni a fókuszeseményeket (//1). A konstruktorban három szürke szövegmezőt teszünk a keretbe; rajtuk keletkeznek majd a fókuszesemények. //2-től a szövegmezők-höz tapasztjuk a panelt mint hallgatót. Amikor egy szövegmező fókuszba kerül (//3), akkor halványsárgára változtatjuk a színét; amikor elveszíti a fókuszt (//4), akkor ismét szürkére. A keret CloseableFrame, ezért a programnak csak akkor lesz vége, ha az utolsó keretet is becsukják (//5). //6-ban két keretet hozunk létre; a konstruktor paramétere a bal felső sarok.

Feladat – Dialógusok mezőinek ellenőrzése (CheckDialogs)

Tegyük lehetővé, hogy a felhasználó egy fő keretből tetszőleges számban hozzon létre dialógusokat (egymással párhuzamosan)! Kezdetben a dialógus Szül. év mezőjén villogjon a kurzor! Ha a felhasználó elhagyja ezt a mezőt, akkor a program ellenőrizze az évszám értékét, és ha az nem 1900 és 2000 közé esik, akkor küldjön figyelmeztést az alsó információs sorban; ha az évszám jó, akkor törölje az üzenetet! A Mégse gomb lenyomására csukódjék be az aktív dialógus, de az Ok gombbal csak akkor lehessen becsukni az ablakot, ha az évszám jó! Az egyik dialógusról bármikor át lehessen térti a másikra!



Forráskód

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class CheckDialog extends JDialog implements
    FocusListener, ActionListener { //1
    private JTextField tfSzulhely;
    private JTextField tfSzulev;
    private JButton btOk, btMegse;
    private JLabel lbInfo;

```

```
public CheckDialog(JFrame owner) {
    super(owner);
    Container cp = getContentPane();
    setTitle("Bevitel");
    setLocation(owner.getX() + 50, owner.getY() + 50);
    cp.setLayout(new GridLayout(0, 1));

    JPanel pn;
    cp.add(pn = new JPanel());
    pn.add(new JLabel("Születési hely: "));
    pn.add(tfSzulhely = new JTextField("Pécs", 10));

    cp.add(pn = new JPanel());
    pn.add(new JLabel("Szül. év: "));
    pn.add(tfSzulev = new JTextField("2000", 4));

    cp.add(pn = new JPanel());
    pn.add(btOk=new JButton("Ok"));
    pn.add(btMegse=new JButton("Mégse"));

    cp.add(lbInfo = new JLabel("", JLabel.CENTER));
    tfSzulev.addFocusListener(this); //2
    btOk.addActionListener(this);
    btMegse.addActionListener(this);
    pack();
    show();

    tfSzulev.requestFocus(); //3
}

boolean isNumber(String str, int also, int felso) { //4
    try {
        int szulev = Integer.parseInt(str);
        if (szulev < also || szulev > felso)
            throw new NumberFormatException();
        return true;
    }
    catch(NumberFormatException ex) {
        return false;
    }
}

// Ha fókuszba került:
public void focusGained(FocusEvent ev) { //5
}

// Ha elveszítette a fókuszt:
public void focusLost(FocusEvent ev) { //6
    if (ev.getComponent() == tfSzulev && !ev.isTemporary())
        if (!isNumber(tfSzulev.getText(), 1900, 2000))
            lbInfo.setText("Évszám 1900-2000!");
        else
            lbInfo.setText("");
}
```

```

public void actionPerformed(ActionEvent ev) { //7
    if (ev.getSource() == btOk) {
        if (lbInfo.getText().equals(""))
            dispose();
        else
            tfSzulev.requestFocus();
    }
    else if (ev.getSource() == btMegse)
        dispose();
}
}

public class CheckDialogs extends JFrame
    implements ActionListener {
private JButton btNew;
public CheckDialogs() {
    setLocation(300, 200);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    getContentPane().add(btNew = new JButton("Új dialógus"));
    btNew.addActionListener(this);
    pack();
    show();
}
public void actionPerformed(ActionEvent e) {
    new CheckDialog(this);
}
public static void main (String args[]) {
    new CheckDialogs();
} // main
}

```

A program elemzése

- ◆ //1: A dialógusablaknak most a fókusz- és az akcióeseményekre is figyelnie kell, minden két hallgatóinterfészt implementáljuk tehát.
- ◆ //2: Figyelnünk kell a `tfSzulev` mezőt, hogy mikor kerül ki a fókusból, és figyelnünk kell a két gombot is, hogy lenyomták-e őket.
- ◆ //3: Kezdetben az évszám kapja meg a fókuszt. A fokuszlásnak csak a `show()` után van hatása.
- ◆ //4: Az `isNumber` függvény választ ad arra, hogy a paraméterben megadott szöveg a szintén paraméterben megadott alsó és felső határ közé eső egész szám-e.
- ◆ //5: A fókusz megszerzésekor nincs tennivalónk.
- ◆ //6: Ha a `Szulev` mező vélegesen kikerül a fókusból (ablakváltással még nem kerül ki), akkor ellenőrizzük, hogy a beírt érték a megadott határok közé eső egész szám-e; ha nem az, akkor kiírjuk az üzenetet; ha az, akkor letöröljük az esetleges előző üzenetet.
- ◆ //7: A felhasználó az `Ok` gombot nyomta meg; az ablakot csak akkor engedjük becsukodni, ha üres az üzenetsor, vagyis jó az `Evszam` mező értéke. A `Mégse` gomb lenyomására mindenképpen hagyjuk becsukodni az ablakot.

10.4. Billentyűesemény – KeyEvent

Billentyűkód, billentyűkarakter



A számítógéphez kapcsolt billentyűzeten (keyboard) billentyűk (key) vannak. A billentyűk lehetnek

- **alfanumerikus billentyűk**, ezeknek jól meghatározott megjelenési formájuk van. Például: Q, A, Z, 1, 2, !, @
- **vezérlő- (akció-) billentyűk**, ezek valamilyen – közvetlen vagy közvetett – vezérlőfunkciót látnak el – kurzormozgatást, soremelést stb. Például: ↑, Enter, Delete, Home, Tab, Esc, F1
- **módosítóbillentyűk**, ezeknek önmagukban általában nincs hatásuk, csak módosítanak valamely más beviteli adatot. Például: Ctrl, Shift, Alt

Minden létező billentyűnek önálló **billentyűkódja** (`keyCode`) van – a megjeleníthető, a vezérlő- és a módosítóbillentyűnek is. minden billentyűhöz külön konstanst definiáltak a KeyEvent osztályban. Az összes billentyűkód-azonosító a `VK_` (Virtual Key) karakterhármasossal kezdődik. A fenti billentyűk például:

- `VK_Q`, `VK_A`, `VK_Z`, `VK_1`, `VK_2`, `VK_EXCLAMATION_MARK`, `VK_AT`
- `VK_UP`, `VK_ENTER`, `VK_DELETE`, `VK_HOME`, `VK_TAB`, `VK_ESCAPE`, `VK_F1`
- `VK_CONTROL`, `VK_SHIFT`, `VK_ALT`

A megjeleníthető billentyűknek **billentyűkarakter** (`keyChar`) értékük is van. A fenti billentyű első sorának értéke: '`'Q'`', '`'A'`', '`'Z'`', '`'1'`', '`'2'`', '`'!'`', '`'@'`'

A billentyűeseményeknek (KeyEvent) három fajtájuk van:

- `KEY_PRESSED`: akkor keletkezik, amikor egy **billentyű lenyomnak** (lenyomás szempontjából nincs különböző megjeleníthető, vezérlő- és módosítóbillentyű között);
- `KEY_RELEASED`: akkor keletkezik, amikor egy **billentyű felengednek** (felengedés szempontjából sincs különböző megjeleníthető, vezérlő- és módosítóbillentyű között);
- `KEY_TYPED`: egy alfanumerikus billentyűt lenyomnak (`KEY_PRESSED`), majd egy kis idő múlva felengednek (`KEY_RELEASED`).

A `KEY_PRESSED` és a `KEY_RELEASED` esemény egyetlen billentyűtől függ. A `KEY_TYPED` (begépelés) magasabb szintű esemény, az valamilyen billentyűkombináció lenyomásával (és bizonyos esetekben felengedésével, például `Alt+NumPad+szám`) jár együtt.

Megjegyzések:

- 101 gombos billentyűzeten az előre definiált billentyükód-konstansokból 101-et használhatunk a programokban.
- A billentyűzeten nincs külön a és A billentyű; de az A billentyűt lenyomhatják önállóan és a Shift módosító billentyűvel együtt is.

• Billentyük azonosítására kizárolag a KeyEvent osztály előre definiált VK_ konstansait használjuk! A Sun fenntartja a jogot a kódok megváltoztatására.

Most lássunk néhány példát billentyűesemények keletkezésére! A kiírt sorok mind egy-egy KeyEvent-objektum szöveges reprezentációi.

Egy alfanumerikus billentyű leütésekor három különböző esemény keletkezik. Például ha az A billentyűt ütjük le:

```
KEY_PRESSED, keyCode=65, keyChar='a'  
KEY_TYPED, keyCode=0, keyChar='a' // keyCode definiálatlan  
KEY_RELEASED, keyCode=65, keyChar='a'
```

Ha egy ismétlőbillentyűt hosszabb ideig tartunk lenyomva, akkor gyors egymásutánban KEY_PRESSED és KEY_TYPED események keletkeznek, majd befejezésük, a billentyű felengedésekor keletkezik még egy KEY_RELEASED esemény is. Ha például a B betűt tartjuk lenyomva, akkor az eseménysor a következő:

```
KEY_PRESSED, keyCode=66, keyChar='b'  
KEY_TYPED, keyCode=0, keyChar='b'  
KEY_PRESSED, keyCode=66, keyChar='b'  
KEY_TYPED, keyCode=0, keyChar='b'  
KEY_PRESSED, keyCode=66, keyChar='b'  
KEY_TYPED, keyCode=0, keyChar='b'  
KEY_RELEASED, keyCode=66, keyChar='b'
```

Vezérlőbillentyű nyomva tartásakor nem keletkezik KEY_TYPED esemény. Ha az F1 billentyűt egy ideig lenyomva tartjuk, akkor a következő események jönnek létre:

```
KEY_PRESSED, keyCode=112, F1  
KEY_PRESSED, keyCode=112, F1  
KEY_RELEASED, keyCode=112, F1
```

Billentyűesemény

Billentyűeseményt (KeyEvent) bármely komponens kelthet, ha rajta van a fókusz. Az eseményt megkapja minden olyan KeyListener, illetve KeyAdapter, amely addKeyListener metódussal csatlakoztatva van a komponens figyelőláncához (a KeyAdapter osztály üres metódusokkal implementálja a KeyListener interfést).

A billentyűesemény keletkezése és kezelése:

Esemény	Mi történt?	Figyelőinterfész	Felfűzőmetódus	Eseménykezelők
KeyEvent	Ha a billentyűt leütötték, felengedték vagy begépeltek.	KeyListener	addKeyListener	keyPressed keyReleased keyTyped

Most végigvesszük a billentyűeseménnyel kapcsolatos osztályok fontosabb deklarációit:

InputEvent

Absztrakt osztály, a billentyű- és egéresemények (a felhasználó által keltett beviteli események) közös őse. Mindkét fajta eseménynek van időpontja, és a módosítóbillentyűkkel vagy valamelyik egérgombbal mindenkorrel módosítani lehet.

Mezők

- ▶ static final int ALT_MASK
 - ▶ static final int ALT_GRAPH_MASK
 - ▶ static final int CTRL_MASK
 - ▶ static final int META_MASK
 - ▶ static final int SHIFT_MASK
- Maszkok a módosítóbillentyűk (modifiers) lekérdezéséhez.
- ▶ static final int BUTTON1_MASK
 - ▶ static final int BUTTON2_MASK
 - ▶ static final int BUTTON3_MASK
- Maszkok az egérgombokhoz. BUTTON1_MASK=bal, BUTTON2_MASK=jobb, valamint BUTTON3_MASK=középső egérgomb.

Metódusok

- ▶ long getWhen()

Visszaadja az esemény keletkezésének időpontját ezredmásodpercben.
- ▶ int getModifiers()

Egy módosítómaszkot ad vissza, vagyis megadja, hogy az esemény keletkezésekor a következők közül mi volt lenyomva: billentyűk: Alt, Alt+Graph, Control, Meta, Shift; egérgombok: bal, középső, jobb.
- ▶ boolean isAltDown()

▶ boolean isAltGraphDown()

▶ boolean isControlDown()

▶ boolean isMetaDown()

▶ boolean isShiftDown()

Az adják vissza, hogy az esemény keletkezésekor le volt-e nyomva az Alt, Alt+Graph, Control, Meta, illetve a Shift billentyű.

KeyEvent

Mezők

- ▶ static final int KEY_PRESSED
- ▶ static final int KEY_RELEASED
- ▶ static final int KEY_TYPED

Konstansok: a billentyűesemény lehetséges azonosítói (`id`).

- ▶ static final int VK_0
- ▶ static final int VK_1
- ▶ ...
- ▶ static final int VK_A
- ▶ static final int VK_B
- ▶ ...
- ▶ static final int VK_F1
- ▶ static final int VK_F2
- ▶ ...
- ▶ static final int VK_ENTER
- ▶ static final int VK_BACK_SPACE
- ▶ static final int VK_CANCEL
- ▶ static final int VK_ALT
- ▶ ...
- ▶ static final int VK_UNDEFINED
- ▶ static final char CHAR_UNDEFINED

A különféle billentyűkhöz rendelt konstansok. A `VK` kezdőbetűk a virtuális billentyű (virtual key) rövidítésből származnak. A konkrét billentyűkódok az API-ból olvashatók ki. A billentyűk azonosításához csak ezeket a mnemonik-konstansokat ajánlatos használni!

Konstruktur, metódusok

- ▶ `KeyEvent(Component source, int id, long when, int modifiers, int keyCode, char keyChar)`

A billentyűeseményt általában a rendszer hozza létre és adja át a programnak. A programozó csak különleges esetben hoz létre billentyűeseményt. Paraméterek:

- `source`, `id`, `when`, `modifiers`: forrás, azonosító, időpont, módosítók.
- `keyCode`: A leütött billentyű kódja. Csak a `KEY_PRESSED` és `KEY_RELEASED` típusú eseményekben van értéke, egyébként definiáltlan (`VK_UNDEFINED`)
- `keyChar`: A leütött billentyű unikódja. Csak a megjeleníthető karakterekhez tartozó eseményekben van értéke, egyébként definiáltlan (`CHAR_UNDEFINED`)

- ▶ `void setKeyCode(int keyCode)`
- ▶ `int getKeyCode()`

Beállítja, illetve visszaadja a leütött billentyű kódját. A `KEY_PRESSED` és `KEY_RELEASED` eseményeknél használatos.

- ▶ void setKeyChar(char keyChar)
- ▶ char getKeyChar()

Beállítja, illetve visszaadja az eseményhez tartozó karaktert. A KEY_TYPED események-nél használatos.
- ▶ boolean isActionKey()

true, ha vezérlőbillentyűt ütöttek le.
- ▶ void setSource(Object newSource)

Megváltoztatja az esemény forrását.
- ▶ static String getKeyText(int keyCode)

Megadja a billentyűkódhoz tartozó szöveget, például G, F1, Home, Page Up, Shift stb.
A leütött billentyű kiírása szöveggel: KeyEvent.getKeyText(e.getKeyCode())
- ▶ static String getKeyModifiersText(int modifiers)

Megadja a módosítók szövegét, például Ctrl, Alt-Shift, Ctrl-Alt-Shift.
A leütött módosító billentyűk kiírása szöveggel:
KeyEvent.getKeyModifiersText((e.getModifiers()))

KeyListener, KeyAdapter

- ▶ void keyPressed(KeyEvent e)
- ▶ void keyReleased(KeyEvent e)
- ▶ void keyTyped(KeyEvent e)

A billentyűesemény kezelőmetódusai: pressed=lenyomták; released=felengedték; typed=begépelték.

Component

- ▶ void addKeyListener(KeyListener l)
- ▶ void removeKeyListener(KeyListener l)

A komponens hozzáadása a billentyűesemény figyelőláncához, s levétele róla.

Feladat – Billentyűteszt (KeyTest)

Tegyük a képernyőre egy üres keretet! A begépelt szöveget a beírással egy időben jelenítsük meg a keret címsorában! A keretet az Alt+F4 billentyűkombinációval csukjuk be!



Forráskód

```
import java.awt.event.*;
import javax.swing.*;
```

```

public class KeyTest extends JFrame
    implements KeyListener {           //1
public KeyTest() {
    setBounds(100,100,500,100);
    setDefaultCloseOperation(JFrame.DO NOTHING ON CLOSE); //2
    addKeyListener(this);
    show();
}

public void keyTyped(KeyEvent e) {           //3
    setTitle(getTitle()+e.getKeyChar());
}

public void keyPressed(KeyEvent e) {          //4
    if (e.getKeyCode()==KeyEvent.VK_F4 &&
        e.getModifiers()==InputEvent.ALT_MASK)
        System.exit(0);
}

public void keyReleased(KeyEvent e) {          //5
}

public static void main (String args[]) {
    new KeyTest();
} // main
} // KeyTest

```

A program elemzése

- ◆ //1 és //2: A keret saját magát fogja figyelni.
- ◆ //3: Ha begépeltek egy billentyűt, akkor azt hozzáteszik a keret címsorához.
- ◆ //4: A billentyűk lenyomását azért kell figyelnünk, mert az Alt+F4 kombinációt csak így lehet „elkapni”. Ha megnyomták az F4-t és vele párhuzamosan az Alt módosítót is, akkor kilépünk a programból.
- ◆ //5: A billentyűfelengedéseket nem kell figyelnünk – a kereten keletkező KEY_RELEASED típusú események az enyészetéi lesznek (a metódusnak persze így is meg kell írni a vázát).

❖ Ha a keretre ráteszünk egy (esetleg láthatatlan) címkét, akkor a program nem fog működni, mert a címke elveszi az ablak elől a billentyűeseményeket. Ebben az esetben a keretnek a címkét kell hallgatnia, vagyis //2-ben a címke hallgatóláncára kell feliratkoznia:

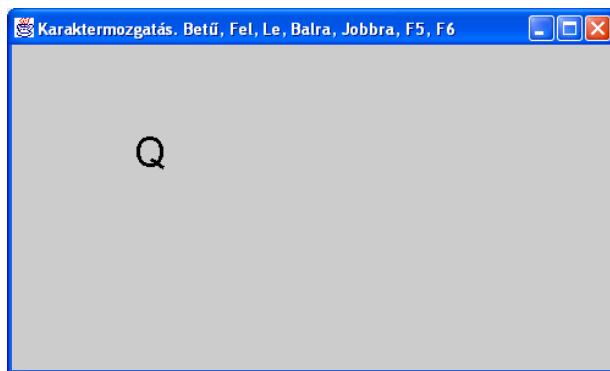
```

JLabel lb;
...
getContentPane().add(lb= new JLabel(" "));
lb.addKeyListener(this);

```

Feladat – Karaktermozgatás

Jelenítsünk meg indulásképpen egy ? karaktert! Ha leütnek egy megjeleníthető billentyűt, akkor cseréljük ki a karaktert a leütött billentyűhöz tartozó karakterre! A fel, le, jobbra és balra kurzorvezérlő billentyűk 10 ponttal tolják el a karaktert a megfelelő irányba! Az F5 leütésére a karakter mérete növekedjék meg 5 ponttal, az F6 leütésére ugyanennyivel csökkenjen! Az Esc billentyű leütésére legyen vége a programnak!



Forráskód

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class KarakterVaszon extends JPanel implements KeyListener {
    private static final int STEP=10; // az elmozdulás léptéke
    private static final int INCR=5; // a betűméret növekménye
    private char karakter='?'; // a vászonon megjelenő karakter
    private int meret=35; // a betű aktuális mérete
    private int x=100,y=100; // a betű aktuális pozíciója

    public KarakterVaszon() {
        addKeyListener(this); //1
    }

    public void paintComponent(Graphics gr) { //2
        super.paintComponent(gr);
        requestFocus();
        gr.setFont(new Font("Dialog",Font.PLAIN,meret));
        gr.drawString(""+karakter,x,y);
    }

    public void keyPressed(KeyEvent ev) { //3
        if (ev.getKeyCode() == ev.VK_UP)
            y -= STEP;
        else if (ev.getKeyCode() == ev.VK_DOWN)
            y += STEP;
        else if (ev.getKeyCode() == ev.VK_LEFT)
            x -= STEP;
        else if (ev.getKeyCode() == ev.VK_RIGHT)
            x += STEP;
    }
}

```

```

        else if (ev.getKeyCode() == ev.VK_F5)
            meret += INCR;
        else if (ev.getKeyCode() == ev.VK_F6 && meret >= INCR)
            meret -= INCR;
        else if (ev.getKeyCode() == ev.VK_ESCAPE)
            System.exit(0);
        repaint();
    }

    public void keyReleased(KeyEvent ev) {

    }

    public void keyTyped(KeyEvent ev) { //4
        karakter = ev.getKeyChar();
        repaint();
    }
}

public class KarakterMozgatas extends JFrame {
    public KarakterMozgatas() {
        setBounds(100,100,500,300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setTitle("Karaktermozgatás. Fel, Le, Balra, +
                  "Jobbra, F5, F6");
        Container cp = getContentPane();
        cp.add(new KarakterVaszon());
        show();
    }

    public static void main(String args[]) {
        new KarakterMozgatas();
    } // main
} // KarakterMozgatas

```

A program elemzése

A karaktert felügyelő és mozgató panelt beillesztjük a tartalompanelbe. A panel megjegyzi a karaktert, a karakter aktuális méretét és pozíóját, és ő kezeli az alkalmazáshoz érkező billentyűeseményeket.

- ◆ //1: A panel figyeli a maga billentyűeseményeit.
- ◆ //2: A paintComponent szolgál a megfelelő méretű és alakú karakter megjelenítésére. A requestFocus-ra azért van szükség, mert alapértelmezésben egy közönséges panel nincs a fókusz birtokában. A fókusz beállításához az kell, hogy a panel látható legyen, a konstruktorban tehát a requestFocus még hatástalan.
- ◆ //3: Ide érkeznek a vezérlőbillentyűk eseményei; azoknak megfelelően változtatjuk a karakter pozíóját, illetve méretét. A megjelenítés a paintComponent dolga, de őt a rendszer csak egy adatváltozás miatt nem hívja meg – az újrarendezést a repaint metódussal kényszerítjük ki.
- ◆ //4: karakter értéke a most begépelt billentyűnek megfelelő karakter lesz. A repaint metódussal érjük el, hogy a karakter azonnal megjelenjen a panelen.

10.5. Egéresemény – MouseEvent

Egér



A számítógéphez kapcsolt egérnek legalább két gombja van (bal és jobb) és egy mozgásérzékelője: annak a közreműködésével határozható meg az egér pillanatnyi helyzete. Az egérrrel a következő akciók végezhetők el:

- **Elmozgatják;**
- **Lenyomják** valamelyik gombját;
- **Felengedik** valamelyik gombját;
- **Kattintanak** valamelyik gombon (gyors egymásutánban lenyomják, majd felengedik). A kattintás lehet egyszeres (szimpla), kétszeres (dupla), vagy többszörös.

Egéresemény

Egéresemény (`MouseEvent`) elvileg bármely komponensen keletkezhet. Az eseménykeletkezésnek az a feltétele, hogy a komponens látható (`visible`) és engedélyezett (`enabled`) legyen. Az egéresemény mindenkor a komponensen keletkezik, amely felett éppen ott van az egér.

Az aktor (felhasználó) az egérrrel egyszerre több akciót is végrehajthat (például nyomja a gombot, s közben mozgatja az egeret). Az egérakció történtekor a rendszer a következő egéreseményeket keltheti – másszóval az egéresemények a következő fajtájuk lehetnek:

- `MOUSE_CLICKED`: Kattintottak egy gombon (egyszer, kétszer vagy többször);
- `MOUSE_ENTERED`: Az egér a komponens fölé került (belépett a komponens területére);
- `MOUSE_EXITED`: Az egér elhagyta a komponenst;
- `MOUSE_PRESSED`: Lenyomtak egy gombot (amíg nem engedik fel a gombot, addig nem keletkezik új `MOUSE_PRESSED` esemény);
- `MOUSE_RELEASED`: Felengedtek egy gombot;
- `MOUSE_DRAGGED`: Vonzolták az egeret (folyamatosan keletkezhet);
- `MOUSE_MOVED`: Elmozgatták az egeret (folyamatosan keletkezhet).

Mivel egéreseményből sokféle (összesen 7-féle) van, azért a Java két külön egérfigyelő lánct definiál:

- az egéreseményeket (`MouseEvent`) megkapja minden olyan `MouseListener`, illetve `MouseAdapter`, amely `addMouseListener` metódussal fel van fűzve a komponens figyelőláncára (a `MouseAdapter` osztály üres metódusokkal implementálja a `MouseListener` interfész).

- az egérmozgás eseményeket (`MouseEvent`) megkapja minden olyan `MouseListener`, illetve `MouseMotionAdapter`, amely az `addMouseListener` metódussal fel van fűzve a komponens figyelőláncára (a `MouseMotionAdapter` osztály üres metódusokkal implementálja a `MouseListener` interfészt).

Az egéresemények keletkezése és kezelése:

Esemény	Mi történt?	Figyelőinterfész	Felfűzőmetódus	Eseménykezelők
MouseEvent	Kattintottak rajta, fölé került, elhagyták, lenyomták, felengedték.	MouseListener	addMouseListener	mouseClicked mouseEntered mouseExited mousePressed mouseReleased
MouseEvent	Vonszolták, elmozdították.	MouseMotionListener	addMouseMotionListener	mouseDragged mouseMoved

Most végignézzük az egéreseményekkel kapcsolatos osztályok fontosabb deklarációit:

InputEvent

Absztrakt osztály, a billentyű- és egéresemények (a felhasználó által keltett beviteli események) közös őse. Leírását lásd az előző pontban, a billentyűesemény tárgyalásánál.

MouseEvent

Mezők

- static final int MOUSE_CLICKED
- static final int MOUSE_DRAGGED
- static final int MOUSE_ENTERED
- static final int MOUSE_EXITED
- static final int MOUSE_MOVED
- static final int MOUSE_PRESSED
- static final int MOUSE_RELEASED

Konstansok: az egéresemény lehetséges azonosítói (`id`).

Konstruktur, metódusok

- `MouseEvent(Component source, int id, long when, int modifiers)`
- Az egéreseményt általában a rendszer hozza létre és adja át a programnak. A programozó csak különleges esetben hoz létre egéreseményt. Paraméterek:
- `source, id, when, modifiers`: forrás, azonosító, időpont, módosítók.

- ▶ int getX()
- ▶ int getY()
- ▶ Point getPoint()

Az egéresemény pozíciója, az esemény forráskomponenséhez képest.

- ▶ void translatePoint(int x, int y)

Eltolja az egéresemény pozíóját: x-et, illetve y-t ad hozzá a koordinátához.

- ▶ int getClickCount()

Visszaadja az egérkattintások számát. Ha a kattintás nem folyamatos, akkor a számolás előlről kezdődik. Ha például az egérrel duplán kattintunk, akkor a függvény visszatérési értéke 2. A kattintások száma akárhány lehet.

MouseListener, MouseAdapter

- ▶ void mouseClicked(MouseEvent e)
- ▶ void mouseEntered(MouseEvent e)
- ▶ void mouseExited(MouseEvent e)
- ▶ void mousePressed(MouseEvent e)
- ▶ void mouseReleased(MouseEvent e)

Az egéresemény kezelőmetódusai: `clicked`=kattintottak rajta; `entered`=fölé került; `exited`=elhagyta; `pressed`=lenyomták; `released`=felengedték.

MouseMotionListener, MouseMotionAdapter

- ▶ void mouseDragged(MouseEvent e)
- ▶ void mouseMoved(MouseEvent e)

Az egérmozgás-esemény kezelőmetódusai: `dragged`=vonszolták; `moved`=elmozgatták.

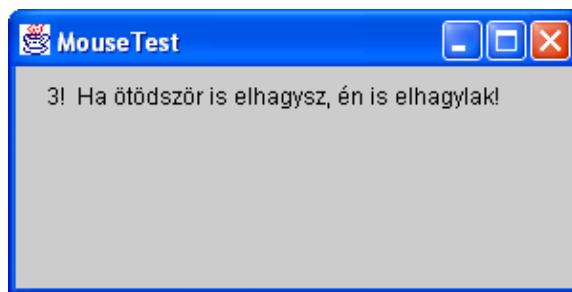
Component

- ▶ void addMouseListener(MouseListener l)
- ▶ void addMouseMotionListener(MouseMotionListener l)
- ▶ void removeMouseListener(MouseListener l)
- ▶ void removeMouseMotionListener(MouseMotionListener l)

A komponens hozzáadása az egér-, illetve az egérmozgás eseménylemezhez, és levétele róla.

Feladat – Egérteszt (MouseTest)

Tegyünk a képernyőre egy üres keretet, majd folyamatosan írjuk ki rá a rajta történt egéreseményeket! Írjuk ki az egérkursor pozíciójába, hogy melyik gombbal (bal, jobb vagy középső) mit művelt az aktor (megnyomta, felengedte, hányat kattintott, vonszolta, elmozdította)! Ha az aktor az egérrel ötödször is elhagyja az ablak területét, akkor csukjuk be az ablakot – s erre minden korábbi „kilépéskor” figyelmeztesük őt!



Forráskód

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MouseTest extends JFrame implements
    MouseListener, MouseMotionListener {           //1
    private int exitSzam;

    public MouseTest() {
        setTitle("MouseTest");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setBounds(0,0,300,150);
        addMouseListener(this);                      //2
        addMouseMotionListener(this);
        show();
    }

    void drawString(String str,int x, int y) {      //3
        Graphics gr = this.getGraphics();
        gr.clearRect(0,0,getWidth(),getHeight());
        gr.drawString(str,x,y);
        gr.dispose();
    }

    String melyikGomb(MouseEvent e) {               //4
        if (e.getModifiers()==e.BUTTON1_MASK)
            return "Bal";
        else if (e.getModifiers()==e.BUTTON2_MASK)
            return "Középső";
        else if (e.getModifiers()==e.BUTTON3_MASK)
            return "Jobb";
        return "";
    }
    // Egyszerű egéresemények:
}

```

```

public void mouseClicked(MouseEvent e) { //5
    drawString(melyikGomb(e)+" gombon "+
        e.getClickCount()+"-t kattintottál",e.getX(),e.getY());
}

public void mousePressed(MouseEvent e) { //6
    drawString(melyikGomb(e)+" gombot megnyomtad",
        e.getX(),e.getY());
}

public void mouseReleased(MouseEvent e) { //7
    drawString(melyikGomb(e)+" gombot felengedted",
        e.getX(),e.getY());
}

public void mouseEntered(MouseEvent e) { //8
}

public void mouseExited(MouseEvent e) { //9
    if (++exitSzam==5)
        System.exit(0);
    drawString(exitSzam+
        "! Ha ötödször is elhagysz, én is elhagy lak!",20,50);
}

// Egérmozgás-események:
public void mouseDragged(MouseEvent e) { //10
    drawString(melyikGomb(e)+" gombbal vonszoltad",
        e.getX(),e.getY());
}

public void mouseMoved(MouseEvent e) { //11
    drawString("Elmozgattad",e.getX(),e.getY());
}

public static void main (String args[]) {
    new MouseTest();
} // main
} // MouseTest

```

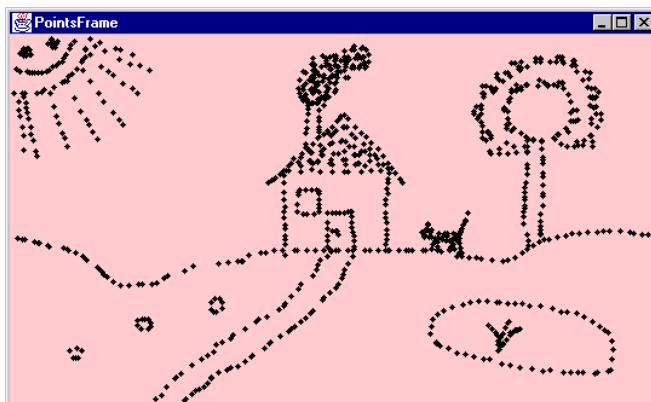
A program elemzése

- ◆ //1: A keret minden egéreseményt figyel, ezért minden két egérinterfészt implementálnia kell!
- ◆ //2: A keretet minden két figyelőláncra felfüzzük.
- ◆ //3: A JFrame nem utódja a JComponent-nek, nincs paintComponent metódusa. A kiírást a drawString metódussal intézzük el úgy, hogy elérjük a kerettől a Graphics objektumát. A drawString metódus a megadott koordinátaban kiírja a megadott szöveget. Az előző írást nem kell megjegyeznünk, sőt törölünk kell az esetlegesen ott maradt kiírást: ezt a clearRect metódussal végeztetjük el.

- ◆ //4: A melyikGomb() metódus kiolvassa a paraméterben megkapott egéreseményből, hogy melyik gombot nyomták le. Az eredményt szövegesen adja vissza.
- ◆ //5: Kattintottak az egérrel. Az eseményből kiolvassuk, hogy melyik gombon és hány-szor kattintottak, és ezt az adatot kiírjuk az egér pozíójában.
- ◆ //6: Lenyomták az egérgombot: Az eseményből kiolvassuk, hogy melyik gombot nyom-ták meg, és ezt az adatot kiírjuk az egér pozíójában.
- ◆ //7: Felengedték az egérgombot: Az eseményből kiolvassuk, hogy melyik gombot engedték fel, és ezt az adatot kiírjuk az egér pozíójában.
- ◆ //8: Az egér feltehetőleg többször is belép a keretre (különben nem tudna többször kimenni), de ezt az eseményt most figyelmen kívül hagyjuk.
- ◆ //9: Kilépéskor növeljük a kilépések számát, és ha az eléri az 5-öt, akkor befejezzük a programot. Az adatot itt nem a kilépés helyére tesszük, mert akkor valószínűleg lecsúszna a keretről.
- ◆ //10: Vonszolták, vagyis mozgatták az egeret, s közben egy gombot nyomva tartottak. Az eseményből kiolvassuk, hogy ez melyik gombbal történt, és ezt az adatot kiírjuk az egér pozíójában.
- ◆ //11: Elmozgatták az egeret. Ezt az adatot kiírjuk az egér pozíójában.

Feladat – Pontozó

Tegyük a keretbe egy vásznat, majd a vászon minden egérkattintás helyére tegyük egy pontot!



Forráskód

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;

class PontozoVaszon extends JPanel {
    private Vector pontok = new Vector();           //1
    
```

```

public PontozoVaszon() {
    setBackground(Color.PINK);
    setOpaque(true);
    addMouseListener(new EgerFigyelo()); //2
}

public void paintComponent(Graphics gr) { //3
    super.paintComponent(gr);
    Point p;
    for (int i=0; i<pontok.size(); i++) {
        p = (Point)(pontok.get(i));
        gr.fillOval(p.x,p.y,5,5);
    }
}

class EgerFigyelo extends MouseAdapter {
    public void mousePressed(MouseEvent e) { //4
        pontok.add(new Point(e.getX(),e.getY()));
        Graphics gr = getGraphics();
        gr.fillOval(e.getX(),e.getY(),5,5);
        gr.dispose();
    }
}
}

public class Pontozo extends JFrame {
    public Pontozo() {
        setTitle("Pontozás");
        setBounds(0,0,640,400);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        getContentPane().add(new PontozoVaszon());
        show();
    }

    public static void main (String args[]) {
        new Pontozo();
    } // main
} // Pontozo

```

A program elemzése

A pontokat meg kell jegyeznünk (//1), hiszen most fontos, hogy a pontozóvászon újrarajzolásakor – a `paintComponent()` metódusban – az összes pont kirajzolódjék (//3). A `points` vektorhoz minden újabb kattintáskor hozzáadjuk az új pontot. Az egéreseményeket most az `EgerFigyelo` adapterrel figyeltetjük, a program így lényegesen rövidebb lesz (//2). A pontozást a `mousePressed()` metódusban végezzük el (//4). Megtehethnénk, hogy minden pontozáskor meghívjuk a `repaint()` metódust, s ezzel az egész kép kirajzolására kényszerítenénk a rendszert. Csakhogy így feleslegesen sokat rajzolnánk, s a kép esetleg rángatózna emiatt. A képet csak akkor kell újrarajzolnunk, amikor az ablakot elmozdítják vagy átméretezik – ezért a `mousePressed()` metódusban előbb megjegyezzük a legutóbbi pontot, majd csak ezt a legutóbbit rajzoltatjuk ki. A teljes kép kirajzolása a `paintComponent()` metódus dolga; azt annak rendje s módja szerint megírjuk, de hívását a rendszerre bízzuk.

Feladat – Rajzolás

Készítsünk egy rajzolóprogramot! Lehessen a vászonon különböző színnel szabadkézi vonalakat húzni, átállítani a rajzolószínt és letörölni a vásznat!



A megoldás terve

Amikor lenyomják a jobb egérgombot (`mousePressed`), akkor elkezdődik egy vonal: az aktuális színnel kirajzolunk egy pontot. Ha az egér gombot mindenkor fel is engedi, akkor a vonal nem folytatódik, de ha a gombot nyomva tartják és húzzák is (`mouseDragged`), akkor az egér pozicióját összekötjük az előző ponttal (`drawLine`). Vonalhúzás közben nem változhat a szín.

Persze meg kell jegyeznünk a teljes rajzot. Erre egy `lines` azonosítójú vektort használunk; a `lines`-nak `Polygon`-objektumok az elemei, s mindegyik egy-egy vonalat ír le. A sokszög utolsó és első pontját nem szabad összekötni, ezért a `drawPolyline` metódust használjuk kirajzolásra. A színeket egy külön `drawColors` vektorban jegyezzük meg. Az egész rajzot csak a `paintComponent` metódusban rajzoljuk ki, egyébként mindenig csak annyit, amennyit éppen kell.

Forráskód

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;

class Rajzlap extends JPanel implements MouseListener,
    MouseMotionListener {
    // minden eleme egy egyszínű, összefüggő vonal:
    private Vector vonalak = new Vector();
    private Polygon aktVonal;

    // A színeket külön vektorban tároljuk:
    Vector vonalSzinek = new Vector();
    Color aktSzin;
}

```

```
public Rajzlap() {
    setSize(400,400);
    addMouseListener(this);
    addMouseMotionListener(this);
}

// A kép törlése:
public void clear() {
    vonalak.clear();
    vonalSzinek.clear();
    repaint();
}

// Aktuális rajzolószín beállítása:
public void setColor(Color c) {
    aktSzin = c;
}

// A teljes kép (összes sokszög) kirajzolása:
public void paintComponent(Graphics gr) {
    super.paintComponent(gr);
    Polygon vonal;
    Color szin;
    for (int i=0; i<vonalak.size(); i++) {
        szin = (Color)(vonalSzinek.get(i));
        vonal = (Polygon)(vonalak.get(i));
        gr.setColor(szin);
        gr.drawPolyline(vonal.xpoints,
                        vonal.ypoints,vonal.npoints);
    }
}

public void mouseClicked(MouseEvent e) {
}

public void mousePressed(MouseEvent e) {
    // Vonalkezdés, egy pont hozzáadása a sokszöghöz.
    // Először adatmódosítás:
    int x=e.getX(), y=e.getY();
    vonalSzinek.add(aktSzin);
    vonalak.add(aktVonal = new Polygon());
    aktVonal.addPoint(x,y);
    // Pont kirajzolása:
    Graphics gr = getGraphics();
    gr.setColor(aktSzin);
    gr.drawLine(x,y,x,y);
    gr.dispose();
}

public void mouseReleased(MouseEvent e) {
}

public void mouseEntered(MouseEvent e) {
}
```

```

public void mouseExited(MouseEvent e) {
}

public void mouseDragged(MouseEvent e) {
    // Vonal folytatása, ha már el van kezdve a vonal:
    if (aktVonal != null && aktVonal.npoints > 0) {
        // Adatmódosítás:
        aktVonal.addPoint(e.getX(),e.getY());
        // Utolsó vonalszakasz kirajzolása:
        Graphics gr = getGraphics();
        gr.setColor(aktSzin);
        gr.drawLine(aktVonal.xpoints[aktVonal.npoints-2],
                    aktVonal.ypoints[aktVonal.npoints-2],
                    aktVonal.xpoints[aktVonal.npoints-1],
                    aktVonal.ypoints[aktVonal.npoints-1]);
        gr.dispose();
    }
}

public void mouseMoved(MouseEvent e) {
}

public class Rajzolas extends JFrame implements ActionListener {
    JButton btTorol, btRajzoloSzin;
    Rajzlap rajzlap;
    Color[] colors = {Color.BLACK,Color.WHITE,Color.RED,
                     Color.PINK,Color.ORANGE,Color.MAGENTA,Color.BLUE};
    int rajzoloSzin = 0;

    public Rajzolas() {
        setSize(500,400);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        // Törlés és színállítás gombok:
        JPanel pn = new JPanel();
        pn.add(btTorol = new JButton("Töröl"));
        pn.add(new JLabel("Rajzolószín",JLabel.RIGHT));
        pn.add(btRajzoloSzin = new JButton(""));
        btRajzoloSzin.setBackground(colors[rajzoloSzin]);

        getContentPane().add(pn,"North");

        // Rajzolóterület:
        getContentPane().add(rajzlap = new Rajzlap(),"Center");
        rajzlap.setBackground(Color.LIGHT_GRAY);
        rajzlap.setColor(colors[rajzoloSzin]);

        btRajzoloSzin.addActionListener(this);
        btTorol.addActionListener(this);
        show();
    }
}

```

```

public void actionPerformed(ActionEvent e) {
    if (e.getSource() == btTorol)
        rajzlap.clear();
    else if (e.getSource() == btRajzoloSzín) {
        if (rajzoloSzín < colors.length-1)
            rajzoloSzín++;
        else
            rajzoloSzín = 0;
        btRajzoloSzín.setBackground(colors[rajzoloSzín]);
        rajzlap.setColor(colors[rajzoloSzín]);
    }
}

public static void main (String args[]) {
    new Rajzolas();
} // main
} // Rajzolas

```

Tesztkérdések

10.1. Mely állítások igazak? Jelölje be az összes helyes állítást!

- a) Az alacsony szintű események valamennyi osztálya a ComponentEvent leszármazottja.
- b) Az AdjustmentEvent alacsony szintű eseményosztály.
- c) A billentyű-, az egér- és az ablakesemények mind az InputEvent példányai.
- d) Ha egy ablakot ikonná változtatnak, akkor WindowEvent esemény keletkezik.

10.2. Mely állítások igazak? Jelölje be az összes helyes állítást!

- a) Ha egy komponens láthatóvá válik, akkor a keletkezett esemény ComponentShown osztályú.
- b) A ComponentAdapter interfész.
- c) A ComponentEvent.getComponent() az esemény forrásobjektumát adja vissza.
- d) Megtörténhet, hogy egy COMPONENT_MOVED és egy COMPONENT_RESIZED esemény pontosan egy időben keletkezzék.

10.3. Mely állítások igazak? Jelölje be az összes helyes állítást!

- a) A felhasználói felületen egyszerre legfeljebb annyi komponens lehet fókuszban, ahány alkalmazás fut.
- b) Egy komponens általában más látványt nyújt, ha fókuszból kerül.
- c) Ha egy komponens látszik és engedélyezve van, akkor fókuszban is van.
- d) Billentyűesemény csak a fókuszban levő komponensen keletkezhet.

10.4. Mely állítások igazak? Jelölje be az összes helyes állítást!

- a) minden konténernek (Container leszármazottnak) van fókuszlánca.
- b) Ha egy komponens nem járható be (requestFocusEnabled==false), akkor az a komponens semmilyen módon nem hozható fókuszbba.

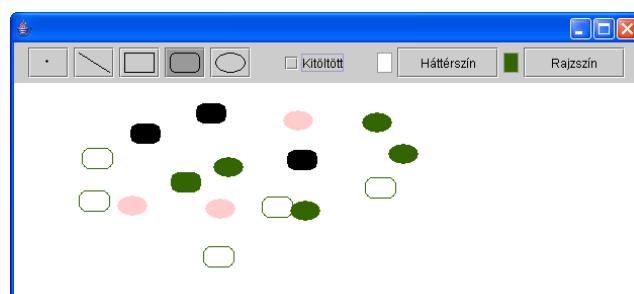
- c) A fókuszlánc bejárási sorrendjét a Swingben egy konténeren belül alapértelmezés szerint a komponensek bal felső sarka határozza meg.
- d) Egy ablak létrehozásakor alapértelmezésben az első olyan gyerekkomponens lesz fókuszban, amelyik látható, engedélyezett és fokuszálható.
- 10.5. Mely állítások igazak? Jelölje be az összes helyes állítást!
- A következő billentyűk mind módosítóbillentyűk: Enter, Delete, Home.
 - KEY_TYPED esemény csak KEY_PRESSED esemény után keletkezhet.
 - A billentyűesemény forrásobjektuma mindenkor az aktív ablak.
 - Ha egy ismétlő alfanumerikus billentyűt hosszabb ideig tartunk nyomva, akkor sok KEY_RELEASED esemény keletkezik.
- 10.6. Hány KeyEvent esemény keletkezik, ha egyszerre leütjük a Shift és az F1 billentyűt és nyomban fel is engedjük őket? Jelölje be az egyetlen lehetséges választ!
- Két esemény keletkezik; azonosítójuk rendre KEY_PRESSED, KEY_RELEASED
 - Három esemény keletkezik; azonosítójuk rendre KEY_PRESSED, KEY_RELEASED, KEY_TYPED
 - Négy esemény keletkezik; azonosítójuk rendre KEY_PRESSED, KEY_PRESSED, KEY_RELEASED, KEY_RELEASED
 - Hat esemény keletkezik; azonosítójuk rendre KEY_PRESSED, KEY_TYPED, KEY_PRESSED, KEY_TYPED, KEY_RELEASED, KEY_RELEASED
- 10.7. Mely állítások igazak? Jelölje be az összes helyes állítást!
- Az egéresemény mindenkor a fókuszban levő komponensen keletkezik.
 - A következő azonosítók mindenkor a MouseEvent statikus adatai: MOUSE_PRESSED, MOUSE_DRAGGED, MOUSE_CLICKED, MOUSE_MOVED
 - A mouseMoved eseménykezelő metódust a MouseListener interfész definiálja.
 - A mouseClicked eseménykezelő metódust a MouseListener interfész definiálja.
- 10.8. Mely állítások igazak? Jelölje be az összes helyes állítást!
- Egy egéresemény módosítható mindenkor módosítóbillentyűvel.
 - Egy billentyűesemény módosítható mindenkor egérgomb lenyomásával.
 - A Component.requestFocus() metódusnak csak akkor van hatása, ha a komponens látható.
 - A fókuszláncban levő szövegmezők bejárhatók mindenkor Enter billentyű sorozatos leütésével..

Feladatok

- 10.1. (A) Figyelje a program futásakor keletkező ComponentEvent, WindowEvent és KeyEvent alacsony szintű eseményeket! Tegyen a keretbe egy szövegterületet, és abba folyamatosan írja bele a kereten keletkező események lényeges tulajdonságait! Az információs sorokat lássa el sorszámmal! Engedje meg a felhasználónak, hogy törölje a szövegterületet, illetve üres sorokat szúrjon be, hogy a sorok tagolhatók legyenek! (*EseményFigyelo.java*)



- 10.2. Tegyen a keretbe néhány fokuszálható komponenst! A program minden fókusztáltáskor
- (A) adjon hangot! (*HangosFokusz.java*)
 - (B) jelenítsen meg egy kis időre a keret jobb felső sarkában s kis herceget (például a "dukewave.gif" ikon)! (*HercegFokusz.java*)
- 10.3. (B) Tegyen a keretbe egy képet, például az "amhappy.jpg"-t, és csökkentse a tizedére a méretét! Mozgassa a képet a fel, le, balra, jobbra billentyűkkel! F1-re a kép tűnjön el, illetve legyen újra látható! Ne engedje, hogy a kép elhagyja a keretet! Ha átméretezik a panelt, akkor a kép pozíciója változzon arányosan a panel méretével!
(*KepMozgatas.java*)
- 10.4. Tegyen a képernyőre egy üres keretet, s ha legalább kettőt kattintanak rajta az egérrel, akkor írassa ki a kattintás helyének koordinátait (x,y) alakban!
- (A) Egyszerre csak egy koordináta legyen látható, de az az ablak elmozdításakor is maradjon a helyén! (*EgerKoord1.java*)
 - (B) Mindig legyenek láthatók az előző kattintás eredményei is! (*EgerKoord2.java*)
- 10.5. (C) Készítsen egy alakzatrajzoló programot! A következő dolgokat lehessen eszköztár-ból választani:
- Alakzat: pont, egyenes, téglalap, lekerekített téglalap, ellipszis
 - Kitöltés van/nincs
 - Színek: háttérszín és az alakzat színe
- Az alakzatok fix méretűek és irányúak legyenek. A program tegye ki az egér pozíciójába a kiválasztott alakzatot! (*AlakzatRajzolo.java*)



- 10.6. (C) Egészítse ki az *AlakzatRajzolo* programot az Undo/Redo funkciókkal! A legutóbb rajzolt vonalak legyenek visszavonhatók és újra kirajzolhatók!
(A KissDraw esettanulmányban megtalálja ezeket a funkciókat.)
- 10.7. (C) Készítsen egy rácsos felületet, és tegye lehetővé, hogy arra vonszolással téglalapot tegyenek: egy egérkattintás adja meg a téglalap egyik sarkát; az átellenes sarok pedig ott legyen, ahol a vonszolást befejezik! (*VonszoltTegla.jpx*)
- 10.8. (C) Készítsen el egy 2000*1000-es, átlósan áthúzott rajzlapot (az egyenesek a sarkokból indulnak), és tegyen rá egy, a lapszélét érintő ellipszist. A rajzlapot megmutató keret legyen átméretezhető, s kezdetben 800*600-as méretű. Tegye lehetővé, hogy a felhasználó görgetősávok segítségével keresztbe-kasul pásztázhassa a rajzlapot!
(RajzlapPasztazas.java)

11. Belső eseménykezelés, komponensgyártás

A fejezet pontjai:

1. Esemény keletkezése és életútja
 2. Események feldolgozása
 3. Komponensgyártás – feladatok
-

Ebben a fejezetben az eseménykezelés egy újabb módját ismerjük meg. Megtudjuk, hogyan lehet egy felhasználói (alacsony szintű, AWT-) eseményt még a figyelő objektumoknak való szétosztás előtt a forrásobjektumon belül feldolgozni – megvizsgálni, megváltoztatni az esemény állapotát, vagy az egész eseményt elpusztítani. A fejezet utolsó pontjában egy kis lépést teszünk a komponensgyártás felé: olyan osztályokat – némi túlzással komponenseket – fogunk gyártani, amelyek részei lehetnének egy használható, „eladható” komponensgyűjteménynek is.

11.1. Esemény keletkezése és életútja

Ebben a pontban azt vizsgáljuk meg, hogy a programozó szempontjából hogyan és hol keletkezik egy alacsony szintű esemény, milyen utat jár be, és mely pontokon „kaphatja el” a programozó.

A különböző eseményobjektumokat az alkalmazás állítja össze, az operációs rendszer segítségével. minden AWT-eseményobjektumra jellemzők a következők:

- ◆ az esemény osztálya;
- ◆ az esemény azonosítója;
- ◆ a forrásobjektum (az esemény keletkezésének helye).

A forrásobjektumon meghatározott típusú események keletkezhetnek – hogy milyenek, azt a forrásobjektum osztálya határozza meg. A forrásobjektumnak figyelőláncai vannak, eseménytípusonként általában egy (a MouseEvent-eseményekhez azonban kettő). A keletkezett eseményt csak maga a forrásobjektum és a megfelelő figyelőlánc objektumai kapják meg.

A keletkezett esemény a következő utat járja be: a forrásobjektumtól indul, és sorban „meglátogatja” a figyelőláncokra felfűzött objektumokat. A „vendéglátó” objektumok (a forrás és a hallgatók) reagálhatnak az eseményre, módosíthatják vagy el is pusztíthatják. Az elpusztított eseményt az úton még hátra levő többi objektum már nem dolgozhatja fel.

A komponenseknek (potenciális forrásobjektumoknak) több figyelőláncuk is lehet. minden komponensnek (Component-leszármazott) megvan például a következő néhány figyelőlánca:

- ◆ ComponentListener-figyelőlánc;
- ◆ FocusListener-figyelőlánc;
- ◆ KeyListener-figyelőlánc;
- ◆ MouseListener-figyelőlánc;
- ◆ MouseMotionListener-figyelőlánc.

Emellett

- ◆ minden Container-komponensnek van egy ContainerListener-figyelőlánca;
- ◆ minden Window-komponensnek van egy WindowListener-figyelőlánca;

A 11.1. ábrán látható példában a bt1 nyomógomb a forrásobjektum (az összes esemény rajta keletkezik), és három nem üres figyelőlánca van:

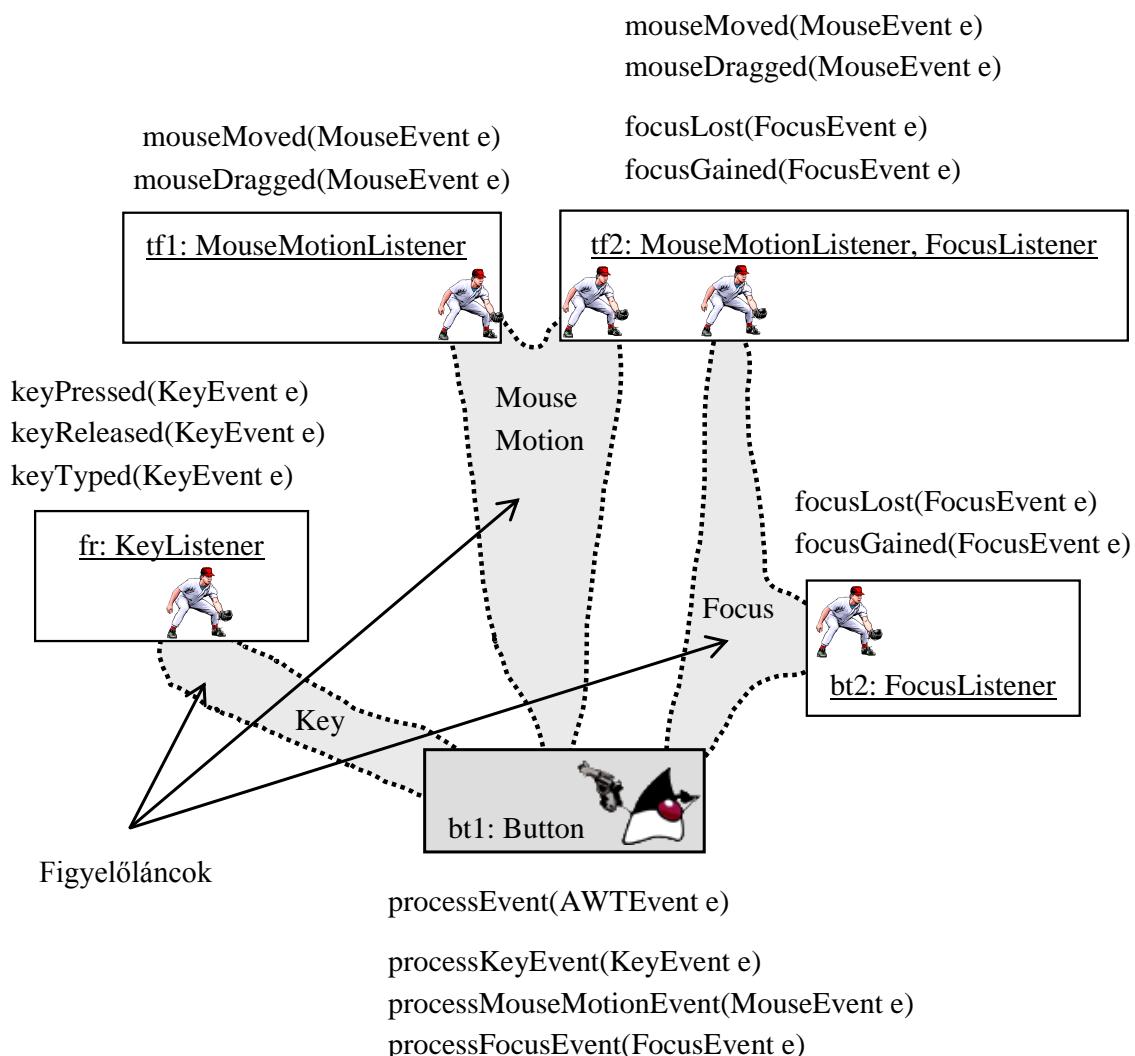
- ◆ A KeyEvent-eseményeket az fr keret figyeli;
- ◆ A MouseMotionEvent-eseményeket a tf1 és a tf2 szövegmező figyeli;
- ◆ A FocusEvent-eseményeket a tf2 szövegmező és a bt2 nyomógomb figyeli.

Látható, hogy a tf2 két eseményláncra is fel van fűzve: a MouseMotion- és a Focus-figyelőláncra. A tf2 objektum csak származtatott osztályú lehet (a JTextField leszármazott-jából való), mert a JTextField magától nem figyeli így ezeket az eseményeket.

A keletkezett eseményt először maga a forrásobjektum kapja meg kezelésre, éspedig azzal, hogy végrehajtódik a forrásobjektum processEvent eseményfeldolgozó metódusa. A metódus paraméterként kapja meg az eseményt, és minden eseményt annyit tesz, hogy ezt az eseményt az eseménytípusnak megfelelően átadja a komponens egy másik eseményfeldolgozó metódusának (processKeyEvent, processMouseMotionEvent...). Ez az eseménytípushoz rendelt metódus pedig úgy kezeli a kapott eseményt, hogy átadja minden figyelőjének – a figyelőláncára felfűzött összes objektumnak.

Az esemény meghatározott utat jár be mint a következő metódusok paramétere:

- A forrásobjektum **processEvent** (AWTEvent e) **eseményfeldolgozó** metódusa;
- A forrásobjektum megfelelő **processXXXEvent** (XXXEvent e) **eseményfeldolgozó** metódusa;
- Az összes figyelőobjektum megfelelő **eseménykezelő** metódusa(i).



11.1. ábra. Figyelőláncok

Könnyebb lesz megérteni a `Component` osztály `processEvent()` és `processXXXEvent()` metódusainak működését, ha tanulmányozzuk a forráskódjukat:

API-forráskód – Component – processEvent

```

protected void processEvent(AWTEvent e) {
    if (e instanceof FocusEvent) {
        processFocusEvent((FocusEvent)e);
    }
    else if (e instanceof MouseEvent) {
        switch(e.getID()) {
            case MouseEvent.MOUSE_PRESSED:
            case MouseEvent.MOUSE_RELEASED:
            case MouseEvent.MOUSE_CLICKED:
            case MouseEvent.MOUSE_ENTERED:
            case MouseEvent.MOUSE_EXITED:
                processMouseEvent((MouseEvent)e);
                break;
            case MouseEvent.MOUSE_MOVED:
            case MouseEvent.MOUSE_DRAGGED:
                processMouseMotionEvent((MouseEvent)e);
                break;
        }
    }
    else if (e instanceof KeyEvent) {
        processKeyEvent((KeyEvent)e);
    }
    else if (e instanceof ComponentEvent) {
        processComponentEvent((ComponentEvent)e);
    }
}

```

API-forráskód – Component – processKeyEvent

```

protected void processKeyEvent(KeyEvent e) {
    KeyListener listener = keyListener; // bill. hallgatólánc

    if (listener != null) {
        int id = e.getID();
        switch(id) {
            case KeyEvent.KEY_TYPED:
                listener.keyTyped(e);
                break;
            case KeyEvent.KEY_PRESSED:
                listener.keyPressed(e);
                break;

            case KeyEvent.KEY_RELEASED:
                listener.keyReleased(e);
                break;
        }
    }
}

```

11.2. Események feldolgozása

Egy komponens osztályában a következő maszkok és eseményfeldolgozó metódusok tartoznak a különféle alacsony szintű eseménycsoportokhoz:

Eseménymaszk	Eseményfeldolgozó metódus
COMPONENT_EVENT_MASK	processComponentEvent (ComponentEvent e)
FOCUS_EVENT_MASK	processFocusEvent (FocusEvent e)
KEY_EVENT_MASK	processKeyEvent (KeyEvent e)
MOUSE_EVENT_MASK	processMouseEvent (MouseEvent e) processMouseMotionEvent (MouseEvent e)
WINDOW_EVENT_MASK	processWindowEvent (WindowEvent e)
CONTAINER_EVENT_MASK	processContainerEvent (ContainerEvent e)
	processEvent (AWTEvent e)

Egy eseménymaszk több eseményt is jellemz egyszerre. A következő utasítással például az összes billentyűeseményt egyszerre engedélyezhetjük egy komponensen:

```
enableEvents (AWTEvent.KEY_EVENT_MASK);
```

Az eseménymaszkokat az AWTEvent, az eseményfeldolgozó metódusokat pedig a Component osztály megfelelő leszármazottja deklarálja. Az eseményfeldolgozó metódusok védettek – közvetlen módon nem lehet meghívni őket, csak felülírni.

Előfordulhat, hogy a forrásobjektumon keletkezett eseményt nem szeretnénk a figyelőknek közvetlenül, minden ellenőrzés nélkül átadni. Az eseményeket sokkal egyszerűbb magában a forrásobjektumban megvizsgálni, megváltoztatni vagy elpusztítani, mint ugyanezt az összes figyelőbe beleprogramozni.

A processEvent és a processXXXEvent **eseményfeldolgozó metódusok**, s valamennyien **felülírhatók**. Felülírásukkal megváltoztatható a forrásobjektum viselkedése.

A processEvent metódushoz minden alacsony szintű esemény eljut; a processXXX-Event metódushoz már csak a megfelelő típusú.

Események engedélyezése: A forrásobjektumon csak olyan esemény keletkezhet, amilyen engedélyezve van. A forrásobjektum eseményeit az enableEvents metódussal lehet engedélyezni, illetve letiltani. A forrásobjektumon, ha van megfigyelő a megfigyelőláncán, akkor eleve keletkezhet (a megfelelő típusú) megfigyelendő esemény – de ha nincs figyelő, akkor külön engedélyezni kell az eseménykeletkezést!

A programozónak világosan látnia kell azt, hogy az ös metódusoknak is van feladatak, ha tehát nem a megfelelő módon hívja őket, akkor lehet, hogy a program váratlan módon fog működni. Ha például a processEvent metódusban nem hívjuk meg az őst, akkor az események nem jutnak el a figyelőkhöz.

Nézzük most sorra a Component osztályban deklarált ide vonatkozó metódusokat!

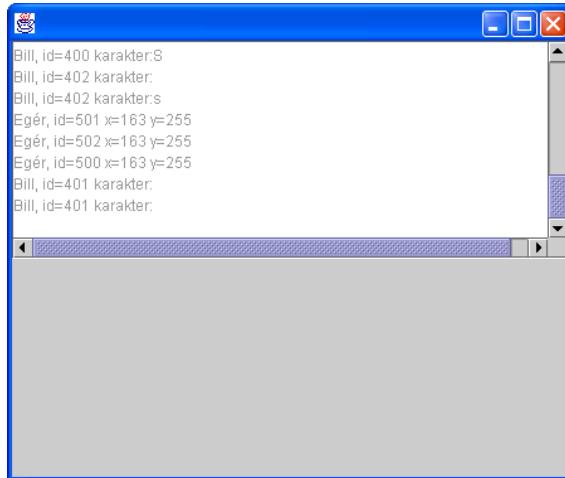
Component metódusok

- ▶ `protected void processEvent (AWTEvent e)`
Feldolgozza az ezen a komponensen keletkezett alacsony szintű eseményt. Alapértelmezésben meghívja az eseménynek megfelelő `processXXXEvent` metódust.
- ▶ `protected void processXXXEvent (XXXEvent e)`
`XXX` jelentése: Component, Focus, Key, Mouse, Window, vagy Container. Feldolgozza a komponensen keletkezett `XXXEvent` eseményt: elküldi az összes bejegyzett `XXXListener` objektumnak. Ez a metódus csak akkor hívódik meg, ha az `XXXEvent` esemény keletkezése engedélyezve van a forrásobjektumon, vagyis teljesül a következő két feltétel valamelyike:
 - A komponenshez az `addXXXListener` metódussal hozzá van fűzve legalább egy `XXXListener`-objektum;
 - Az `enableEvents` metódus révén engedélyezve van `XXXEvent`-események keletkezése.
- ▶ `protected final void enableEvents (long eventsToEnable)`
Engedélyezi, hogy a komponens fogadja a paraméterben megadott eseményeket. Egy esemény keletkezése automatikusan engedélyezetté válik, ha ilyen típusú figyelőt kapcsolnak a komponenshez. Az `eventsToEnable` paraméter megadásához felhasználhatók az `AWTEvent` osztály eseménymaszkjai: `KEY_EVENT_MASK`, `MOUSE_EVENT_MASK`, stb. A metódus védett, azt csak a Component osztály valamely leszármazottja hívhatja meg.
- ▶ `protected final void disableEvents (long eventsToDelete)`
Megtiltja, hogy a komponens fogadja paraméterben megadott eseményeket.
- ▶ `void setEnabled (boolean b)`
Engedélyezi, illetve letiltja a komponenst. Ha a paraméter értéke `false`, akkor az objektumon nem keletkezhet semmiféle esemény. Az események potenciális engedélyezése/tiltása megmarad.

Feladat – Key- és Mouse-események feldolgozása (KeyMouseFeldolg)

Készítsünk egy keretet, s figyeljük a kereten keletkezett `KeyEvent` és `MouseEvent` eseményeket (kivéve a mozgással járó egéreseményeket)! Sorban írjuk ki az események azonosítóját; ha az esemény billentyűesemény, akkor a leütött karaktert is, ha egéresemény, akkor az egér koordinátáját! Ha leütik az Escape billentyűt, akkor legyen vége a programnak! Osszuk ketté a keretet, és tegyük a felső felébe egy szövegterületet; oda íródjanak ki az adatok.

Az eseményeket a keret `processKeyEvent` és `processMouseEvent` metódusában dolgozzuk fel!



Forráskód

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class KeyMouseFeldolg extends JFrame {
    private Container cp = getContentPane();
    private JTextArea taSzoveg = new JTextArea("",10,40);

    public KeyMouseFeldolg() {
        setLocation(200,200);
        cp.setLayout(new GridLayout(2,1));
        cp.add(new JScrollPane(taSzoveg));
        taSzoveg.setEnabled(false);
        enableEvents(AWTEvent.KEY_EVENT_MASK); //1
        enableEvents(AWTEvent.MOUSE_EVENT_MASK);
        pack();
        show();
    }

    public void processKeyEvent(KeyEvent e) { //2
        taSzoveg.append("Bill, id="+e.getID()+
            " karakter:"+e.getKeyChar()+"\n");
        if (e.getKeyCode() == e.VK_ESCAPE)
            System.exit(0);
        super.processKeyEvent(e);
    }

    public void processMouseEvent(MouseEvent e) { //3
        taSzoveg.append("Egér, id="+e.getID()+
            " x="+e.getX()+" y="+e.getY()+"\n");
        super.processMouseEvent(e);
    }

    public static void main (String args[]) {
        new KeyMouseFeldolg();
    } // main
} // KeyMouseFeldolg

```

A program elemzése

A szövegterületet letiltjuk, hogy ne lehessen „belepiszkítani”. A keret eseményei ténylegesen csak a keret alsó felén és az „igazi” kereten (a keret szegélyén) keletkezhetnek.

- ◆ //1: Engedélyezzük, hogy a kereten billentyű- és egéresemények keletkezzenek.
- ◆ //2: A processKeyEvent a kereten keletkező összes billentyűeseményt megkapja, más eseményt viszont nem. Beírjuk a keretbe az esemény adatait, és ha az esemény az Escape leütése volt, akkor kilépünk a programból.
- ◆ //3: A processMouseEvent a kereten keletkező összes egéreseményt megkapja, más eseményt viszont nem. Beírjuk a keretbe az esemény adatait.

Feladat – Események feldolgozása (EventFeldolg)

Oldjuk meg az előző feladatot úgy, hogy a keret eseményeit a processEvent metódusban kezeljük!

Forráskód

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class EventFeldolg extends JFrame {
    private Container cp = getContentPane();
    private JTextArea taSzoveg = new JTextArea("", 10, 40);

    public EventFeldolg() {
        setLocation(200, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        cp.setLayout(new GridLayout(2, 1));
        cp.add(new JScrollPane(taSzoveg));
        taSzoveg.setEnabled(false);
        enableEvents(AWTEvent.KEY_EVENT_MASK);
        enableEvents(AWTEvent.MOUSE_EVENT_MASK);
        pack();
        show();
    }

    public void processEvent(AWTEvent e) {
        if (e instanceof KeyEvent) {
            KeyEvent ke = (KeyEvent)e;
            taSzoveg.append("Bill, id="+ke.getID()+
                " karakter:"+ke.getKeyChar()+"\n");
            if (ke.getKeyCode() == ke.VK_ESCAPE)
                System.exit(0);
        }
        else if (e instanceof MouseEvent) {
            MouseEvent me = (MouseEvent)e;
            if (me.getID() != MouseEvent.MOUSE_DRAGGED &&
                me.getID() != MouseEvent.MOUSE_MOVED) {

```

```

        taSzoveg.append("Egér, id="+me.getID()+
                          " x="+me.getX()+" y="+me.getY()+"\n");
    }
}
super.processEvent(e);
}

public static void main (String args[]) {
    new EventFeldolg();
} // main
} // EventFeldolg

```

A program elemzése

Most „egy kalap alá” vettük a különböző eseményeket. A `processEvent` metódusnak ezzel összhangban `AWTEvent` a paramétere, az minden eseményre ráhúzható (//2). A kalapban persze összekeverednek az események: a `processEvent` metóduson minden esemény keresztülmegy. Ha a feladat úgy kívánja, akkor itt szét kell válogatnunk őket, és ami még nehézkesebbé teszi a dolgot, az üzenet küldéséhez az eseményekre rá kell húznunk a megfelelő eseményosztályt (//3). //1-ben számlálót deklarálunk az események sorszámnak számontartásához.

A specifikus (`processXXXEvent`) eseményfeldolgozó metódusokban egyszerűbb az eseménykezelés – nem kell rákényszeríteni az eseményre a tényleges típusukat!

Megjegyzés: ha a megoldásban benne hagynánk `processKeyEvent` és `processMouseEvent` metódust, akkor a billentyű- és egéresemények kétszer lennének feldolgozva.

Feladat – Ablak becsukása belülről (AblakBecsuk)

Csukjuk be a keretet az új módszerrel! Használjuk a keret saját `processWindowEvent` metódusát!

Forráskód

```

import javax.swing.JFrame;
import java.awt.event.*;
public class AblakBecsuk extends JFrame {
    public AblakBecsuk() {
        setBounds(100,100,300,200);
        enableEvents(WindowEvent.WINDOW_CLOSING);
        show();
    }
    public void processWindowEvent(WindowEvent e) {
        if (e.getID()==WindowEvent.WINDOW_CLOSING)
            System.exit(0);
        super.processWindowEvent(e);
    }
    public static void main(String[] args) {
        new AblakBecsuk();
    }
} // AblakBecsuk

```

Esemény módosítása

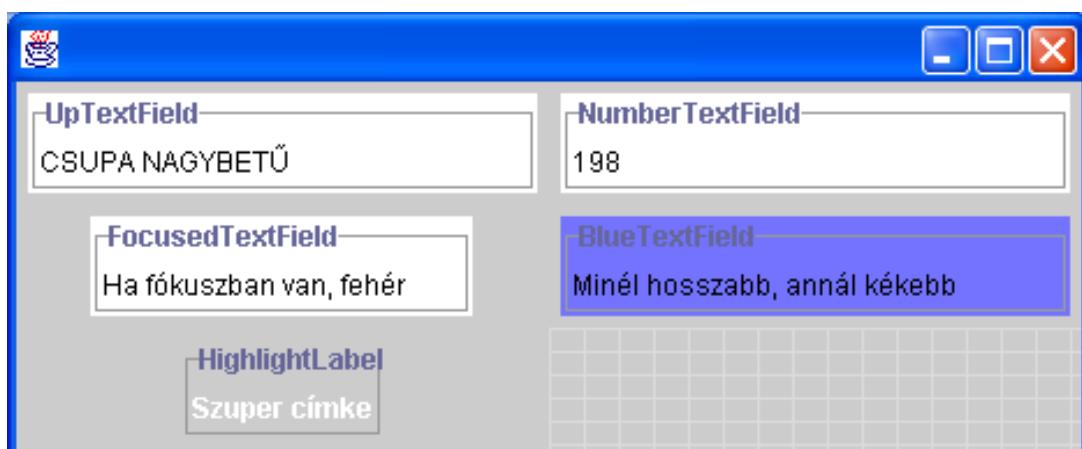
Az eseményfeldolgozó metódusokban megkapjuk feldolgozásra az eseményt. A következőket tehetjük:

- ◆ az eseményt megváltoztathatjuk; a megváltoztatott eseményből azután senki ki nem olvassa, hogy mi történt valójában. Lásd Komponensgyártás pont, UpTextField feladat – ott a leütött billentyűt nagybetűssé változtatjuk az `e.setKeyChar` metódussal.
- ◆ az eseményt elpusztíthatjuk; Lásd Komponensgyártás pont, NumberTextField feladat – ott azokat a billentyűeseményeket, amelyek nem számok, egyszerűen elpusztítjuk a `consume` metódussal.

11.3. Komponensgyártás – feladatok

A következőkben elkészítünk néhány osztályt; minden egy-egy swingbeli komponensosztály kiterjesztése lesz. A feladatokban a komponensen keletkezett eseményt helyben, a forrásban dolgozzuk fel (az eseményekre persze az eseményfeldolgozó és az eseménykezelő metódusokban is reagálhatunk).

A komponensek gyártását a Komponens projektben fogjuk össze. Az osztályokat (UpTextField, NumberTextField...) az extra.controls csomagba tesszük. A csomagot később átmásolhatjuk egy alkalmas könyvtárba (például a javalib alá), hogy az osztályokat később újra felhasználhassuk. A komponenseket a TestControls.java osztály segítségével próbáljuk ki. A tesztprogram futása és forráskódja a következő:



Forráskód

Projekt: Komponens

TestControls.java

```
import extra.controls.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class TestPanel extends JPanel {
    /* Hozzáadja a panelhez a comp komponenst. Előbb ráteszí egy
     * belső panelre.
     * Bekeretezi, a keret címe a komponens osztálya lesz:
     */
    void create(JComponent comp) {
        JPanel pnBelso = new JPanel();
        String className = comp.getClass().getName();
        className=className.substring(className.lastIndexOf(".") + 1);
        comp.setBorder(BorderFactory.createTitledBorder(className));
        pnBelso.add(comp);
        add(pnBelso);
    }

    // Konstruktor:
    public TestPanel() {
        setLayout(new GridLayout(0,2));

        // A tesztelendő komponensek
        create(new UpTextField(20));
        create(new NumberTextField(20));
        create(new FocusedTextField(15));
        create(new BlueTextField(20));
        create(new HighlightLabel("Szuper címke",Color.black,
            Color.yellow));
        add(new GridPanel(15,10));
    }
}

public class TestControls extends JFrame {
    public TestControls() {
        setLocation(200,0);
        getContentPane().add(new TestPanel());

        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        pack();
        show();
    }

    public static void main (String args[]) {
        new TestControls();
    } // main
} // TestControls
```

Feladat – UpTextField

Készítsünk egy UpTextField osztályt; ez abban térjen el a JTextField szövegmezőtől, hogy a leütött karakterek automatikusan nagybetűsek legyenek!

Forráskód

```
package extra.controls;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class UpTextField extends JTextField {
    public UpTextField(int columns) {
        super(columns);
        enableEvents(AWTEvent.KEY_EVENT_MASK); //1
    }

    public void processKeyEvent(KeyEvent e) { //2
        if (e.getID() == e.KEY_TYPED) {
            char c = Character.toUpperCase(e.getKeyChar()); //3
            e.setKeyChar(c);
        }
        super.processKeyEvent(e); //4
    }
} // UpTextField
```

A program elemzése

A konstruktörben engedélyeznünk kell, hogy a szövegmezőn billentyűesemények keletkezhessenek (//1). Ezután csak a processKeyEvent metódust kell felülírnunk, mert ha a komponensen billentyűesemény keletkezik, akkor az ezen a metóduson biztosan „keresztülmegy” (//2). Ha az esemény KEY_TYPED azonosítójú, akkor a billentyűesemény karakterét egyszerűen átírjuk nagybetűsre (//3). A metódus végén mindenkiéppen meghívjuk az processKeyEvent metódus ősét, hiszen az osztja szét az eseményeket (//4). Ez akkor lehet érdekes, ha a komponens billentyűeseményeit más is akarja hallgatni.

A megoldás lényege tehát az, hogy a beütött karaktert elkapjuk, és nagybetűssé alakítjuk még azelőtt, hogy a szövegmező bevenné a szövegébe!

Feladat – NumberTextField

Készítsünk egy szövegmezőt számok bevitelére. A mező a megjeleníthető billentyűvel beírható karakterek közül csak a számokat fogadja el!

Forráskód

```
package extra.controls;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

```

public class NumberTextField extends JTextField {
    public NumberTextField(int columns) {
        super(columns);
        enableEvents(AWTEvent.KEY_EVENT_MASK);
    }

    public void processKeyEvent(KeyEvent e) {
        if (e.getID() == e.KEY_TYPED) { //1
            char c = e.getKeyChar();
            if ((c < '0' || c > '9') & (c != e.VK_BACK_SPACE)) //2
                e.consume(); //3
        }
        super.processKeyEvent(e);
    }
} // NumberTextField

```

A program elemzése

Most a megjeleníthető billentyűkkel bírható karakterek közül csak a számokat fogadhatjuk el, vigyázunk kell azonban arra, hogy a JTextField-ben is működő szerkesztőbillentyűk minden érvényben maradjanak: a KEY_PRESSED-billentyűket tehát nem bántjuk. A mezőszerkesztő billentyűk közül az ENTER, a TAB és a BACK_SPACE KEY_TYPED jellegű, s a BACK_SPACE karaktert el kell fogadnunk közülük, mert az az eredeti mezőben is működik. //1-ben azt a feltételeztet fogalmaztuk meg, hogy az esemény egy billentyüleütés, //2-ben pedig azt, hogy az eseményben tárolt karakter nem szám és nem a BACK_SPACE. Az ilyen eseményt egyszerűen hatástanítjuk (//3), az tehát nem vesz részt a további feldolgozásban (a hatástanított esemény továbbadódik ugyan, de minden feldolgozómetódus figyelmen kívül hagyja). A processKeyEvent ōsa a megjeleníthető nem szám karakterekkel már nem dolgozik.

Feladat – FocusedTextField

Készítsünk olyan szövegmező-komponenst, amely fehér, ha fókuszon van és szürke, ha nincs fókuszon!

Forráskód

```

package extra.controls;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class FocusedTextField extends JTextField {
    Color szin=Color.LIGHT_GRAY;

    public FocusedTextField(int columns) {
        super(columns);
        enableEvents(AWTEvent.FOCUS_EVENT_MASK);
        setBackground(szin);
    }
}

```

```

public void processFocusEvent(FocusEvent e) {
    if (e.getID() == FocusEvent.FOCUS_GAINED)
        setBackground(Color.WHITE);
    else
        setBackground(szin);
}
} // FocusedTextField

```

A program elemzése

A szövegmmezőnek figyelnie kell magát, hogy fókuszba került-e vagy éppen elveszítette a fókuszt. A konstruktörben engedélyezzük a fókuszeseményeket, a processFocusEvent metódusban pedig a fókuszesemény jellegének megfelelően fehérre vagy szürkére állítjuk a mező háttérsínét.

Feladat – HighLightLabel

Készítsen egy olyan HighLightLabel(String,Color,Color) címkét, amelynek a szövege elszíneződik, ha a címke felett elmegy az egér! A címkét a szöveggel és annak kétféle színével inicializáljuk!

Forráskód

```

package extra.controls;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class HighlightLabel extends JLabel {
    private Color baseColor, highLightColor;

    public HighlightLabel(String text, Color baseColor,
                          Color highLightColor) {
        super(text);
        this.baseColor = baseColor;
        this.highLightColor = highLightColor;
        enableEvents(AWTEvent.MOUSE_EVENT_MASK);
    }

    public void processMouseEvent(MouseEvent e) {
        if (e.getID() == e.MOUSE_ENTERED)
            setForeground(highLightColor);
        else if (e.getID() == e.MOUSE_EXITED)
            setForeground(baseColor);
        super.processMouseEvent(e);
    }
} // HighlightLabel

```

A program elemzése

A megoldás rendkívül egyszerű: a címkén figyeljük a `MOUSE_ENTERED` és a `MOUSE_EXITED` eseményt, és velük összhangban állítjuk be a címke színét.

Végül gyártunk két olyan komponenst, amelyben magas szintűek a feldolgozandó események. Ezeknek az eseményeknek nincs eseményfeldolgozó metódusuk, a forrásobjektum tehát megfigyelője lesz magának.

Feladat – BlueTextField

Készítsen egy szövegmezőt, amely a benne levő szöveg hosszúságától függően elkéül – ha nincs benne szöveg, akkor fehér, ha a szöveg legalább 30 hosszú, akkor már sötétkék!

Forráskód

```
package extra.controls;

import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

public class BlueTextField extends JTextField implements
                               DocumentListener {
    public BlueTextField(int columns) {
        super(columns);
        this.getDocument().addDocumentListener(this);
    }

    void update() {
        int len = getText().length();
        int n = (255-len*5>100) ? 255-len*5:100 ;
        setBackground(new Color(n,n,255));
    }

    public void insertUpdate(DocumentEvent e) {
        update();
    }
    public void removeUpdate(DocumentEvent e) {
        update();
    }
    public void changedUpdate(DocumentEvent e) {
    }
} // BlueTextField
```

A program elemzése

Ebben a feladatban a forrásobjektum figyeli a maga dokumentummodelljét. Ha változott a szöveg, akkor kiszámítjuk a mező kékségét, és elvégezzük a színezést.

Feladat – JPanel

Készítsünk olyan JPanel osztályt, amely egy halványszürke, a konstruktörben megadott vízszintes és függőleges lépésközű rácsot tesz a panelre!

Forráskód

```
package extra.controls;

import javax.swing.*;
import java.awt.*;

public class JPanel extends JPanel {
    private int gridX, gridY;

    public JPanel(int gridX, int gridY) {
        this.gridx = gridX;
        this.gridy = gridY;
    }

    public void paintComponent(Graphics g) {
        g.setColor(new Color(220,220,220));
        for (int x=0; x<getWidth(); x+=gridX)
            g.drawLine(x,0,xgetHeight());
        for (int y=0; y<getHeight(); y+=gridY) {
            g.drawLine(0,y,getWidth(),y);
        }
    }
} // JPanel
```

Tesztkérdések

- 11.1. Mely állítások igazak? Jelölje be az összes igaz állítást!
 - a) Egy forrásobjektum hallgatóinak száma az objektum osztályától függ.
 - b) Egy forrásobjektum hallgatóinak száma az objektum osztályától függ.
 - c) Egy komponensen keletkezett alacsony szintű esemény feldolgozható a komponens processEvent metódusával.
 - d) A forrásobjektum osztályában egyetlen olyan metódus van, amellyel a komponensen keletkezett billentyűesemény feldolgozható, és ez a processKeyEvent.

- 11.2. Jelölje be az összes igaz állítást!
 - a) Egy forrásobjektumon csak egyfélle esemény keletkezhet.
 - b) A processEvent a figyelő eseménykezelő metódusa.
 - c) A processXXXEvent metódus felülírható.
 - d) A forrásobjektumon csak olyan esemény keletkezhet, amely ott engedélyezve van.

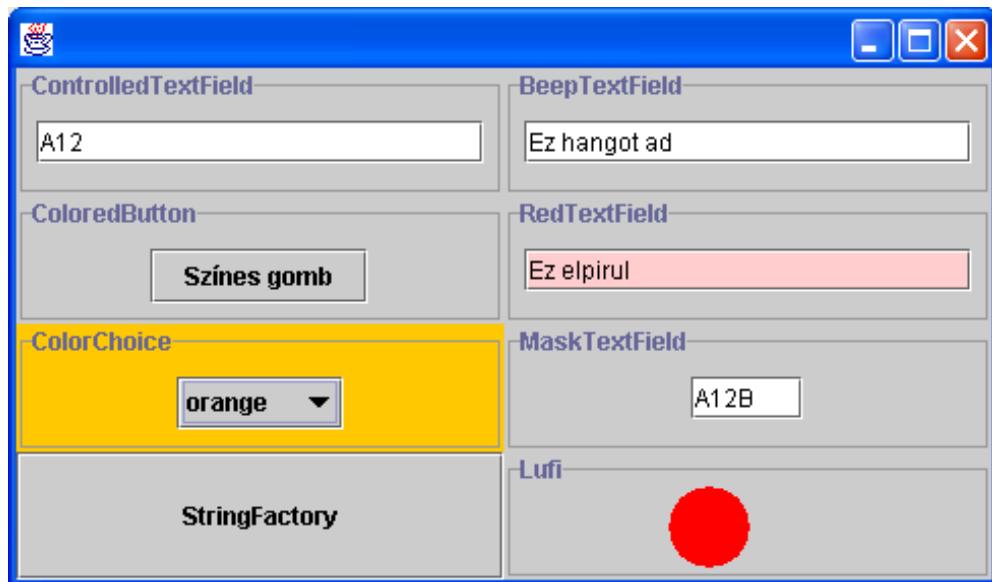
- 11.3. Ha egy forrásobjektumnak egyetlen MouseListener hallgatója van, akkor normális esetben milyen metódusok kapják meg majd egymás után az egér felengedése eseményt? Jelölje be az egyetlen jó választ (a sorrend is számít)!
 - a) processMouseEvent, processEvent, mouseReleased
 - b) mouseReleased, processEvent, processMouseEvent

- c) `processEvent`, `processMouseEvent`, `mouseExited`
- d) `processEvent`, `processMouseEvent`, `mouseReleased`

- 11.4. Hogyan lehet egy komponensen keletkező billentyűeseményt feldolgozni? Jelölje be az összes olyan választ, amelyben nem hiányos a feladatsor!
- a) Engedélyezzük a komponensen a billentyűeseményeket, és felülírjuk a komponens `processKeyEvent` metódusát.
 - b) Engedélyezzük a komponensen a billentyűeseményeket, és felülírjuk a komponens `processEvent` metódusát.
 - c) Az osztály fejében megadjuk az `implements KeyListener-t`, és a komponenst hozzáadjuk saját `KeyListener` hallgatóláncához.
 - d) Az osztály fejében megadjuk az `implements KeyListener-t`, megírjuk a `KeyListener` interfész három metódusát, és a komponenst hozzáadjuk saját `KeyListener` hallgatóláncához.

Feladatok

Az itt található feladatok osztályait az `extra.controls` csomagba tettük; tesztelőprogramjuk a `TestControls.java`.



- 11.1. (A) Készítsünk ellenőrzött szövegmezőt! A mező tartalma akkor minősüljön helyesnek, ha az első karakter 'A' vagy 'B'. A felhasználó addig ne hagyhassa el a mező fókuszát, ameddig helyesen ki nem tölti a mezőt. Ha helytelen a kitöltés, akkor a program adjon hangjelzést és állítsa a kurzort az első karakterre! (`ControlledTextField.java`)
- 11.2. (A) Készítsen olyan szövegmezőt, amely minden billentyűleütéskor sípol egyet! (`BeepTextField.java`)

11.3. (A) Készítsen olyan `ColoredButton` nyomógombot, amely színeződjön el, ha nyomják és válton vissza eredeti színére, ha felengedik! Konstruktorában a címke szövegén kívül lehessen a színt is megadni! (`ColoredButton.java`)

11.4. (B) Készítsen olyan szövegmezőt, amely a kurzor pozíciójától függően fokozatosan elpirul – ha a kurzor az első karakteren áll, akkor a mező fehér, ha a kurzor az 50. karakteren áll, akkor már sötétpiros! (`RedTextField.java`)

11.5. (B) Készítsen olyan `ColorChoice` osztályt, amellyel választani lehet a benne tárolt színekből! A színekből szövegesen lehessen választani, a kiválasztott szín `Color` típusú legyen! (`ColorChoice.java`)

A feladat továbbfejlesztése: a színeket kívülről, az `addItem(String, Color)` metódussal lehessen az objektumba beledobálni.

11.6. (C) Készítsen egy adott hosszúságú, maszkolható szövegmező-komponenst! A maszk megadja, hogy melyik karakterhelyen milyen karakter vihető be. A maszk-karakterek jelentése:

- A: Betűk: A-tól Z-ig
- 9: Számok: 0-tól 9-ig
- X: Bármilyen karakter

Minden más maszk-karakter azt jelenti, hogy azon a helyen csak ez a megadott karakter állhat. Például: "AAA-999" egy rendszám: 3 betű, kötőjel, azután három szám. A mező felülíró üzemmódban működjön, és ne engedjük meg a törlést! (`MaskTextField.java`)

11.7. (C) Készítsen egy rendezett szövegeket összeállító dialógust! A dialógust a `showDialog` statikus metódus jelenítse meg, visszatérési értéke egy szövegeket tartalmazó vektor legyen. A dialógusban a szövegeket interaktív módon lehessen szerkeszteni: Enter: szöveg módosítása; Delete: szöveg törlése; Ins: új szöveg beszúrása. A dialógus a hívó komponens a tulajdonosi hierarchiában legfelül álló keret közepére kerüljön. Hívása például:

```
Vector strings = StringFactory.showDialog(this, "Szövegek");  
(StringFactory.java)
```

11.8. (B) Készítsen egy felfűható lufit! A lufi egy megadott nagyságú színes, kitöltött kör legyen; a középpontjával és színével lehessen inicializálni. Kezdetben a lufi sugara legyen 10 pont. A lufit lehessen felfűjni: ha az egérrel rákattintunk, akkor sugara minden 5 ponttal nagyobb legyen. Ha a sugár elérte a 100 pontot, akkor a lufi essen össze eredeti méretére. A lufit lehessen egérrel vonszolni a képernyőn! (`Lufi.java`)

12. Applet

A fejezet pontjai:

1. Mi az applet?
 2. Applet-futtató környezetek
 3. Az Applet és a JApplet osztály
 4. Az alkalmazás átalakítása appletté
 5. Hanglejtészás – AudioClip
 6. Az applet életciklusa
 7. Az applet paraméterei
 8. Biztonság
-

Az applet olyan Java program, amely HTML-oldalba van beágyazva. Az applet tehát ügyfélalkalmazás, és kapcsolatba is léphet különböző kiszolgálóprogramokkal. A Java programnyelv fejlesztői legfőképpen olyan környezetet szerettek volna létrehozni, amellyel távoli gépről is egyszerűen lehet programot futtatni. Azt gondolták, hogy ezzel megszabadíthatják a felhasználót az alkalmazások helyi gépre való telepítésének terhétől, mert a felhasználó majd távolról is használhatja megszokott programjait. Ez az álom sajnos nem vált egészen valóra, de bizonyos helyzetekben mégis érdemes appleteket készíteni.

Mivel az applet is Java program, azért **elkészítéséhez felhasználható az összes eddig megszerzett ismeret**. Ez a fejezet az appletkészítés alapjait tárgyalja: az Olvasó megtanulhatja belőle, hogyan kell az eddigiekhez hasonló, de távolról is futtatható programokat készíteni. Ma, az Internet világában a böngészőben futó programok nem csupán szórakoztatásra valók, s létrehozásuk sem csak szórakozás (bár annak sem utolsó éppen), hiszen komolyabb cégek is alkalmazzák oldalaikon az appletkészítés technikáját.

A fejezet appletjei „élőben” is futtathatók a <http://4kor.hu/applets> oldalról.

12.1. Mi az applet?

Az **applet** olyan Java program, amely egy HTML-oldalba (weboldalba) van beágyazva. Az applet forráskódja hasonlít az alkalmazásokéhoz, és az appletnek is van egy fő osztálya; de:

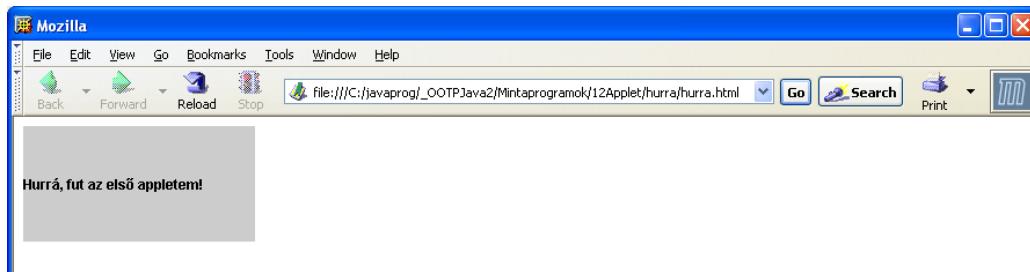
- a Java alkalmazás (application) önállóan fut; a futás a fő osztály szabványos `main` metódusának meghívásával indul, és a publikus osztály ősosztálya bármi lehet.
- **a Java applet viszont csak egy másik alkalmazásba** (például a böngészőbe vagy az appletnézőbe) **ágazva futhat**; fő osztálya a `java.applet.Applet` osztály egy lezármazottja. Az applet osztályában hiába volna `main` matódus, mert a JVM azt nem hívja meg. Az applet objektumot a rendszer hozza létre: lefuttatja az applet osztályának paraméter nélküli konstruktőrét, majd utolsó utasításként meghívja az `init` metódust. Az appletbeli inicializálást az `init` metódusba szokás betenni.

Egy HTML-lapba akárhány applet beágyazható.

Az appletbe biztonsági okokból korlátozások vannak beleépítve. Az applet futási környezetét **homokozónak** (sandbox) szokták nevezni.

Feladat

Írunk olyan appletet, amely megjeleníti a "Hurrá, fut az első appletem!" szöveget! Jelenítsük meg az appletet a böngészőben!



Hozza létre a következő Java forráskódot, például a `_MyPrograms/hurra` könyvtárban:

Forráskód

```
import javax.swing.*;

public class Hurra extends JApplet { //1
    public void init() { //2
        getContentPane().add( //3
            new JLabel("Hurrá, fut az első appletem!"));
    }
}
```

A forráskód készen megtalálható a fejezet mintaprogramjai között is (`hurra/Hurra.java`).

A forráskód elemzése

- ◆ //1: A fő osztály a `javax.swing.JApplet`-ből származik. A `JApplet` közvetlen őse a `java.applet.Applet`; abba van beleprogramozva az applet képességeinek a zöme.
- ◆ //2: Az inicializáló tevékenységek az `init` metódusban vannak.
- ◆ //3: Az appletnek is van tartalompanelje, abba tesszük a szöveget megjelenítő címkét.

Fordítsa le az osztályt (a fordítási egységet) akár parancssorból, akár a javaprojektből: *Make "Hurra.java"*. Tegye a lefordított class állományt a hurra könyvtárba!

Megjegyzés: Ha javaproggal fordított, akkor a class állományt a c:/javaproj/javaproj_classes könyvtárban találja meg.

HTML-állomány

Ezután készítsen a hurra könyvtárban (ott, ahol a class állomány van) egy HTML-állományt, s abba írjon egy `<applet>` HTML-címkepárt! Az `<applet>` címke (parancs) mondja meg a böngészőnek, hogy melyik class állományt kell betöltenie és futtatnia, és azt is, hogy mekkora legyen a lapon az applet által elfoglalt terület. A HTML-állomány neve elvileg bármí lehet, de az a szokás, hogy vagy az applet fő osztályáról kapja a nevét (hurra.html), vagy az index.html nevet adjuk neki, és akkor automatikusan indulhat. Az Interneten közzétett könyvtárak és állományok nevét **ajánlatos kisbetűvel írni**; ezt az ajánlást ebben a könyvben is követjük. Megtehetjük, hogy a hurra/hurra.html állományba az `<applet>` címkekben kívül más nem teszünk (minden egyebet „kispórolunk”); ekkor ez lesz a tartalma:

```
<applet code="Hurra.class" width="200" height="100">
</applet>
```

Eszerint az applet kódja a `Hurra.class`; és ez a kód a HTML-állomány mellett van (a HTML-állománytól indul a `code` relatív útvonala); az applet szélessége (`width`) 200 pixel, magassága (`height`) pedig 100 pixel.

Futtatás

Futtassa a `hurra/hurra.html` állományt! Kattintson kettőt az állományon vagy nyissa meg az állományt a böngészőben! Ha a futtatás sikeres, akkor az applet megjelenik a lapon. A böngészőbe általában bele van építve az applet futtatásának a képessége: csatlakoztatva vannak a Java futtatókörnyezethez, a JRE-hez. Ha az applet mégsem jelenik meg a böngészőben, akkor lehet, hogy

- ◆ hiba csúszott a HTML-állományba vagy az appletbe;
- ◆ a böngésző valami miatt nem kommunikál helyesen a Java futtatókörnyezettel;
- ◆ a futtatókörnyezet túl régi az applethez képest.

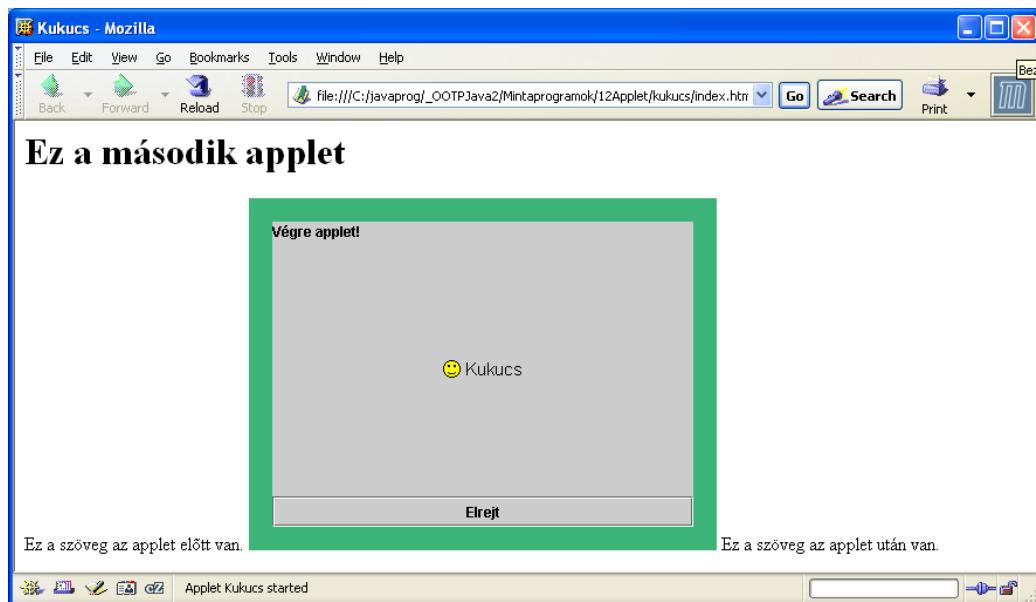
Megjegyzések:

- Feltesszük, hogy Ön alapszinten ismeri a HTML-lap szerkezetét és a böngésző használatát.
- Az `<applet>` címke elvileg elavult; helyette az `<object>` címkét kell használni. Az `<object>` címke azonban még nem mindenhol működik.
- Az applet a weblapnak csak egy eleme. Az appletet körülvevő részt is gondosan meg kell tervezni!
- A JBuilderben csak a `main` metódust tartalmazó osztályt lehet önállóan futtatni. Mivel az applet nem tartalmaz `main` metódust, azért csak a teljes projektet futthatjuk (a projekt-fán nincs futtatási lehetőség).

Bonyolítsuk egy kicsit a programot: tegyünk az appletbe komponenseket és eseményvezérlést!

Feladat – Kukucs

Készítsünk olyan appletet, amely gombnyomásra elrejt, illetve megmutat egy kis ikont és a hozzá tartozó szöveget! Az applet tetejére írjuk ki, hogy "Végre applet"!



Applet-projekt létrehozása

Hozzunk létre egy `kukucs.jpx` projektet! A projekt útvonalait hagyjuk meg alapértelmezésűnek (path: `src`, `classes` stb.)! Ezután hozzunk létre egy új osztályt (*File/New/Class*): *Class name*: `Kukucs`; *package*: hagyjuk üresen; *Base class*: `javax.swing.JApplet`. Írjuk be az alábbi kódot!

A JBuilderben használhatjuk az appletvarázslót is (*File/New/Web/Applet*); ez előállít egy forrásállományt (az applet vázát), és készít hozzá egy alapértelmezés szerinti HTML-tesztállományt is. Ezt a lehetőséget természetesen nem kötelező használni.

Forráskód

```

import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class Kukucs extends JApplet implements ActionListener {
    JPanel cp = (JPanel) getContentPane();
    JLabel lbKukucs;
    JButton btMutat = new JButton("Elrejt");
    boolean mutat = true;
}

```

```
public void init() {
    cp.setBorder(BorderFactory.createLineBorder(
        new Color(60,180,120),20));
    cp.add(new JLabel("Végre Applet!"), "North");
    Image img = getImage(getCodeBase(), "asit.gif");           //1
    ImageIcon iiKep = new ImageIcon(img);
    lbKukucs = new JLabel("Kukucs",iiKep,JLabel.CENTER);
    lbKukucs.setFont(new Font("Dialog",Font.PLAIN,15));
    cp.add(lbKukucs);
    cp.add(btMutat,"South");
    btMutat.addActionListener(this);
}

public void actionPerformed(ActionEvent ev) {
    mutat = !mutat;
    lbKukucs.setVisible(mutat);
    btMutat.setText(mutat?"Elrejt":"Mutat");
}
}
```

A forráskód elemzése

A vezérlők bepakolása a tartalompanelbe és az eseményvezérlés pontosan úgy megy, mint az alkalmazások körében. Ebben a feladatban egyetlen újdonság van: a `getCodeBase()` függvény (//1). Appletben csak a betöltött kód vagy HTML-állomány könyvtárhoz képest adhatunk meg állományspecifikációt. Ezeket a könyvtárakat mutatja meg a `getCodeBase`, illetve a `getDocumentBase` függvény.

Fordítsa le a projektet! A `Kukucs.class` állomány a projekt `classes` könyvtárába kerül.

HTML-kód (`kukucs/index.html`)

A HTML-kódba most beletesszük a szokásos részeket is (fej, törzs). A lapnak címet adunk: `<title>`; az applet fölé egy címsort (headert) teszünk: `<h1>`; és az applet előtt és után kiírunk egy-egy szöveget. Az `<applet>` címkében a `codebase="classes"` azt mondja, hogy a code-beli állományspecifikáció, a `Kukucs.class` fő osztály a HTML állomány alatti `classes` könyvtárban van:

```
<html>
<head>
    <title>Kukucs</title>
</head>
<body>
    <h1>Ez a második applet</h1>
    Ez a szöveg az applet előtt van.
    <applet codebase="classes" code="Kukucs.class"
        width="400" height="300">
    </applet>
    Ez a szöveg az applet után van.
</body>
</html>
```

Jelenítse meg a böngészőben a `kukucs/index.html` állományt!

12.2. Applet-futtató környezetek

Böngésző, HTML-kód

Az applet böngészőben fut, ezért HTML-állományba kell beleágyazni. Ha a böngésző a HTML-állomány betöltésekor szabályos <applet> címkepárt talál, akkor az ott megadott adatok alapján a JVM meghívásával létrehozza az applet objektumot, majd betölти a böngészőbe és futtatja. Az applet az Internet bármely pontjáról letölthető.

Az <applet> címkepár szerkezete a következő:

```
<applet
    codebase = "URL"                                // kódbázis
    archive  = "jarfilename"                          // JAR állomány
    code     = "classfilespec"                        // fő osztály
    width   = "400"                                  // applet szélessége
    height  = "300"                                  // applet magassága
    hspace  = "0"                                    // vízszintes margó az applet mellett
    vspace  = "0"                                    // függőleges margó az applet felett
    align   = "justification"                      // applet igazítása a lapon
    alt     = "text"
  >
  <param name="paraméternév" value="paraméterérték"/>
  <param name="paraméternév" value="paraméterérték"/>
  ...
</applet>
```

Az applet-paraméterek jelentése:

- **codebase** (kódbázis): A kód relatív könyvtára. Lehet abszolút vagy relatív útvonal. A relatív útvonal a betöltött HTML-lap könyvtárától indul. Ha nem adjuk meg, akkor ez a HTML-állomány könyvtára.
- **archive**: A JAR-állomány specifikációja (a codebase-hez viszonyítva); kiterjesztése jar. Ez tartalmazza a teljes appletet, összecsomagolva. Ha nem adjuk meg, akkor a JVM kibontott class állományokat keres.
- **code**: A fő osztály specifikációja (a codebase-hez viszonyítva); kiterjesztése class. Itt adjuk meg a böngészőnek, illetve a JVM-nek, hogy ez a fő osztály. Az itt megadott osztály az Applet –, illetve a JApplet – osztály utódja kell, hogy legyen. Ezt a paramétert kötelező megadni.
- **width, height**: Az applet szélessége és magassága képernyőpontban (pixelben). Ilyen méretben jelenik meg az applet a böngészőben.
- **align**: Az applet igazítása a lapon: left, right, top, texttop, middle, absmiddle, baseline, bottom vagy absbottom.
- **alt**: Ha az applet nem tud futni, akkor ez a szöveg jelenik meg helyette a lapon.

A JAR állomány használatával azt érjük el, hogy a belecsomagolt állományok (class fájlok és erőforrások) a weblappal együtt egyszerre letöltődnek; így a letöltés összességében kevesebb időt vesz igénybe, és nincs szükség további internetkapcsolatra.

Példák:

1.: codebase a HTML-lap könyvtára. Ebben van a `Hurra.class` fő osztály:

```
<applet code="Hurra.class" width="200" height="100">
</applet>
```

2.: codebase a HTML-lap könyvtára alatti `classes` könyvtár. Ebben van a `ViewPictures.class`:

```
<applet codebase="classes" code="ViewPictures.class"
width="1000" height="600">
</applet>
```

3.: codebase a HTML-állomány mappája, ott van a `Mikro.jar` archive állomány. A fő osztály állománya a `Mikro.jar`-ba csomagolt `MikroVezerlo.class`:

```
<applet archive="Mikro.jar" code="MikroVezerlo.class"
width=600 height=400></applet>
```

A JRE és a Java Plug-In

Az első appletet az 1990-es évek elején fejlesztették; akkor még csak a Sun HotJava böngészőjében lehetett őket futtatni. Az appletek akkor lettek igazán népszerűek, amikor a Netscape, majd nem sokkal utána a Microsoft is beépítette böngészőjébe a Java virtuális gépet (a JVM-et) és vele együtt az API osztályait – s ezzel a weblapok „életre keltek”. Sajnos a valóság kissé letörte a kezdeti álmokat: a nagy böngészőgyártók nem igazán követték a Sun Java-fejlesztését. A probléma megoldására a Sun kifejlesztett egy rugalmasabb módszert: a Java Plug-In összekapcsolja a böngészőt a külső Java futtatókörnyezettel. Sajnos az appletek igen nagy hányada még így sem fut a böngészőkben.

A Java futtatókörnyezetre (JRE, Java Runtime Environment) a legtöbb felhasználónak szüksége van, ha önálló Java alkalmazást vagy – böngészőben – appletet szeretne futtatni. A JRE használata ingyenes, és letölthető a <http://java.com> címről (Download). A JRE minden platformra elkészült, hardverkövetelményei: 166 MHz-es Pentium, 125 MB szabad lemezterület, 32 MB RAM.

A Java appletek futtatásához a böngészőnek el kell érnie a Java futtatási környezetet (a JRE-t). Több eset lehetséges:

- A böngészőbe bele van építve a JRE. Ez a megoldás azért nem jó, mert a böngészők nem követik a változásokat és nem egységesek.
- A böngészőbe bele van építve egy Java bedolgozó (Java Plug-In), s az kapcsolatot teremt egy külső JRE-vel. Ezzel a böngésző függetlenné válik a JRE-től. A felhasználónak egyébként nem kell foglalkoznia a JRE letöltésével és a telepítésével, mert a bö-

gésző és a látogatott lapok ezt szükség esetén elvégzik helyette. Ennek a megoldásnak az a hibája, hogy újabb és újabb JRE telepítésére „kényszeríti” a felhasználót.

- A Java Web Start egy Internet alapú alkalmazásindító: ez szükségtelenné teszi a Java Plug-In használatát.

A Plug-In (bedolgozó) szoftver kiterjeszti a böngésző képességeit. A már telepített böngészőt így különleges szoftverekkel egészíthetjük ki. Néhány ismert Plug-In:

- ◆ Apple QuickTime: Zeneszámokat vagy videofilmeket játszik le;
- ◆ Adobe Acrobat: PDF-állományt olvas;
- ◆ Macromedia Flash Player: A Macromedia Flash fejlesztőeszközzel elkészített állományokat (animációkat és vektorgrafikát) jelenít meg (webes szabvány);
- ◆ RealPlayer by RealNetworks: Lejátssza az audio, videó, animáció és multimédia állományokat;
- ◆ Sun Java Plug-In: Kapcsolatot teremt a böngésző és a Java platform között.

A JRE tartalmazza a JVM-et és a Java Plug-Int; ezeket nem kell, és nem is lehet külön telepíteni. A Plug-Int a böngészőben engedélyezni kell!

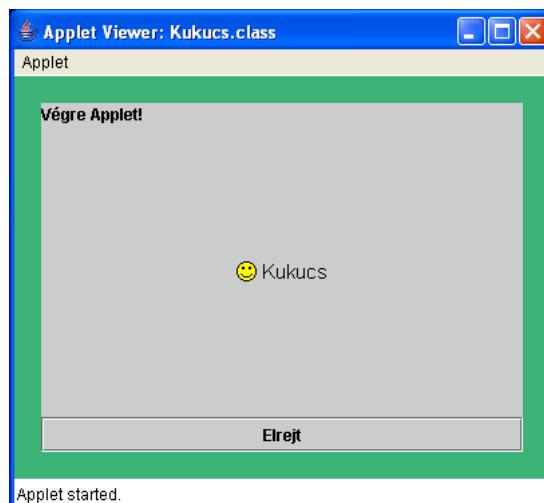
Appletnéző (appletviewer)

Az appletnéző a Java fejlesztőkörnyezet (JDK) része; futtatáskor a paraméterként megkapott HTML-állományból kiolvassa az <applet> címkét, és megjeleníti az appletet a HTML-lap nélkül. Hívása:

```
%JAVA_HOME%\bin\appletviewer index.html
```

Ha a JBuilderben appletet futtatunk (a futtatási konfigurációban appletet adunk meg), akkor is az appletnéző hívódik meg.

Az appletnéző paramétere egy HTML-állomány. Ha abban a fő osztály a Kukucs.class, akkor ez jelenik meg:



Futtatás JBuilderben

Ha az appletet JBuilderben futtatjuk, akkor a futtatási konfigurációban a futási típus (*Run/Configurations/Run Type*) most Applet lesz, és meg kell adni a fő osztályt (Main class) vagy a weblapot (HTML file).

Érdemes minden applethez külön projektet készíteni. A javaprof használatával most nem nyerünk semmit, mert csak a teljes appletet lehet futtatni – külön az applet osztályát nem (a projektfán nincs Run lehetőség).

A böngészőbe való betöltés előtt ajánlatos az appletnézővel megnéznünk az appletet. Az appletnéző előnyei: ha nincs kéznél böngésző, ez talán kéznél van; gyorsabb a megjelenítés; és biztosan a legfrissebb változatot mutatja. Hátránya, hogy nem pontosan úgy működik, mint a böngésző. De ez nem is olyan meglepő, hiszen böngészőből sincs két egyforma. **Éles futás előtt mindenképpen több, különböző böngészőben is ki kell próbálni az appleteket.**

Megjegyzés: Előfordulhat hogy a böngészőben nem a várt, legutoljára módosított applet fut. Ennek az lehet az oka, hogy a gyorsítótárban még ott „ül” az előző applet kódja, és a böngésző mindenkorán azt futtatja, merő „spórolásból”. Ilyenkor kényszeríteni kell a böngészőt, hogy ne a régi lapot, illetve programot futtassa, hanem a frissen letöltöttet:

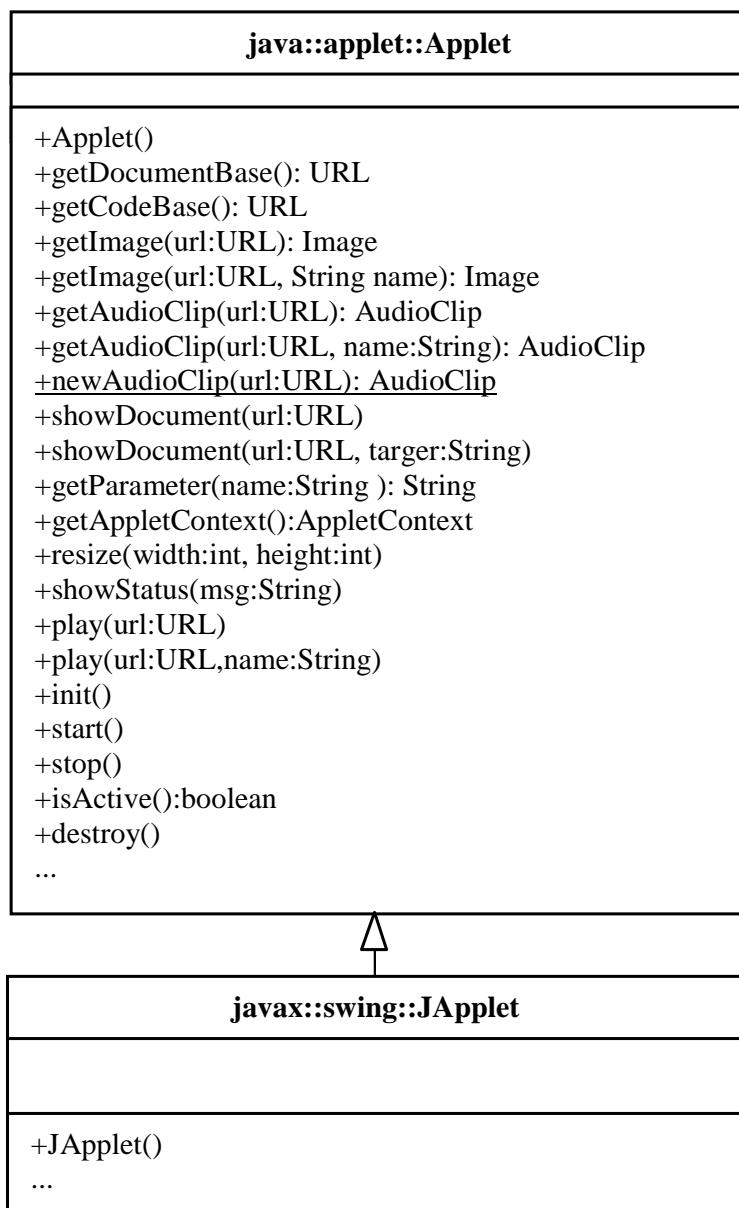
- Frissítse a lapot!
- Indítsa újra a böngészőt!
- Törölje a gyorsítótárat!

Mikor érdemes appletet írni?

Mielőtt appletet írnánk, gondoljuk végig: ha a feladat egyszerűen megoldható HTML-úrlapokkal, JavaScript kóddal és animált képekkel, akkor ne tegyünk felesleges terhet a felhasználókra és magunkra: használjuk inkább ezeket a lehetőségeket! Bonyolultabb programot azonban esetleg már nagyon nehézkes lehet ezekre a technikákra alapozni. A Java nyelv lehetőséget ad arra, hogy tiszta szerkezetű, objektumorientált kódot állítsunk elő. Választhatunk: applet legyen vagy alkalmazás? Ha nincs szükség internetkapcsolatra és a programot nem a nagyközönségnek készítjük, akkor írunk Java alkalmazást; abban is használhatunk távoli elérést. Az applet nagyon jó eszköz, ha az ügyfél (kliens) oldalon interaktív programot akarunk írni webes környezetben, oktató vagy játék céllal.

Appletet elsősorban akkor érdemes használni, ha a program közepes bonyolultságú, és nem szeretnénk az ügyféloldalon telepítési nehézségekkel küszködni.

12.3. Az Applet és a JApplet osztály



12.1. ábra. Az Applet és a JApplet osztály

Az Applet osztály

Csomag: `java.applet` Deklaráció: `public class Applet`
 Közvetlen ős: `java.awt.Panel`

Az **Applet** osztály tartalmazza az applet készítéséhez szükséges legfontosabb tulajdonságokat és metódusokat. Az applet olyan, általában kisebb méretű Java program, amely csak egy másik alkalmazásba (böngészőbe vagy appletnézőbe) beleágyazva futhat. Az applet osztálya minden leszármazottja a `java.applet.Applet` osztálynak. Az appletet a beágyazó program hozza létre, lefuttatja az applet osztályának paraméter nélküli konstruktörét, majd a konstruktur utolsó utasításaként az `init` metódust. Az applet nem ablak: nincs kerete, címe, ikonja, s nem lehet ablakesemény forrása. Az `Applet` osztálynak a `Panel` a közvetlen őse, a `JApplet` pedig közvetlen leszármazottja:

```
java.lang.Object
  +-- java.awt.Component
    |   +-- java.awt.Container
    |   |   +--java.awt.Panel
    |   |   |   +--java.applet.Applet
    |   |   |   |   +--javax.swing.JApplet
```

Az appletnek kell, hogy legyen egy publikus, paraméter nélküli konstruktora; a JVM ezt hívja meg az applet létrehozásakor. Az applethez általában nem írunk konstruktort, s ez esetben az alapértelmezés szerinti konstruktur hívódik meg, az pedig meghívja az `init` metódust. Elegendő tehát az `init` metódust felülírnunk.

Konstruktur

► `public Applet()`

Konstruktur. Létrehoz egy új applet objektumot, és meghívja az `init` metódust. A beágyazó alkalmazás hívja meg.

Metódusok

► `init()`

A böngésző, illetve az appletnéző hívja meg, amikor az applet betöltődik az oldalra. A `start` metódus előtt mindenki meghívódik. Felül lehet – és szokás – írni. Az `Applet` osztály `init` metódusa üres.

► `URL getDocumentBase()`

Visszaadja annak a HTML-dokumentumnak az URL-jét, amelybe ez az applet bele vagy ágyazva.

► `URL getCodeBase()`

Visszaadja a kódbázist: annak a könyvtárnak az URL-jét, amelyben az applet kódja van.

► `void resize(int width, int height)`

Átméretezi az appletet.

► `void showStatus(String msg)`

Kiírja `msg`-t a böngésző státuszsorába.

- ▶ `Image getImage(URL url)`
- ▶ `Image getImage(URL url, String name)`
Betölti az URL (és az állománynév) által meghatározott képállományt.
- ▶ `AudioClip getAudioClip(URL url)`
- ▶ `AudioClip getAudioClip(URL url, String name)`
- ▶ `static AudioClip newAudioClip(URL url)`
Betölti az URL (és az állománynév) által meghatározott hangállományt.
- ▶ `void play(URL url)`
- ▶ `void play(URL url, String name)`
Lejátssza a megadott helyen található hangállományt.
- ▶ `void showDocument(URL url)`
- ▶ `void showDocument(URL url, String target)`
Betölti a megadott lapot. target: "`_self`", "`_parent`", "`_top`", vagy "`_blank`"

A többi metódust később fejtjük ki.

A JApplet osztály

Csomag: `javax.swing` Deklaráció: `public class JApplet`
 Közvetlen ős: `java.awt.Applet`

A JApplet minden össze néhány képességgel egészült ki az Applethez képest. A JAppletnek már van menüsora, gyökérpanele, üvegpanele stb. A JFrame-hez hasonlóan csak a tartalompaneljébe (`contentPane`) lehet elemeket tenni.

A JApplet együttműködik a Swing komponensarchitektúrával. Ha az appletbe Swing-komponenst is teszünk, akkor feltétlenül a JAppletből származtatunk, másképp esetleg helytelenül fog működni.

A könyv appletei a JAppletből származnak.

Konstruktur

- ▶ `public JApplet()`
Konstruktur. Létrehoz egy új applet objektumot. A beágyazó alkalmazás hívja meg.

12.4. Az alkalmazás átalakítása appletté

Egy grafikus felhasználói interfészű alkalmazás osztálya a Frame osztály utódja, az applet osztálya pedig az Applet osztályé. Az alkalmazás appletté alakításakor figyelembe kell vennünk, hogy az applet nem ablak: nem alkalmazhatók benne az ablak és a keret metódusai. Az appletben nincs például `title`, `iconImage`, `resizable` és `state` tulajdonság sem, és nincsenek ablakesemények. Nem használhatók a következő – a keretben megszokott – metódusok: `pack()`, `show()`, `setDefaultCloseOperation(...)` és `addWindowListener(...)`.

Az „appletesítés” folyamatát egy példán keresztül mutatjuk be: Írjuk át a 9., *Grafika, képek fejezet végén* levő feladatok közül a `ForgoEgyenes.jpx` alkalmazást appletté!

Rajzlap.java (változatlanul hagyjuk):

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Rajzlap extends JPanel implements ActionListener {
    //...
}
```

ForgoEgyenes.java

Az eredeti kódot meghagytuk, de a szükségtelenné vált kódrészleteket megjegyzéssé fokoztuk le; az újonnan beszúrt kódrészleteket itt, a nyomtatásban kiemeltük.

```
import javax.swing.*;

public class ForgoEgyenes extends /*JFrame*/ JApplet {
    public /*ForgoEgyenes()*/void init() {
        //setBounds(100,100,300,300);
        //setDefaultCloseOperation(EXIT_ON_CLOSE);
        getContentPane().add(new Rajzlap());
        //show();
    }

    //public static void main (String args[]) {
    //    new ForgoEgyenes();
    //}
}
```

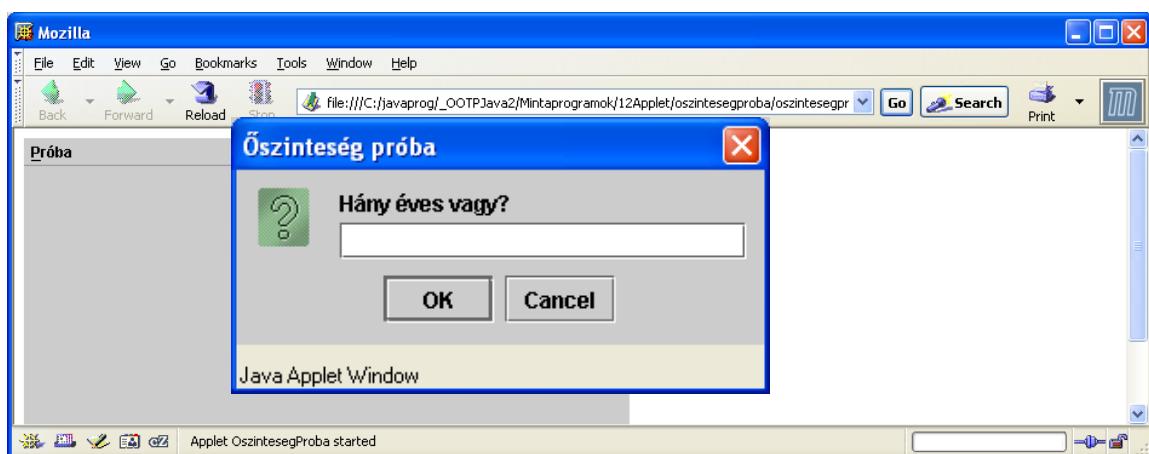
A program elemzése

A `Rajzlap` osztály tökéletes, azt nem módosítjuk. A fő osztály azonban nem jó: most nem önállóan futó keretet készítünk, hanem egy beágyazott appletet. A következő módosításokat kell elvégeznünk a `ForgoEgyenes` fő osztályban:

- ◆ Az ősosztályt kicséréljük `JFrame`-ről `JApplet`re! A böngésző ebből az osztályból hozza létre majd a beágyazott objektumot.
- ◆ Kitöröljük a `main` metódust! Nem okozna bajt, ha itt maradna, de nem hívódna meg.
- ◆ A konstruktur törzsét meghagyjuk, de a fejét átkereszteljük `public void init()`-re!
- ◆ Kitöröljük az `init` metódusból a következő utasításokat:
 - `setBounds(100,100,300,300);` Az applet méretét a HTML-lapon adjuk meg. Nem okozna hibát, de nem volna hatása.
 - `setDefaultCloseOperation(EXIT_ON_CLOSE);` Az applet nem keret, ezért ez szintaktikailag hibás utasítás.
 - `show();` Az applet nem ablak, ezért ez hibás utasítás (van ugyan `Component.show`, de az elavult).

Keretek, ablakok az appletben

Létrehozhatunk az appletből keretet, ablakot vagy dialógust – szülőablaknak ilyenkor a null objektumot kell megadnunk. A new SpecDialog(this) helyett például most ezt írjuk: new SpecDialog(null). Használhatók a JOptionPane dialógusai és a JColorChooser. showDialog is. Biztonsági okokból azonban a böngészőben megjelenő ablakon megjelenik egy figyelmeztető szöveg arról, hogy ez az ablak egy appleté. A következő kép az OszintesegProba.jpx projekt képe; éppen egy dialógust látunk megnyílni rajta:



12.5. Hanglejátszás – AudioClip

Az applet csomag `AudioClip` interfésze lehetőséget ad a hanglejátszássra. Hangot persze nem csak appletben lehet lejátszani; az alkalmazások is támaszkodhatnak erre az interfészre. Ebben a pontban minden össze azt mutatjuk meg, hogyan lehet egy hangállományt lejátszani. A lejátszás igen egyszerű, és sokszor nagyon hatásossá teszi a programot. A teljes hangfeldolgozás a `javax.sound` csomagban található, de azzal most nem foglalkozunk.

A Java a következő hangformátumokat kezeli:

- Audioformátumok: AIFF, AU és WAV
- MIDI (Musical Instrument Digital Interface) formátumok: MID, MIDI és RMF

A hangot az `AudioClip` interfész valamely implementációja játssza le. A lejátszás elindítható, újraindítható és megállítható. Az `AudioClip` példány saját szálón futhat, vagyis a program egyéb működése közben zenélhetünk is. Egy `AudioClip` példányt az `Applet` osztály `getAudioClip` vagy a statikus `newAudioClip` metódusával hozhatunk létre és tölthetünk be egy saját vagy távoli gépen levő állományból.

A betöltő metódusok állományazonosító paramétere egy URL, például:

- ◆ Ha távoli állományról van szó, akkor "http://4kor.hu/applets/Zene.wav";
- ◆ Ha helyi állományról van szó, akkor "file:/// [útvonal]"

A következő utasítás például létrehoz egy, a Windows csöngötését magában foglaló hang objektumot (a hangot magát külön kell megszólaltatni):

```
AudioClip hang = Applet.newAudioClip(  
    new URL("file:///c:/windows/Media/ringout.wav"));
```

Ha érvénytelen címet adunk meg, az URL konstruktora `MalformedURLException` kivételt ejt.

AudioClip interfész

Csomag: `java.applet`

Deklaráció: `public interface AudioClip`

Metódusok

- `void loop()`
Folyamatosan játszza a hangállományt. Ha a lejátszás befejeződött, újrakezdi a lejátszást.
- `void play()`
Elkezdi a hang lejátszását.
- `void stop()`
Megállítja a lejátszást. Ezután a loop ott kezdi, ahol a lejátszás abbamaradt.

Megjegyzések:

- Az operációs rendszerek általában tartalmaznak beépített hangfelvétő–lejátszó programot. Annak a használatához az kell, hogy a számítógépen legyen mikrofonnal és meghajtószoftverrel ellátott hangkártya. A Windowst futtató számítógépeken ez a program a Sound Recorder (Hangrögzítő): *Programs/Accessories/Entertainment/Sound Recorder*. A könyv mellékletében az itt következő `AudioClipTest` mintaprogram könyvtárában talál egy `HangFelvetel.txt` állományt; az röviden leírja a Sound Recorder használatát.
- Számítógépen rengeteg hangállomány van; keresse meg őket!

Feladat – AudioClipTest

Játsszuk le a `c:/windows/Media` könyvtárban levő "ringout.wav" hangállományt, majd a könyv elektronikus mellékletének `sounds` könyvtárában levő `vau.wav` állományt is!

Megjegyzés: Futtassa a `javaproj` projekttel: ott az aktuális könyvtár a `javaproj`!

Forráskód

```

import java.applet.Applet;
import java.applet.AudioClip;
import java.io.*;
import java.net.URL;

public class AudioClipTest {
    public static void main(String[] args) {
        try {
            AudioClip hang;
            hang = Applet.newAudioClip(
                new URL("file:///c:/windows/Media/ringout.wav"));
            hang.play();

            // A hang relativ útjából abszolút útvonalat készítünk,
            // és azzal képezzük az URL-t:
            String fName =
                new File("sounds/vau.wav").getAbsolutePath();
            Applet.newAudioClip(new URL("file:///"+fName)).play();
        }
        catch (Exception ex) {
            System.out.println(ex);
        }
    }
}

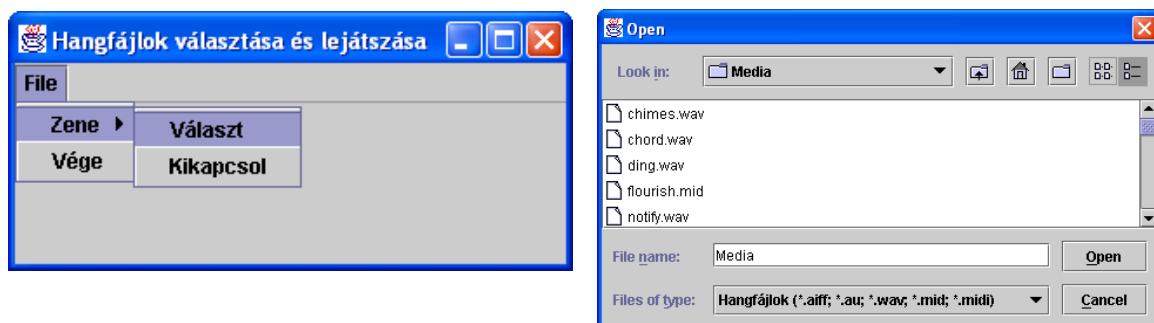
```

A program futása

Előbb lejátszódik a ringout, majd rögtön az ugatás. Ha a csengetés (ringout) nem volna ilyen rövid, akkor az ugatás (vau) kioltaná.

Feladat – Választ, játszik

Készítünk egy alkalmazást a lemezen levő audioállományok közötti válogatásra és a kiválasztott állomány lejátszására. A válogatást állománykiválasztó dialógussal végezzük; csak a hangállományok jelenjenek meg benne!



Megjegyzés: Ezúttal nem appletet írunk, hanem alkalmazást – a könyvtárbejegyzések applet-ből való olvasása meghaladja a könyv kereteit. A következő program, az *AudioShow* azonban már zenélő applet lesz.

Forráskód

Projekt: ValasztJatszik
Csomagok: -

AudioFilter.java

```
import javax.swing.filechooser.FileFilter;
import java.io.*;

public class AudioFilter extends FileFilter { //1

    public boolean accept(File f) {
        String fName = f.getName().toLowerCase();
        return f.isDirectory() ||
            fName.endsWith(".aiff") ||
            fName.endsWith(".au") ||
            fName.endsWith(".wav") ||
            fName.endsWith(".mid") ||
            fName.endsWith(".midi");
    }

    public String getDescription() {
        return "Hangfájlok (*.aiff; *.au; *.wav; *.mid; *.midi)";
    }
}
```

ValasztJatszik.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.net.URL;

public class ValasztJatszik extends JFrame
    implements ActionListener {
    private Container cp;
    private JMenuItem miValaszt, miKikapcsol, miVege; //2
    private AudioClip ac;
    private JFileChooser fc =
        new JFileChooser("c:/javaprogsounds");

    public ValasztJatszik() {
        setBounds(100,100,400,200);
        setTitle("Hangfájlok választása és lejátszása");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        cp = getContentPane();

        JMenuBar mb;
        JMenu mFile, mZene;
        setJMenuBar(mb=new JMenuBar());
        mb.add(mFile=new JMenu("File"));
        mFile.add(mZene = new JMenu("Zene"));
        mZene.add(miValaszt=new JMenuItem("Választ"));
        mZene.add(miKikapcsol=new JMenuItem("Kikapcsol"));
        mFile.add(miVege = new JMenuItem("Vége"));
```

```

        miVege.addActionListener(this);
        miValaszt.addActionListener(this);
        miKikapcsol.addActionListener(this);

        miKikapcsol.setEnabled(false);
        fc.setFileFilter(new AudioFilter());                                //3
        show();
    }

    void zeneValaszt() {
        if (fc.showOpenDialog(this) != fc.APPROVE_OPTION)
            return;
        try {
            URL url = new URL("file:///"+fc.getSelectedFile());      //4
            ac = Applet.newAudioClip(url);
            miKikapcsol.setEnabled(true);
            ac.play();
        }
        catch(Exception ex) {
            System.out.println(ex);
        }
    }

    public void actionPerformed(ActionEvent ev) {
        if (ev.getSource() == miValaszt) {
            zeneValaszt();
        }
        else if (ev.getSource() == miKikapcsol) {                         //5
            ac.stop();
            miKikapcsol.setEnabled(false);
        }
        else if (ev.getSource() == miVege)
            System.exit(0);
    }

    public static void main(String[] args) {
        new ValasztJatszik();
    }
}

```

A forráskód elemzése

- ◆ //1: Az állománydialógus szűrője. Csak a könyvtárakat és a hangállományokat lehet majd kiválasztani.
- ◆ //2: Deklarálunk egy AudioClip típusú változót, rajta keresztül hivatkozunk majd az aktuális hangra. Két metódusban is használni fogjuk: egyszer a lejátszás elindításakor, egyszer a lejátszás megállításakor.
- ◆ //3: Az állománydialógus szűrője az //1-ben megírt AudioFilter.
- ◆ //4: A kiválasztott állománnyal létrehozzuk az url objektumot. A newAudioClip segítségével betöljük az url-lel azonosított hangállományt, majd megkezdjük a lejátszását. Kezeljük az URL keltette esetleges kivételt.
- ◆ //5: A Kikapcsol menüponttal leállítjuk a hangot.

12.6. Az applet életciklusa

Ha a felhasználó elhagyja az appletet tartalmazó weboldalt (másik oldalra lép), minden meghívódik az applet `stop()` metódusa; és valahányszor visszatér erre a weboldalra, minden meghívódik a `start()` metódus. Ezzel elérhető, hogy az applet csak akkor működjön, amikor a felhasználó éppen „nézi”. A háttérben feleslegesen működő appletek csak „tékozolják” a processzor energiáját.

Életciklus metodusok (java.applet.Applet)

► `start()`

A böngésző, illetve az appletnéző hívja meg az `init` metódus után és minden alkalommal, valahányszor a felhasználó rááll az appletet tartalmazó lapra. Alapértelmezésben nem csinál semmit. Akkor szokás felülírni, ha visszatéréskor a futást újra akarjuk indítani.

► `stop()`

A böngésző, illetve az appletnéző hívja meg a `destroy` előtt, és minden alkalommal, valahányszor a felhasználó elhagyja az appletet tartalmazó lapot. Alapértelmezésben nem csinál semmit. Akkor szokás felülírni, ha a futást le akarjuk állítani a lapelhagyás idejére.

► `destroy()`

Megszünteti az appletet. A böngésző, illetve az appletnéző hívja meg, amikor becsukják. Előtte meghívódik a `stop` metódus.

► `boolean isActive()`

Megadja, hogy az applet éppen aktív-e. A `start` metódus meghívásával az applet aktív lesz, a `stop` meghívásával pedig inaktív.

Feladat – AudioShow

Az appletben szóljon a zene; egy gombbal lehessen ki- és bekapcsolni. Ha a felhasználó ellapoz az oldalról, akkor a zene átmenetileg némuljon el! Informáljuk a felhasználót arról, hogy éppen szól-e a zene.

Tegyünk az appletbe egy a "<http://sdp-city.hu>" oldalra elirányító gombot!

Forráskód

```
import java.awt.*;
import javax.swing.*;
import java.applet.*;
import java.awt.event.*;
import java.net.*;

public class AudioShow extends JApplet implements ActionListener {
    Container cp = getContentPane();
    JButton btSdp = new JButton("Go to SDP-city");
    JLabel lbInfo = new JLabel("", JLabel.CENTER);
    JButton btStart = new JButton("Start");
```

```
 JButton btStop = new JButton("Stop");
 AudioClip ac;
 boolean on = false;

 public void init() {
    ac = getAudioClip(getCodeBase(),"esobot.wav");
    btSdp.setBackground(Color.WHITE);
    lbInfo.setFont(new Font("Dialog",Font.PLAIN,15));
    cp.add(lbInfo);

    JPanel pnControl = new JPanel();
    cp.add(pnControl,"Center");
    pnControl.add(btStart);
    pnControl.add(btStop);
    pnControl.add(btSdp);
    btSdp.addActionListener(this);
    btStart.addActionListener(this);
    btStop.addActionListener(this);
}

void sound() {
    on = true;
    lbInfo.setText("Esik az eső");
    ac.loop();
}

void noSound() {
    on = false;
    lbInfo.setText("Süt a nap");
    ac.stop();
}

public void start() {
    if (on) ac.loop();
}

public void stop() {
    if (on) ac.stop();
}

public void actionPerformed(ActionEvent ae) {
    if (ae.getSource()==btStart)
        sound();
    else if (ae.getSource()==btStop)
        noSound();
    else if (ae.getSource()==btSdp) {
        try {
            getAppletContext().showDocument(
                new URL("http://sdp-city.hu"));
        }
        catch (MalformedURLException me) {
            showStatus(""+me);
        }
    }
}
```

⚠ Vigyázat! Az applet egészen addig fut, amíg be nem csukjuk a böngészőt. A számítógép erőforrásainak kímélésére a folyamatosan dolgozó (mozgó, számoló stb.) appletekhez illik stop és start metódust írni. Ezeknek pont az a dolguk, hogy amikor a lap éppen nem aktív (a felhasználó arrébb lapoz), akkor a programot ideiglenesen leállítsák, majd megint újraindításá.

12.7. Az applet paraméterei

Az appletnek a HTML lapon adhatunk meg paramétereket az <applet> címkepár közötti a <param> címkékben:

```
<applet>
    <param name="paraméternév" value="paraméterérték"/>
    <param name="paraméternév" value="paraméterérték"/>
    ...
</applet>
```

A paramétereket az Applet osztály getParameter metódusával kérhetjük el:

java.applet.Applet metódus

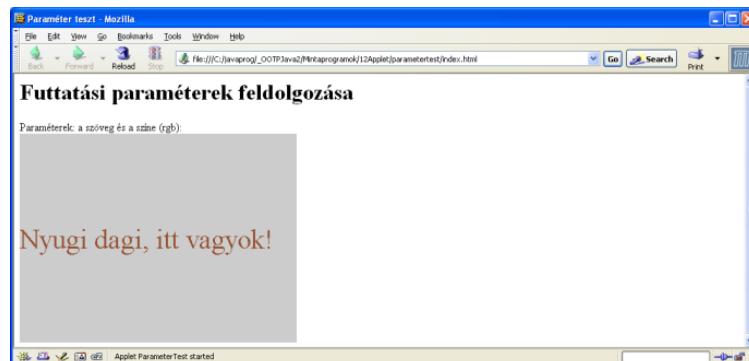
- ▶ String getParameter(String name)

Visszaadja a HTML <param> címkék közül a name nevű paraméter value értékét.

Például ha <param name="r" value="100"/>, akkor
getParameter("r") == "100"

Feladat – Paraméterteszt

Írunk ki egy szöveget; a szöveget és a szöveg RGB színeit az applet paramétere-ként adjuk meg!



Forráskód

```
import javax.swing.*;
import java.awt.*;
```

```

public class ParameterTest extends JApplet {
    int r, g, b;
    JLabel lbSzoveg = new JLabel();

    public void init() {
        try {
            // Futtatási paraméterek: szöveg, x, y
            lbSzoveg.setText(getParameter("szöveg"));
            r = Integer.parseInt(getParameter("r"));
            g = Integer.parseInt(getParameter("g"));
            b = Integer.parseInt(getParameter("b"));
        }
        catch (Exception e) {
            JOptionPane.showMessageDialog(null,
                "Adja meg a szöveg és a szín (rgb) paramétereit!");
        }
        lbSzoveg.setFont(new Font("Serif", Font.PLAIN, 40));
        lbSzoveg.setForeground(new Color(r,g,b));
        getContentPane().add(lbSzoveg);
    }
}

```

HTML-kód

```

<applet codebase="classes" code="ParameterTest.class"
        width="400" height="300">
<param name="szöveg" value="Nyugi dagi, itt vagyok!"/>
<param name="r" value="150"/>
<param name="g" value="80"/>
<param name="b" value="50"/>
</applet>

```

12.8. Biztonság

Az appleteket arra terveztek, hogy a böngészőben fussanak. Sok felhasználó nincs is tudatában annak, hogy böngészője engedélyezi a Java futtatókörnyezet beépülését és működését, vagyis az appletek futását. Normális esetben ha a felhasználó rááll egy appletet tartalmazó oldalra, akkor az applet automatikusan betöltődik és elkezd futni; a felhasználónak nincs is módja megállítani. Ezért nagyon fontos, hogy az applet még véletlenül se okozasson bajt a felhasználó gépének.

Az appletbe biztonsági korlátok vannak beépítve – éppen azért, mert az applet a lap betöltésével automatikusan fut a böngészőben. Hogy mit tehet egy applet és mit nem, az a böngészőbeállításoktól is függ.

Amit egy applet **mindig** megtehet:

- Betölthet és megjeleníthet képet;
- Betölthet és lejátszhat hangot;
- Fogadhat felhasználói eseményeket és adatokat.

Amit egy applet **sohasem** tehet meg:

- Nem futtathat külső programot;
- Nem kommunikálhat más géppel, csak azzal a gazdagéppel, amelyikről letöltődött;
- Nem írhat és olvashat állományokat a helyi gépen (ott, ahol fut);
- Nem olvashat ki adatot a helyi gépről (néhány alapadatot kivéve, mint pl. a JRE verziója). Nem tudja például kiolvasni a felhasználó személyes adatait.
- Csak úgy jeleníthet meg ablakot (keretet), hogy abban egy biztonsági figyelmeztetés is feltűnik.

A biztonsági korlátozásokat az teszi lehetővé, hogy az applet nem közvetlenül a CPU-n fut. Az appletet a JVM (Java virtuális gép) futtatja, s az egyszersmind szűrő is: megvizsgál minden kritikus utasítást, és ha az applet megsért egy biztonsági szabályt, akkor a biztonsági menedzser (security manager) `SecurityException` kivételt kelt.

Tesztkérdések

12.1. Mi igaz az appletre? Jelölje be az összes helyes választ!

- a) Az applet kis méretű alkalmazás.
- b) Az applet csak böngészőben futhat.
- c) Az applet fő osztálya az Applet utódja.
- d) Egy HTML-lapba csak egy appletet lehet beleágazni.

12.2. Az applet a következő módokon futtatható. Jelölje be az összes helyes választ!

- a) Az appletviewer programmal.
- b) Böngészőben.
- c) Egyszerre elindítható több példányban böngészőkben és appletnézőkben.
- d) Közvetlenül az operációs rendszeren (ebben tér el a Java alkalmazásoktól).

12.3. Mely állítások igazak? Jelölje be az összes helyes állítást!

- a) Az Applet osztály az Object közvetlen leszármazottja.
- b) Az appletben használt állományoknak abban a könyvtárban kell lenniük, ahol az appletet tartalmazó HTML-állomány van.
- c) Az aktuális könyvtárban levő "gag.wav" hangfájlt a következőképpen játszhatjuk le:
`AudioClip ac = new AudioClip("gag.wav");
ac.play();`
- d) Az AudioClip interfész az applet csomagban van deklarálva.

12.4. Jelölje be az összes helyes állítást!

- a) Az appletben kötelező konstruktort írni.
- b) Az appletben kötelező init metódust írni.
- c) Az appletben kötelező start metódust írni.
- d) Az appletet a beagyazó program hozza létre.

12.5. Jelölje be az összes helyes állítást!

- a) Az applet csak azzal a géppel kommunikálhat, amelyikről letöltődött.
- b) Az applet olvashat az ügyfélgépen levő állományokból.
- c) Az applet képet is megjeleníthet.
- d) Az applet nem futtathat külső programot, még a gazdagépen sem.

Feladatok

12.1. (A)! Írja át appletté a 8., *Swing-komponensek* fejezet StopperApp.java programját!

(stopper.jpx)

12.2. (A)! Írja át appletté a 8., *Swing-komponensek* fejezet TextAreaTest.java programját!

(textareatest.jpx)

12.3. (A)! Írja át appletté a 8., *Swing-komponensek* fejezet ColorChooserTest.java programját! (colorchoosertest.jpx)

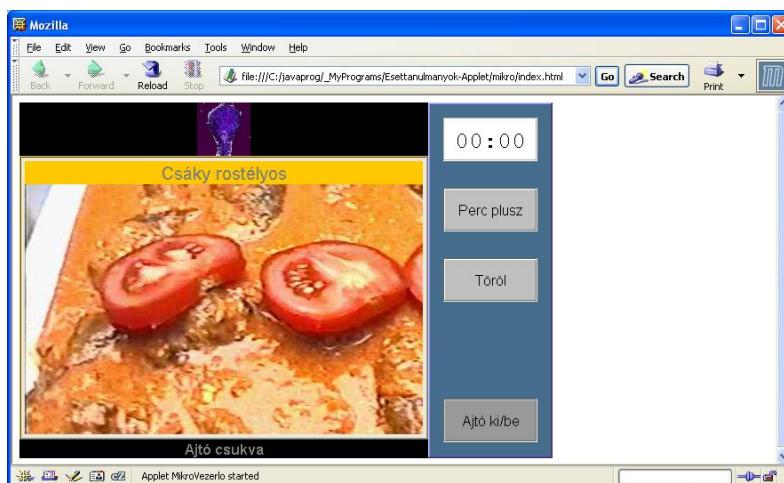
12.4. (A) Játssza le a javaprogsound mappa "taps.au" hangját! (HangTaps.java)

12.5. (B) Játsszon le ötször egymás után egy tetszőleges hangállományt ! (HangOtszor.java)

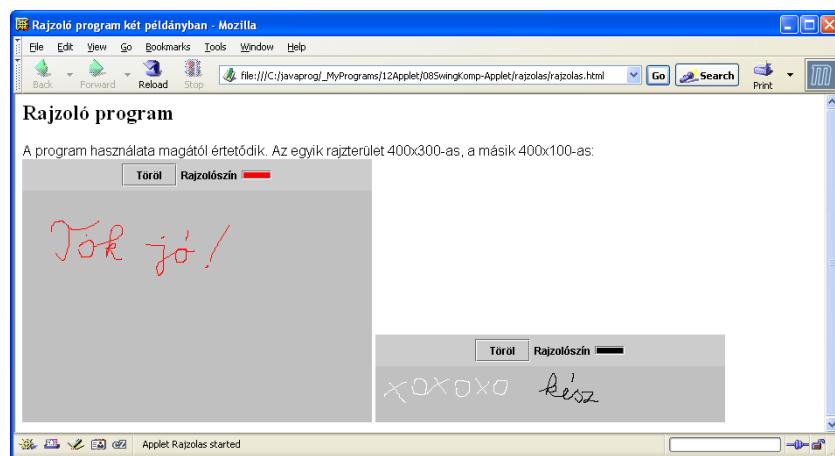
12.6. (B) Készítsen olyan appletet, amely sorban betölt képeket, és mindegyikhez más zenét játszik! (HangMesek.java)

12.7. (B) Olvasson fel néhány mesét, és vegye fel őket hangállományba, majd készítsen olyan appletet, amelyben választhat ezek közül az „esti mesék” közül! (HangSajat.java)

12.8. (B) Írja át appletté a könyv forráskód mellékletének Esettanulmányok könyvtárából a Mikro.jpx alkalmazást! (mikro.jpx)



12.9. (B)! Írja át appletté a 10., Alacsony szintű események fejezet Rajzolas.java programját! Jelenítsen meg két rajzolóappletet a böngészőben! (rajzolas.jpx)



I. OBJEKTUMORIENTÁLT TECHNIKÁK

1. Csomagolás, projektkezelés
2. Örökítés
3. Interfészek, belső osztályok
4. Kivételkezelés

II. GRAFIKUS FELHASZNÁLÓI INTERFÉSZ

5. A felhasználói interfész felépítése
6. Elrendezésmenedzserek
7. Eseményvezérelt programozás
8. Swing-komponensek
9. Grafika, képek
10. Alacsony szintű események
11. Belső eseménykezelés, komponensgyártás
12. Applet



III. ÁLLOMÁNYKEZELÉS

13. Állományok, bejegyzések
14. Folyamok
15. Közvetlen hozzáférésű állomány

III.

IV. VEGYES TECHNOLÓGIÁK

16. Rekurzió
17. Többszálú programozás
18. Nyomtatás
19. Hasznos osztályok

V. ADATSZERKEZETEK, KOLLEKCIÓK

20. Klasszikus adatszerkezetek
21. Kölcsönös keretrendszer

FELADATOK

FÜGGELÉK

- A tesztkérdések megoldása
Irodalomjegyzék
Tárgymutató

13. Állományok, bejegyzések

A fejezet pontjai:

1. A `java.io` csomag
 2. Útvonalak
 3. A `File` osztály
 4. Állományműveletek
 5. Szűrés – `FilenameFilter` interfész
 6. Állománykiválasztó dialógus – `JFileChooser`
 7. Könyvtár felderítése rekurzióval
-

A programokban gyakran kell adatokat lemezre (háttértárra) menteni, visszaolvasni őket, egyszóval állományokat (fájlokat) feldolgozni. A állományok a helyi vagy egy távoli számítógépen is lehetnek. Az állományokat kétféleképpen dolgozhatjuk fel: az állományfolyammal (file stream) sorban, egymás után és egyszerre csak egy irányban (ki/be) érjük el az adatokat; a közvetlen elérésű állománnyal bármely pozícióra írhatunk adatot és be is olvashatunk bármely pozícióról. De akármelyik módszert választjuk is, a szóban forgó állományt azonosítanunk kell. A programokban az állományokat logikailag azonosítjuk a `File` osztály egy-egy példányával. A logikai állományazonosító „mögött” egy fizikai állomány- vagy könyvtárbejegyzés van.

A állománykezeléssel kapcsolatos osztályok a `java.io` csomagban kaptak helyet. A fejezet először egy rövid áttekintést ad a `java.io` csomag osztályairól. Ezután a könyvtárbejegyzések következnek: megvizsgáljuk a könyvtárak, állományok tulajdonságait (írható/olvasható, hossz, létrehozás dátuma stb.), és állományműveleteket (létrehozás, törlés, átnevezés) végzünk. Megmutatjuk az állománykiválasztó dialógus használatát, s végül felderítjük egy adott háttértár vagy könyvtár összes állományát, beleértve a tartalmazott könyvtárat is. Az állománybejegyzésekhez tartozó konkrét adatok feldolgozásával a 14. és 15. fejezet foglalkozik majd.

13.1. A `java.io` csomag

A 13.1. ábra a `java.io` csomag osztályhierarchiáját ábrázolja.



13.1. ábra. A java.io csomag

Nézzük végig röviden a hierarchia tetején álló osztályokat, az Object közvetlen utódait!

- ◆ A `File` osztály egy példánya **logikailag azonosít** egy bejegyzést. A logikai bejegyzés-azonosító tárolja a bejegyzés útvonalát és rajta keresztül zajlanak le a bejegyzéssel kapcsolatos műveletek. Megmondja, hogy a bejegyzés létezik-e, állomány-e vagy könyvtár, rejtett-e, tudja a hosszát, törölheti, átnevezheti stb. A bejegyzéshez tartozó állomány konkrét tartalmának feldolgozását más osztályok végeznek. Egy `File` objektum létreho-

zása még nem teremt kapcsolatot a hardvereszközzel – egy rossz útvonal megadása csak később, a metódusok végrehajtásakor okoz bajt.

- ◆ A `FilenameFilter` interfész implementáló osztály bejegyzéseket szűr: kigyűjtethető vele az összes olyan bejegyzés, amely eleget tesz az implementációban megadott kritériumnak (pl. `*.java`).
- ◆ Az `InputStream` és `OutputStream` a beviteli és kiviteli bájtfolyamok (byte streamek) űzosztálya; segítségükkel bájtsorozatok dolgozhatók fel, egy irányban. A leszármazottak speciális bájtfolyamok:
 - A `File`-lal kezdődő nevű folyamoknak állomány a forrás-, illetve célhelyük;
 - a `Data`-val kezdődő nevű folyamok beviteli/kiviteli adategységeként már nem csak bájtokat, hanem primitív típusú adatokat is kezelnek;
 - az `Object`-tel kezdődő nevű folyamok objektumokat is írhatnak/olvashatnak.
- ◆ A `Reader` és a `Writer` a beviteli és kiviteli karakterfolyamok (character streamek) űzosztályai. Segítségükkel az állományok karakterenként dolgozhatók fel, egy irányban.
- ◆ A `RandomAccessFile` osztály közvetlenül elérhetővé teszi egy állomány adatait. Ez azt jelenti, hogy a megadott pozícion tárolt egység (bájt, karakter, adat, objektum) írható is, olvasható is.

A `Filter`- előnevű folyamok szűrőmunkálatokat végeznek; megszűrik, átalakítják a folyam elemeit. A `Buffered`-del kezdődő nevű folyamok a gyors bevitel/kivitel érdekében pufferelik az elemeket: bevitelkor nagyobb mennyiséggű adatot előre beolvasnak a memóriába, kivitelkor pedig összegyűjtik a kiírandó elemeket. A hardverműveletek minimálisra csökkentésével a program lényegesen gyorsítható.

Ebben a fejezetben csak a `File` osztállyal foglalkozunk.

13.2. Útvonalak

Mivel a `File` osztály útvonalukkal azonosítja a bejegyzéseket, tisztáznunk kell a különböző útvonalak fogalmát.

Egy útvonal helyi vagy távoli útvonal lehet.

Valódi és absztrakt útvonal: A `File` objektum absztrakt útvonalként tárolja az operációs rendszerben ténylegesen megadott valódi útvonalat; az absztrakt útvonal már független az operációs rendszertől. A valódi és absztrakt útvonal közötti konverziót a `File` objektum végzi.

A valódi útvonal részei (13.2. ábra):

- **Prefix:** A helyi vagy távoli számítógép, illetve meghajtó azonosítása (Windowsban "`C:\`" vagy "`\\"; Unixban egységesen "/")`

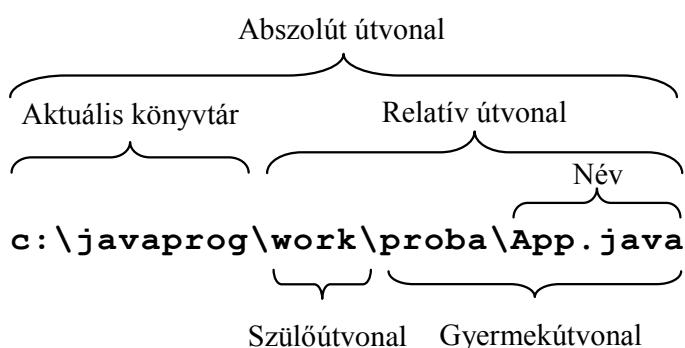
- **Nulla vagy több név**, szeparátorokkal elválasztva. Az utolsó névnek lehet kiterjesztése.
- **Szeparátor**: Egy útvonalon belül a neveket a szeparátor választja el (Windowsban "\"; Unixban "/").
- **Útvonalszeparátor**: Az útvonalszeparátor két útvonalat választ el (Windowsban ";" , Unixban ":"). A szeparátorok az operációs rendszerre jellemző adatok, azok a `File.separator`, illetve a `File.pathSeparator` statikus függvényekkel lekérdezhetők.

Abszolút és relatív útvonal:

- Abszolút útvonal: A számítógép gyökérkönyvtárából indul.
- Relatív útvonal: A számítógép aktuális könyvtárából ("..") indul. Az aktuális könyvtár fejlesztőkörnyezetenként más: lehet az a könyvtár, ahol a class állomány van; JBuilderben a Working directory.

Szülő- és gyermekútvonal: Az útvonal (az abszolút és relatív is) megadható két útvonal összeadásával: az első rész a szülőútvonal (parent path), a második rész a gyermekútvonal (child path); a kettőt egy szeparátor kapcsolja egybe. Egy útvonal többféleképpen is felbontható szülő- és gyermekútvonalra.

❖ Windowsban óvatosnak kell lennünk az útvonal megadásával, mert a karakterláncban a Java a \ karakter után escape szekvenciát vár – ezért a \ szeparátor megadásához a \ karaktert meg kell kettőznünk vagy / jelet kell használnunk. A `c:\jdk1.4\bin` könyvtárat tehát kétféleképpen adhatjuk meg: "`c:\\\\jdk1.4\\\\bin`" vagy "`c:/jdk1.4/bin`". A könyvben többnyire a "/" szeparátort használjuk.



13.2. ábra. Az útvonalak fajtái

Példaként nézzük a 13.2. ábra útvonalát! Az aktuális könyvtár a `c:/javaprog`, ezért a `work/proba/App.java` relatív útvonal e könyvtár alatt értendő. A teljes útvonal így a

c:/javaprogram/work/proba/App.java. A work/proba/App.java útvonalat többféleképpen is összeállíthatjuk szülő- és gyermekútvonalból; a . az aktuális, a .. a felette levő könyvtárat jelenti (a konkrét útvonal-megadási szabályokat lásd a File osztály konstruktoraiban leírásában):

Szülőútvonal	Gyermekekútvonal
"work/proba/App.java"	""
"work/proba"	"App.java"
"work"	"proba/App.java"
".."	"work/proba/App.java"

13.3. A File osztály

Csomag: java.io Deklaráció: public class File

Közvetlen ős: java.lang.Object

Fontosabb implementált interfeszek: Comparable, Serializable

A File osztály példánya egy logikai bejegyzésazonosító: az állomány- vagy könyvtárbejegyzést azonosítja az útvonalával. Az osztály metódusaival lekérdezhetők és módosíthatók a bejegyzésben tárolt információk. Az osztály segítségével fel lehet deríteni a könyvtárrendszert, állományokat lehet törölni, átnevezni, állományattribútumokat lehet megváltoztatni, sőt üres állományt is létre lehet hozni.

Megjegyzés: A File osztály bejegyzést azonosít; neve onnan ered, hogy Unix-ban minden bejegyzés, így a könyvtár is állomány.

Mezők (publikus osztályadatok)

- ▶ static String separator
- ▶ static char separatorChar

Az útvonal részeit elválasztó szeparátorjel; az operációs rendszerre jellemző. Windowsban ez a "\\" (backslash), Unixban a "/" (per).

- ▶ static String pathSeparator
- ▶ static char pathSeparatorChar

Az útvonalakat elválasztó szeparátorjel; az operációs rendszerre jellemző. Windowsban ez a ";", Unixban a ":".

Konstruktörök

- ▶ File(String pathname)
- ▶ File(String parent, String child)
- ▶ File(File parent, String child)

Létrehoz egy File objektumot. A paraméterben megadott (valódi vagy absztrakt) útvonalak alapján eltárol egy absztrakt útvonalat. Ha szülő- és gyermekútvonalat adunk

meg, akkor összeadja őket és egy szeparátorjelet tesz közéjük. Az útvonalban szerepelhet a "... könyvtárnév is: az aktuális feletti könyvtár. Az aktuális könyvtár: "..". A tárolt útvonalra jellemző, hogy abszolút-e vagy relatív. Az útvonal a létrehozás után már nem változtatható meg. A háttértárral még nincs kapcsolat, ezért egy nem létező útvonal megadása itt még nem okoz hibát. Paraméterek:

- pathname: abszolút vagy relatív útvonal;
- parent: abszolút vagy relatív szülőútvonal (csak könyvtárbejegyzés lehet).
- child: gyermekútvonal. `NullPointerException` kivétel keletkezik, ha `child` értéke null.

Például a következő utasítások egyenértékűek:

```
File f = new File("work", "proba/App.java");
File f = new File(".", "work/proba/App.java");
File f = new File("work/proba/App.java", "");
```

A tárolt absztrakt útvonal ez lesz:

```
"work/proba/App.java"
```

Az osztályleírásban erre a "work/proba/App.java" útvonalat tartalmazó `File` objektumra fogunk hivatkozni.

Metódusok – A bejegyzés útvonala

- `boolean isAbsolute()`

A visszaadott érték `true`, ha az objektumban tárolt útvonal abszolút, egyébként `false`.

Példánkban: `f.isAbsolute() == false`

- `String getAbsolutePath()`
- `File getAbsoluteFile()`

Visszaadja az abszolút (valódi) útvonalat, illetve egy ilyen állapotú `File` objektumot.

Ha az útvonal relatív, akkor azt balról kiegészíti az aktuális könyvtár útvonalával.

Példánkban: `f.getAbsolutePath() == "c:\javaprogram\work\proba\App.java"`

- `String getPath()`

Visszaadja az objektumban tárolt (abszolút vagy relatív) útvonalat. Példánkban:

`f.getPath() == "work\proba\App.java"`

- `String getCanonicalPath() throws IOException`
- `File getCanonicalFile() throws IOException`

Visszaad egy abszolút és egyedi útvonalat. Az útvonalban feloldja az esetleges relatív ("..") hivatkozásokat.

- `String getParent()`
- `File getParentFile()`

Visszaadja a bejegyzés feletti könyvtárat úgy, hogy az útvonalból leveszi a nevet a szeparátorral együtt. Ha nincs ilyen (például a tárolt útvonal relatív és a szülőkönyvtár nem

része), akkor a visszaadott érték null. Példánkban: `f.getParent() == "work\proba"`. A proba felett a work van, a felett a null.

► `String getName()`

Visszaadja a bejegyzés nevét, vagyis az abszolút útvonal utolsó szeparátor utáni részét, kiterjesztéssel együtt, ha van. A főkönyvtár neve az üres karakterlánc. Példánkban: `f.getName() == "App.java"`

Metódusok – A bejegyzés tulajdonságai

E metódusokban nem keletkezik kivétel, bár a háttértárral való kapcsolat már létrejön. Ha a bejegyzés nem létezik, akkor a visszaadott érték nullaszerű (`false/0`).

- `boolean isDirectory()`
- `boolean isFile()`

Megadja, hogy a bejegyzés könyvtár-e, állomány-e. Ha a bejegyzés létezik, akkor a kettő közül pontosan az egyik igaz.

Példánkban: `f.isDirectory() == false, f.isFile() == true`

► `long length()`

Megadja, hogy a bejegyzéshez tartozó állomány hánnyal foglal el a háttértáron. Könyvtárbejegyzés esetén az érték 0.

- `long lastModified()`
- `boolean setLastModified(long time)`

Visszaadja, illetve beállítja az utolsó módosítás időpontját. Az idő kezelésével a 17., a Hasznos osztályok című fejezet foglalkozik. A beállítás értéke `true`, ha sikerült.

- `boolean canRead()`
- `boolean canWrite()`
- `boolean exists()`
- `boolean isHidden()`

Megadja, hogy az alkalmazás olvashatja-e, írhatja-e az állományt; az állomány létezik-e, rejtett-e.

► `boolean setReadOnly()`

Csak olvashatóvá (írásvédetté) teszi a bejegyzést. Értéke `true`, ha sikerült.

Metódusok – Képesség

- `boolean mkdir()`
- `boolean mkdirs()`

Létrehozza a tárolt útvonallal (könyvtár!) megadott könyvtára(ka)t: az `mkdir` egyet, az `mkdirs` több egymás alattit. A visszaadott érték `true`, ha a létrehozás (az összes) sikeresült, egyébként `false`. Példánkban: `f.mkdirs()` létrehozza az aktuális könyvtár alatt a következő könyvtárakat (ha eddig még nem léteztek): `work, proba, App.java` (ez az utolsó is egy könyvtár lesz!).

- ▶ `boolean createNewFile() throws IOException`
Létrehoz egy új, üres (nulla hosszúságú) állományt, de csak akkor, ha ilyen állomány még nem létezik. A visszaadott érték `true`, ha sikerült a létrehozás.
- ▶ `boolean delete()`
Letörli az állományt, illetve a könyvtárat a háttértárról. A visszaadott érték `true`, ha a törlés sikerült, egyébként `false`. Egy könyvtár csak akkor törölhető, ha üres.
- ▶ `void deleteOnExit()`
Letörli az állományt, illetve a könyvtárat a háttértárról, amikor a virtuális gép befejezi futását.
- ▶ `boolean renameTo(File dest)`
Átnevezi az állományt, illetve a könyvtárat a `dest`-ben megadottra. A visszaadott érték `true`, ha az átnevezés sikerült, egyébként `false`. Ha a `dest` egy könyvtár bejegyzését azonosítja, akkor az átnevezés nem sikerülhet.

Metódusok – Bejegyzéslisták

- ▶ `String[] list()`
- ▶ `String[] list(FilenameFilter filter)`
Ha a bejegyzés könyvtárat azonosít, akkor visszaadja annak bejegyzésneveit. Ha az objektum nem könyvtár, akkor a visszaadott érték `null`. `filter` megadásakor csak a `filter` szűrési feltételének megfelelő bejegyzésneveket adja vissza.
- ▶ `File[] listFiles()`
- ▶ `File[] listFiles(FilenameFilter filter)`
Ha a bejegyzés könyvtárat azonosít, akkor visszaadja annak `File` típusú bejegyzéseit. Ha az objektum nem könyvtár, akkor a visszaadott érték `null`. `filter` megadásakor a `filter` szűrési feltételének megfelelő bejegyzésneveket adja vissza. A szűrést lásd később.
- ▶ `static File[] listRoots()`
Visszaadja az operációs rendszer által felismert gyökérkönyvtárakat (pl. `A:\`, `C:\` stb.)

Metódusok – Egyéb

- ▶ `equals(Object o)`
- ▶ `compareTo(File pathname)`
- ▶ `compareTo(Object o)`
Egyenlőségvizsgálat, illetve összehasonlítások az absztrakt útvonal alapján, lexikografikusan.
- ▶ `String toString()`
Visszaadja a bejegyzés karakterláncformáját. Meghívja a `getPath()` metódust.

Feladat – Bejegyzések tulajdonságai (FileAttributes)

A programnak egy szülő- és egy gyermekútvonal a paramétere. A szülőútvonalat nem kötelező megadni, a gyermekútvonalat igen. Ha nem adnak meg paramétert, akkor írunk ki egy információs szöveget a program használatáról! Az útvonalakkal megadott bejegyzésről állapítsuk meg, hogy létezik-e, majd írjuk ki a tulajdonságait:

- Könyvtár-e vagy állomány
- Útvonal (a konstruktörben megadott szülő- és gyermekútvonal)
- Abszolút útvonal (gyökérkönyvtárból induló útvonal)
- Bennfoglaló könyvtár (amelyben a bejegyzés van)
- Bejegyzés neve
- Az állomány hossza
- Az utolsó módosítás dátuma

Forráskód

```

import java.io.*; //1
import java.text.DateFormat; //2
import java.util.Date;

public class FileAttributes {
    public static void main (String args[]) {
        if (args.length == 0) {
            System.out.println(
                "Használat: [Szülőútvonal] Gyermekútvonal");
            return;
        }
        File f;
        if (args.length == 1)
            f = new File(args[0]);
        else
            f = new File(args[0],args[1]);

        if (f.exists()) {
            System.out.println("Létezik, adatai:");
            System.out.println(f.isDirectory()?"Könyvtár":"Állomány");
            System.out.println("Útvonal: "+f.getPath());
            System.out.println("Abszolút útvonal: "+
                f.getAbsoluteFilePath());
            System.out.println("Bennfoglaló könyvtár: "+
                f.getParent());
            System.out.println("Bejegyzés neve: "+f.getName());
            System.out.println("Hossza (byte): "+f.length());
            DateFormat df = DateFormat.getDateInstance();
            String datum = df.format(new Date(f.lastModified()));
            System.out.println("Utolsó módosítás dátuma: "+datum);
        }
        else
            System.out.println("Nincs ilyen bejegyzés.");
    } // main
} // FileAttributes

```

A program futása

A programot két paramétergyüttessel is lefuttatjuk (az aktuális könyvtár a c:/javaprogram, alatta van a work könyvtár.):

1. eset: A program paraméterei: c:/JBuilder9/jdk1.4/bin/javac.exe

```
Létezik, adatai:  
Állomány  
Útvonal: C:\JBuilder9\jdk1.4\bin\javac.exe  
Abszolút útvonal: C:\JBuilder9\jdk1.4\bin\javac.exe  
Bennfoglaló könyvtár: C:\JBuilder9\jdk1.4\bin  
Bejegyzés neve: javac.exe  
Hossza (byte): 28794  
Utolsó módosítás dátuma: 2003.11.09. 7:25:46
```

2. eset: A program paraméterei: work

```
Létezik, adatai:  
Könyvtár  
Útvonal: work  
Abszolút útvonal: C:\javaprogram\work  
Bennfoglaló könyvtár: null  
Bejegyzés neve: work  
Hossza (byte): 0  
Utolsó módosítás dátuma: 2003.11.19. 7:18:14
```

A program elemzése

- ◆ //1: A java.io csomagban vannak az állománykezeléssel kapcsolatos osztályok.
- ◆ //2: A java.text.DateFormat és a java.util.Date a dátum megjelenítéséhez szükséges – róluk a 19., a Hasznos osztályok című fejezetben lesz szó.

A program többi része magáért beszél – nézze meg a megfelelő metódusleírásokat!

Figyelje meg, hogy a program az útvonalakat windowsos módon írja ki a konzolra, noha az útvonalat unixos vagy absztrakt módon adtuk meg!

Feladat – Jellemzők

Írjuk ki az

- operációs rendszerre jellemző szeparátort és útvonal-szeparátort;
- aktuális könyvtárat; végül
- az aktuális könyvtártól felfelé egyenként az összes könyvtárat (először az abszolút útvonalát, majd szögletes zárójelben a nevét)!

Forráskód

```
import java.io.*;  
  
public class Jellemzok {  
    public static void main (String args[]) {  
        System.out.println("Szeparátor: "+File.separator);           //1  
        System.out.println("Útv. szeparátor: "+File.pathSeparator);
```

```

        File f = new File("");
        System.out.println("Aktuális könyvtár: "+
            f.getAbsolutePath());
        f = f.getAbsoluteFile(); //3
        while (f != null) {
            System.out.println(f.getPath()+" ["+f.getName()+"])");
            f = f.getParentFile();
        }
    } // main
} // Jellemzok

```

A program futása

Tegyük fel, hogy az aktuális könyvtár a `c:/javaprog/work`. Állítsa át a Working Directoryt!

```

Szeparátor: \
Útvonal szeparátor: ;
Aktuális könyvtár: C:\javaprog\work
C:\javaprog\work [work]
C:\javaprog [javaprog]
C:\ []

```

A program elemzése

- ◆ //1: Az operációs rendszerre jellemző szeparátorokat a megfelelő statikus adatok tartalmazzák.
- ◆ //2: Ha a konstruktorban csak egy karakterláncot adunk meg, az a gyermekútvonalnak felel meg. Mivel az nem a gyökérkönyvtárból indul, a rendszer relatív útvonalnak tekinti – az abszolút útvonal tehát az aktuális könyvtárnak és ennek az útvonalnak a konkatenációja, vagyis pont az aktuális könyvtár.
- ◆ //3: Egy új `File` objektumot hozunk létre, amelyben már abszolút az útvonal. Ha abszolút útvonalról van szó, akkor a `getPath()` és `getAbsolutePath()` értéke megegyezik. Kiírjuk ezt az útvonalat, majd zárójelben a nevet: az útvonal utolsó szeletét. Ezután létrehozunk egy újabb `File` objektumot; annak az útvonala az eddigi bejegyzés feletti könyvtárat azonosítja. Ezt addig ismétljük, ameddig még létező (nem `null`) könyvtár ról van szó.

13.4. Állományműveletek

Ebben a pontban állományokat fogunk törölni és átnevezni, majd ki fogjuk listázni egy adott könyvtár összes bejegyzését. Adva van egy `c:/javaprog/work/proba` könyvtár, s abban 3 állomány meg egy `icons` nevű alkönyvtár. Az alkönyvtárban további 2 állomány van:

```

c:/javaprog/work/proba/App.java
c:/javaprog/work/proba/FileAttributes.class
c:/javaprog/work/proba/FileAttributes.java
c:/javaprog/work/proba/icons/delete.gif
c:/javaprog/work/proba/icons/App.java

```

Feladat – Átnevezés, törlés, létrehozás (RenDel)

A c:/javaprog/work/proba könyvtárban végezzük el a következő állományműveleteket:

- Nevezzük át az App.java állományt App..~jav-ra!
- Töröljük ki a teljes icons könyvtárat! Feltesszük, hogy abban nincs további könyvtár.
- Hozzunk létre egy üres readme.txt szöveges állományt!

Minden művelet után listázzuk ki a könyvtár tartalmát!

Forráskód

```
import java.io.*;

public class RenDel {

    // A megadott könyvtár listázása:
    static void printDir(String dir) {
        File f = new File(dir);
        if (!f.exists() || !f.isFile()) {
            System.out.println(dir+" nem egy könyvtár.");
            return;
        }

        System.out.println(dir+" bejegyzései:");
        String[] list = f.list();
        // A könyvtár bejegyzéseinek kiírása:
        for (int i=0; i<list.length; i++)
            System.out.println(list[i]);
        System.out.println();
    }

    // Main:
    public static void main (String args[]) {
        String dir = "c:/javaprog/work/proba";
        printDir(dir);

        // dir-ben az App.java file átnevezése App..~jav -ra:
        File f = new File(dir,"App.java");
        if (f.renameTo(new File(dir,"App..~jav")))
            System.out.println("Az átnevezés sikeresült.");
        else
            System.out.println("Az átnevezés nem sikeresült.");
        printDir(dir);

        // Az icons könyvtár törlése, előtte ürítése:
        f = new File(dir,"icons");
        File[] list = f.listFiles();
```

```

        if (list!=null) {
            for (int i=0; i<list.length; i++)
                list[i].delete();
            if (f.delete())
                System.out.println("A törlés sikerült.");
        }
        else
            System.out.println("A törlés nem sikerült.");
    }
    else
        System.out.println("Nincs ilyen könyvtár.");
}

// A readme.txt létrehozása:
f = new File(dir,"readme.txt");
try {
    f.createNewFile();
    System.out.println("A létrehozás sikerült.");
}

catch(IOException e) {
    System.out.println("A létrehozás nem sikerült.");
}
printDir(dir);
} // main
} // RenDel

```

A program futása

Ha a könyvtár valóban olyan szerkezetű volt, amilyennek leírtuk, akkor a következőre változott:

```
c:/javaprog/work/proba/App.java
c:/javaprog/work/proba/FileAttributes.class
c:/javaprog/work/proba/FileAttributes.java
c:/javaprog/work/proba/readme.txt
```

13.5. Szűrés – FilenameFilter interfész

Csomag: java.io

Deklaráció: public interface FilenameFilter

A `FilenameFilter` interfész bejegyzések szűrését szolgálja. Egyetlen metódusa, az `accept` (elfogadás) a szűrés kritériumát adja meg. Megadásával szűrhetjük a bejegyzéseket például a `File` osztály `list` és `listFiles` metódusában.

Metódus

- `boolean accept(File dir, String name)`

A `dir` könyvtár `name` bejegyzéséről megállapítja, hogy legyen-e eleme a listának.

Feladat – Szűrés

Írjuk ki konzolra a program paramétereként megadott könyvtár összes exe állományát! Ha nem adnak meg paramétert, akkor az aktuális könyvtárból listázzunk!
Futtatáskor a program paramétere legyen a java home könyvtár bin alkönyvtára!

Forráskód

```
import java.io.*;

class FiletypeFilter implements FilenameFilter { //1
    private String extension;

    public FiletypeFilter(String extension) {
        this.extension = extension.toUpperCase();
    }

    public boolean accept(File dir, String name) { //2
        return name.toUpperCase().endsWith('.'+extension);
    }
}

public class Szures {
    public static void main(String[] args) { //3
        File f;
        if (args.length>0)
            f = new File(args[0]);
        else
            f = new File(".");
        // Aktuális könyvtár

        if (!f.exists() || !f.isDirectory()) {
            System.out.println("Nincs, vagy nem könyvtár.");
            return;
        }
        String[] list = f.list(new FiletypeFilter("exe")); //4
        System.out.println(f.getAbsolutePath()+
                           " könyvtár exe állományai:");
        for(int i=0; i<list.length; i++)
            System.out.println(list[i]);
    }
} // Szures
```

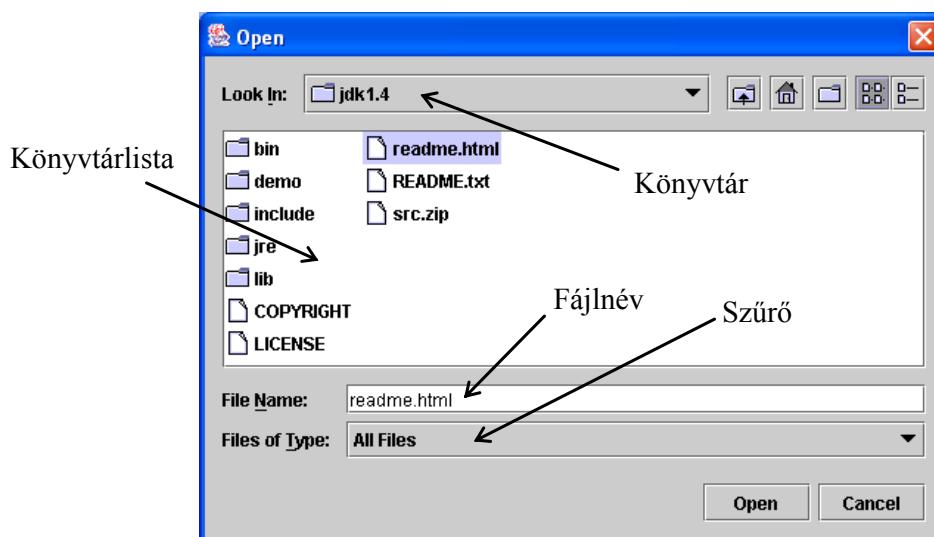
A program futása – a program paramétere: c:/JBuilder9/jdk1.4/bin

```
C:\JBuilder9\jdk1.4\bin könyvtár exe állományai:
appletviewer.exe
jar.exe
java.exe
javac.exe
javadoc.exe
javaw.exe
...
```

A program elemzése

- ◆ //1: A `FiletypeFilter` osztállyal végeztetjük el a szűrést, implementáljuk tehát a `FilenameFilter` interfészt. A továbbfejleszthetőség kedvéért az osztály konstruktőrben megadjuk a kiterjesztést, hogy a szűrést bármilyen kiterjesztésre el tudjuk majd végezni.
- ◆ //2: Ha a név a . (pont) + kiterjesztéssel végződik, akkor elfogadjuk, egyébként nem.
- ◆ //3: A program úgy kezdődik, hogy létrehozunk egy `File` objektumot, amelyben a program paramétere az útvonal – vagy az aktuális könyvtár útvonala. Ha az útvonal által azonosított bejegyzés nem létezik, akkor egy üzenet kíséretében leállítjuk a programot.
- ◆ //4: Az `f:File` objektumnak paraméterben adjuk át a szűrőobjektumot, és elkérjük tőle a szűrőn „fennakadt” neveket. Az így kapott neveket konzolra listázzuk.

13.6. Állománykiválasztó dialógus – JFileChooser



13.3. ábra. JFileChooser – állománykiválasztó dialógus

Csomag: `javax.swing`

Deklaráció: `public class JFileChooser`

Közvetlen ős: `javax.swing.JComponent`

Fontosabb implementált interfések: `Serializable`

Az **állománykiválasztó dialógus** (`JFileChooser`) egy modális dialógusablak; feladata egy abszolút útvonal bekérése (13.3. ábra).

A dialógus elemei:

- **Könyvtár** (Look in): Aktuális könyvtár (current directory). E könyvtár bejegyzéseit jeleníti meg a könyvtárlista.
- **Könyvtárlista**: A könyvtár bejegyzései. Ha a dialógushoz szűrő van rendelve, akkor csak a szűrt bejegyzések jelennek meg.
- **Szűrő** (Files of Type): A dialógus szűrője. A lehetséges szűrők programból állíthatók.
- **Állománynév** (File name): A könyvtárlistából kiválasztott állomány neve. A mező szerkeszthető; olyan állománynév is beírható, amely nincs a megjelenített könyvtárlistában.

A dialógusnak két megjelenítő metódusa van: `showOpenDialog` és `showSaveDialog` – különbség csak a feliratokban van (cím, gombok). A megjelenítő metódus három módon térhet vissza:

- `APPROVE_OPTION`: kiválasztottak egy bejegyzést. Ekkor a dialógustól le lehet kérdezni a kiválasztott könyvtárat (`currentDirectory`) és a bejegyzést (`selectedFile`).
- `CANCEL_OPTION`: elvetették a kiválasztást, a kiválasztott könyvtár és állomány marad az eredeti.
- `ERROR_OPTION`: hiba lépett fel.

Jellemzők (set és get metódusokkal állíthatók, illetve lekérdezhetők)

- ▶ `String dialogTitle`
A dialógus címe.
- ▶ `int dialogType`
A dialógus típusa. Lehetséges értékei: `OPEN_DIALOG` és `SAVE_DIALOG`.
- ▶ `File currentDirectory`
Az aktuális könyvtárbejegyzés, abszolút útvonal. Beállításnál, ha a megadott könyvtár nem létezik, akkor a szülökönyvtár lesz az aktuális könyvtár (felmehet egészen a gyökérkönyvtárig). Alapértelmezés: ha nem állítjuk be, akkor a `user home` könyvtár (operációs rendszertől függ).
- ▶ `int fileSelectionMode`
Kiválasztási mód; csak a megadott típusú bejegyzéseket lehet választani. Lehetséges értékek: `FILES_ONLY` (csak állományokat), `DIRECTORIES_ONLY` (csak könyvtárokat), `FILES_AND_DIRECTORIES` (mindkettőt). Alapértelmezés: `FILES_ONLY`.
- ▶ `boolean multiSelectionEnabled`
Többszörös választás engedélyezése.
- ▶ `File selectedFile`
- ▶ `File[] selectedFiles`
A kiválasztott állomány, illetve állományok (ha `multiSelectionEnabled==true`). Alapértelmezés: `null`.

► `FileFilter fileFilter`

A dialógushoz rendelt szűrőobjektum; osztálya a `FileFilter` absztrakt osztály leszármazottja. Alapértelmezés: nincs szűrés. Lásd a `FileFilter` osztályt.

Mezők

- `static int APPROVE_OPTION`
- `static int CANCEL_OPTION`
- `static int ERROR_OPTION`

A dialógus konstansai.

Konstruktorok, metódusok

- `JFileChooser()`
- `JFileChooser(File currentDirectory)`
- `JFileChooser(String currentDirectoryPath)`

A dialógus létrehozása induló könyvtárbejegyzés megadásával.

- `int showOpenDialog(Component parent)`
- `int showSaveDialog(Component parent)`

Megjeleníti a dialógust, hogy válasszanak belőle állományt. A dialógus tulajdonosa `parent`. A két metódus a dialógus címében és a gombfeliratokban tér el egymástól. A visszatérési érték lehetséges értékei: `APPROVE_OPTION` (választottak), `CANCEL_OPTION` (elvetették), `ERROR_OPTION` (hiba állt elő).

- `String getName(File f)`

Visszaadja a `File` bejegyzés nevét, vagyis `f.getName()`-et.

FileFilter osztály

Csomag: `javax.swing.filechooser`

Deklaráció: `public abstract class FileFilter`

A `FileFilter` absztrakt osztály; a `JFileChooser` dialógus könyvtárlistájának szűrését szolgálja. Az utódosztályban két absztrakt metódust kell implementálni.

Metódusok

- `public boolean accept(File f)`

Az `f` bejegyzéséről megállapítja, hogy megjelenhet-e a könyvtárlistában vagy sem.

- `public String getDescription()`

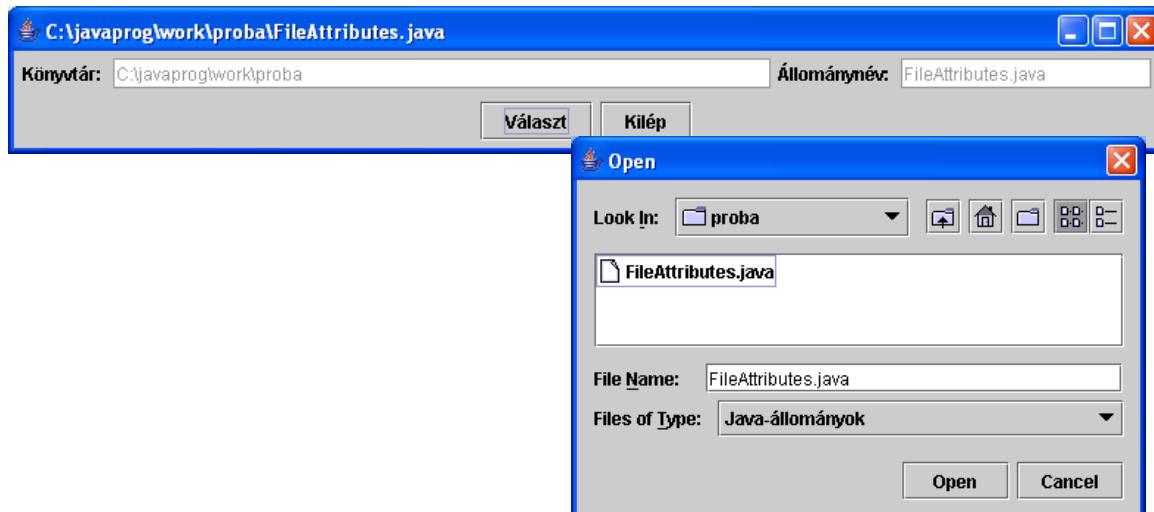
Megadja a szűrőmezőbe írt szöveget.

Megjegyzés: Ne tévessze össze ezt a `FileFilter` osztályt a `java.io.FileFilter`-rel!

Feladat – FileChooserTest

Készítsünk alkalmazást Java forrásállományok keresgélésére!! Tegyünk a keretre két szövegmezőt és két nyomógombot:

- a szövegmezőkben az éppen kiválasztott könyvtár és java állomány neve legyen, és egyiket se lehessen szerkeszteni!
- a Választ gomb lenyomására jelenjen meg az állománykiválasztó dialógus! A szövegmezők értéke ezután a kiválasztott könyvtár és állománynév legyen! A keret címe legyen a kiválasztott állomány teljes útvonala! A dialógus kezdeti könyvtára az aktuális könyvtár legyen, ezután pedig minden az előző állapotot jelenjen meg!
- a Kilép gombra csukjuk be a keretet!



Forráskód

```

import javax.swing.*;
import javax.swing.filechooser.FileFilter;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

class JavaFilter extends FileFilter { //1

    public boolean accept(File f) {
        return f.getName().toUpperCase().endsWith(".JAVA") || f.isDirectory();
    }

    public String getDescription() {
        return "Java-állományok";
    }
}

public class FileChooserTest extends JFrame
    implements ActionListener {
    private Container cp = getContentPane();
    private JTextField tfDir, tfFile;
    private JButton btValaszt, btKilep;
    private JFileChooser fc = new JFileChooser(); //2
}

```

```
public FileChooserTest() {
    setLocation(100,100);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    fc.setCurrentDirectory(new File(".")); //3
    fc.setFileFilter(new JavaFilter());

    JPanel pnInfo = new JPanel(); //4
    pnInfo.add(new JLabel("Könyvtár: "));
    pnInfo.add(tfDir = new JTextField(40));
    pnInfo.add(new JLabel("Állománynév: "));
    pnInfo.add(tfFile = new JTextField(15));
    tfDir.setEnabled(false);
    tfFile.setEnabled(false);
    cp.add(pnInfo);

    JPanel pnGombok = new JPanel();
    pnGombok.add(btValaszt = new JButton("Választ"));
    pnGombok.add(btKilep= new JButton("Kilép"));
    cp.add(pnGombok,"South");
    btValaszt.addActionListener(this);
    btKilep.addActionListener(this);

    pack();
    show();
}

void update() { //5
    File sf = fc.getSelectedFile();
    if (sf != null) {
        tfDir.setText(fc.getCurrentDirectory().getAbsolutePath());
        tfFile.setText(sf.getName());
        setTitle(sf.getAbsolutePath());
    }
}

public void actionPerformed(ActionEvent e) { //6
    if (e.getSource()==btValaszt) {
        if (fc.showOpenDialog(this)==
            JFileChooser.APPROVE_OPTION) {
            update();
        }
    }
    else if (e.getSource()==btKilep)
        System.exit(0);
}

public static void main(String[] args) {
    new FileChooserTest();
}
} // FileChooserTest
```

A program elemzése

- ◆ //1: A JavaFilter osztály megszűri a dialógusablak könyvtárlistájának elemeit. A szűrőn csak a java kiterjesztésű állományok és a könyvtárak akadnak fenn.
- ◆ //2: Létrehozzuk a JFileChooser dialógust.
- ◆ //3: Beállítjuk a dialógusban a kezdeti könyvtárat és a szűrőt.
- ◆ //4: Összeállítjuk a keretet. A szövegmezőket letiltjuk.
- ◆ //6: Ha lenyomták a Választ gombot, akkor megjelenítjük a dialógust megnyitási (OPEN) üzemmódban. A dialógus állapotán nem változtatunk, érvényben marad az előző beállítás. Ha az Open gomb lenyomásával hagyták el a dialógust, vagyis jóváhagyták azt (APPROVE_OPTION) akkor az update metódus meghívásával módosítjuk a kijelzéseket.
- ◆ //5: update metódus: A tfDir szövegmező értéke az aktuális könyvtár abszolút útvonala lesz. A kiválasztott állomány nevét a tfFile mezőbe tesszük, a keret címe annak abszolút útvonala lesz. A metódust a konstruktorból is meghívjuk.

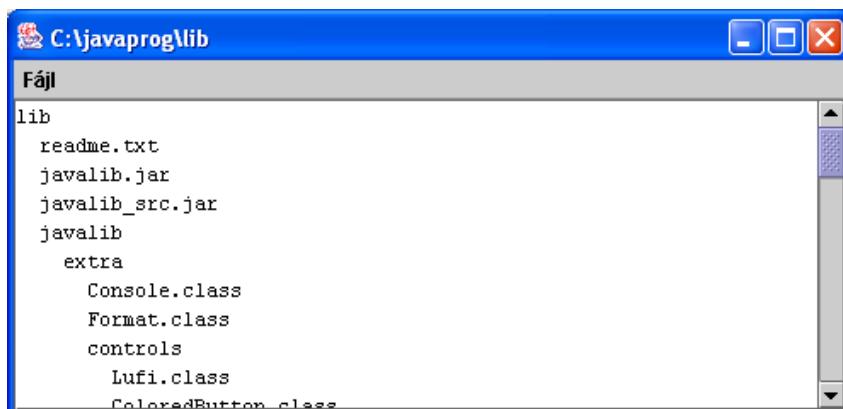
Figyelje meg, hogy a dialógusból nem tud könyvtárat kiválasztani! Ez azért van így, mert a dialógus alapértelmezett kiválasztási módja: fileSelectionMode==FILES_ONLY!

13.7. Könyvtár felderítése rekurzióval

Feladat – Könyvtárlista

Készítsünk egy könyvtárfelderítő alkalmazást! Listázzuk ki strukturáltan egy szövegterületre a felhasználó által megadott könyvtárat: az alkönyvtárak bejegyzései minden két karakterhellyel beljebb kezdődjenek!

A feladatot rekurzióval oldjuk meg, felhasználva a 16., Rekurzió című fejezetben közölt ismereteket. A feladatot mégis ebben a fejezetben helyeztem el, mert az állománykezelés miatt ide kívánkozott. Lapozzon előre nyugodtan a Rekurzió fejezetre!



A program terve

Készítünk egy KonyvtarArea osztályt; ez a JTextArea leszármazottja lesz. A setKonyvtar(File dir) metódusnak az lesz a feladata, hogy a kért könyvtárat kíllistázza a területre. A listázást rekurzív módon készítjük el: egy könyvtár kíllistázását a konyvtarLista metódus végzi el. A listázandó könyvtárat menüből kérjük be, egy állománydialógus segítségével.

Forráskód

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

class KonyvtarArea extends JTextArea {
    private int maxSzint = 5; // kiírandó szintek maximális száma
    private File dir;
    private int szint;

    public void setKonyvtar(File dir) {
        this.dir = dir;
        szint = 0;
       setFont(new Font("Monospaced",Font.PLAIN,12));
        setText("");
        konyvtarLista(dir);
    }

    private void konyvtarLista(File f) {
        if (!f.exists()) {
            append("Nemlétező bejegyzés!");
            return;
        }
        append(szokoz(szint)+f.getName()+"\n");
        if (f.isFile())
            return;

        if (szint < maxSzint) {
            szint++;
            String[] fList = f.list();
            for (int i=0; i<fList.length; i++)
                konyvtarLista(new File(f.getAbsoluteFilePath()+
                    File.separator+fList[i]));
            szint--;
        }
    }

    private String szokoz(int eltolas) {
        String str = "";
        for (int i=0; i<eltolas; i++)
            str = str+" ";
        return str;
    }
}
```

```

public class KonyvtarLista extends JFrame
    implements ActionListener {
    private JMenuBar mb;
    private JMenuItem miUjKonyvtar;
    private JMenuItem miBezaras;
    private KonyvtarArea taKonyvtar;
    private JFileChooser fc = new JFileChooser();

    public KonyvtarLista() {
        setTitle("Könyvtárlista");
        setBounds(200,100,500,300);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setJMenuBar(mb = new JMenuBar());
        JMenu mFajl = new JMenu("Állomány");
        mb.add(mFajl);
        mFajl.add(miUjKonyvtar = new JMenuItem("Új könyvtár"));
        mFajl.add(miBezaras = new JMenuItem("Bezárás"));
        miUjKonyvtar.addActionListener(this);
        miBezaras.addActionListener(this);
        taKonyvtar = new KonyvtarArea();
        getContentPane().add(new JScrollPane(taKonyvtar));
        fc.setDialogTitle("Válasszon ki egy könyvtárat!");
        fc.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
        fc.setCurrentDirectory(new File("C:/"));
        show();
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource()==miUjKonyvtar) {
            if (fc.showOpenDialog(this)==fc.APPROVE_OPTION) {
                taKonyvtar.setKonyvtar(fc.getSelectedFile());
                setTitle(fc.getSelectedFile().getAbsolutePath());
            }
        }
        else if (e.getSource()==miBezaras)
            System.exit(0);
    }

    public static void main(String[] args) {
        new KonyvtarLista();
    }
} // KonyvtarLista

```

Tesztkérdések

- 13.1. Mely állítások igazak? Jelölje be az összes helyes állítást!
- Az absztrakt útvonal függ az operációs rendszertől.
 - A valódi útvonal szeparátorjele függ az operációs rendszertől.
 - Az abszolút útvonal sosem egyezhet meg a relatív útvonallal.
 - A relatív útvonal egy szülő és egy gyermekútvonal megadásával elvileg többféleképpen is összeállítható.

13.2. Mely állítások igazak? Jelölje be az összes helyes állítást!

- a) A `File` objektum segítségével egy állomány teljes tartalma kiolvasható.
- b) A `File` objektum segítségével megtudható az állomány hossza.
- c) A `File` objektum segítségével megtudható, hogy a bejegyzést mikor módosították utoljára.
- d) A `File` objektum segítségével megtudható, hogy a bejegyzés milyen könyvtárban van.

13.3. Mely állítások igazak? Jelölje be az összes helyes állítást!

- a) Egy `File` objektum külön tárolja a bejegyzéshez tartozó szülő- és gyermekútvonalat, s minden lekérdezhető.
- b) minden `File` objektum külön tárolja az állományszeparátort.
- c) A `File` objektum útvonala az objektum létrehozása után nem változtatható meg.
- d) A `File` objektum segítségével létrehozhatók a megadott útvonalhoz tartozó könyvtárak.

13.4. Mely állítások igazak? Jelölje be az összes helyes állítást!

- a) A `java.io.FilenameFilter` egy absztrakt osztály.
- b) A `FilenameFilter` egyetlen metódusa az `accept`.
- c) A `JFileChooser` állománykiválasztó dialógusban mindenki lehet választani állományt és könyvtárat is.
- d) A `JFileChooser` állománykiválasztó dialógushoz csatolható szűrőobjektum `java.io.FilenameFilter` típusú.

Feladatok

13.1. (A) Írja ki az aktuális mappa abszolút útvonalát, valamint a mappa szülőmappájának abszolút útvonalát! (*MappaVizsg.java*)

13.2. (A) Listázza ki a `c` lemezegység főkönyvtárában található

- a) könyvtárbejegyzéseket és azok legutolsó módosításának dátumát!
- b) a `com` kiterjesztésű bejegyzéseket és az állományok hosszát!

(*Listazas.java*)

13.3. (A) Egy, a program paramétereként megadható könyvtárból törölje ki az összes `~jav` kiterjesztésű (Java forrásprogram biztonsági másolata) állományt! (*Torles1.java*)

13.4. (B) A program paramétereként megadható egy könyvtár és egy vagy több állománykiterjesztés. A könyvtárból törölje ki az összes megadott kiterjesztésű állományt! (*Torles2.java*)

13.5. (B) Vizsgálja meg, hogy létezik-e valamelyik lemezegységének főkönyvtárában jdk-val kezdődő könyvtár! (*JDKKereses.java*)

13.6. (A) Listázza ki egy adott könyvtár JPG kiterjesztésű állományait! (*ListJpg.java*)

- 13.7. **(B)** Egy adott könyvtárban vannak jpg állományok. Nevezze át ezeket az állományokat sorban 1.jpg, 2.jpg, 3.jpg stb. nevűre! (*Atnevezes.java*)
- 13.8. **(A)** Jelenítsen meg egy olyan megnyitási állománydialógust, amelynek aktuális könyvtára az Ön JDK könyvtára és csak a txt és html állományok jelennek meg benne! A könyvtárak között lehet ide-oda lépdelni. Ha a felhasználó kiválasztott egy állományt, akkor a dialógus írja ki a kiválasztott könyvtár és állomány nevét! (*SzuroDialog.java*)
- 13.9. **(B)** Készítsen egy bejegyzésvizsgáló programot! A felhasználó kiválaszthat egy tetszőleges bejegyzést, s mi egy szövegterületen megjelenítjük annak a bejegyzésnek a tulajdonságait: abszolút útvonal, könyvtár, állománynév, méret, utolsó módosítás dátuma, írható/olvasható stb.! (*BejegyzésVizsgalat.java*)
- 13.10. **(C)** Készítsen olyan függvényt, amely visszaadja a paraméterként megkapott bejegyzés-hez tartozó állomány vagy könyvtár teljes méretét! (*EntrySize.java*)
- 13.11. **(C)** Készítsünk képnézegető programot! A könyvtár egy állománydialógus segítségével választható ki. A program jelenítse meg sorban a könyvtárban található összes jpg és gif kiterjesztésű képállomány képét, nevét és méretét! A képek között lehessen előre-hátra lapozni! (*KepNezegeto.jpx*)

14. Folyamok

A fejezet pontjai:

1. A folyam fogalma
 2. Bájtfolyam
 3. Karakterfolyam, szöveges állomány
 4. Adatfolyam
 5. Pufferező folyam
 6. Objektumfolyam
-

Gyakori eset, hogy a programnak egy adatsorozatot (bájtokat, karaktereket, primitív adatokat, objektumokat stb.) kell fogadnia valamely adattárból (például állományból), vagy adatokat kell oda eljuttatnia. A forrás-, illetve célhely elvileg bármi lehet – a billentyűzet például forráshely, a konzol célhely; egy állomány vagy a memória lehet forrás- és célhely is. Az adatsorozat tárolása lehet maradandó (perzisztens) vagy ideiglenes. A Javában a folyamobjektum (stream) arra való, hogy sorban egymás után adatokat hozzon be a tárolóból, illetve adatokat írjon ki rá. A folyam csak egy irányba mozgathatja az adatokat: vagy folyamatosan olvas, vagy folyamatosan ír. Az olvasás, illetve az írás egysége lehet bájt, karakter, primitív adat vagy objektum. A fejezetben szó lesz bájtfolyamokról, karakterfolyamokról, objektumfolyamokról és szűrőfolyamokról. A könyv elsősorban olyan folyamokkal foglalkozik, amelyeknek egy állomány a „végállomásuk”.

14.1. A folyam fogalma

A **folyam** (**stream**) olyan objektum, amely sorosan adatokat írhat egy célhelyre, illetve sorosan adatokat olvashat egy forráshelyről. Aszerint, hogy a folyam milyen típusú elemeket kezel, a következő folyamokat különböztetjük meg egymástól:

- **Bájtfolyam** (byte stream): Az írás/olvasás egysége a bájt, vagyis egy 0 és 255 közötti érték. A bájtfolyam bájtokat olvas, illetve ír. minden adatsorozat feldolgozható bájtfolyamként. Egy állomány tartalmát például úgy másolhatjuk át egy másik állományba, hogy a forrásállomány tartalmát bájtonként „befolyatjuk” a memóriába, majd onnan

„kifolyatjuk” azt az eredményállományba. A bájtfolyamok absztrakt őse az `InputStream` és az `OutputStream`. A folyamosztályokat a 14.1. ábra mutatja.

- **Karakterfolyam** (character stream): Az írás/olvasás egysége az unikódos karakter, azaz két bájt. A karakterfolyam karaktereket olvas, illetve ír. A célhelyen való tárolás a karakterkódolási szabvánnyal függ (a Windows például egy karaktert egyetlen bájton tárol). A szöveges (megjeleníthető karakterekből álló) állományokat karakterfolyammal szokás feldolgozni. A karakterfolyamok absztrakt őse a `Reader` és a `Writer`.
- **Adatfolyam** (data stream): Az olvasás/írásnak a típusos adatok az egységei: primitív adatok (`boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, `double`) és a `String`. Az adatfolyam típusos adatokat olvas/ír. Adatfolyam a `DataInputStream` és a `DataOutputStream`.
- **Objektumfolyam** (object stream): Az olvasás/írás egysége az objektum. Az objektumfolyam szerializálható objektumokat olvas, illetve ír. Egy objektum akkor szerializálható, ha osztálya implementálta a `Serializable` interfész. Objektumfolyam az `ObjectInputStream`, illetve az `ObjectOutputStream`.

Beviteli/kiviteli folyam: Az adatfeldolgozás irányá szerint kétféle folyam van:

- Beviteli folyam (input stream): Csak olvasni tud. Az adatokat kizárolag sorban, egymás után, a tárolás sorrendjében olvashatja.
- Kiviteli folyam (output stream): Csak írni tud. Az adatokat kizárolag sorban, egymás után írhatja (tárolhatja).

A folyam mutatója: A folyamnak van egy aktuális mutatója, ez mutatja, hogy hol tart a folyam az elemek olvasásában, illetve írásában.

Forráshely, célhely: Egy beviteli folyam forráshelye, illetve egy kiviteli folyam célhelye a következő lehet:

- egy fizikai hely (például állomány, konzol, billentyűzet, nyomtató);
- egy másik folyam.

A forráshelyet/célhelyet a folyam létrehozásakor kell megadni. A kapcsolatot a folyam konstruktora határozza meg.

Szűrőfolyamok: A beviteli, illetve kiviteli folyamok egymáshoz kapcsolhatók: az egyik folyam által olvasott vagy írt adatsorozatot egy másik folyam rögtön megkapja és feldolgozza. Ilyenkor a folyam forrása, illetve célhelye egy másik folyam. A továbbító, szűrő szerepet játszó folyamot szűrőfolyamnak (filter stream) nevezzük. A szűrők ősosztálya a `FilterInputStream` és a `FilterOutputStream`; mindenktőnek privát a konstruktora. A szűrőobjektum konstruktornak egy másik folyam a paramétere. A következő folyamok szűrőfolyamok:

- a `Buffered` szóval kezdődő nevű osztályok pufferelnek;
- a `Data` szóval kezdődő nevű osztályok bájtsorozatot konvertálnak primitív adatokká vagy fordítva.

A 14.1. ábra a 13.1. ábra egy kivonata, és a `java.io` csomag lényegesebb folyamosztályait ábrázolja.



14.1. ábra. A `java.io` csomag alapvető stream osztályai

Áttekintésként nézzünk egy példát. A 14.2. ábra két folyamláncot mutat:

- ◆ **Kiviteli folyamlánc** (felül): Az első folyam egy adatfolyam (`dos:DataOutputStream`), s ahhoz egy állományba író bájtfolyam (`FileOutputStream`) kapcsolódik. Az összekapcsolást a konstruktörben végezzük el: az első folyamnak paraméterként megadjuk a második folyamot. A programból az első, `DataOutputStream` folyamra primitív adatokat írunk (`dos.writeDouble`). A `DataOutputStream` folyamnak az a dolga, hogy a primitív adatokat bájtökká alakítsa, majd a célhelyre írja őket. A célhely most a második folyam; annak az a dolga, hogy a kapott bájtokat lemezre írja. A

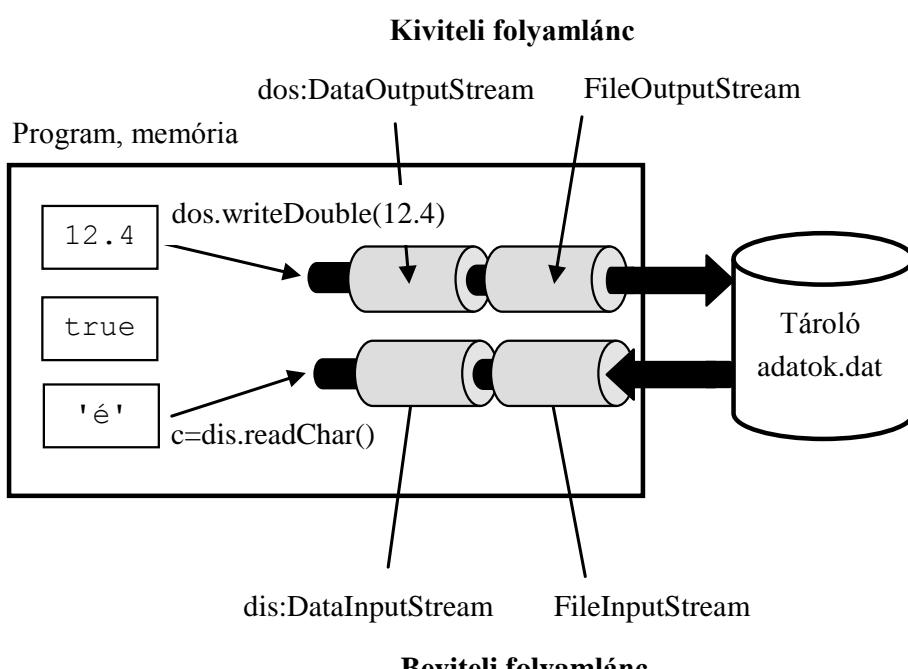
programozó csak az első folyamra ír, s ezzel beindul a gépezet: a kapcsolt folyamok a beléjük táplált módon rögtön továbbítják a kapott adatokat. A megfelelő, primitív adatokat állományba író programrészlet a következő:

```
DataOutputStream dos = new DataOutputStream(
    new FileOutputStream("adatok.dat"));
dos.writeDouble(12.4);
dos.writeBoolean(true);
dos.writeChar('é');
dos.close();
```

- ◆ **Beviteli folyamlánc** (alul): Ez a kiviteli folyamlánc tükörképe. A primitív adatokat az állományból beolvasó programrészlet a következő:

```
DataInputStream dis = new DataInputStream(
    new FileInputStream("adatok.dat"));
double d = dis.readDouble(); // -> 12.4
boolean b = dis.readBoolean(); // -> true
char c = dis.readChar(); // -> 'é'
dis.close();
```

Visszaolvasáskor tisztában kell lennünk azzal, hogy mi van a lemezen – a `readDouble` például fogja a soron következő 8 bajtot, és betölti a megadott változóba. Ezért az adatokat ugyanolyan sorrendben kell visszaolvasni, amilyenben kiírtuk őket!



14.2. ábra. A folyamok áttekintése

14.2. Bájtfolyam

Az `InputStream` az összes beviteli bájtfolyam absztrakt őse, az `OutputStream` pedig az összes kiviteli bájtfolyamé. Mindenekelőtt nézzük sorra ezeket az osztályokat, hiszen a bennük deklarált metódusokat minden bájtfolyamban használhatjuk. A konkrét leszármazott folyam-osztályoknál ezeket a metódusokat már nem fogjuk külön elmagyarázni. Ebben a pontban két konkrét bájtfolyamot tárgyalunk – mindenkor közvetlenül egy állománnyal lép kapcsolatba: a `FileInputStream` bájtokat olvas az állományból, a `FileOutputStream` bájtokat ír az állományba.

InputStream – absztrakt ősosztály

Csomag: `java.io`

Deklaráció: `public abstract class InputStream`

Közvetlen ős: `java.lang.Object`

Az `InputStream` osztály az összes beviteli bájtfolyam absztrakt őse.

Az osztály `read` metódusa absztrakt: a leszármazott osztály ebben a metódusban fogja megadni, hogy a folyam hogyan olvasson be egy bájtot.

Metódusok

- ▶ `abstract int read() throws IOException`

Absztrakt metódus; a leszármazottnak mindenkor implementálnia kell. Beolvass a folyamból egy bájtot. Visszaad egy 0 és 255 közötti értéket, de nem `byte`, hanem `int` típusút: abban a felső bájt értéke 0 lesz. A folyam végén a visszaadott érték -1. A metódus addig vár (blokkolja a program futását), ameddig be nem olvashat egy bájtot, vagy nincs vége a folyamnak, vagy kivétel nem keletkezik. Például a `FileInputStream` osztály `read` metódusa egy állományból olvas be egy bájtot.

- ▶ `int read(byte[] b) throws IOException`
- ▶ `int read(byte[] b, int off, int len) throws IOException`

A folyamból több bájtot olvas be egyszerre, és beteszi őket a `b` bájttömbbe. Az első esetben `b.length` számú bájtot olvas és megtölți a tömböt (ha van annyi bájt a folyamban); a második esetben `len` számú bájtot olvas, és azokat `off`-tól kezdve helyezi el a tömbben. A visszaadott érték a ténylegesen beolvasott bájtok száma. A folyam végéhez érve -1-et ad vissza. A metódus addig vár (blokkolja a program futását), amíg be nem olvashat elegendő számú bájtot, vagy nincs vége a folyamnak, vagy kivétel nem keletkezik. Ha hibás a megadott `off` vagy `len`, akkor `IndexOutOfBoundsException` keletkezhet.

- ▶ `int available() throws IOException`
Visszaadja a pillanatnyilag elérhető – blokkolás nélkül beolvasható (`read`) vagy átugorható (`skip`) – bájtok számát.
- ▶ `long skip(long n) throws IOException`
A stream mutatóját `n` bájttal előbbre viszi.
- ▶ `void close() throws IOException`
Bezárja a folyamot és felszabadít minden hozzá kapcsolt erőforrást. A bezárt folyamról tovább már nem lehet olvasni, és a folyamot nem lehet újra megnyitni (vagyis új objektumot kell létrehozni).

OutputStream – absztrakt ősosztály

Csomag: `java.io`

Deklaráció: `public abstract class OutputStream`

Közvetlen ős: `java.lang.Object`

Az `OutputStream` osztály az összes kiviteli bájtfolyam absztrakt őse.

Az osztály `write` metódusa absztrakt: a leszármazott osztály ebben a metódusban fogja megadni, hogy a folyam hogyan írjon ki egy bájtot.

Metódusok

- ▶ `abstract void write(int b) throws IOException`
Kiírja a célhelyre az `int` típusú paraméter alsó bájtját; a felső bájtot figyelmen kívül hagyja.
- ▶ `void write(byte[] b) throws IOException`
- ▶ `void write(byte[] b, int off, int len) throws IOException`
Kiírja a megadott bájttömböt a célhelyre – második esetben `off`-tól kezdve egy `len` hosszúságú részt. Ha `off` vagy `len` nem esik a megfelelő tartományba, akkor `IndexOutOfBoundsException` keletkezik.
- ▶ `void flush() throws IOException`
Ha a folyam pufferelt, akkor a pufferben összegyűlt adatokat azonnal kiírja a célhelyre.
- ▶ `void close() throws IOException`
Bezárja a folyamot és felszabadít minden hozzá kapcsolt erőforrást. A bezárt folyamra tovább már nem lehet írni, és a folyamot nem lehet újra megnyitni (vagyis új objektumot kell létrehozni).

FileInputStream – állományból olvasó bájtfolyam

Csomag: java.io

Deklaráció: public class FileInputStream

Közvetlen ős: java.io.InputStream

A `FileInputStream` egy beviteli bájtfolyam, az `InputStream` közvetlen leszármazottja. Bájtonként olvastathatjuk vele egy állomány tartalmát. A `read` metódusok a soron következő bájtot vagy bájtokat olvassák be. Ha az olvasás az állomány végére ér, akkor a következő olvasás nem hoz be adatot – ez a `read` visszatérési értékéből kiolvasható (-1), kivétel nem keletkezik.

Ha az állomány nem létezik, a konstruktur `FileNotFoundException` kivételt kelt. A metódusok I/O hiba esetén `IOException` kivételt ejtenek.

FileInputStream

```
+FileInputStream(name: String) {FileNotFoundException}  
+FileInputStream(file: File) {FileNotFoundException}  
+read():int {IOException}  
+read(b: byte[]): int {IOException}  
+read(b: byte[], off: int, len: int) : int {IOException}  
+available(): int {IOException}  
+skip(n: long): long {IOException}  
+close() {IOException}  
...
```

Konstruktörök

- ▶ `FileInputStream(String name)` throws `FileNotFoundException`
- ▶ `FileInputStream(File file)` throws `FileNotFoundException`

Megnyitja a paraméterben megadott állományt, vagyis fizikailag hozzáférhetővé teszi az állomány adatait. Ha nincs ilyen állomány, akkor `FileNotFoundException` kivétel keletkezik. A `FileNotFoundException` kivételestály a `java.io.IOException` közvetlen leszármazottja.

A metódusok leírását lásd az absztrakt `InputStream` osztálynál. A `read` metódusok most állományból olvasnak.

FileOutputStream – állományba író bájtfolyam

Csomag: java.io

Deklaráció: public class FileOutputStream

Közvetlenős: java.io.OutputStream

A **FileOutputStream** egy kiviteli bájtfolyam, az OutputStream közvetlen leszárma-zottja; bájtonként írhatunk vele egy állományba. A write metódusok a soron következő helyre írják a bájto(ka)t.

Ha az állományt nem sikerült létrehozni, akkor a konstrukturor FileNotFoundException kivételt kelt. A metódusok I/O hiba esetén IOException kivételt ejtenek.

FileOutputStream

```
+FileOutputStream(name: String) {FileNotFoundException}
+FileOutputStream(file: File) {FileNotFoundException}
+FileOutputStream(name: String, append: boolean) {FileNotFoundException}
+write(b: int):int {IOException}
+write(b: byte[]): int {IOException}
+write(b: byte[], off: int, len: int) : int {IOException}
+flush() {IOException}
+close() {IOException}
...
...
```

Konstruktorok, metódusok

- ▶ `FileOutputStream(String name) throws FileNotFoundException`
- ▶ `FileOutputStream(File file) throws FileNotFoundException`
- ▶ `FileOutputStream(String name, boolean append) throws FileNotFoundException`

Nyit egy kiviteli folyamot. Létrehozza a paraméterben megadott állományt, és fizikailag hozzáférhetővé teszi az adatok kiírásához. A már létező állományt felülírja, kivéve, ha a hozzáfűzés lehetőségét választjuk: ha a harmadik konstruktorban `append==true`, akkor a kiírás az állomány végétől folytatódik. Ha az állományt valamelyen okból nem lehet létrehozni, FileNotFoundException kivétel keletkezik.

A metódusok leírását lásd az absztrakt OutputStream osztálynál. A write metódusok most állományba írnak.

Állomány másolása

Feladat – CopyFile

Másoljunk át egy megadott állományt ugyanabba a könyvtárba, egy másik állományba; az újabb állomány neve csak annyiban térjen el az eredetiétől, hogy a kiterjesztése előtt legyen egy ~ karakter!

Három megoldást fogunk adni: először egyenként olvassuk és írjuk a bájtokat, másodszor az állományt egy darabban hozzuk be a memóriába és írjuk is ki onnan. A harmadik megoldásban az állományokat „tisztességesebben” zárjuk le.

1. megoldás – bevitel és kivitel bájtonként (CopyFile1) – Forráskód

```
import java.io.*;

public class CopyFile1 {
    public static void main (String args[]) {
        String fSource = "c:/javaprog/work/proba/App.java";
        String fDest; //1
        int index = fSource.lastIndexOf('.');
        fDest = fSource.substring(0,index) +
            ".~"+fSource.substring(index+1);

        try {
            FileInputStream in = new FileInputStream(fSource); //2
            FileOutputStream out = new FileOutputStream(fDest);

            int b; //3
            while ((b = in.read()) != -1) {
                out.write(b);
            }
            in.close(); //4
            out.close();
        }
        catch (IOException e) {
            System.out.println(e.getMessage()); //5
        }
    } // main
} // CopyFile1
```

A program elemzése

- ◆ //1: Meghatározzuk a célútvonalat (fDest) a forrásútvonal (fSource) alapján.
- ◆ //2: A try blokkban deklaráljuk és létrehozzuk a beviteli (in) és kiviteli (out) folyamot.
- ◆ //3: A beolvasott bájtok a b változóba kerülnek. A ciklusban in-ből beolvasható bájt, s mindenki írjuk out-ra. Ez addig megy, amíg in-ben még van beolvasható bájt, vagyis a beolvasással kapott érték nem -1. Megjegyezzük, hogy az int típus felső bájtja beolvasáskor és kiíráskor is kihasználatlan mindaddig, amíg végelet nem olvasunk.
- ◆ //4: Lezárjuk az állományokat.

- ◆ //5: A bevitelkor vagy kivitelkor keletkezett kivételt egyetlen üzenettel intézzük el. Két kivétel keletkezhet: `FileNotFoundException` és `IOException`, de mivel az `IOException` közös ősosztályuk, azért azon minden kivétel „fennakad”.

2. megoldás – bevitel és kivitel egy adagban (CopyFile2)

Az állományt egyszerre beolvashatjuk egy bájttömbbe. Az `in.available` megadja az állomány hosszát, és éppen ilyen hosszú tömböt deklarálunk (feltesszük, hogy az állomány befér a memóriába). A megfelelő `read` és `write` metódussal akárhány bájtot bekérhetünk egyszerre. Ehhez elég kicserélnünk a program //3-as részét a következő utasításokra:

```
byte[] bajtTomb = new byte[in.available()]; //3
in.read(bajtTomb);
out.write(bajtTomb);
```

3. megoldás – biztonságos lezárás (CopyFile3)

A megnyitott állományokat le kell zárni, különben adatot veszíthetünk! Mi a `try` ágban egyszerűen elintéztük a lezárást, de meg kell jegyeznünk, hogy a lezárásnak van egy biztonságosabb, bár enyhén szólva is „macerásabb” módja. Ha az állományokat a `try` ágban zárjuk le, akkor előfordulhat, hogy valamelyik művelet kivételt ejt, s amiatt nem jut idáig a vezérlés, bizonyos megnyitott, „jó” állományok tehát nem zárulelnak le. Szerencsére a folyam `finalize()` metódusa szemétyűtéskor lezárja az esetleg nyitva maradt állományt.

Biztonságosabb az állományokat a `try` blokk `finally` ágában lezárni. Ilyenkor a folyamokat a `try` blokk előtt kell deklárnunk, és kezdőértéket is kell adnunk nekik. Ezenkívül a `close()` meghívása előtt meg kell győzödnünk arról, hogy a folyam egyáltalán létrejött-e. A lezáras maga is kelthet `IOException` kivételt, ezért a `finally` ágban újabb `try-catch` blokkot kell alkalmaznunk:

```
FileInputStream in = null;
FileOutputStream out = null;
try {
    in = new FileInputStream(fSource);
    out = new FileOutputStream(fDest);
    byte[] bajtTomb = new byte[in.available()];
    in.read(bajtTomb);
    out.write(bajtTomb);
}
catch (IOException e) {
    System.out.println(e.getMessage());
}
finally {
    try {
        if (in != null)
            in.close();
        if (out != null)
            out.close();
    }
}
```

```
        catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
```

14.3. Karakterfolyam, szöveges állomány

A Reader absztrakt őse az összes beviteli karakterfolyamnak, a Writer pedig az összes kiviteli karakterfolyamnak. Először is lássuk ezt a két osztályt!

Reader – absztrakt ősosztály

Csomag: java.io

Deklaráció: public abstract class Reader

Közvetlen ős: java.lang.Object

A Reader osztály az összes beviteli karakterfolyam absztrakt őse.

Az osztály háromparaméteres `read` metódusa absztrakt: az utódosztály ebben a metódusban adja meg, hogy a folyam hogyan olvasson karaktereket. Absztrakt a `close` metódus is.

Metódusok

- ▶ `int read() throws IOException`

Beolvás a folyamból egy karaktert. Visszaad egy `int` típusú unikódos értéket (0 és 16383 között). A folyam végén a visszaadott érték -1. A metódus addig vár (blokkolja a program futását), amíg be nem olvashat egy karaktert, a folyamnak nincs vége, vagy kivétel nem keletkezik. Az állomány végénél nem keletkezik kivétel.

- ▶ `int read(char[] cbuf) throws IOException`
- ▶ `abstract int read(char[] cbuf, int off, int len) throws IOException`

Az első metódus beolvás `cbuf.length` számú karaktert a folyamból, vagyis megtölti az egész tömböt (ha van annyi bájt a folyamban). A második metódus `len` számú karaktert olvas be, és azt a `cbuf` tömbbe teszi, az `off` pozíciótól kezdve. A visszaadott érték a ténylegesen beolvastott karakterek száma. A folyam végéhez érve -1-et ad vissza. Ha az `off` vagy a `len` értéke hibás, akkor `IndexOutOfBoundsException` keletkezhet.

- ▶ `boolean ready() throws IOException`
A viszaadott érték `true`, ha van blokkolás (várakozás) nélkül beolvasható karakter, egyébként `false`.
- ▶ `long skip(long n) throws IOException`
A folyam pozíóját `n` karakterrel előrébb viszi, vagyis `n` karaktert kihagy az olvasásból.
- ▶ `abstract void close() throws IOException`
Bezárja a folyamot és felszabadít minden hozzá tartozó külső erőforrást.

Writer – absztrakt űsosztály

Csomag: java.io

Deklaráció: public abstract class Writer

Közvetlen ős: java.lang.Object

A **Writer** osztály az összes kiviteli karakterfolyam absztrakt őse.

Az osztály háromparaméteres `write` metódusa absztrakt: a leszármazott osztály ebben a metódusban fogja megadni, hogy a folyam hogyan írjon ki karaktereket. Absztrakt metódus a `flush` és a `close` is.

Metódusok

- ▶ `void write(int c) throws IOException`
Kiírja az int típusú unikódos karaktert (2 bajtot) a célhelyre.
- ▶ `void write(char[] cbuf) throws IOException`
- ▶ `abstract void write(char[] cbuf, int off, int len) throws IOException`
Kiírja a megadott karaktertömböt a célhelyre – a második esetben `off`-tól kezdve egy `len` hosszúságú részt. Ha hibás `off` vagy `len` értéket adunk meg, akkor `IndexOutOfBoundsException` keletkezhet.
- ▶ `void write(String str) throws IOException`
- ▶ `void write(String str, int off, int len) throws IOException`
Kiírja a célhelyre a megadott karakterláncot; a második esetben `off`-tól kezdve egy `len` hosszúságú részt.
- ▶ `abstract void flush() throws IOException`
Ha a folyam pufferelt, akkor a pufferben összegyűlt adatokat azonnal kiírja a célhelyre.
- ▶ `abstract void close() throws IOException`
Lezárja a célhelyet.

Szöveges állomány

Szöveges állomány: A csak olvasható karaktereket tartalmazó állományokat szöveges állományoknak nevezzük. A szöveges állomány sorokból áll, a sorok pedig karakterekből. minden sor végén egy-egy „sor vége” jel (`<EOLN>` = End Of Line) található. Az állományt az „állomány vége” jel (`<EOF>` = End Of File) zárja. Az `<EOLN>` és az `<EOF>` logikai jelölések, fizikai megfelelőjük függ az operációs rendszertől.

A szöveges állomány szervezése soros, vagyis a sorokat, illetve karaktereket csak egymás után lehet elérni, és az állományt nem lehet egyszerre írni és olvasni. Az állománynak van egy mutatója: az mondja meg, hogy hol áll éppen az olvasás, illetve az írás.

A szöveges állomány logikai és fizikai szerkezetét a 14.3. ábra mutatja.

Windowsban a sorvégjelek és az állományvégjelek a következők:

- ◆ <EOLN> (sorvégjel): egy CR és egy LF karakter egymás után. CR = #13 (Carriage Return = kocsi vissza); LF = #10 (Line Feed = soremelés).
- ◆ <EOF> (állomány-végjel): Ctrl-Z = #26 karakter.

Az ábrán az állomány háromsoros: az első sor négy betűből áll: „ELSÓ”; a második sor egyetlen számból áll; a harmadik és egyben az utolsó sor az „UTOLSÓ” szövegből.

Állománymutató

Logikai szerkezet:



E	L	S	Ő	<EOLN>	2	<EOLN>	U	T	O	L	S	Ó	<EOLN>	<EOF>
---	---	---	---	--------	---	--------	---	---	---	---	---	---	--------	-------

Fizikai szerkezet (a Windows operációs rendszerben):

E	L	S	Ő	#13	#10	2	#13	#10	U	T	O	L	S	Ó	#13	#10	#26
---	---	---	---	-----	-----	---	-----	-----	---	---	---	---	---	---	-----	-----	-----

14.3. ábra. A szöveges állomány logikai és fizikai szerkezete

A Javában a szöveges állományokat két karakterfolyam testesíti meg: a `FileReader` karaktereket olvas a szöveges állományból; a `FileWriter` karaktereket ír a szöveges állományba.

InputStreamReader, OutputStreamWriter – karakterkódolás

A `FileReader` a szöveges állományból unikódos karaktersorozatot gyárt, a `FileWriter` pedig unikódos karaktereket ír ki. A szöveges állományok tárolási formátuma azonban függ az operációs rendszertől. A lemezen a karakterek nem minden operációs rendszerben unikódos karakterek – a Windows például a karektort egy bájton tárolja. **A lemezen tárolt karakter és a Java karakterei között kódolás/dekódolás szükséges.** Ezt a feladatot látja el a `FileReader` közvetlen őse, az `InputStreamReader`, illetve a `FileWriter` közvetlen őse, az `OutputStreamWriter`. Ezek az osztályok alakítják át a beolvasott bájtokat karaktersorozattá a megadott, vagy az operációs rendszer alapértelmezés szerinti karakterkódolási szabályának megfelelően. Nekünk elég azzal tisztában lennünk, hogy a tárolt állomány szabványos-e vagy sem – hacsak nem akarjuk megváltoztatni az alapértelmezés szerinti kódolást.

FileReader – állományból olvasó karakterfolyam

Csomag: java.io

Deklaráció: public class FileReader

Közvetlen ős: java.io.InputStreamReader

A **FileReader** karaktereket olvas egy (szöveges) állományból.

FileReader
+FileReader(fileName: String) {FileNotFoundException}
+FileReader(file: File) {FileNotFoundException}
+read():int {IOException}
+read(cbuf: char[]): int {IOException}
+read(cbuf: char[],off: int, len: int): int {IOException}
+ready(): boolean {IOException}
+skip(n: long): long {IOException}
+close() {IOException}
...

Konstruktőrök

- ▶ FileReader(String fileName) throws FileNotFoundException
- ▶ FileReader(File file) throws FileNotFoundException

Létrehoz egy beviteli karakterfolyamot, vagyis kapcsolatot teremt a megadott szöveges állománnyal. Ha az állomány nem létezik, FileNotFoundException kivétel keletkezik.

A metódusok leírását lásd a Reader absztrakt osztály leírásánál.

FileWriter – állományba író karakterfolyam

Csomag: java.io

Deklaráció: public class FileWriter

Közvetlen ős: java.io.OutputStreamWriter

A **FileWriter** karaktereket ír egy (szöveges) állományba.

FileWriter
<pre>+FileWriter(fileName: String) {IOException} +FileWriter(file: File) {IOException} +FileWriter(String fileName, boolean append) {IOException} +write(c: int) {IOException} +write(cbuf: char[]) {IOException} +write(cbuf: char[], off: int, len: int) {IOException} +write(str: String) {IOException} +write(str: String, off: int, len: int) {IOException} +flush() {IOException} +close() {IOException}</pre>

Konstruktörök

- ▶ `FileWriter(String fileName) throws IOException`
- ▶ `FileWriter(File file) throws IOException`
- ▶ `FileWriter(String fileName, boolean append) throws IOException`

Létrehoz egy kiviteli karakterfolyamot, vagyis kapcsolatot teremt a megadott szöveges állománnyal. A már létező állományt felülírja, kivéve, ha a hozzáfűzés lehetőségét választjuk: ha a harmadik konstruktőrben `append==true`, akkor a kiírás az állomány végétől folytatódik. Ha az állományt valamilyen okból nem lehet létrehozni, akkor `FileNotFoundException` kivétel keletkezik.

A metódusok leírását lásd a `Writer` absztrakt osztály leírásánál.

Megjegyzés: A `FileReader` és a `FileWriter` osztályban nincsenek `readLine`, illetve `writeline` metódus, vagyis ezek az osztályok nem kezelnek sorokat. Sorok olvasásához a `BufferedReader` és `BufferedWriter` osztályt használhatjuk.

Feladat – AppendAndList

Készítsen egy programot, amellyel egy szöveges állományt lehet bővíteni (első alkalommal létrehozni), majd a konzolra kilistázni! Bővítéskor konzolról viszünk fel sorokat – végejel az üres szöveg. A szöveges állomány legyen a "work/Szoveg.txt"!

Forráskód

```
import java.io.*;
import extra.Console; //1

public class AppendAndList {
    private String path;

    public AppendAndList(String path) { //2
        this.path = path;
        appendText();
        listText();
    }
}
```

```

void appendText() { //3
    try {
        FileWriter fw = new FileWriter(path,true);
        String line;
        while (!(line = Console.readLine("Sor: ")).equals("")) {
            fw.write(line+"\r\n");
        }
        fw.close();
    }
    catch (IOException ex) {
        System.out.println(ex);
    }
}

void listText() { //4
    try {
        FileReader fr = new FileReader(path);
        System.out.println("Lista:");
        while (fr.ready())
            System.out.print((char)fr.read());
        fr.close();
    }
    catch (IOException ex) {
        System.out.println(ex);
    }
}

public static void main (String args[]) {
    new AppendAndList("work/Szoveg.txt");
} // main
} // AppendAndList

```

A program elemzése

- ◆ //1: Használni fogjuk a `Console.readLine()` metódust; azzal a `System.in` szabványos beviteli folyamról olvasunk be egy sort. Tanulmányozza a metódus forráskódját!
- ◆ //2: Az objektum megjegyzi az írandó és olvasandó állomány útvonalát. Nincs más dolga, mint először kibővíteni a állományt, majd az egész állományt rögtön kilistázni.
- ◆ //3: Az állomány bővítése: Az `fr` folyamot bővítésre nyitjuk meg (`append==true`). A ciklusban konzolról bekérünk egy-egy sort, egészen addig, amíg már csak az Entert ütik le. A `String` nem tartalmazza a sorvégejét, azt nekünk kell a sor karakterei után az állományba írni.
- ◆ //4: Az állomány listázása: Az olvasás karakterenként megy mindaddig, amíg van még karakter (`ready`). A karaktert a `read` metódus `int`-ként adja vissza, s ennek a karakterformáját írjuk ki a konzolra.

PrintWriter

Csomag: java.io

Deklaráció: public class PrintWriter

Közvetlen ōs: java.io.Writer

A **PrintWriter** egy karakterfolyam, amely számos print és println metódust definiál. Közreműködésével a különböző típusú primitív adatok és objektumok olvasható formában kerülnek ki a karakterfolyamra. Ha a PrintWriter-t egy FileWriter-hez kapcsoljuk, akkor a karakterek egy szöveges állományba kerülnek.

A print és println metódusokat a System.out.println-nél már megszoktuk. A System.out és a System.err objektumot a rendszer automatikusan hozza létre, osztályuk a PrintStream leszármazottja. A PrintStream a FileWriter-hez hasonló funkciókat ad, de a **PrintStream** elavult osztály, az ó helyébe lépett a **PrintWriter**. A PrintStream már csak a System.out és System.err szabványos eszközre van fenntartva.

Konstruktörök

- ▶ PrintWriter(OutputStream out)
- ▶ PrintWriter(Writer out)
- ▶ ...

Metódusok (kivonat)

- ▶ print(<type> value) // type bármilyen primitív típus lehet
- ▶ print(Object obj) // obj bármilyen objektum lehet
- ▶ print(String str)
- ▶ print(char[] chars)
- ▶ println(<type> value)
- ▶ println(Object obj)
- ▶ println(String str)
- ▶ println(char[] chars)
- ▶ println()
- ▶ ...

A PrintWriter használatát egy programmal tesszük szemléletessé.

Feladat – PrintWriterTest

Írunk ki adatokat a "print.txt" állományba a print és println metódusokkal ugyanúgy, ahogy a System.out-ra szoktunk írni! Ezután olvassuk vissza az állományt, és írjuk ki a konzolra!

Forráskód

```
import java.io.*;  
  
public class PrintWriterTest {  
    public static void main(String[] args) {
```

```
try {
    PrintWriter writer = new PrintWriter(
        new FileWriter("work/print.txt"));
    writer.println("Double: "+12.78);
    writer.print('A');
    writer.println(new Integer(66));
    writer.close();

    FileReader reader = new FileReader("work/print.txt");
    while (reader.ready()) {
        char ch = (char)reader.read();
        System.out.print(ch);
    }
    reader.close();
}
catch (IOException ex) {
}
}
```

A program futása

```
| Double: 12.78
| A66
```

14.4. Adatfolyam

A DataInput és a DataOutput interfész a primitív típusú adatok és a String írását és olvasását segíti. A beolvasó metódusok az állomány végénél EOFException kivételt ejtenek. Az adatfolyamokon kívül az objektumfolyamok is implementálják ezeket az interféseket.

Az interfészek tárgyalása előtt tisztáznunk kell az **UTF-8-as karakterkódolás** fogalmát. Az UTF-8-as egy unikód-átviteli formátum (Unicode Transfer Format); segítségével egy unikódos karaktersorozat veszteségmentesen átalakítható ASCII karakterekké és vissza. Az UTF-8-as formátumú karakterlánc első két bájtja tartalmazza az UTF lánc hosszát bájtokban; ezután következnek sorban a karakterek reprezentációi. minden unikódos karakter 1, 2 vagy 3 bájton van tárolva; hogy hányon, azt az első karakter mondja meg (ha például az első bájt 0 és 127 közé esik, akkor a reprezentáció egybájtos, és a karakter egy ASCII karakter). A konkrét formátummal most nem foglalkozunk, de annyit érdemes tudni, hogy az UTF-8-as karakterkódolás az unikódos karaktersorozatoknak gyakori tömörítési módja. Ha a karakterlánc karaktereinek többsége ASCII karakter (kódja 0 és 127 közé esik), akkor az UTF-8-as karakterlánc nagyon gazdaságosan fogja tárolni őket.

DataInput interfész

Csomag: java.io

Deklaráció: public interface DataInput

Metódusok

- ▶ boolean readBoolean() throws IOException
- ▶ byte readByte() throws IOException
- ▶ char readChar() throws IOException
- ▶ int readInt() throws IOException
- ▶ long readLong() throws IOException
- ▶ float readFloat() throws IOException
- ▶ double readDouble() throws IOException
- ▶ String readLine() throws IOException

Beolvassa a folyamból a típusnak megfelelő számú bajtot. A függvény visszatérési értéke az adott típusú érték lesz. String típusból sorvégjelig olvas. Ha az állomány végéhez ér, EOFException keletkezik. Egyéb I/O hiba esetén IOException keletkezhet.

- ▶ String readUTF() throws IOException

Beolvas a folyamból egy UTF-8-as formátumú karaktersorozatot, és unikódos karaktereket tartalmazó Stringgé alakítja. Ha az állomány végéhez ér, EOFException keletkezik. Ha a folyamon található karakersorozat nem UTF-8-as formátumú, akkor UTFDataFormatException keletkezik. Egyéb I/O hiba esetén IOException keletkezhet.

DataOutput interfész

Csomag: java.io

Deklaráció: public interface DataOutput

Metódusok

- ▶ void write(int b) throws IOException
- ▶ void write(byte[] b) throws IOException
- ▶ void write(byte[] b, int off, int len) throws IOException
- ▶ void writeBoolean(boolean v) throws IOException
- ▶ void writeByte(int v) throws IOException
- ▶ void writeShort(int v) throws IOException
- ▶ void writeChar(int v) throws IOException
- ▶ void writeInt(int v) throws IOException
- ▶ void writeLong(long v) throws IOException
- ▶ void writeFloat(float v) throws IOException
- ▶ void writeDouble(double v) throws IOException
- ▶ void writeBytes(String s) throws IOException
- ▶ void writeChars(String s) throws IOException

Bájtokká konvertálja a megfelelő adatot és kiírja a folyamra. I/O hiba esetén IOException keletkezhet.

- ▶ String writeUTF() throws IOException

Kiírja a karakterláncot UTF-8-as formátumban a folyamra. I/O hiba esetén IOException keletkezhet.

FilterInputStream, FilterOutputStream

A **FilterInputStream** és a **FilterOutputStream** a szűrőfolyamok ősosztálya. A szűrőfolyamok továbbító, szűrő szerepet játszanak. A szűrőfolyam konstruktorának paramétere mindenkor egy másik folyam. A szűrőfolyam által olvasott vagy írt adatsorozatot a másik folyam rögtön megkapja és feldolgozza.

DataInputStream

Csomag: `java.io`

Deklaráció: `public class DataInputStream`

Közvetlen ős: `java.io.FilterInputStream`

Fontosabb implementált interfész: `DataInput`

A **DataInputStream** egy beviteli szűrő adatfolyam. Egy tetszőleges adatfolyamról bájtokat olvas be, s azokat primitív típusú változókká vagy `String` típusú objektumokká konvertálja. Az adatfolyam csak egy másik folyamból olvashat.

Konstuktor

- ▶ `DataInputStream(InputStream in)`

A metódusok leírását lásd a `DataInput` interfész leírásánál.

DataOutputStream

Csomag: `java.io`

Deklaráció: `public class DataOutputStream`

Közvetlen ős: `java.io.FilterOutputStream`

Fontosabb implementált interfész: `DataOutput`

A **DataOutputStream** egy kiviteli szűrő adatfolyam. Primitív típusú változókat és `String` típusú objektumokat bájtokká alakít, és kiírja őket egy tetszőleges adatfolyamra. Az adatfolyam csak egy másik folyamba írhat.

Konstuktor

- ▶ `DataOutputStream(OutputStream out)`

A metódusok leírását lásd a `DataOutput` interfész leírásánál.

Feladat – Egész számok írása, olvasása (SzamIrOlvas)

Kérjünk be a felhasználótól `int` típusú értékeket, és írjuk fel őket egy állományba!

Ezután listázzuk ki az egészeket a konzolra és vele párhuzamosan egy szöveges állományba is!

Forráskód

```
import java.io.*;
import extra.Console;

public class SzamIrOlvas {

    static void felvisz(String fileName) {
        try {
            int szam = 0;
            System.out.println("Egesz szamok felvitele. Vegjel: -1");
            DataOutputStream szamok=new DataOutputStream(
                new FileOutputStream(fileName+".dat"));           //1

            while ((szam=Console.readInt()) != -1)
                szamok.writeInt(szam);                         //2
            szamok.close();
        }
        catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }

    static void listaz(String fileName) {
        DataInputStream szamok = null;                      //3
        PrintWriter lista = null;
        int szam = 0;

        try {
            szamok=new DataInputStream(
                new FileInputStream(fileName+".dat"));
            lista=new PrintWriter(
                new FileOutputStream(fileName+".txt"));
            while (true) {
                szam = szamok.readInt();                     //4
                System.out.print(szam+" ");
                lista.print(szam+" ");                       //5
            }
        }
        catch (EOFException e) {
        }
        catch (IOException e) {
            System.out.println(e.getMessage());
        }
        finally {
            try {
                szamok.close();
                lista.close();
            }
            catch(IOException e) {
            }
        }
    }
}
```

```

public static void main (String[] args) {
    String fNev="work/egeszek";
    felvisz(fNev);
    listaz(fNev);
}
} // SzamIrOlvas

```

A program elemzése

- ◆ //1: A `felvisz` metódus fogja felírni a számokat. Létrehozunk egy állományhoz csatolt `DataOutputStream`-objektumot.
- ◆ //2: Az adatfolyamra a `writeInt` metódussal írogatjuk fel az egész számokat.
- ◆ //3: A `listaz` metódus olvasni fogja az előzőleg megírt állományt. Létrehozunk egy, a megírt állományhoz csatolt `DataInputStream`-objektumot. A `PrintWriter`-objektum szöveges állományt készít a számokból, azt tehát egy `txt` állományhoz csatoljuk.
- ◆ //4, 5: A ciklusban a `readInt` metódussal beolvassunk egy-egy egész számot a beviteli állományból, majd a `print` metódussal felírjuk a szöveges állományba.

14.5. Pufferező folyam

A **pufferező folyamnak** az a szerepe, hogy csökkentse az író/olvasó műveletek számát a memória és egy külső erőforrás (pl. állomány) között. Evégett bevitelkor a folyam előre beolvashat egy bájtsorozatot, kivitelkor pedig egyszerre írja ki az összegyűjtött bájtokat a fizikai helyre. Pufferező folyam a `BufferedInputStream`, a `BufferedOutputStream`, a `BufferedReader` és a `BufferedWriter`.

BufferedInputStream, BufferedOutputStream

A `BufferedInputStream` bájtfolyamot pufferező beviteli szűrőfolyam. Őse a `FilterInputStream`.

A `BufferedOutputStream` bájtfolyamot pufferező kiviteli szűrőfolyam. Őse a `FilterOutputStream`.

Konstruktörök

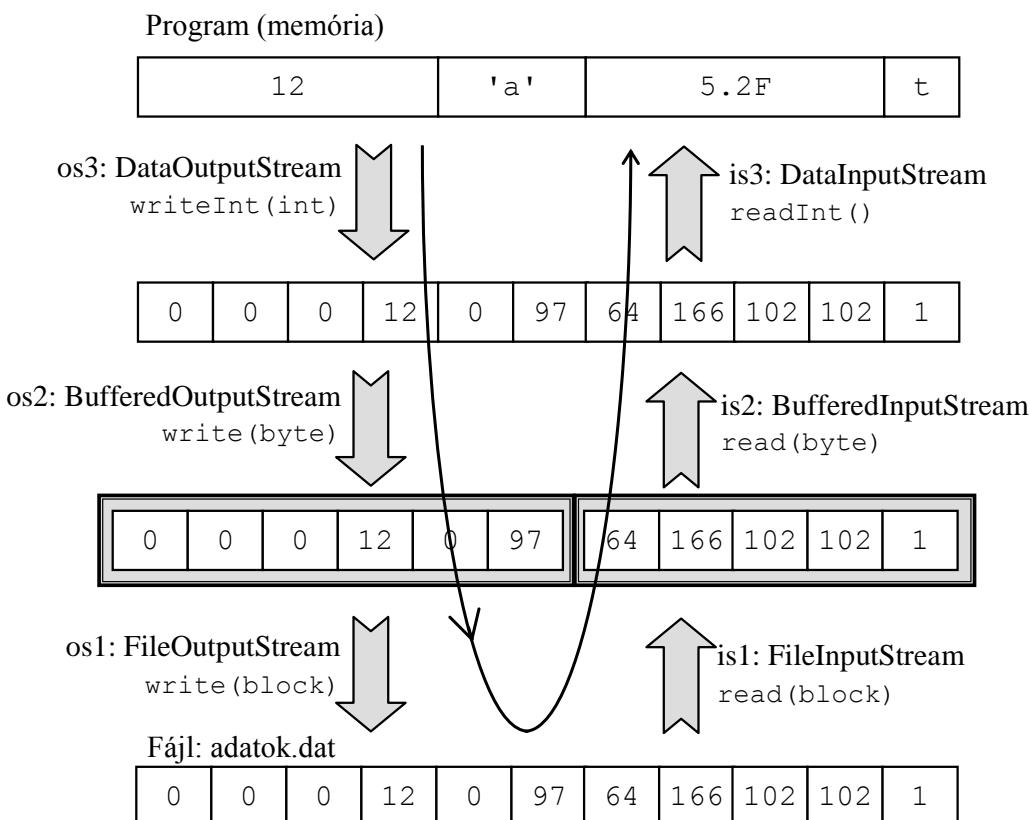
- `BufferedInputStream(InputStream in)`
- `BufferedInputStream(InputStream in, int size)`

Létrehoz egy, az `in`-ről olvasó pufferező szűrőfolyamot. A puffer mérete `size`, alapértelmezésben 2048.

- `BufferedOutputStream(OutputStream out)`
- `BufferedOutputStream(OutputStream out, int size)`

Létrehoz egy, az `out`-ra író pufferező szűrőfolyamot. A puffer mérete `size`, alapértelmezésben 2048.

A következő példában egy adatfolyamot kötünk egy pufferező folyamhoz, azt pedig egy "adatok.dat" nevű állományhoz. Először adatokat írunk a folyamra, majd visszaolvassuk őket. A folyamok egymáshoz kapcsolását és a pufferezés technikáját a 14.4. ábra szemlélteti.



14.4. ábra. Folyamok egymáshoz kapcsolása, pufferezés

Adatok kiírása: A kiviteli folyam vagy külső erőforráshoz kapcsolódik, vagy egy másik folyamhoz – hogy pontosan mihez kapcsolódhat, azt a folyam konstruktora határozza meg. Példánkban a `DataOutputStream`hez egy `BufferedOutputStream`et kapcsolunk, azt meg egy, az "adatok.dat" állományhoz kötött `FileOutputStream`hez. Az `os3` adatfolyamra primitív típusú adatokat írunk, például így: `os3.writeInt(12)`. Az `os3` folyam bájtokat készít a primitív adatokból, és azokat továbbítja az `os2` folyamnak. `os2` a bájtokat blokkosítja, és a blokkokat `os1`-nek adja át. `os1` a végállomás,ő az adatokat lemezre írja:

```
FileOutputStream os1 = new FileOutputStream("adatok.dat");
BufferedOutputStream os2 = new BufferedOutputStream(os1);
DataOutputStream os3 = new DataOutputStream(os2);
```

```
os3.writeInt(12);
os3.writeChar('a');
os3.writeFloat(5.2F);
os3.writeBoolean(true);
os3.close();
```

A három helyett egyetlen utasítással is létrehozhatjuk a folyamokat (ez a követendő módszer):

```
DataOutputStream os3 = new DataOutputStream(new
    BufferedOutputStream(new FileOutputStream("adatok.dat")));
```

Adatok visszaolvasása: A beviteli folyam vagy külső erőforráshoz kapcsolódik, vagy egy másik folyamhoz – hogy pontosan mihez kapcsolódhat, azt a folyam konstruktora határozza meg. Példánkban a DataInputStreamhez egy BufferedInputStreamet kapocsolunk, azt meg egy, az "adatok.dat" állományhoz kötött FileInputStreamhez. Az is3 adatfolyamról primitív típusú adatokat olvasunk, például így: int szam=is3.readInt(). Az is3 szűrőfolyam az is2 folyamtól kéri el a primitív adat összeállításához szükséges bajtokat. is2 rendelkezésre bocsátja a bajtokat. Ha kifogyott a buffere, akkor újabb adag bajtot kér is1-től. is1 a kért bajtokat lemezről fogja beolvasni:

```
FileInputStream is1 = new FileInputStream("adatok.dat");
BufferedInputStream is2 = new BufferedInputStream(is1);
DataInputStream is3 = new DataInputStream(is2);
int i = is3.readInt();
char c = is3.readChar();
float f = is3.readFloat();
boolean b = is3.readBoolean();
```

A három helyett egyetlen utasítással is létrehozhatjuk a folyamokat (ez a követendő módszer):

```
DataInputStream is3 = new DataInputStream(new
    BufferedInputStream(new FileInputStream("adatok.dat")));
```

BufferedReader

A BufferedReader karakterfolyam pufferező beviteli folyam. Őse a Reader.

Konstruktorok, metódusok

- BufferedReader(Reader in)
- BufferedReader(Reader in, int size)

Létrehoz egy, az in-ről olvasó pufferező karakterfolyamot. A puffer mérete size, alapértelmezésben 2048.
- String readLine() throws IOException

Beolvashat egy sort a szövegből. A sor végét az LF ('\n') vagy a CR ('\r') karakter adja meg, vagy a CR+LF karakterpár. A függvény visszaad egy sorvégjel nélküli karakterláncot; illetve a null értéket, ha a folyamnak vége van.

BufferedWriter

A BufferedWriter karakterfolyam pufferező kiviteli folyam. Őse a Writer.

Konstruktorok, metódusok

- ▶ `BufferedWriter(Writer out)`
- ▶ `BufferedWriter(Writer out, int size)`
Létrehoz egy, az `out`-ra író pufferező karakterfolyamot. A puffer mérete `size`, alapértelmezésben 2048.
- ▶ `void newLine() throws IOException`
Ír egy sorvégjelet. A sorvégjel rendszerfüggő, azt a `line.separator` rendszerjellemző (System property) definiálja.

Feladat – Sorok másolása

Írunk olyan programot, amely egy szöveges állomány sorait átmásolja egy másik szöveges állományba, de minden átmásolt sor után kiír egy csillagot tartalmazó sort is! A másolás végén listázzuk ki konzolra a forrás- és a célállományt! Legyen most a forrásállomány a "work/Szoveg.txt", a célállomány pedig a "work/Szoveg2.txt"!

Forráskód

```
import java.io.*;  
  
public class SorokMasolasa {  
    String sourcePath = "work/Szoveg.txt";  
    String destPath = "work/Szoveg2.txt";  
    // Szöveges állomány listázása:  
    void listFile(String fileName) {  
        System.out.println("\n"+fileName+" állomány sorai:");  
        String sor;  
        try {  
            BufferedReader br = new BufferedReader(  
                new FileReader(fileName));  
            while (br.ready())  
                System.out.println(br.readLine());  
            br.close();  
        }  
        catch (IOException ex) {  
            ex.printStackTrace();  
        }  
    }  
  
    public SorokMasolasa() {  
        try {  
            BufferedReader br = new BufferedReader(  
                new FileReader(sourcePath));  
            BufferedWriter bw = new BufferedWriter(  
                new FileWriter(destPath));  
            String sor;
```

```

        while (br.ready()) {
            sor = br.readLine();
            bw.write(sor); bw.newLine();
            bw.write("*"); bw.newLine();
        }
        bw.close();
    }
    catch (IOException ex) {
        ex.printStackTrace();
    }
    listFile(sourcePath);
    listFile(destPath);
}

public static void main (String args[]) {
    new SorokMasolasa();
} // main
} // SorokMasolasa

```

A program futása

```

work/Szoveg.txt állomány sorai:
Első sor
Második sor

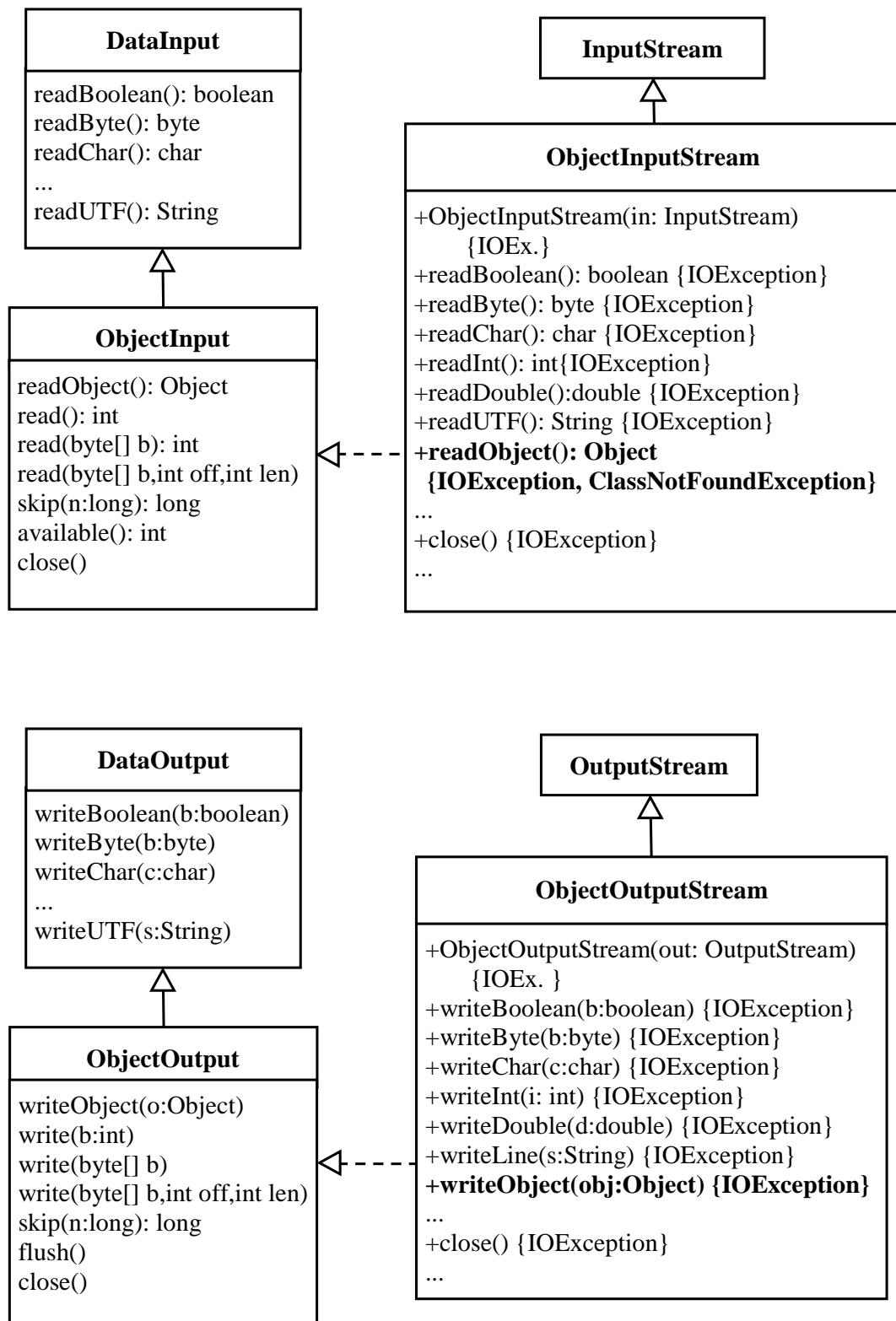
work/Szoveg2.txt állomány sorai:
Első sor
*
Második sor
*
```

14.6. Objektumfolyam

Szerializációval az objektumok eltárolhatók, perzisztenssé tehetők. A szerializációval az objektumból egy bájtsorozat lesz, s abból azután visszaállítható majd az eredeti objektum. Az objektumokat az `ObjectOutputStream`-re kell írni, és az `ObjectInputStream`-ből kell beolvasni. A kiírást a `writeObject` metódus végzi, a beolvasást a `readObject`. Ezek a metódusok automatizálva vannak – tudják, hogy a különféle objektumokat hogyan kell kiírni, illetve beolvasni. A szerializáció elnevezés onnan ered, hogy az algoritmus szigorú sorrendbe állítja az objektum alkotóelemeit – még akkor is, ha azok igen bonyolult fastruktúrát alkotnak.

Az objektumfolyamok implementálják az `ObjectInput`, illetve az `ObjectOutput` interfész (egyszersmind a `DataInput`, illetve a `DataOutput` interfész is). Az objektumoknak szerializálhatóknak kell lenniük: implementálniuk kell a `Serializable` interfész.

Az objektumfolyamokkal kapcsolatos osztályokat a 14.5. ábra mutatja.



14.5. ábra. ObjectInputStream és ObjectOutputStream osztálydiagramja

Az alapértelmezés szerinti szerializáló algoritmus egyetlen `writeObject` utasításra tökéletesen felírja a több szinten egymásba ágyazott objektumszerkezeteket is. Például:

```
Vector v = new Vector();
v.add(...);
JFileChooser fc = new JFileChooser();
fc.setCurrentDirectory(...);
...
ObjectOutputStream out = new ObjectOutputStream(
    new FileOutputStream("work/objektumok.dat"));
out.writeObject(v);
out.writeObject(fc);
out.writeObject("Marci");
out.writeObject(new Integer(5));
out.close();
```

A szerializált objektumok visszaolvashatók a `readObject` utasításokkal, de ügyelni kell a sorrendre és arra, hogy a program ismerje a beolvasott objektum osztályát.

Figyelje meg, hogy a `Vector`, a `JFileChooser`, a `String` és az `Integer` osztály minden szerializálható: deklarációjukban megtalálható az `implements Serializable`!

ObjectInputStream

Csomag: `java.io` Deklaráció: `public class ObjectInputStream`

Közvetlen ōs: `java.io.InputStream`

Fontosabb implementált interfészek: `DataInput`, `ObjectInput`

Az `ObjectInputStream` primitív adatokat és objektumokat olvashat egy bájtfolyamból.

Az objektumokat a `readObject` metódus olvassa.

Konstruktur, metódus

- ▶ `public ObjectInputStream(InputStream in) throws IOException`
Létrehozza a beviteli objektumfolyamot. A folyamhoz csatolja az `in` beviteli bájtfolyamot.
- ▶ `public final Object readObject() throws IOException,`
`ClassNotFoundException`
Beolvass egy objektumot a folyamból. Ha az objektum osztálya nincs meg a programban, akkor `java.lang ClassNotFoundException` kivétel keletkezik.

ObjectOutputStream

Csomag: `java.io` Deklaráció: `public class ObjectOutputStream`

Közvetlen ōs: `java.io.OutputStream`

Fontosabb implementált interfészek: `DataOutput`, `ObjectOutput`

Az `ObjectOutputStream` primitív adatokat és objektumokat is kiírhat egy bájtfolyamra.

Az objektumokat a `writeObject` metódusa írja.

Konstruktur, metódus

- ▶ `public ObjectOutputStream(OutputStream out) throws IOException`
Létrehozza a kiviteli objektumfolyamot. A folyamhoz csatolja az `out` kiviteli bájtfolyamot. `out`-ra sorban ki lehet írni az objektumokat.
- ▶ `public final void writeObject(Object obj) throws IOException`
Kiír egy objektumot a folyamra. Ha az objektum nem implementálta a `Serializable` interfészét, akkor `java.io.NotSerializableException` keletkezik. Ha szerializáció közben kiderül, hogy hibás az osztály, akkor `InvalidClassException` kivétel keletkezik.

Egy objektum szerializálásának szabályai:

- Az objektum osztályának (vagy valamely ősének) implementálnia kell a `Serializable` jelölőinterfészet, egyébként `NotSerializableException` kivétel keletkezik.
- A szerializálhatóság öröklődik. Ha egy osztály implementálja a `Serializable` interfészet, akkor a teljes objektum elmentése az ő dolga, ha szerializálható az őse, ha nem.
- Szerializáláskor a bájtfolyamra kerülnek az osztályt azonosító adatok és az objektum adatai. Az objektum `static` (statikus) és `transient` (ideiglenes) módosítóval ellátott adatai azonban nem íródnak ki!
- Az objektum adatai és belső objektumai a deklarálás sorrendjében íródnak ki, és ez igaz a belső objektumokra is. Így egészen bonyolult, rekurzív objektumgráfok is eltárolhatók. Ha a gráfban ugyanannak az objektumnak több hivatkozása is szerepel, akkor az objektum csak egyszer tárolódik (ennek jóvoltából az algoritmus nem lehet végtelen).
- Van egy alapértelmezés szerinti szerializáló mechanizmus; a programozók általában nem szoktak ettől eltérni, de ha akarnak, eltérhetnek.
- Az objektumokat ugyanolyan sorrendben kell visszaolvasni, ahogyan kiírtuk őket – fontos, hogy a beolvasott objektum osztálya az legyen, amit a program kér. Ha az osztály megváltozott, akkor beolvasáskor `java.lang.ClassNotFoundException` kivétel keletkezik.

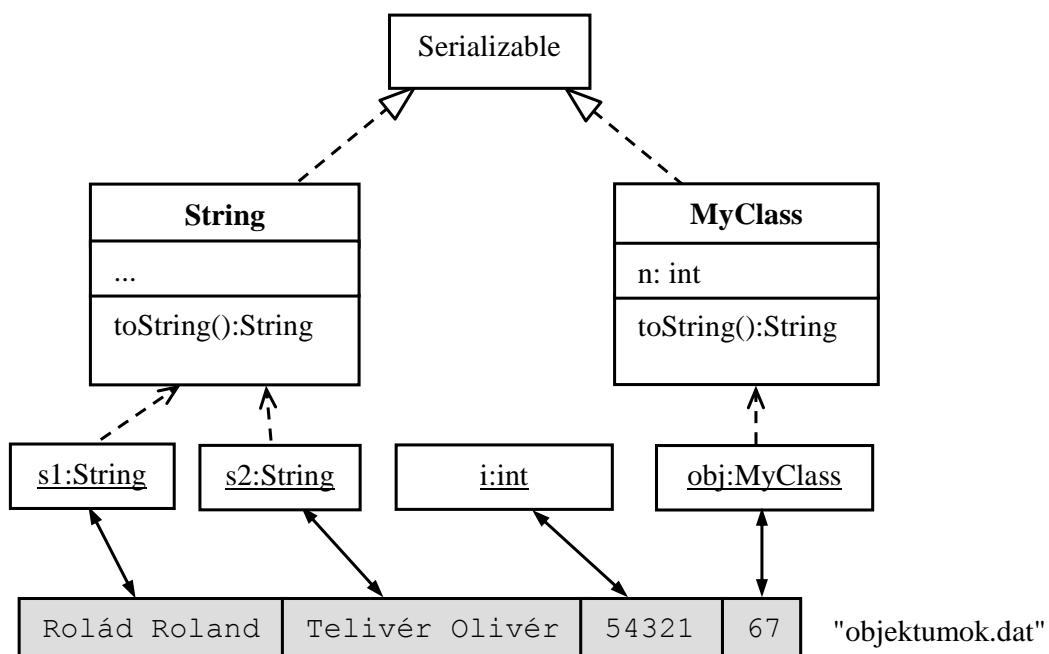
A `Serializable` interfész jelölő szerepet játszik, s egyetlen metódust sem definiál. A szerializáló mechanizmus figyeli, hogy a különféle objektumok osztályai implementálták-e ezt az interfészet; ha nem, akkor `NotSerializableException` kivétel keletkezik. A Java osztályok többsége szerializálható: a Swing-komponensek többsége, a `String`, a csomagolóosztályok, a tömb, a kollekciók (például a `Vector`) stb. A Java írói eredetileg az összes Java osztályt szerializálhatóvá akarták tenni, de kiderült, hogy bizonyos osztályok szerializációja vagy túl bonyolult lenne, vagy nem eléggyé biztonságos.

Objektumok elmentése, majd visszaolvasása

Feladat – ObjectStreamDemo

Írunk ki egy "work/objektumok.dat" állományba két String-objektumot, egy primitív egész számot, és egy saját, MyClass osztályú objektumot. Olvassuk vissza az állomány tartalmát!

A feladat osztályait, objektumait és az állomány hozzávetőleges tartalmát a 14.6. ábra mutatja.



14.6. ábra. SzerIALIZÁCIÓ

Forráskód

```

import java.io.*;

class MyClass implements Serializable {
    int n=67;

    public String toString() {
        return Integer.toString(n);
    }
}

public class ObjStreamDemo {
    static void letrehoz() {
  
```

```
try {
    ObjectOutputStream out = new ObjectOutputStream(
        new FileOutputStream("work/objektumok.dat")); //1
    // A folyamra objektumot és primitív adatot is írunk:
    out.writeObject(new String("Rolád Roland"));
    out.writeObject(new String("Telivér Olivér"));
    out.writeInt(54321); // primitív adat
    out.writeObject(new MyClass());
    out.close();
}
catch (FileNotFoundException ex) {
    System.out.println("Az állományt nem lehet létrehozni!");
}
catch (IOException ex) {
    System.out.println("I/O hiba! "+ex);
}
}

static void listaz() {
try {
    ObjectOutput o;
    ObjectInputStream in = new ObjectInputStream(
        new FileInputStream("work/objektumok.dat"));
    // A visszaolvasás a felírási sorrendben történik:
    String s1 = (String)in.readObject(); //2
    String s2 = (String)in.readObject();
    int i = in.readInt();
    MyClass obj = (MyClass)in.readObject();
    in.close();
    System.out.println(s1+" "+s2+" "+i+" "+obj);
}
catch (FileNotFoundException ex) {
    System.out.println("Nincs ilyen állomány!");
}
catch (ClassNotFoundException ex) {
    System.out.println("Rossz osztály!");
}
catch (IOException ex) {
    System.out.println("I/O hiba! "+ex);
}
}

public static void main(String[] args) {
    letrehoz();
    listaz();
}
```

A folyam csak objektumokkal (ObjStreamDemo2.java)

Ha csak objektumokat írunk a folyamra (például a primitív `int` helyett `Integer`-t írunk ki), akkor a visszaolvasást ciklusban is végezhetjük, hiszen akkor minden visszaolvasott típus `Object`. Ekkor //2-től a blokk végéig a következő szerepelhetnek:

```
try {
    while (true)
        System.out.println(in.readObject());
}
catch (IOException ex) {
    in.close();
}
```

Alapértelmezés szerinti szerelmeztetés

Egy objektum szerelmeztető mechanizmusa alapértelmezésben a következőképpen működik:

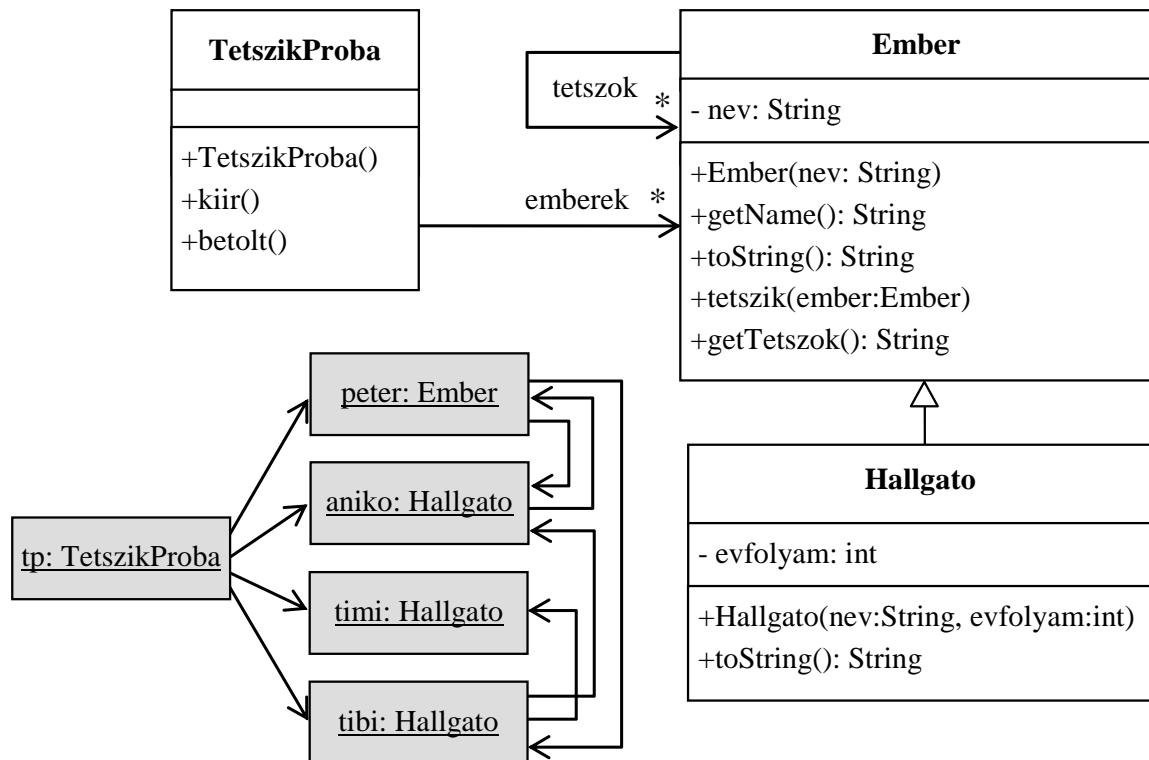
- Felírja az objektum osztályának nevét, azonosítóját.
- Felírja az objektum összes ősadtát és saját adatát a deklaráció sorrendjében, kivéve a `transient` (`transient`) és statikus (`static`) deklarációkat. A felírandó adat lehet primitív adat vagy objektum. A belső objektum ugyanígy tárolódik.
- Ha a folyamra írandó objektum osztálya nem implementálja a `Serializable` interfész, akkor `NotSerializableException` kivétel keletkezik és megszakad a szerelmeztetés.
- Ha a folyamra írandó objektum ősosztálya nem implementálja a `Serializable` interfész, akkor az ősosztályban lennie kell egy paraméter nélküli konstruktornak, mert beolvasáskor a `readObject` metódus az objektum létrehozásakor ezt a konstruktort fogja meghívni. Ha nincs ilyen konstruktur, akkor `NotSerializableException` keletkezik.

Ha azt szeretnénk, hogy egy példányadat ne kerüljön a folyamra, akkor azt az adatot a `transient` módosítóval kell ellátnunk. A `transient` jelentése: múlandó, átmeneti. Az ilyen mező mindenkor a program futása során kap értéket. Visszaolvasáskor a rendszer tudja, hogy ennek a mezőnek helyet kell foglalni, de értéket nem ad neki.

Feladat – TetszikPróba

Hozzunk létre egy `Ember` osztályt, s egy-egy példányába tároljuk az ember nevét és mindeneket az embereket, akik ennek az embernek tetszenek! A Hallgató osztály az `Ember` leszármazottja legyen, s a példányainban tároljuk a hallgató évfolyamát is!

Állítsunk össze emberekből és hallgatókból egy vektort! minden emberhez adjuk meg a neki tetsző embereket! Tároljuk el a vektort, majd olvassuk vissza a lemezről!



14.7. ábra. A TetszikProba objektum- és osztálydiagramja

A program terve

Már első ránézésre is látszik, hogy a kinek-kinek tetsző emberek megadásával meglehetősen kusza, rekurzív objektumhierarchiát kaphatunk. De semmi baj, a helyes tárolás a szerializáló algoritmus feladata! A program terve a 14.7. ábrán látható. A forráskód magáért beszél.

Forráskód

```

import java.io.*;
import java.util.Vector;

class Ember implements Serializable {
    private String nev;
    private Vector tetszok;

    public Ember(String nev) {
        this.nev = nev;
        tetszok = new Vector();
    }

    public String getNev() {
        return nev;
    }
}

```

```
public String toString() {
    return getNev() + ". Tetszik neki: " + getTetszok() + "\n";
}

public void tetszik(Ember ember) {
    tetszok.add(ember);
}

public String getTetszok() {
    StringBuffer sb = new StringBuffer("");
    Ember ember;
    int n = tetszok.size();
    for (int i=0; i<n; i++) {
        ember = (Ember)(tetszok.get(i));
        if (i<n-1)
            sb.append(ember.getNev() + ", ");
        else
            sb.append(ember.getNev());
    }
    return sb.toString();
}

class Hallgato extends Ember implements Serializable {
    private int evfolyam;

    public Hallgato(String nev, int evfolyam) {
        super(nev);
        this.evfolyam = evfolyam;
    }

    public String toString() {
        return evfolyam + ". éves " + super.toString();
    }
}

public class TetszikProba {
    private Vector emberek = new Vector();

    public TetszikProba() {
        // emberek vektor összeállítása:
        Ember peter;
        Hallgato aniko, timi, tibi;
        peter = new Ember("Peter");
        emberek.add(peter);
        emberek.add(aniko = new Hallgato("Aniko", 2));
        emberek.add(timi = new Hallgato("Timi", 3));
        emberek.add(tibi = new Hallgato("Tibi", 2));
        aniko.tetszik(peter);
        peter.tetszik(aniko);
        tibi.tetszik(timi);
        tibi.tetszik(aniko);
        peter.tetszik(tibi);
    }
}
```

```

// Az emberek vektor kiírása:
public void kiir() {
    try {
        ObjectOutputStream os = new ObjectOutputStream(
            new FileOutputStream("work/emberek.dat"));
        os.writeObject(emberek);
        os.close();
    }
    catch (Exception e) {
        System.out.println(e);
    }
}

// Az emberek vektor visszaolvasása:
public void betolt() {
    try {
        ObjectInputStream os = new ObjectInputStream(
            new FileInputStream("work/emberek.dat"));
        System.out.println(os.readObject());
        os.close();
    }
    catch (Exception e) {
        System.out.println(e);
    }
}

public static void main(String[] args) {
    TetszikProba tp = new TetszikProba();
    tp.kiir();
    tp.betolt();
}
}

```

A program futása

```

[Peter. Tetszik neki: Aniko, Tibi
, 2. éves Aniko. Tetszik neki: Peter
, 3. éves Timi. Tetszik neki:
, 2. éves Tibi. Tetszik neki: Timi, Aniko
]

```

Megjegyzés:

Nem csak az alapértelmezés szerinti módszerrel szerializálhatjuk az objektumokat. Csak meg kell adnunk az elmentendő objektum osztályában egy `writeObject` és egy `readObject` metódust:

```

private void writeObject(ObjectOutputStream out) throws
    IOException, ClassNotFoundException {
    // Egyedi felirások
    out.defaultWriteObject();
}

```

```
private void readObject(ObjectInputStream in) throws
    IOException, ClassNotFoundException {
    // Egyedi visszaolvasások
    in.defaultReadObject();
}
```

Ha a szerializáló mechanizmus talál ilyen metódusokat, akkor velük végzi a szerializálást. A metódusokból meghívható az alapértelmezés szerinti szerializációt elvégző `defaultReadObject`, illetve a `defaultWriteObject` metódus. A fenti `readObject` és `writeObject` metódus nem tévesztendő össze az objektumfolyam alábbi metódusaival:

```
void ObjectOutputStream.writeObject(Object)
Object ObjectInputStream.readObject()
```

A forráskód melléklet tartalmaz egy `SajatSzerializacio.java` programot; abban a `CustomVector` osztálynak saját szerializáló algoritmusa van.

Tesztkérdések

14.1. Mi igaz a `DataOutputStream` folyamra? Jelölje be az összes helyes választ!

- a) Beviteli folyam.
- b) Karakterfolyam.
- c) Szűrőfolyam.
- d) Lehet állománynév a paramétere.

14.2. Mi igaz a `Reader` folyamra? Jelölje be az összes helyes választ!

- a) Beviteli folyam.
- b) Karakterfolyam.
- c) Szűrőfolyam.
- d) Lehet állománynév a konstruktur paramétere.

14.3. Mely állítások igazak? Jelölje be az összes helyes állítást!

- a) A szűrőfolyamhoz mindenkorábban kell csatolni egy másik folyamot.
- b) A karakterfolyamhoz mindenkorábban kell csatolni egy másik folyamot.
- c) A bájtfolyamhoz mindenkorábban kell csatolni egy másik folyamot.
- d) A pufferező folyamhoz mindenkorábban kell csatolni egy másik folyamot.

14.4. Mely állítások igazak? Jelölje be az összes helyes állítást!

- a) A beviteli bájtfolyam őse a `Reader`.
- b) A beviteli karakterfolyam őse a `Reader`.
- c) A kiviteli objektumfolyam őse a `Writer`.
- d) A kiviteli karakterfolyam őse a `Writer`.

14.5. Mely állítások igazak? Jelölje be az összes helyes állítást!

- a) A pufferező folyam feladata, hogy összegyűjtse a primitív adat összeállításához szükséges bájtokat.
- b) A pufferező folyamhoz mindenkorábban kell csatolni egy másik folyamot.

- c) A szöveges állomány visszafelé is olvasható.
d) A szöveges állomány sorvégjele minden operációs rendszerben a CR+LF karakterpár.
- 14.6. Mely állítások igazak? Jelölje be az összes helyes állítást!
- Az objektumfolyamra írható primitív adat is.
 - Az objektumfolyamra írt objektumnak implementálnia kell a `Serializable` interfészét.
 - Az objektumfolyam alapértelmezés szerinti szerializáló mechanizmusa nem írja fel a `transient` adatokat.
 - Az objektumfolyam alapértelmezés szerinti szerializáló mechanizmusa felírja a `static` adatokat.

Feladatok

- 14.1. (A) Próbáljon meg felfedezni egy teljesen ismeretlen állományt (előzetesen azt sem tudni róla, hogy szöveges-e)! Listázza ki konzolra bájtonként úgy, hogy a karakterként megjeleníthető bájtokat megjeleníti, a többi helyére pontot tesz! 40 karakterenként emeljen sort! Fedezze fel például a `c:/config.sys` vagy a `c:/javaprog/lib/javalib.jar` állományt! (*Felfedelez.java*)
- 14.2. (A) Fűzzön össze két tetszőleges állományt! Készítsen belőlük egyetlen állományt; annak hossza a két állomány hosszának az összege legyen, és tartalma bájtról bájtra egyezzen meg az első, majd a második állományéval! A végén ellenőrizze, hogy az összefűzött állomány hossza megegyezik-e a két forrásállomány hosszának összegével! (*Osszefuz.java*)
- 14.3. (A) Listázza ki konzolra a program paramétereként megadott szöveges állományt! (*ListFile.java*)
- 14.4. (B) Listázzon ki egy szöveges állományt karakterenként egy `JTextArea` komponensre, lassítva. A feladatot időzítővel oldja meg! (*LassanListaz.java*)
- 14.5. (B) Kérjen be konzolról egy állománynevet! Készítsen el egy `dump.txt` szöveges állományt; annak az első sora tartalmazza a megadott állomány nevét, a többi sora pedig az állomány dump-ját, vagyis az állomány összes bájtját szám és karakter formájában! Egy sorba 10 bájtot írjon egymás alá igazítva! Az állomány utolsó sora a "Vége" szó legyen!

Ha például a `proba.txt` tartalma:

```
ABC ÓÚŰ 123
01 23 45
XXX
```

akkor a dump a következő:

```
work/proba.txt
65 A| 66 B| 67 C| 32 | 32 | 213 ?| 218 Ú| 219 ?| 32 | 32 |
49 1| 50 2| 51 3| 13 □| 10 □| 48 0| 49 1| 32 | 50 2| 51 3|
32 | 52 4| 53 5| 13 □| 10 □| 88 X| 88 X| 88 X| 13 □| 10 □|
Vége
```

(*Dump.java*)

- 14.6. (B) A megadott szöveges állományban (*SzovegStatisztika.java*)
a) számolja meg a sorok, a szavak és a leütések számát!
b) keressen egy adott szöveget! Írja ki, van-e benne, és ha van, akkor hányadik sorban!
- 14.7. (B) Egy szöveges állományban (Java forráskódban) cserélje ki az összes HT (horizontal tab) karaktert szóközökre. A program paraméterként kapja meg a szóközök számát!
(*TabToSpace.java*)
- 14.8. (C) Egy Java forráskódból törölje ki az összes megjegyzést! (*MegjegyzesTorol.java*)
- 14.9. (B) Gépeljünk vakon! A begépelt szöveget mentsük el egy szöveges állományba!
Ezután jelenítsük meg az állományt a képernyön egy `JTextArea` komponensen!
(*Vakon.java*)
- 14.10. (B) Kérjen be fizetéseket, és mentse el őket egy adatfolyamra! Később írja szöveges állományba a következő statisztikát:
Fizetések:
1. 160000
2. 56000
3. 320500
A fizetések átlaga: 178833 Ft
(*Fizetesek.java*)
- 14.11. (A) Tegyen egy `Vector` konténerbe `String`, `Integer` és `Double` típusú objektumokat! Mentse el a vektort egy "vektor.dat" állományba. Később listázza ki a vektor elemeit! (*VectorTarol.java*)
- 14.12. (B) Készítsen egy alkalmazást nevek eltárolására. A keretben legyen egy lista egy rendezetlen névsorral. A keret felső részéről, egy beviteli mezőről lehessen nevet adni a listához. A keret alsó részén négy gomb legyen:
– Elment: Névsor lemezre mentése állománykiválasztó dialógus segítségével.
– Betölt: Névsor betöltése állománykiválasztó dialógus segítségével.
– Töröl: Névsor törlése.
– Kilép: Program befejezése.
(*NevekTarol.java*)
- 14.13. (B) Készítsünk egy szövegszerkesztőt! A kiválasztott szöveges állományt (txt, java és pas kiterjesztések) egy szövegerületen jelenítsük meg; a felhasználó menthesse is el a megváltoztatott állományt! Az állomány betöltésekor és elmentésekor használunk pufferelést!
(*SimpleEditor.java*)
- 14.14. (C) Bővítsük ki az előző, SimpleEditor programot úgy, hogy minél jobban hasonlítsa a Jegyzettömb (NotePad) programhoz! (*Esettanulmányok/KissEditor.jpx*)

15. Közvetlen hozzáférésű állomány

A fejezet pontjai:

1. Állományszervezési és -hozzáférési módok
 2. A RandomAccessFile osztály
-

A felhasználó lehetőség szerint gyorsan szeretné megtalálni az eltárolt adatokat. De az adatelérést csak bonyolultabb tárolási eljárásokkal lehet felgyorsítani. Az állományok logikai egyéinek tárolására és hozzáférésére szabályokat felállítani: ez állományszervezési feladat. Az állomány szervezését főként az adatelérési igények szerint kell megválasztani.

A fejezetben először röviden összefoglaljuk az állományok szervezésének és hozzáférésének módjait. A Java soros szervezésű állománya a folyam; arról az előző fejezetben már volt szó. A közvetlen szervezésű állományban – a sorossal ellentétben – az elemeket közvetlenül el tudjuk érni, azt írhatjuk és olvashatjuk. A Java közvetlen szervezésű állománya a RandomAccessFile; a második pontban róla lesz szó. Sajnos a RandomAccessFile csak bájtonként enged hozzáférést az állományhoz; a logikai egységek címét tehát nekünk kell kiszámítanunk. A RandomAccessFile működését egy egyszerű konzolos alkalmazáson mutatjuk be. Az indexelt szekvenciális szervezésre a könyv végén a *Feladatok* részben adunk példát.

15.1. Állományszervezési és -hozzáférési módok

Az adatok tárolása és karbantartása kulcskérdés az alkalmazásfejlesztésben. Stratégiai döntéseket kell hoznunk: adatbázisban tároljuk-e az adatokat, s ha igen, akkor melyiket válasszuk. Bizonyos feladatok megoldásához bőven elegendő az állománykezelés alapfokú ismerete – néhány egyszerű adat elmentéséért és visszaolvasásáért nem érdemes „ágyúval verébre löni”, és nagy, drága adatbázist használni. Előfordulhat azonban, hogy egyszerű szerkezetű, de nagy mennyiségű adatot akarunk gyorsan mozgatni: ilyenkor hasznos lehet az állományszervezés néhány ismert fogása.

Hozzáférési (elérési) módok

Az állomány logikai egysége logikailag összetartozó elemeket tartalmaz, s ezt az egységet egy-szerre akarjuk kezelni: eltárolni, megkeresni, feldolgozni. A logikai egység lehet objektum, rekord (adatok összessége), esetleg elemi adat is. Logikai egység lehet például egy személy összes adata vagy egy beviteli ablak.

Az állományba felírt logikai egységeket később el szeretnénk olvasni – „el szeretnénk érni”. A felhasználó általában a következőket követeli meg:

- ◆ Az összes egység elérése, feldolgozása; a sorrend nem számít;
- ◆ Az összes egység elérése valamilyen logikai (rendezési) sorrendben;
- ◆ Adott kulcsú (azonosítójú) egység elérése;
- ◆ Adott tulajdonságú egység(ek) elérése.

Az állomány szervezését legfőképpen az elérési (hozzáférési) igények szerint kell megválasztani.

Állományok szervezése

Az adatkezelés csak akkor lehet hatékony, ha minden feladatban végig gondoljuk, hogyan tároljuk majd a logikai egységeket és milyen módszerrel fogjuk őket visszakeresni.

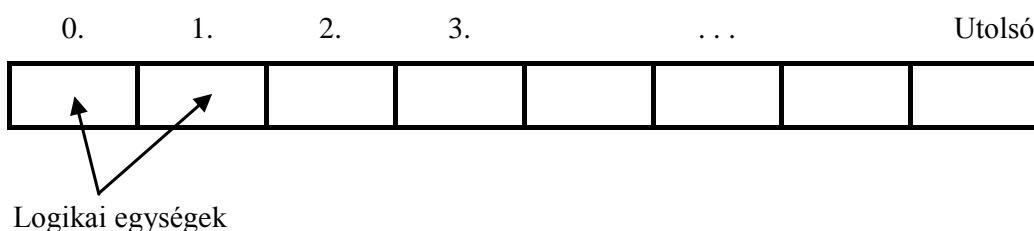
Az operációs rendszer állománykezelő rendszere bájtokat írhat ki egy megadott fizikai címre vagy bájtokat olvashat be onnan. Minél magasabb szintű szoftver kezeli a kivitel/beolvasást, annál közelebb állhat az állománykezelés az emberhez.

Állományszervezésnek nevezzük azt a műveletet, amellyel meghatározzuk az állomány egységeit kiíró és elérő algoritmusait, a kiírás és a hozzáférés szabályait. Alapjában a következő állományszervezási módokat különböztetjük meg:

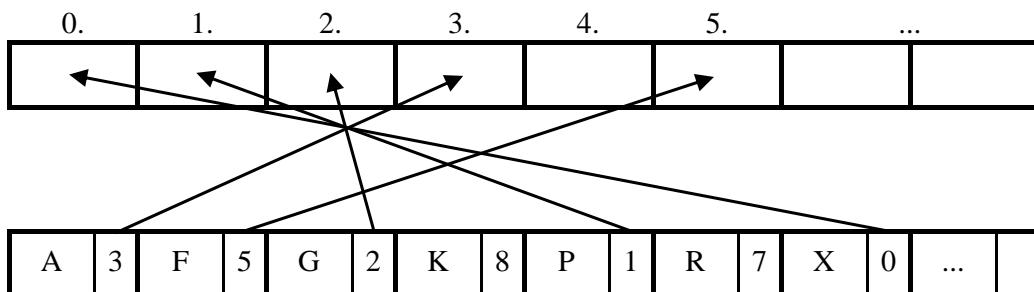
- **Soros szervezés:** A logikai egység és a tárolási (fizikai) cím között nincs kapcsolat. Az állományban lévő adatokat csak a fizikai tárolás sorrendjében vihetjük fel, illetve érhetjük el. A folyamok soros szervezésű állományok.
- **Közvetlen szervezés** (15.1. ábra): A logikai egység és a tárolási cím között kölcsönösen egyértelmű a megfeleltetés (pl. a névsorban az 5. helyen van az ötödik ember). Közvetlenül elérhetjük, írhatjuk és olvashatjuk az állomány bármelyik elemét. A Javában a RandomAccessFile közvetlen szervezésű állomány, lásd a következő pontban.
- **Véletlen (hasító) szervezés:** Egy algoritmus egy hasítókódnak nevezett leképezéssel tárolási címeknek felelteti meg a logikai egységeket. Egy egységnak egyértelmű hasító-kódja van; az egység következetesen minden ugyanarra a helyre kerül. A véletlen elnevezés onnan ered, hogy a hasítókód „össze-vissza” szórja szét az egységeket. Ha ez az algoritmus két logikai egységet véletlenül ugyanarra a helyre képezte le, akkor korrekciós algoritmusokkal kell kezelnünk a „túlcordulási területet”. Ez a szervezés akkor jó,

ha találunk olyan algoritmust, amely aránylag egyenletesen „teríti” az állományban a logikai egységeket.

- **Indexelt szekvenciális szervezés** (15.2. ábra): Kigyűjtjük az ún. törzsállomány egységeinek a kulcsait, és külön indextáblában vagy indexállományban (memóriában vagy háttértárolón) tároljuk őket. A kulcshoz szorosan hozzátartozik az „anyarekord” törzsállománybeli helye, indexe. Az index alapján a logikai egységek közvetlenül, tehát gyorsan elérhetők. Egy egység keresésekor a kulcsot először az indexek között keresük, és a megtalált kulcs melletti index alapján a logikai egység elérhető, feldolgozható. Ez a szervezés akkor igazán hatékony, ha a törzsállomány nagy, és a kulcsok kicsik a logikai egységekhez képest.



15.1. ábra Közvetlen szervezés



15.2. ábra Indexelt szekvenciális szervezés

Az állomány logikai egységeihez való hozzáférés módjait a következőképpen osztályozzuk:

- **Soros hozzáférés:** Az egységeket a fizikai tárolás sorrendjében érjük el.
- **Szekvenciális hozzáférés:** Az egységeket meghatározott logikai sorrend (például rendezettség) szerint érjük el. A szekvenciális feldolgozhatóságot az állomány megfelelő szervezésével érhetjük el, vagy valamilyen algoritmussal – például rendezéssel.

- **Közvetlen hozzáférés:** Az egységeket nem kell sorban beolvasnunk, mert közvetlenül férhetünk hozzá azokhoz, amelyekre szükségünk van. Az egységekhez való közvetlen hozzáférésről szintén szervezássel gondoskodhatunk.

Az indexállomány lehet többszintű, ezt azonban itt nem részletezzük. Indexelt szekvenciális állományra a könyv végén a *Feladatok* részben hozunk példát.

15.2. A RandomAccessFile osztály

Csomag: `java.io` Deklaráció: `public class RandomAccessFile`

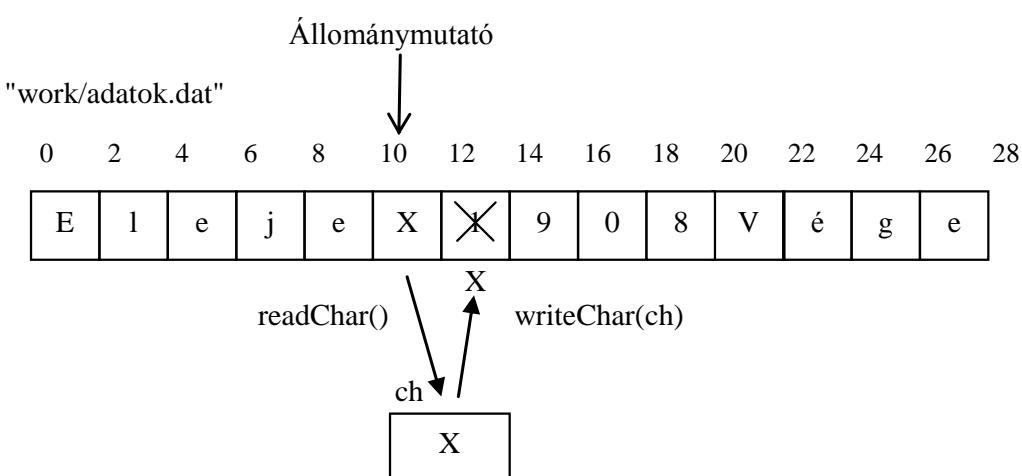
Közvetlen ős: `java.lang.Object`

Közvetlenül implementált interfések: `DataInput`, `DataOutput`

A `RandomAccessFile` osztály segítségével az állomány bármelyik része közvetlenül feldolgozható, és a benne levő adatok közvetlenül írhatók is, olvashatók is. Az állománynak minden van egy **aktuális mutatója** (állománymutató, file pointer, állománypozíció), a mutatónak az állomány elejétől számított bájtsorszám az értéke. Az állomány mutatóját lehet kérdezni, és be lehet állítani. Az olvasó, illetve író utasítások minden az állománymutató által megadott pozíciótól olvasnak, illetve írnak, majd az állománymutatót a beolvasott, illetve a kiírt bájtszámmal előbbre viszik.

A `RandomAccessFile` osztály az `Object` közvetlen leszármazottja, és implementálja a `DataInput` és `DataOutput` interfést. Az `ObjectInput` és `ObjectOutput` interfész azonban nem implementálja, az objektumok automatikus kiírásához tehát nem ad eszközt.

Olvasáskor az állomány végén `EOFException` keletkezik.



15.3. ábra. RandomAccessFile

Tesztprogram (RandomTest.java)

A RandomAccessFile működését a 15.3. ábra érzékelteti. A teszprogram forráskódja:

```
import java.io.*;

public class RandomTest {
    public static void main(String[] args) {
        try {
            RandomAccessFile raf =
                new RandomAccessFile("work/adatok.dat", "rw");
            raf.writeChars("ElejeX1908Vege"); //1
            System.out.println("Állományhossz: "+raf.length()); //2
            raf.seek(10); //3
            char ch = raf.readChar(); //4
            System.out.println(ch);
            raf.writeChar(ch); //5
            long poz = raf.getFilePointer(); //6
            System.out.println("Állománypozíció: "+poz);
        }
        catch (IOException ex) {
            System.out.println(ex);
        }
    }
}
```

A közvetlenül elérhető állományt "rw" (read-write) módban hozzuk létre, az tehát írható és olvasható is. Ezután felírunk 14 darab karaktert (//1), így az állomány hossza 28 lesz (//2). Az állomány mutatóját a 10. bájtra állítjuk (//3). A soron következő karaktert, vagyis az X-et beolvassuk a ch változóba (//4). Az állomány mutatója kettővel előbbre lépett, mert a karakter kétbájtos. //5-ben a ch tartalmát, vagyis az X-et kiírjuk az állomány aktuális pozíójába – ezzel felülírjuk az 1-est. Az állománymutató megint kettőt lépett, így poz értéke végül 14 (//6).

Konstruktorok

- ▶ `RandomAccessFile(String name, String mode)` throws `FileNotFoundException`
- ▶ `RandomAccessFile(File file, String mode)` throws `FileNotFoundException`
mode: "r" vagy "rw". r (read) módban az állományt csak olvasni lehet, rw (read/write) módban írni is.

Metódusok

- ▶ `long getFilePointer()` throws `IOException`
Visszaadja az állomány aktuális mutatóját (pozíóját). A legközelebbi írás vagy olvasás ettől a bájttól kezdődik.
- ▶ `void seek(long pos)` throws `IOException`
Beállítja az állomány mutatóját a paraméterben megadott pozícióra. A legközelebbi írás vagy olvasás ettől a bájttól kezdődik. pos lehet nagyobb is, mint az állomány aktuális hossza. Az állomány a legközelebbi íráskor bővülni fog.

- ▶ `long length() throws IOException`
Visszaadja az állomány hosszát bájtokban.
- ▶ `void setLength(long newLength) throws IOException`
Beállítja az állomány hosszát bájtokban. Ha kell, levágja az állományt, ha kell, kibővíti (az új rész tartalma definiálatlan).
- ▶ `int read() throws IOException, EOFException`
- ▶ `int read(byte[] b, int off, int len) throws IOException, EOFException`
- ▶ `int read(byte[] b) throws IOException, EOFException`
Egy vagy több bájt beolvasása – mint az `InputStream` osztály `read` metódusában. A visszaadott érték a sikeresen beolvasott bájtok száma. Az állomány mutatója ennyivel előrébb áll. Az állomány végéhez érve `EOFException` kivétel keletkezik.
- ▶ `void write(int b) throws IOException`
- ▶ `void write(byte[] b) throws IOException`
- ▶ `void write(byte[] b, int off, int len) throws IOException`
Egy vagy több bájt kiírása – mint az `OutputStream` osztály `write` metódusában. Az állomány mutatója ennyivel előrébb áll.
- ▶ `int skipBytes(int n) throws IOException`
Az állománymutató n bájttal előrébb áll; ha nincs annyi bájt, akkor az állomány végére. Visszaadja azt az értéket, amennyivel sikerült előrébb állnia.
- ▶ `void close() throws IOException`
Kiírja a pufferben maradt adatokat, és lezárja az állományt. Az állományt nem lehet újra megnyitni (új objektumot kell létrehozni).

Emlékeztetőül – kivonat a `DataInput` és `DataOutput` metódusaiból:

- ▶ `boolean readBoolean() throws IOException`
- ▶ `char readChar() throws IOException`
- ▶ `int readInt() throws IOException`
- ▶ `double readDouble() throws IOException`
- ▶ `String readLine() throws IOException`
- ▶ `void writeBoolean(boolean v) throws IOException`
- ▶ `void writeChar(int v) throws IOException`
- ▶ `void writeInt(int v) throws IOException`
- ▶ `void writeDouble(double v) throws IOException`
- ▶ `void writeChars(String s) throws IOException`

Az állomány mutatója a beolvasott, illetve a kiírt bájtok számával előrébb kerül. Beolvasákor az állomány végén `EOFException` kivétel keletkezik.

Feladat – Hőmérsékletek

A hónap minden napján délben egytized fok pontossággal megmérjük a levegő hőmérsékletét. Vigyük fel az adatokat egy "HaviHomers.dat" állományba! Ezután listázzuk ki az állományt, és írjuk ki a havi átlaghőmérsékletet! Kérdezhesse meg a felhasználó, hogy melyik napon mekkora volt a hőmérséklet! Ha rossz napot üt be, akkor írjuk ki, hogy "Nem jó nap!"; ha nullát üt be, akkor fejezzük be a programot! A hőmérsékletadatok a program befejezése után se vesszenek el!

A hőmérsékletértékeket véletlenszám-generátorral fogjuk előállítani, és double értékekkel fogjuk eltárolni. Valamely nap hőmérsékletének könnyen kiszámítható az állománybeli pozíciója: $(\text{nap}-1) * 8$, hiszen a double 8 bájtnyi helyet foglal el a lemezen, és az állománypozíció 0-ról indul.

Forráskód

```
import java.io.*;
import extra.*;

public class Homersekletek {

    static void felvisz() {
        try {
            DataOutputStream havihom = new DataOutputStream(
                new FileOutputStream("work/HaviHomers.dat")); //1
            for (int i=0; i<31; i++) { //2
                double hom = (int)(Math.random()*100)/10.0+10;
                havihom.writeDouble(hom);
            }
            havihom.close();
        }
        catch (IOException ex) {
            System.out.println(ex);
        }
    }

    static void kerdez() {
        try {
            RandomAccessFile havihom =
                new RandomAccessFile("work/HaviHomers.dat", "r"); //3
            System.out.println("Hőmérsékletek a hónapban"); //4
            int i=0;
            double osszeg = 0;
            try {
                while (true) {
                    double hom = havihom.readDouble();
                    i++;
                    osszeg += hom;
                    if (i%8==0) System.out.println();
                    System.out.print(i+": "+hom+"\t");
                }
            }
        }
    }
}
```

```

        catch (EOFException ex) {
            System.out.println("\nÁtlaghőmerséklet 2 tizedesre: "+
                ((int)(osszeg/31*100)/100.0));
        }

        System.out.println("\n\nKérdezzzen 0 végjelig!");
        long poz;
        while ((poz = Console.readInt("\nNap: ")) != 0) {
            try {
                havihom.seek(8*(poz-1));                                //5
                System.out.println(havihom.readDouble());
            }
            catch (IOException e) {
                System.out.println("Nem jó nap!");
            }
        }
        havihom.close();
    }
    catch (IOException ex) {
        System.out.println(ex);
    }
}

public static void main(String[] args) {
    felvisz();
    kerdez();
}
}

```

A program elemzése

- ◆ //1: A számokat most egy adatfolyamra „folytatjuk” ki, bár használhatnánk éppen RandomAccessFile-t is.
- ◆ //2: A ciklusban a hónap minden napjához gyártunk egy 10 és 20 közötti véletlenszerű, egyszínes valós számot (hogy ne kelljen az Olvasónak fáradnia a 31 darab szám beütésével). A hom valós számot a writeDouble(hom) utasítással írjuk ki a havihom állományba.
- ◆ //3: A kerdez metódusban a számok feldolgozásához egy RandomAccessFile-t nyitunk meg, mégpedig olvasásra (r), mert nem akarjuk megmásítani az eredményeket.
- ◆ //4: A kérdezgetés előtt ellenőrzésképpen kiírjuk a 31 felvitt számot. A visszaolvasás szekvenciális, a mutató léptetése automatikus. Az állomány végénél elkapjuk az EOFException-t, és kiírjuk a számok átlagát.
- ◆ //5: Itt jön a közvetlen hozzáférésű állomány eleje, mert most különböző pozíciókról fogjuk beolvasni a számokat. A felhasználó a napot 1-től számolja, az állománypozíció azonban 0-ról indul, és figyelembe kell vennünk azt is, hogy a double 8 bájtnyi helyet foglal el. A poz. hőmérséklet állománybeli pozíciója tehát 8*(poz-1). A seek metódussal ráállítjuk erre a bájtra az állomány mutatóját, majd a readDouble behozza onnan a várt értéket.

Tesztkérdések

15.1. Jelölje be az összes helyes állítást!

- a) A soros szervezésű állományban a logikai egység egyértelmű kapcsolatban áll a tárolási (fizikai) címmel.
- b) A közvetlen szervezésű állomány egységei közvetlenül is elérhetők.
- c) A véletlen szervezésű állomány egységei véletlenszerű helyekre kerülnek, de egy egység következetesen minden ugyanoda.
- d) Az indexelt szekvenciális szervezésű állomány egységei a kúlcukon keresztül érhetők el.

15.2. Jelölje be az összes helyes állítást!

- a) A soros szervezésű állomány egységei a kiírás sorrendjével ellentétes sorrendben is visszaolvashatók.
- b) A szekvenciális elérés azt jelenti, hogy az adatokat a tárolási sorrendjében érhetjük el.
- c) Ha az elérés közvetlen, akkor a logikai egységet elérhetjük a többi elem olvasása nélkül is.
- d) Az állományszervezés olyan művelet, amelyben rendezzük az állományt.

15.3. Mely állítások igazak a RandomAccessFile-ra? Jelölje be az összes helyes állítást!

- a) Ha az állományt "r" módban nyitjuk meg, akkor azt csak sorasan lehet olvasni.
- b) Fel lehet írni double típusú adatot.
- c) Fel lehet írni egy String-et.
- d) Fel lehet írni objektumot a writeObject utasítással.

15.4. Mely állítások igazak? Jelölje be az összes helyes állítást!

- a) A közvetlen elérésű állomány mutatóját lehet állítani.
- b) A közvetlen elérésű állomány segítségével egy állománynak bármelyik bájtja kijavítható.
- c) A RandomAccessFile osztály read metódusai EOFException kivételt dobnak.
- d) A RandomAccessFile osztály write metódusai EOFException kivételt dobnak.

15.5. Mely állítások igazak? Jelölje be az összes helyes állítást!

- a) A RandomAccessFile egy szűrőfolyam.
- b) A RandomAccessFile implementálja a DataInput interfést.
- c) A RandomAccessFile implementálja az ObjectInput interfést.
- d) A RandomAccessFile osztály a DataStream osztály leszármazottja.

Feladatok

15.1. (A) Írjon fel long típusú számokat a lemezre! A számokat a felhasználótól konzolról kérje be, "*" végjelig! Ezután

- a) Listázza ki a számokat a kozolra!
- b) Írja ki az 5. számot, ha van ilyen!
- c) Írjon a számok után további számokat, majd listázza ki az összes számot!
- d) Listázza ki a számokat visszafelé!

- e) Egészítse ki az állományt nullákkal vagy vágja le a végét úgy, hogy az állomány hossza pontosan 800 bájt legyen!
- f) Válogassa szét a számokat két állományba: az egyikbe az 1000-nél kisebbeket, a másikba az 1000-nél nagyobbakat tegye!
- (*Egeszek.java*)
- 15.2. (B) A "SzazadHom.dat" állományban legyenek múlt századi (1901-2000) éves hőmérsékletadatok:
- Az évi maximális hőmérséklet és pontos dátuma.
 - Az évi minimális hőmérséklet és pontos dátuma.
 - Az évi átlaghőmérséklet.
 - Egyéb adatok, ízlés szerint
- Minden évhez ugyanannyi és ugyanolyan formátumú adat tartozzon.
- a) Készítse el az állományt bármilyen módszerrel! Lehetnek definiáltlan (kitöltetlen) évek is.
- b) Készítsen statisztikát az évszázadról: maximális és minimális hőmérséklet, átlaghőmérséklet (átlagok átlaga)! A statisztikában csak a kitöltött évek szerepeljenek.
- (*SzazadHom.java*)
- 15.3. (C) Készítsen egy alkalmazást, amellyel egy totóárus munkáját segíti! A program funkciói:
- a) Nyertes tippek rögzítése: minden héten nyertes tippeinek felvitele.
 - b) Adatok rögzítése: a beérkezett totószelvények adatainak felvitele (szelvény száma, héten, tippek, név és cím).
 - c) Adatszolgáltatás: egy szelvény adatainak megadása a héten és a szelvényszám alapján.
 - d) Valamely adott héten nyertesei.
- (*Totozo.java*)

I. OBJEKTUMORIENTÁLT TECHNIKÁK

1. Csomagolás, projektkezelés
2. Örökítés
3. Interfészek, belső osztályok
4. Kivételkezelés

II. GRAFIKUS FELHASZNÁLÓI INTERFÉSZ

5. A felhasználói interfész felépítése
6. Elrendezésmenedzserek
7. Eseményvezérelt programozás
8. Swing-komponensek
9. Grafika, képek
10. Alacsony szintű események
11. Belső eseménykezelés, komponensgyártás
12. Applet

III. ÁLLOMÁNYKEZELÉS

13. Állományok, bejegyzések
14. Folyamok
15. Közvetlen hozzáférésű állomány



IV. VEGYES TECHNOLÓGIÁK

16. Rekurzió
17. Többszálú programozás
18. Nyomtatás
19. Hasznos osztályok

IV.

V. ADATSZERKEZETEK, KOLLEKCIÓK

20. Klasszikus adatszerkezetek
21. Kölcsönös keretrendszer

FELADATOK

FÜGGELÉK

- A tesztkérdések megoldása
Irodalomjegyzék
Tárgymutató

16. Rekurzió

A fejezet pontjai:

1. A teljes indukció elve
 2. Rekurzív feladat
 3. Rekurzív eljárás, függvény
 4. A rekurzió megállítása
 5. Feladat – Hanoi tornyai
 6. Feladat – Gyorsrendezés
-

Eddigi tanulmányainkban mindig úgy alkottunk ciklust, hogy egy vezérlőutasítással valamely feltétel teljesüléséig újra meg újra végrehajtottunk egy utasítást vagy utasításblokkot. Van a tevékenységek ismétlésének egy másik, sokkal elegánsabb módja is: a rekurzió – ha az eljárás vagy függvény önmagát hívja meg. A folyamatot persze ebben az esetben is véges számú lépés után meg kell állítani. Ennek a fejezetnek az a célja, hogy néhány klasszikus problémán át bemutassa a rekurzió leglényegesebb tulajdonságait.

A rekurzív feladatok többsége ismétléssel (iterációval) is megoldható. A rekurzió sok esetben lassítja a programot, és a vermet is nagyon intenzíven használja. Vannak azonban olyan feladatok, amelyeknek a megoldásában a rekurzió a maga tiszta logikai szerkezetével és rövid programkódjával bőven kárpótlást ad ezekért a nehézségekért.

16.1. A teljes indukció elve

A teljes indukció elve a következő:

Egy állítás igaz az $n = n_0, n_0 + 1, n_0 + 2, n_0 + 3, \dots$ értékekre (n_0 -tól kezdve bármely természetes számra), ha sikerül bebizonyítani a következőket:

- az állítás igaz $n = n_0$ -ra
- ha az állítás igaz egy tetszőleg n -re, akkor abból következik, hogy igaz $n + 1$ -re is.

Nézzünk egy példát!

Állítás:

Az első n természetes szám összege, $1 + 2 + \dots + n = \frac{n * (n + 1)}{2}$

Például: $1 + 2 + \dots + 8 = (8*9)/2 = 36$

Bizonyítás teljes indukcióval:

$n = 1$ -re ez az állítás igaz, hiszen ekkor az összeg $1*(1+1) / 2 = 1$.

Bizonyítandó, hogy ha ez az állítás igaz n -re, akkor igaz $n + 1$ -re is. Adjunk hozzá az első n szám összegéhez $n + 1$ -et; a többi már adódik:

$$\frac{n * (n + 1)}{2} + n + 1 = \frac{n * (n + 1) + 2 * (n + 1)}{2} = \frac{(n + 1) * (n + 2)}{2},$$

s ez az állítás megfogalmazása $n + 1$ -re.

Az indukciót két oldalról is megközelíthetjük:

- ◆ Általánosítási folyamat: Bebizonyítjuk, hogy egy eset megoldható. Ezután bebizonyítjuk, hogy ha egy eset megoldható, akkor a következő eset is megoldható. Így minden eset megoldható.
- ◆ Egyszerűsítési folyamat: Bebizonyítjuk, hogy egy eset megoldható, ha az előző eset megoldható (visszavezetés egy előző esetre). Ha találunk egy legegyszerűbb megoldható esetet, akkor minden eset megoldható.

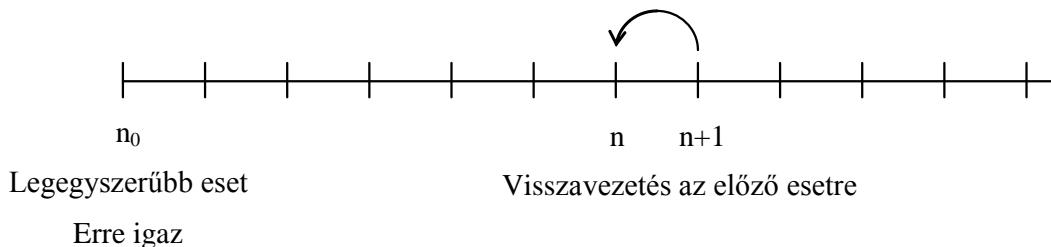
16.2. Rekurzív feladat

Egy feladat **rekurzív**, ha a feladat megoldásához vezető lépésekkel

- találunk egy **legegyszerűbb esetet**, amelyben a megoldás magától értetődik, és
- találunk egy olyan **ismételt egyszerűsítési folyamatot**, amely alapján véges sok lépéssben eljuthatunk a legegyszerűbb esethez. minden lépéssben feltesszük, hogy a következő, egyszerűbb esetnek már megvan a megoldása.

Ez a módszer az indukció elvén alapszik. A feladatot minden visszavezetjük egy előző, egy fokkal már egyszerűbben megoldható feladatra. Véges sok lépéssben elérjük a legegyszerűbb esetet, és így a feladatot megoldottuk.

Az egyszerűsítési folyamatot a 16.1. ábra szemlélteti.



16.1. ábra. Egyszerűsítési folyamat

Az első n természetes szám összege

Példaként nézzük meg az első n természetes szám összegének meghatározását!

Gondolják el, hogy van n ember. Az n -edik embernek az a feladata, hogy kiszámolja 1-től n -ig a számok összegét. Ő azonban a feladatot valamelyest leegyszerűsítve továbbadja szomszédjának, az $n-1$ -edik embernek – tőle várja az első $n-1$ szám összegét. Ha választ kap, akkor neki már csak a kapott összeghez hozzá kell adnia n -et, és már meg is oldotta a feladatot. Az $n-1$ -edik ember ugyanígy tesz: ő is átadja a leegyszerűsített feladatot szomszédjának, az $n-2$ -edik embernek, majd a kapott összeghez hozzáadja a maga számát, $n-1$ -et. A feladat átruházása addig megy, amíg van „kinek” átadni a feladatot. Az 1. ember már nem adja át senki-nek, hiszen ő a legegyszerűbb eset. A feladat tehát általánosan, az n -edik emberre megfogalmazva a következő:

```
if n == 1
    összeg(n) = 1
else
    összeg(n) = összeg(n-1) + n
end if
```

Miután a feladatot az emberek szépen egymásra ruházták, meg kell várniuk a választ ahhoz, hogy feladataikat megoldják. A sorban mindenki ki van szolgáltatva a szomszédjának, kivéve azt az embert, aki utoljára kapja meg a feladatot, hiszen ő kapásból tud válaszolni. Az utolsó emberre mint legegyszerűbb esetre nagy szükség van, hiszen egyébként a feladat sohasem göngyölődhene fel. A visszagöngyölítésben a legegyszerűbb esetnek megfelelő ember elmondja a megoldást a 2. embernek. A 2. ember a kapott megoldást beleépítve megoldja a maga feladatát, és az eredményt továbbadja a 3. embernek. Legutoljára az $n-1$ -edik ember adja oda a már kiszámolt eredményt az n -edik embernek, s az végre megoldja a feladatot: megmondja a kért összeget.

16.3. Rekurzív eljárás, függvény

Egy metódust (eljárást vagy függvényt) rekurzívnak nevezünk, ha az meghívja önmagát.

- **Közvetlen rekurzió:** a metódus közvetlenül hívja magát. Például `r1` metódus hívja `r1`-et.
- **Közvetett rekurzió:** a metódus csak közvetve, egy másik metódus hívásán keresztül hívja meg magát. Például `r1` hívja `r2-t`, s `r2` hívja majd `r1`-et.

Egy rekurzív feladat megoldásában minden lépés ugyanazzal a függvénnel oldható meg, ezért természetes, hogy a függvényt újra és újra meghívjuk.

Az előző pont feladatában az `összeg(n)` függvény magát hívja meg, minden eggyel kisebb paraméterrel. A feladat és annak Java megoldása tehát a következő:

Feladat – Összeg

Határozzuk meg az első `n` természetes szám összegét!

Forráskód

```
public class Osszeg {

    static int osszeg(int n) {
        if (n==1)
            return 1;
        else
            return osszeg(n-1)+n;
    }

    public static void main(String[] args) {
        System.out.println("1+2+3= "+osszeg(3));
    }
}
```

A program elemzése

A függvény első hívásakor `n==3`. Mivel `n!=1`, azért az `else` ág hajtódiik végre, s ott újra az `osszeg` függvényt hívjuk, de már eggyel kisebb, vagyis `n==2` aktuális paraméterrel. A függvénynek keletkezik egy következő „példánya”, s abban megint az `else` ágra kerül a vezérlés. `osszeg`-nek most megint egy újabb példánya hívódik meg, `n==1` aktuális paraméterrel. De ekkor már az igaz ág hajtódiik végre, a függvény felveszi az 1 értéket, és a blokk végére kerül a vezérlés. Ezzel megszűnik a függvény legutolsó példánya. A vezérlés most az előző példány + műveletéhez kerül vissza, majd az összeadás elvégzése után a metódus végére „csorog” a vezérlés. Az összes elkezdett példány lefutásával a rekurzió véget ér. A program futásának eredményeként a képernyön egy 6-os ($3+2+1$) jelenik meg.

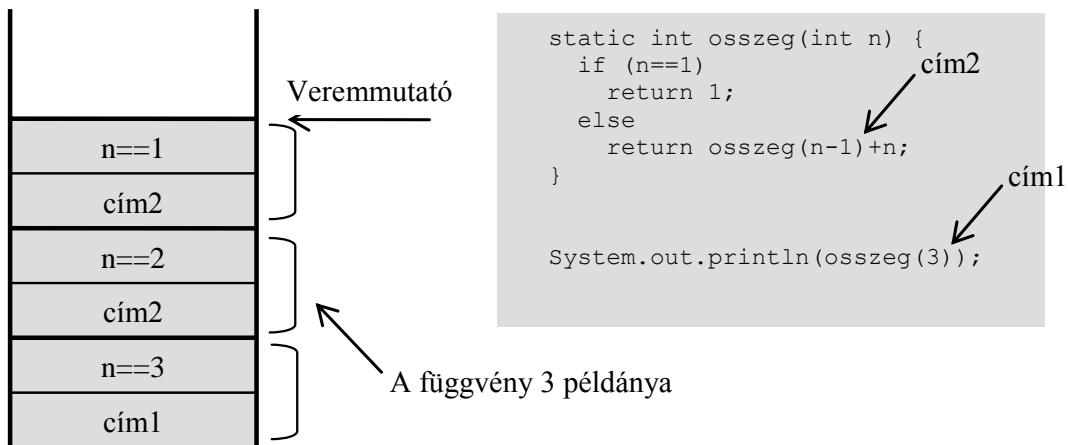
Verem (stack)

A **verem (stack)** az alkalmazás által fenntartott, adatok ideiglenes tárolására használt memóriaterület. Az az egyik leglényegesebb tulajdonsága, hogy minden csatlakoztatott adatot csak a legutoljára betett lehet belőle „kivenni”. Angol szóhasználattal: a verem LIFO (Last In First Out) adatszerkezet.

Egy metódus hívásakor a veremre kerül:

- a metódus visszatérési címe (a blokk végén a vezérlés ide kerül vissza);
- a metódus paraméterei és lokális változói.

Ha a metódus befejezte a futást (a vezérlés a blokk végére kerül), akkor a rendszer automatikusan felszabadítja a veremben lefoglalt területeket (16.2. ábra). Ha a verem betelik (túlcsoportul), `StackOverflowError` hiba keletkezik, és a program leáll.



16.2. ábra. Verem

A veremnek van egy mutatója, s az mindenkorra a verem első szabad helyére mutat. A fordítóprogram minden metódus elejéhez és végéhez hozzátesz egy programrészletet. Amikor a program vezérlése a metódus elejére kerül, akkor a program a verembe teszi a megfelelő adatokat: a visszatérési címet, a paramétereit, illetve az összes lokális változót. Az eljárás ezeket a változókat használja, és lefutása után (ha a vezérlés a blokk végére ér) a metódus visszatér az eltárolt címre, a veremmutató pedig felveszi a metódus hívása előtti értéket. A lefoglalt területeket a rendszer tehát automatikusan felszabadítja.

Ha rekurziót használunk, akkor a metódust még futásának befejeződése előtt újból meghívjuk. Ezzel újabb adatok kerülnek a veremre. A második hívás adatai tehát nem azonosak az előzőivel, jóllehet az azonosítójuk ugyanaz. Az adatok mindenkorra az aktuális hívás lokális adatai. Ami-

kor a második futás sikeresen véget ér, akkor az első futásnak pontosan arra a pontjára érünk vissza, ahol azt abbahagytuk, és az adatokat is pontosan abban az állapotban fogjuk viszontláttni, ahogy otthagytuk őket. Képzeljük el úgy a rekurzív hívást, mintha ezekben az egymás utáni hívásokban mindenig más metódust hívánk meg. Ebben az a nagyszerű, hogy a metódust mégiscsak egyszer kell leírnunk, és ha tetszik, milliószor is meghívhatjuk. Csak arra kell vigyázunk, hogy a hívásokat előbb-utóbb be is fejezzük, mert a verem és a türelmünk sem végig telén kapacitású.

A 16.2. ábrán látható, hogy az `osszeg(3)` hívásakor a veremre kerülnek a függvény adatai: `cím1` és az `n==3` paraméter. Amikor az `osszeg(n-1)`-re kerül a vezérlés, akkor az `osszeg(3)` még nem fejezi be futását, a függvénynek tehát egy újabb példánya kerül a veremre, `cím2` és `n==2` adatokkal. Ez még egyszer eljátszódik, de most már `n==1` lesz, s az a rekurzió leállító feltétele. Megszűnik a veremben a függvény utolsó példánya, a vezérlés viszszakerül egy korábbi híváshoz. A hívások sorban lezajlanak, és a verem teljesen felszabadul.

16.4. A rekurzió megállítása

A rekurzív algoritmusok tervezésében figyelnünk kell arra, hogy minden lépéssel közelebb jussunk a megoldáshoz. Egy rekurzív folyamat csak akkor fejeződhet be, ha valami megállít parancsol a további rekurzióknak, s akkor az elkezdett metódusok sorban végigfuthatnak. A rekurzív metódusban mindenig lennie kell egy **küsöbfeltételnek** vagy **leállító feltételnek**: az dönti el, hogy lesz-e további rekurzív hívás vagy sem. Módot kell adni arra, hogy a metódus blokkja valamikor egészen a végéig fusson. Megállít parancsolhat például olyasvalami, ami minden rekurzív hívással biztosan nő vagy csökken. Ha ez a valami elér egy megadott küsöbüöt, akkor nem hívjuk többször a szóban forgó metódust, és így az egymásból hívott metódusok sorban befejeződhetnek.

Rekurzív eljárások, illetve függvények futtatásakor mindenig megvan az a veszély, hogy a hívásokat semmi sem állítja le, hiszen az egymásból való hívások száma elvileg végtelen lehet. Gyakorlatilag azonban végtelen rekurziót nem könnyű előállítani, hiszen a rekurzió használja a vermet, és ha mást nem is, a visszatérési címet mindenig oda teszi. Mivel a verem nagysága véges, az előbb-utóbb „túlcodul” és futási hibával leáll (`java.lang.StackOverflowError`).

Összefoglalva:

Egy rekurzív metódusban szerepelnie kell

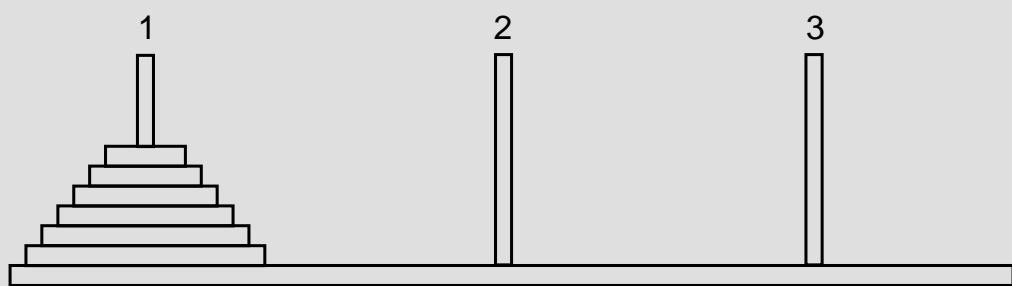
- valaminek, ami a hívások egymásutánjában mindenig változik, és elvileg elérhet egy **küsöbüöt**.
- egy olyan utasításnak, amely ezt a valamit a **küsöb felé viszi**.
- egy **leállító feltételnek**, mely **eldönti** arról a bizonyos valamiről, hogy **elérte-e a küsöbüöt**. S ha igen, akkor nem enged több rekurzív hívást.

16.5. Feladat – Hanoi tornyai

A rekurzív feladatmegoldás verhetetlenül legszebb példája a következő:

Feladat – Hanoi tornyai

Van három oszlop. Az első oszlopon van valahány lyukas korong, s ezek a korongok annál kisebbek, minél feljebb vannak:



A feladat az, hogy az 1. oszlopról át kell tenni az összes korongot a 2. oszlopra, éspedig úgy, hogy egyszerre csak egy korongot tehetünk át egy másik oszlopra, és kisebb korongra soha nem tehetünk nagyobbat. A megoldáshoz átmenetileg felhasználható a 3. oszlop. Az oszlopokon felfelé egyre kisebbeknek kell lenniük a korongoknak a játék bármely pillanatában.

Három koronggal a megoldás a következő (lépésről lépésre kiírjuk, hogy melyik oszlopról melyik oszlopra kell áttenni a legfelső a korongot):

1. lépés: 1-ről 2-re
2. lépés: 1-ről 3-ra
3. lépés: 2-ről 3-ra
4. lépés: 1-ről 2-re
5. lépés: 3-ról 1-re
6. lépés: 3-ról 2-re
7. lépés: 1-ről 2-re

Megjegyzések:

- **A feladathoz egy legenda kapcsolódik:** ezt a játékot Isten találta ki az idők kezdetén, hogy próbára tegye vele az emberiséget. A feladatot történetesen Hanoi kolostorának szerzetesei kapták. A kolostor temploma előtt három oszlop áll, az első oszlopon 64 darab kőkorong van. A feladatot ezekkel a korongokkal kell elvégezni. A legenda szerint akkor lesz vége a világnak, ha az emberiség ezt a legnagyobb feladatát elvégezte.
- Nyugodtan megpróbálhatjuk megoldani a feladatot, **nem kell közeli világvégétől tartanunk**. Az Olvasó nemsokára meggyőződhet róla, hogy ha egy korongot 1 másodperc alatt tesz át egyik oszlopról a másikra, akkor a 64 korongos feladat megoldásához 585 milliárd évre lesz szüksége. Egyébként is csak akkor lesz világvége, ha a korongok kőből vannak, és a színhely Hanoi.

Képzeljük el a megoldást teljes indukcióval:

Állítsunk sorba 64 szerzetest. A 64 korong áthelyezése az 1. oszlopról a 2. oszlopra a 64. szerzetes feladata. Ő azonban kellően bőlcs lévén a 63. szerzeteshez fordul, és így szól:

- ◆ Tedd át a felső 63 korongot az 1. oszlopról a 3. oszlopra.
- ◆ Ha ez megvan, akkor én átteszem a nagy korongot az 1. oszlopról a 2. oszlopra.
- ◆ Végül tess vissza az előbb átemelt 63 korongot a 3. oszlopról a 2. oszlopra.

Ezzel a feladat meg is oldódott. Feltettük persze, hogy a 63. szerzetesnek is sikerül megoldania a feladatát. De ő ugyanúgy gondolkodik, mint a 64. szerzetes – ő 62 korong áthelyezését kéri a 62. szerzetestől. Ő már négy alkalommal fogja ezt kérni, hiszen a 63 korongot kétszer kell áthelyeznie, s mindenkor áthelyezéshez egyszer oda kell tenni a 62 korongot, egyszer meg vissza. Mindegyik szerzetes szomszédját kéri meg tehát az eggyel egyszerűbb feladat elvégzésére. Az 1. szerzetes már meg tudja oldani a maga feladatát, hiszen 1 korong áthelyezése a legegyszerűbb feladat. Valamikor a játéknak vége lesz, idő kérdése csupán.

A program futása így is elképzelhető: a 64. szerzetes a verem először lefoglalt egységeben „tartózkodik”, a 63. szerzetes a másodikban, az első szerzetes pedig a verem 64. rekeszében „foglal helyet” – persze mindegyik csak akkor, ha éppen van feladata. minden hívásra újabb rekesz foglalódik le, és minden részfeladat megoldásakor felszabadul egy rekesz. A megoldás közben a verem foglalt rekeszeinek száma mindig 1 és 64 közé esik. A verem „lélegzik” futás közben.

Hány lépésre van szükség, ha a korongok száma 2, 3, 4, ...?

n korong áthelyezéséhez $f(n)$ lépésre van szükség. Határozzuk meg $f(n)$ -t!

át kell helyezni $n-1$ korongot:	$f(n-1)$ lépés
át kell helyezni 1 korongot:	1 lépés
vissza kell helyezni $n-1$ korongot:	$f(n-1)$ lépés

Összesen tehát $2*f(n-1)+1$ lépésre van szükség. Próbáljuk meghatározni a lépések számát konkrét korongszámokra:

Korongok száma	Lépések száma
2	$3 = 3$
3	$2*3+1 = 7$
4	$2*7+1 = 15$
5	$2*15+1 = 31$
6	$2*31+1 = 63$
...	
N	sejtés: $= 2^N - 1$

Ez utóbbi sejtés a teljes indukció elvével könnyen bizonyítható:

$n=2$ -re igaz, hiszen $2^2 - 1 = 3$

Ha n -re a lépések száma $2^n - 1$, akkor $n+1$ korongra a szükséges lépések száma:

$2 * (2^n - 1) + 1$, vagyis $2 * 2^n - 2 + 1 = 2^{n+1} - 1$.

Lássuk most a Hanoi tornyocskák megoldásának programját; a **megoldás „lelke” alig néhány sor**. A programot 4 koronggal teszteljük. Az `attesz` nevű eljárás n darab korongot tesz át a `mirol` oszlopról a `mire` oszlopra, s közben használja a `manko` oszlopot. Az oszlopoknak persze mindenkor más a funkciójuk az eljárások példányaiban:

Forráskód

```
public class HanoiKonzol {
    static int lepes = 0;

    static void attesz(int n, int mirol, int mire, int manko) {
        if (n == 1)
            System.out.print(++lepes+". lepes: "+mirol+"-"+mire) ;
        else {
            attesz(n-1,mirol,manko,mire) ;
            System.out.print(++lepes+". lepes: "+mirol+"-"+mire) ;
            attesz(n-1,manko,mire,mirol) ;
        }
    }

    public static void main(String[] args) {
        attesz(4,1,2,3) ;
    }
}
```

A program futása:

1. lepes: 1-3	6. lepes: 2-3	11. lepes: 2-1
2. lepes: 1-2	7. lepes: 1-3	12. lepes: 3-2
3. lepes: 3-2	8. lepes: 1-2	13. lepes: 1-3
4. lepes: 1-3	9. lepes: 3-2	14. lepes: 1-2
5. lepes: 2-1	10. lepes: 3-1	15. lepes: 3-2

64 koronggal a lépések száma összesen: $2^{64} - 1 \sim 18\ 446\ 744\ 073\ 709\ 551\ 600$, vagyis 1 másodperces lépésekkel 584 942 417 355 év, azaz 585 milliárd év.

☛ Ne próbálja kivárni!

Próbálja meg a programot grafikusan szimulálni! A megoldás megtalálható a mellékletben..

16.6. Feladat – Gyorsrendezés

Az egyik leghatékonyabb rendezés elvét C.A.R. Hoare adta meg, 1962-ben.. A gyorsrendezés (quicksort) rekurzív algoritmusát most egy egész értékeket tartalmazó tömb rendezésén mutatjuk be.

A gyorsrendezésnek az a lényege, hogy a tömböt az elemek cserélgetéseivel két részre osztjuk fel: a bal oldali rész egy konkrét értéknél kisebb, a jobb oldali rész ennél az érték-nél nagyobb elemekből fog állni. Ha ezt sikerül elérnünk, akkor használhatjuk ugyanezt a módszert a bal oldali, majd a jobb oldali részre is. Az így keletkezett résztömbököt ismételten szétosztva végül 1 elemű résztömbök keletkeznek, és addigra a teljes tömb rendezett lesz. Mivel a résztömbök elemeit minden ugyanazzal a módszerrel „rendezgetjük”, kézenfekvő rekurziót használni.

Legyen a tömb legkisebb indexe **bal**, legnagyobb indexe pedig **jobb**:

bal						közép					
	6	12	8	9	7	3	22	21	5	20	
	i.				7				j.		

elválasztó érték

Válasszunk ki szúrópróbaszerűen egy elválasztó értéket. Ez lehet például a tömb középső indexű eleme, most éppen a 7-es.

Legyen **i** alulról felfelé futó, **j** pedig felülről lefelé futó index. **i**-vel addig megyünk felfelé, ameddig az elemek még kisebbek az *elválasztó* értéknél. Mivel a 12 nagyobb a 7-nél, azért itt már leállunk. **j**-vel meg lefelé haladunk, s azon az elemen állunk meg, amely kisebb, mint a 7. Most felcseréljük az **i**-edik és a **j**-edik elemet, s ezzel mindenkor a jó „térfélre” kerül. Az indexekkel továbbhaladva most a 8-as és a 3-as elemen állunk meg, s őket is felcseréljük:

bal						közép					
	6	5	8	9	7	3	22	21	12	20	
	i.				7				j.		

Végül a 7-es és a 9-es elemeket kicserélve egy olyan tömböt kapunk, amelyben az alsó rész összes eleme kisebb 7-nél vagy egyenlő vele, a felső rész elemei pedig mindenkor nagyobbak 7-nél vagy egyenlők vele.

A teljes tömb szétosztása után most rekurzív módon ugyanezt az algoritmust alkalmazzuk először az alsó, majd a felső résztömbre. Egy tömbrész rendezésének akkor van vége, ha az eljárás végére kerül a vezérlés, vagyis ha az összes belső rendezést sikerült elvégezni:

bal	jobb			
6	5	3	7	
i.		j.		

Feladat – Gyorsrendezés (QuickSort)

Rendezzünk gyorsrendezéssel egy long típusú elemekből álló tömböt!

Forráskód

```

public class QuickSort {
    static long[] t = {6,12,8,9,7,3,22,21,5,20};

    static void quickSort(int bal, int jobb) {
        int i, j; // futó indexek
        long elvalaszto; // a tömb kiválasztott eleme
        long temp;

        elvalaszto = t[(bal+jobb)/2];
        i = bal;
        j = jobb;
        while (i<=j) {
            while (t[i]<elvalaszto)
                i++;
            while (t[j]>elvalaszto)
                j--;
            if (i<=j) {
                // i. és j. elem cseréje:
                temp = t[i]; t[i] = t[j]; t[j] = temp;
                i++; j--;
            }
        }
        if (bal<j)
            quickSort(bal,j);
        if (i<jobb)
            quickSort(i,jobb);
    }

    public static void main(String[] args) {
        quickSort(0,t.length-1);
        System.out.println("Rendezve: ");
        for (int i=0; i<t.length; i++)
            System.out.print(t[i]+" ");
        System.out.println();
    }
}

```

Tesztkérdések

- 16.1. Mely állítások igazak? Jelölje be az összes jó választ!
- A teljes indukció szerint: ha n-re és n+1-re bebizonyítjuk az állítás helyességét, akkor az állítás minden n-re igaz.
 - Egy rekurzív feladatban mindenkor kell lennie egy legegyszerűbb esetnek.
 - A közvetlen rekurzióban a függvény önmagát hívja meg.
 - A rekurzív függvény mindenkor önmagát hívja meg.
- 16.2. Mely állítások igazak? Jelölje be az összes jó választ!
- Minden függvény használja a rendszer által fenntartott vermet.
 - A veremből mindenkor betett elemet lehet kivenni.
 - Egy metódus hívásakor csak a metódus paramétereit kerülnek a verembe, a metódus lokális változói nem.
 - Ha a verem betelik, akkor `StackOverflowError` hiba keletkezik.
- 16.3. Mely állítások igazak? Jelölje be az összes jó választ!
- Közvetett rekurzióban a rekurzív függvény teljes egészében átadja a feladatot egy másik függvénynek.
 - Rekurzív metódusban mindenkor kell egy leállító feltételnek.
 - A rekurzív feladatok ciklussal is megoldhatók.
 - A rekurzív metódus intenzíven használja a vermet.

Feladatok

- 16.1. Készítsen olyan rekurzív függvényt, amely kiszámítja (*RekurzivFugg.java*)
- (A) egy egész szám faktoriálisát. $N! = 1*2*3*...*(N-1)*N$.
 - (B) két szám között az egész számok szorzatát, beleértve a határokat!
- 16.2. (A) Rendezzen gyorsrendezéssel egy szövegekből álló tömböt! (*RendezSzoveg.java*)
- 16.3. (B) Rendezzen gyorsrendezéssel egy objektumokból álló tömböt; a rendezés az objektum egyik tulajdonsága szerint csökkenő, s azon belül egy másik tulajdonsága szerint növekvő legyen! Rendezze például a Versenyző (pont, név) objektumokat! (*RendezObj.java*)
- 16.4. (C) Oldja meg a „Hanoi tornyai” programot grafikus felületen! Tegyen a keretbe három oszlopot: az elsőn legyenek a korongok, a másik két oszlop legyen üres. A korongokat csak a megadott szabályok szerint lehessen raktatni.
- A feladatot oldja meg a gép, rekurzív algoritmussal; a felhasználó csak nézi a korongok automatikus raktatását! (*HanoiGep.jpx*)
 - A feladatot oldja meg a felhasználó; ő raktassa a korongokat a maga elképzelése szerint! (*HanoiFeh.jpx*)
- 16.5. (B) Keressen meg a felhasználói interfész tulajdonosi hierarchiáján (a komponensfán) egy megadott tulajdonságú komponensem! Keresse meg például a `JFileChooser` komponensei között a "File name:" feliratú címkét (*KomponensKeres.jpx*)

17. Többszálú programozás

A fejezet pontjai:

1. A programszál fogalma
 2. A Thread osztály és a Runnable interfész
 3. Szinkronizáció: wait, notify
 4. Programszálak appletben
-

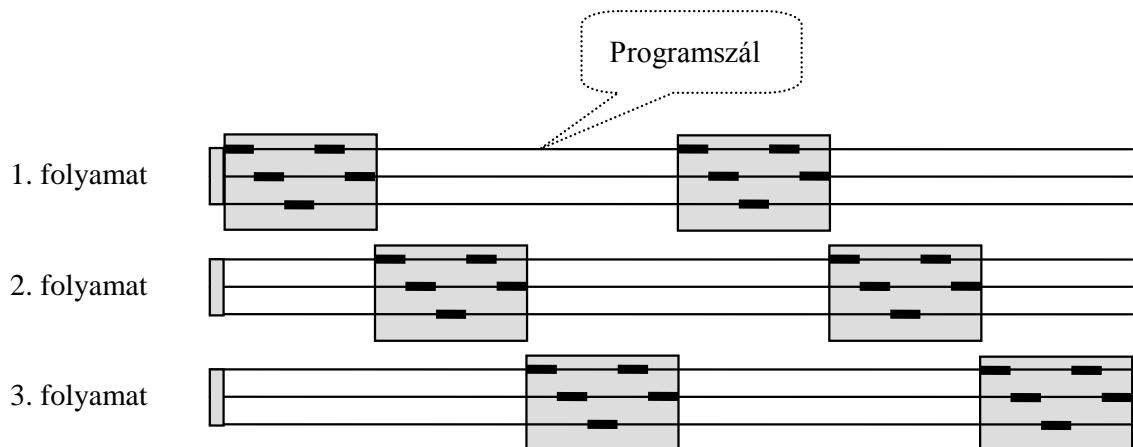
A többszálú programozás jóvoltából programunkban egyszerre több programszál (utasítás-sorozat) is futhat. A program ezáltal sokkal jobban kihasználja a processzort, vagyis sokkal rugalmasabb és hatékonyabb lehet. Egy programszál időről időre blokkolt állapotba jut: ha például adatot vár a felhasználótól, akkor lényegében nem csinál semmit. A felhasználónak már természetes, hogy számítógépe egyszerre több feladatot lát el: miközben a gép betölt egy képet, ő átkapcsol a szövegszerkesztőbe, és levelet ír. A többszálú programozás különösen jól használható animációk készítésére.

Ebben a fejezetben először egy egyszerű, konzolos példán mutatjuk be a szálkészítés technikáját. Ezután arról lesz szó, hogyan osztozkodhat több szál „összegabalyodás” – logikai vagy futási hiba nélkül – egy-egy objektumon. Végül appletben is működtetni fogunk szálakat.

17.1. A programszál fogalma

A **folyamat** (process, task, job) a számítógép által végrehajtott utasítássorozat. Az egyfeladatos operációs rendszerben egyszerre csak egy folyamat futhat, a többfeladatos operációs rendszerben egymással párhuzamosan több is. A párhuzamos működést a felhasználó alighanem egyidejűnek látja; pedig ha a számítógépen csak egy processzor van, akkor szó sincs egyidejűségről: az operációs rendszer megfelelő szabályok szerint bizonyos időtartamokra – „időszeletekre” – odaadja a vezérlést a programoknak. A **programszál** (thread) hasonlít az operációs rendszerben futó folyamatra: ő is csak időnként kapja meg a vezérlést. A 17.1. ábrán látható, hogy a folyamatok szálakból állnak. Amikor ez vagy az folyamat (program) fut, akkor az ő szálai közt oszlik meg a vezérlés. Ha a folyamat éppen blokkolva van (valami miatt nem futhat, még ha megkapná is a CPU-tól a vezérlést), akkor nem futhat a benne „futó” programszál sem.

Az éppen futó szálszeleteket (utasítássorozatokat) az ábrán vastagon jelöltük – egy időben láthatólag minden csak egy ilyen utasítássorozat fut.



17.1. ábra. Folyamatok, programszálak

Alapfogalmak

A **programszál** (thread, egyszerűen szál) egy utasítássorozat, amely valahol elkezdődik és valahol befejeződik. A programszálak párhuzamosan futnak; futásukat a szálütemező ütemezi. A programszál megszakítható, megadott ideig felfüggeszthető és újraindítható. Több programszál párhuzamos működését szinkronizálni is lehet.

A programszál állapota: A programszál megszületik, működik és meghal; születésétől a haláláig a következő állapotokba juthat:

- ◆ **Új** (new): A szál a megszületése után nyomban ebbe az állapotba kerül és addig marad meg benne, amíg el nem indítják (start).
- ◆ **Futtatható** (runnable): A szál akkor futtatható, ha elindították és még él. Ennek az állapotnak két alállapota van:
 - **Futó** (run): A szál aktív, legfeljebb a CPU-ra vár. Ha egy programszál nincs blokkolva, akkor aktív (fut).
 - **Blokkolt**, ha valami akadályozza a szál futását – egy hosszabb metódus fut, vagy:
 - **Alvó** (sleep): A szálat el lehet altatni valamennyi időre. Ha ez az időtartam letelik, akkor a programszál „felébred”, és fut tovább – mintha mi sem történt volna.
 - **Várakozó** (wait): A szál addig vár, amíg fel nem ébresztik (notify). Ha jön az értesítés, akkor a programszál fut tovább.
- ◆ **Halott** (dead): Egy szál halott, ha befejezte futását (a run végére ért).

Szinkronizáció: A folyamatok, szálak futását összehangoló kapcsolat: az egyik programszál mindaddig felfüggeszti a futását (vár – wait), ameddig egy másik programszál el nem jut egy megadott pontig (s erről értesítést nem küld – notify).

Monitor: A szálaknak egy megadott objektumhoz való hozzáférését figyeli, ellenőrzi.

Szálcsoport (thread group): A szálak szálcsoportba foghatók, és ezáltal együtt kezelhetők. A csopotok egymásba ágyazhatók.

Prioritás (priority): A szálhoz prioritásérték tartozik. A szálütemező két szál közül minden a nagyobb prioritású szálnak ad engedélyt a futásra. Egy szál prioritása alapértelmezésben ugyanakkora, mint a szülőjéé – azé a szálé, amely őt létrehozta. A prioritás menet közben megváltoztatható.

Démonszál (daemon thread): Abban különbözik egy „normális” száltól, hogy csak kiszolgáló szerepet játszik: ha más szál már nem fut a programban, akkor létezése „értelmetlené válik” és meghal.

| *Megjegyzés:* Ebben a könyvben nem tárgyaljuk meg a száltechnika összes lehetőségét.

Java programszál

A Javában a programszál is egy objektum; osztálya a `Thread` osztály leszármazottja. S a szálobjektumok egy-egy objektumot „futtatnak”. A futtatott objektum – ha lehetőséget kap rá – végzi a dolgát; a szál pedig fut, alszik, vár, meghal... A következő programszálakkal már eddigi programjainkban is találkoztunk:

- ◆ fő szál: a `main` metódus – általában hamar lefut;
- ◆ AWT-szál: akkor keletkezik, ha létrehozunk egy keretet vagy appletet. Lényegében egy megszakítható végtelen ciklus.

A programozó maga is készíthet szálobjektumot; csak meg kell adnia, hogy a születendő szálobjektumnak milyen objektumot kell majd futtatnia. A futtatott objektumnak futtathatónak (`Runnable`) kell lennie; ez a biztosíték arra, hogy lesz `run()` metódusa. A szálat ennek a `run` metódusnak a meghívásával lehet futtatható állapotba hozni; s ha a `run` lefutott, akkor a szál meghal.

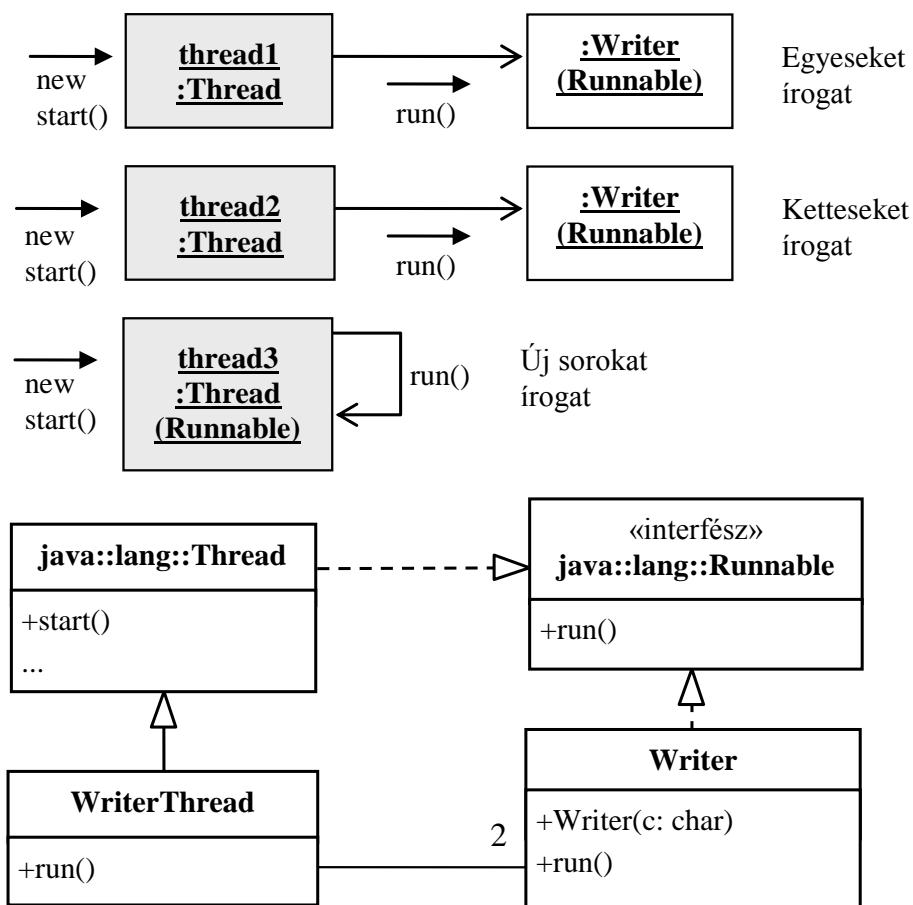
A következő egyszerű mintaprogramban kétféle szálat hozunk létre: (1) a szálobjektum más lesz, mint a futtatható objektum; (2) a szálobjektum egybeesik a futtatható objektummal, vagyis a szál önmagát futtatja.

Mintaprogram

A programszálak működésének lényegét egy egyszerű karakterírogató programon mutatjuk be:

Feladat – ThreadTest

Hozzunk létre három programszálat; az első kettő 1/20 másodpercenként írjon ki a konzolra egy megadott karaktert, a harmadik pedig 1/10 másodpercenként emeljen sort (írjon ki egy Enter-t)! Futtassuk a szálakat!



17.2. ábra. A programszálak a futtatható objektumaikat futtatják

Programterv

A program tervét a 17.2. ábra mutatja. Felül az látható, hogy létrehozunk egy `thread1` programszálat: `new Thread(new Writer('1'))` – a konstruktorral átadtunk neki egy újonnan létrehozott `Writer` objektumot, amely „vég nélkül” egyeseket írogat a konzolra. Ezt a programszálat a `start` metódussal nyomban útjára is bocsátjuk. A `Writer` objektum futtatható, vagyis implementálja a `Runnable` interfészét, a programszál tehát meghívhatja a `Writer`-objektum `run` metódusát.

Ezután létrehozzuk a `thread2` programszálat, és azt is elindítjuk. Most már két szál fut.

Végül létrehozzuk a `thread3` programszálat, de ő nem kap futtatandó objektumot, ezért magamagát futtatja: `new WriterThread()` – megteheti, mert már az ős `Thread` implementálja a `Runnable` interfészt egy üres `run` metódussal; a `WriterThread` osztályban ezt az üres metódust írtuk felül.

Forráskód – ThreadTest projekt

Writer.java

```
public class Writer implements Runnable {
    char c;
    public Writer(char c)
        this.c = c;
    }
    public void run() {
        while (true) {
            System.out.print(c);
            try {
                Thread.sleep(50);
            }
            catch (InterruptedException ie) {
            }
        }
    }
}
```

A `run` metódus (//1) egy végtelen ciklus. Az objektum addig írogatja a konstruktorban megkaptott karaktert, amíg valamilyen módon meg nem szakad a program futása. Az írást úgy lassítjuk, hogy két karakter kiírása között egy kicsit altatjuk a szálat: a `Thread.sleep` statikus metódus az éppen futó szálat altatja a megadott időtartamra (//2). A metódus `InterruptedException` ellenőrzött kivéttel ejthet, ha az éppen futó szálat egy másik szál meg akarná szakítani. A kivéttel elkapjuk, és nem törődünk vele (alszunk tovább ☺).

WriterThread.java

```
public class WriterThread extends Thread {
    public void run() {
        while (true) {
            System.out.println();
            try {
                Thread.sleep(100);
            }
            catch (InterruptedException ie) {
            }
        }
    }
}
```

A `WriterThread` csak a `run` metódust definiálja; ő sorvégjeleket írogat, egy kicsit lassúbb ütemben. A másik két szálnak marad tehát ideje néhány karaktert kiírni azelőtt, hogy ő „leütné” az Entert!

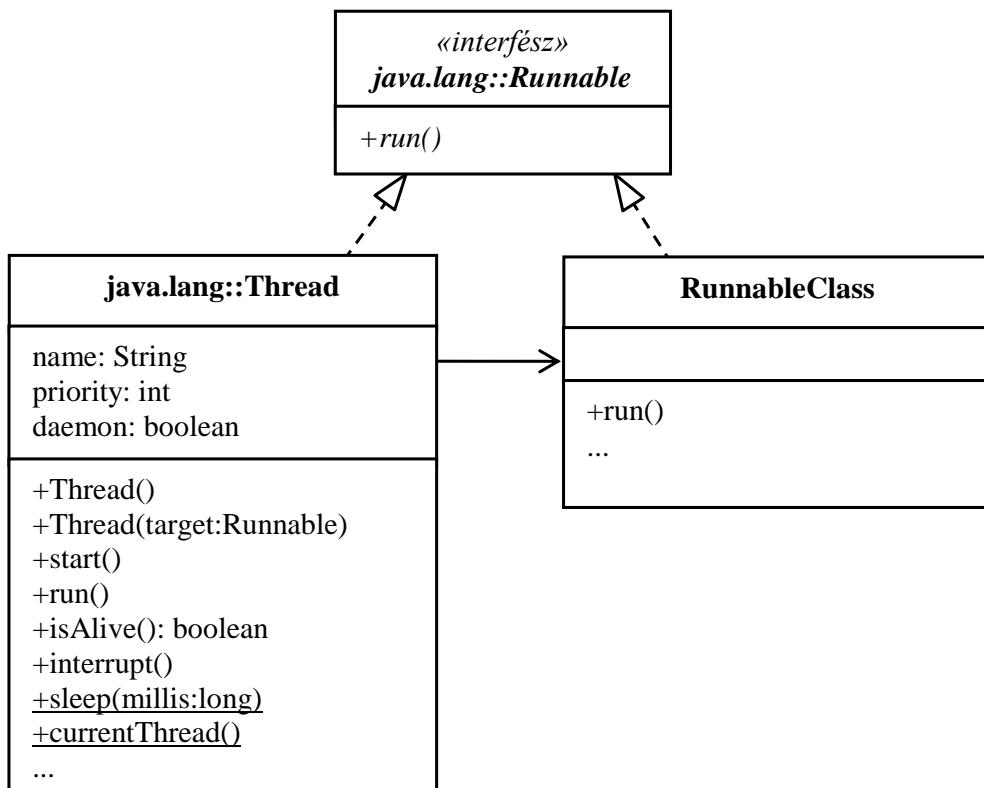
ThreadTest.java

```
public class ThreadTest {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new Writer('1'));
        thread1.start();
        Thread thread2 = new Thread(new Writer('2'));
        thread2.start();
        Thread thread3 = new WriterThread();
        thread3.start();
    }
}
```

A ThreadTest csak létrehozza a szálakat és elindítja őket. A programnak akkor lesz csak vége, ha kívülről „lelöjük”.

A program futása

```
12
12
1212
1212
...
...
```

17.2. A Thread osztály és a Runnable interfész

17.3. ábra. A Thread és a Runnable

A Runnable interfész

Csomag: `java.lang`

Deklaráció: `public interface Runnable`

Egy programszál csak olyan objektumot futtathat, amelynek az osztálya implementálta a Runnable interfészt (17.3. ábra). A Runnable interfészt a Thread osztály is implementálja, objektumai tehát önmagukat is futtathatják.

Metódus

- ▶ `public abstract void run()`

Amikor egy szál elkezd futni, akkor ennek a metódusnak az implementációját futtatja. A programszál maga is addig fut, ameddig a run.

Gondoljuk végig, mi célt is szolgál a Runnable interfész? Miért nem felel meg nekünk az a `run` metódus, amely a `Thread` osztályban eleve benne van? Nem elég az, hogy a futtatható objektum osztálya leszármazottja a `Thread`-nek? Azért nem elég, mert előfordulhat, hogy mondjuk, egy grafikus komponenst szeretnénk futtatni; s akkor a grafikus komponens osztályának két osztályból kellene leszármaznia: a `JComponent`-ből és a `Thread`-ből – csakhogy a Javában nincs többszörös öröklés! A Runnable interfész bevezetése erre a nehézségre ad feloldást.

A Thread osztály

Csomag: `java.lang`

Deklaráció: `public class Thread implements Runnable`

A Javában a programszál osztálya a `java.lang.Thread` osztály utódja. minden programszálnak van egy futtatható objektuma. A futtathatoságról a Runnable interfész `run()` metódusa gondoskodik: a szál újra és újra megakasztja és megint elindítja ezt a metódust. A programszálat úgy lehet végleg megállítani, hogy értékét `null`-ra állítsuk.

A JVM elindításakor a `main` hívásával elindul a program fő szála. Az első keret létrehozása az AWT-szálat indítja el; s a programozó további szálakat indíthat (például képbetöltéssel). A JVM addig futtatja a szálakat, amíg el nem hagyjuk a programot (`System.exit`), vagy az esetleges démonszálakat leszámítva az összes szál abba nem hagyja a futást (a `run` végére ér vagy kivételt ejt).

Konstruktur

- ▶ `Thread ()`
- ▶ `Thread (Runnable target)`

Létrehozza a szálat. `target` a futtatandó – egy, a Runnable interfészt implementáló osztályból való – objektum. Alapértelmezés: `this`

Jellemzők

- ▶ `name`
A szál neve. Alapértelmezés: Thread-1, Thread-2 stb.
- ▶ `priority`
A szál prioritása, 1 és 10 közötti érték. Alapértelmezés: a szülőszál prioritása – azé, amely őt létrehozta őt. A fő szál prioritása 5.
- ▶ `daemon`
Ha értéke true, akkor a szál démonszál. Alapértelmezés: false.

Metódusok

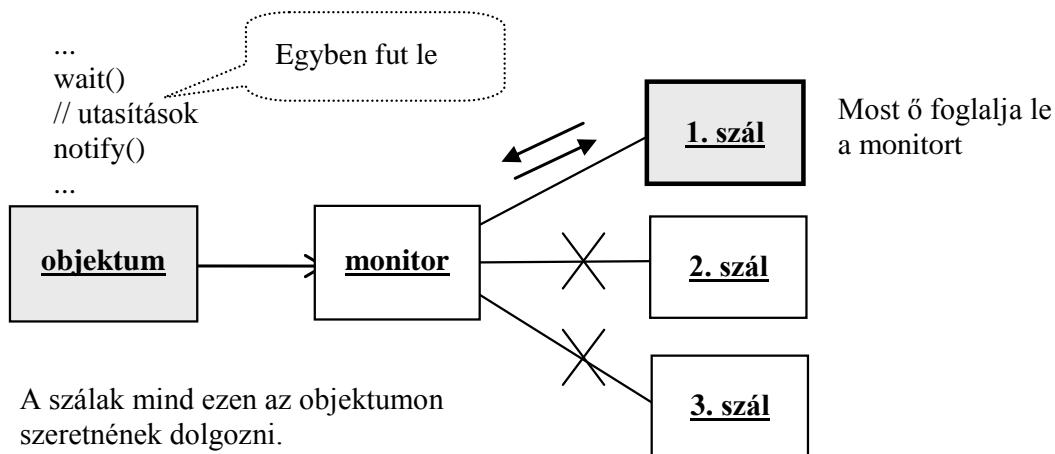
- ▶ `public void start()`
Elindítja a programszálat. Meghívja a `Runnable` típusú objektum `run()` metódusát. A `start` metódus azonnal visszatér; az új szál párhuzamosan fut a többi programszállal.
- ▶ `public void run()`
A `Runnable` interfész üres implementációja. Akkor írjuk felül, ha a szál önmagát futtatja.
- ▶ `final boolean isAlive()`
Viszaadja, hogy a szál élő-e. Egy szál akkor élő, ha elindították (`start`) és még nem halott (nincs vége a `run`-nak).
- ▶ `void interrupt()`
Egy másik szál megkísérli megszakítani a szálat. Ha a szál blokkolva van, akkor beállítja a megszakításjelzöt, és egy `InterruptedException` kivételt ejt – ezt a futó szál figyelembe veheti vagy elejtheti.
- ▶ `static void sleep(long millis) throws InterruptedException`
`long` ezredmásodpercre felfüggeszti az éppen futó szál futását (elaltatja). A szál nem veszíti el monitorjait.
- ▶ `static Thread currentThread()`
Visszaadja az éppen futó szálobjektumot.

Megjegyzés: **Nem biztonságosak**, és ezért elavultak a `Thread` osztály következő metódusai: `suspend` (felfüggeszt), `resume` (újraindít), `stop` (megállít).

17.3. Szinkronizáció: `wait`, `notify`

Egy objektumhoz több programszál is hozzáférhet – s hol az egyik szál dolgozik rajta, hol a másik. Ilyenkor fontos lehet, hogy bizonyos metódusok egyvégtében fussanak le – nem lehet félbehagyni például egy banki tranzakciót (az egyik számláról levenni valamekkora összeget, de a másikra már nem rátenni) vagy egy raktári bevételezést (magnézni, hogy befér-e a megadott mennyiség a raktárba, de csak később betenni, amikor már esetleg nem fér be). Az

objektum megteheti, hogy addig várat egy szálat, amíg nem teljesülnek bizonyos feltételek: az objektum addig blokkolja a várakozó szálat (az nem tud továbbfutni). Amíg a várakozó szál le nem fut, addig más szál sem fér az objektumhoz.



17.4. ábra. Az objektum monitora szinkronizálja a szálakat

A **szinkronizáció** folyamatok, szálak futásának összehangolása (17.4. ábra). A szinkronizációt az objektum **monitora** végzi. Az objektum blokkjai szinkronizálhatók. Egy szinkronizált (**synchronized**) blokk utasításokat fog össze és csak együtt engedi végrehajtani őket.

Metódus szinkronizálása (a blokkolt objektum a this):

```
synchronized <void/típus> <metódusnév>() {
    // utasítások
}
```

Blokk szinkronizálása (a megadott objektumot blokkolja):

```
synchronized(Object) {
    // utasítások
}
```

Ha egy szál meghívja az objektum valamely szinkronizált metódusát, akkor azzal zárolja az objektumot: amíg ez a szinkronizált metódus le nem fut, addig más programszál nem hívhatja meg az objektum egyetlen szinkronizált metódusát sem.

Az objektum monitora az objektumot futtató programszálakat figyeli, monitorozza. Ha egy objektumnak van legalább egy szinkronizált blokkja, akkor van monitora is.

Az Object osztályban van egy `wait` (várakozás) és egy `notify` (értesítés) metódus: a `wait` bejelenti igényét az objektumnak; a `notify` pedig felébreszti a futó szálat.

A wait és a notify metódus párból áll: ugyanabban a szinkronizált blokkban (metódusban) kell végrehajtódniuk – előbb jön a `wait`, azután a `notify`. Ha a blokk meghív egy másik blokkot, akkor azt is szinkronizálni kell! Ez kezeskedik arról, hogy a várakozás után a monitort lefoglaló programszál kapja meg a vezérlést!

Osztályszinten is lehetséges szinkronizáció: minden osztálynak lehet egy, a szinkronizált statikus metódusokat felügyelő monitora.

Az Object osztály metódusai

- ▶ `final void wait() throws InterruptedException`
- ▶ `final void wait(long timeout) throws InterruptedException`

Az éppen futó szál bejelenti igényét erre az objektumra. A szál addig blokkolt állapotba kerül, ameddig az objektum monitora fel nem ébreszti a `notify()` vagy a `notifyAll()` meghívásával, illetve le nem telik a `timeout` ezredmásodpercnyi idő. A `wait` metódus csak szinkronizált metódusban szerepelhet!

- ▶ `final void notify()`
- ▶ `final void notifyAll()`

Felébreszti azt a programszálat, amelyik ennek az objektumnak a monitorára vár. Ha más szál is várakozik erre az objektumra, akkor a nagyobb prioritású szál kapja meg a vezérlést; ha a prioritások megegyeznek, akkor a választás véletlenszerű. A `notify` metódus csak szinkronizált metódusban szerepelhet, a `wait` után!

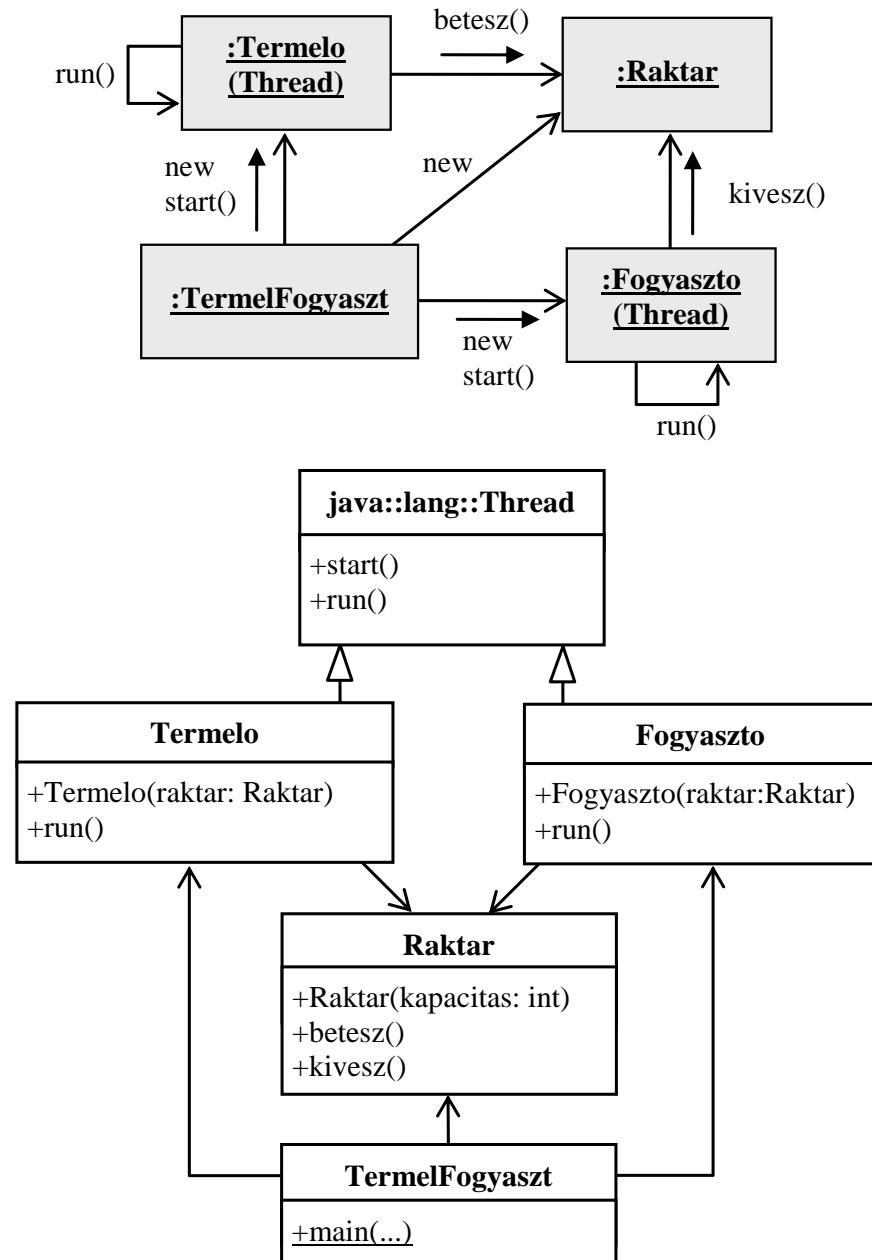
Feladat

Egy raktárban összesen 1000 árut lehet tárolni (az egyszerűség kedvéért csak egyféle árut). Kezdetben nincs benne áru. A termelő termeli az árut: 5 másodpercenként 1 és 100 közötti számú ár elő és beteszi a raktárba. A fogyasztó fogyaszt a raktárból: másodpercenként 1 és 20 közötti számú árut vesz ki.

Szimuláljuk a raktár működését!

A feladat terve a 17.5. ábrán látható. A `TermelFogyaszt` osztálynak csupán az a feladata, hogy létrehozzon egy raktárat, egy termelőt és egy fogyasztót (többet is létrehozhatna belőlük, de akkor nehezebb lenne áttekinteni a feladatot). A termelő és a fogyasztó önálló programszál: mindenki végzi a dolgát a másik száltól (majdnem) függetlenül. A termelő folyamatosan rakosztja be áruját a raktárba, a fogyasztó pedig folyamatosan fogyasztja a raktárkészletet. Mindkettő a maga tempójában dolgozik, s nem aggódik előre a „túltermelés” vagy a „túlfogyasztás” miatt. Akkor lesz csak fennakadás, ha üres a raktár – mert akkor nem lehet belőle árut kivenni –, vagy ha tele van – mert akkor nem lehet árut beletenni.

Programterv



17.5. ábra. A TermelFogyaszt programterve

Projekt: TermelFogyaszt.jpx**Termelo.java**

A termelő külön programszál, önállóan működik: másodpercenként véletlen számú árut termel, s azt megpróbálja betenni a raktárba. A termelő hozzá van „kötve” egy raktárhoz; hogy melyikhez, azt a konstruktorban adjuk meg. A betevés eltarthat egy ideig, ha az áruja éppen most nem fér be a raktárba (lásd a `Raktar.betesz` metódust). A termelő az „idők végezetéig” termel:

```
public class Termelo extends Thread {
    private Raktar raktar;

    public Termelo (Raktar raktar) {
        this.raktar=raktar;
    }

    public void run() {
        try {
            while (true) {
                sleep(5000); // "alvás közben" termel
                int mennyiseg = (int) (Math.random() * 100) + 1;
                System.out.println("Betennék: " + mennyiseg);
                raktar.betesz(mennyiseg); // beteszi a raktárba
            }
        } catch (InterruptedException e) {
        }
    }
}
```

Fogyaszt.java

A fogyasztó külön programszál, önállóan működik: másodpercenként véletlen számú árut igyekszik a raktárból kivenni és elfogyasztani. A fogyasztó hozzá van „kötve” egy raktárhoz; hogy melyikhez, azt a konstruktorban adjuk meg. A vételezés eltarthat egy ideig, ha éppen most nincs a raktáron elegendő áru (lásd a `Raktar.kivesz` metódust). A fogyasztó az „idők végezetéig” fogyaszt:

```
class Fogyaszt extends Thread {
    private Raktar raktar;

    public Fogyaszt(Raktar raktar) {
        this.raktar = raktar;
    }
}
```

```
public void run() {
    try {
        while (true) {
            sleep(1000); // "alvás közben" fogyaszt
            int mennyiseg = (int) (Math.random() * 20) + 1;
            System.out.println("Kivennek: " + mennyiseg);
            raktar.kivesz(mennyiseg); // kivesz a raktárból
        }
    } catch (InterruptedException e) {
    }
}
```

Raktar.java

A raktár nem programszál – csak akkor dolgozik, ha külön kérík rá. A betesz és a kivesz szinkronizált metódus, vagyis egyvégtében futnak le, más programszál nem szakíthatja meg őket:

```
public class Raktar {
    private int kapacitas;
    private int keszlet;

    public Raktar(int kapacitas) {
        this.kapacitas = kapacitas;
        keszlet = 0;
    }

    public synchronized void kivesz(int mennyiseg) {
        // Ha van elegendő áru, kiszolgáljuk:
        if (mennyiseg <= keszlet) {
            keszlet -= mennyiseg;
            System.out.println("Kivettek, készlet: " + keszlet);
        } else {
            System.out.println("Sajnos nem tudunk adni!");
            // Vár, amíg lesz ennyi. Addig nem szolgál ki más:
            try {
                while (keszlet < mennyiseg)
                    wait();
            }
            catch (InterruptedException ie) {
            }
        }
        // Értesíti a szálakat, hátha vár rá valaki:
        notifyAll();
    }
}
```

```
public synchronized void betesz(int mennyiseg) {  
    // Ha van elegendő hely, betesszük:  
    if (keszlet + mennyiseg <= kapacitas) {  
        keszlet += mennyiseg;  
        System.out.println("Betettek, készlet: " + keszlet);  
    }  
    else {  
        System.out.println("Nem fér be, várni kell!");  
        // Vár, amíg lesz annyi hely:  
        try {  
            while (keszlet + mennyiseg <= kapacitas)  
                wait();  
        }  
        catch (InterruptedException ie) {}  
    }  
    notifyAll();  
}
```

TermelFogyaszt.java

```
public class TermelFogyaszt {  
    public static void main(String[] args) {  
        // A raktár 1000 darabbal indul:  
        Raktar raktar = new Raktar(1000);  
        // A termelő a raktárból vesz:  
        Termelo termelo = new Termelo(raktar);  
        // A fogyasztó a raktárból fogyaszt:  
        Fogyaszto fogyaszto = new Fogyaszto(raktar);  
        // Indul a gépezet:  
        termelo.start();  
        fogyaszto.start();  
    }  
}
```

A program futása

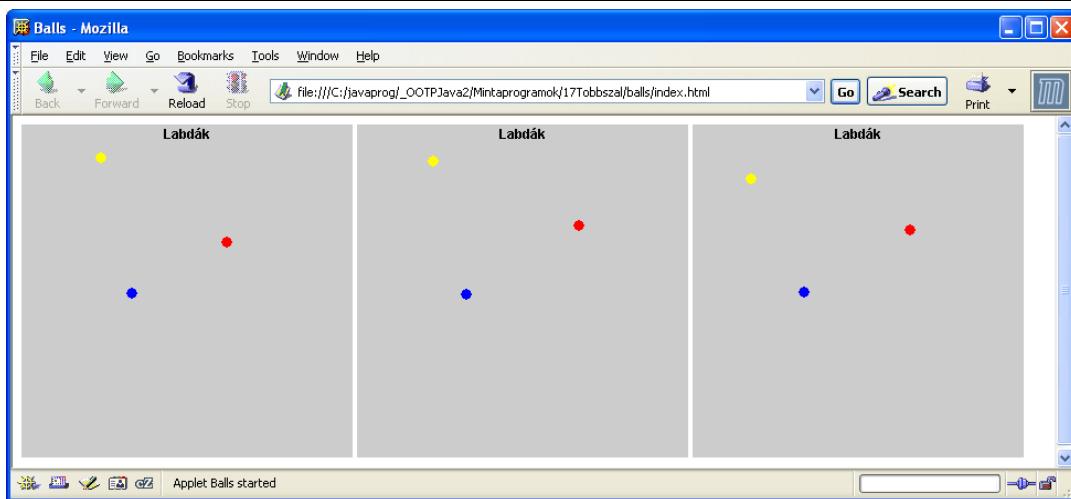
```
Kivennék: 6  
Sajnos nem tudunk adni!  
Betennék: 1  
Betettek, készlet: 1  
Betennék: 95  
Betettek, készlet: 96  
Kivennék: 7  
Kivettek, készlet: 89
```

17.4. Programszálak appletben

Gyakorlásképpen most fésüljünk össze két technikát: hozzunk létre sok szálat egy appletben!

Feladat

Készítsünk egy olyan appletet, amelyben apró, színes labdácskák pattognak rugalmas ütközéssel! Tegyük három példányt belőle a böngészőbe!



Projekt: Balls

Ball.java

A Ball egy futtatható labda. Van szülőpanelje – azon van rajta; van helye, színe, iránya és sugara. El mozdulhat (//1), s közben rugalmasan visszaverődik a panel szegélyéről. A labda szálobjektum (a Thread osztályból származik), nem lehet tehát komponens; de hozzákötöttük a szülőpanelhez – majd az megjeleníti. A labdaobjektum ezenfelül arrébb mehet (//1), kirajzolja magát a kapott panelre (//4), és futhat is (//3), a `setActive(boolean)` metódussal leállítható és újraindítható. Sajnos a szálat kívülről nem lehet felfüggeszteni; a `suspend` metódus nem kívánatossá vált (deprecated), mert nem volt biztonságos. A felfüggesztést most a `setActive` metódus helyettesíti, s ezt a metódust szinkronizálnunk kell (//2) – rosszul járnánk, ha a labda `active` tulajdonságát `true`-ra állítanánk és azután egy másik labdát indítanánk el. A `run` metódusban a labda egy lépését is szinkronizáljuk.

```
import javax.swing.*;
import java.awt.*;
import java.awt.geom.*;

public class Ball extends Thread {
    JPanel panel; // ezen a panelen pattog a labda
    int x, y;      // középpont
    Color color;
    int dx=1, dy=1; // irány
    int RADIUS = 10; // sugar
    boolean active = false;
```

```

public Ball(int x, int y, Color color, JPanel panel) {
    this.x = x;
    this.y = y;
    this.color = color;
    this.panel = panel;
}

public void move() { //1
    x += dx;
    if (x < 0 || x > panel.getWidth())
        dx *= -1;
    y += dy;
    if (y < 0 || y > panel.getHeight())
        dy *= -1;
    panel.repaint();
}

public synchronized void setActive(boolean active) { //2
    this.active = active;
    if (active)
        notify();
}
}

public void run() { //3
    while (true) {
        synchronized(this) {
            try {
                move();
                sleep(5);
                while (!active)
                    wait();
            }
            catch (InterruptedException ie) {
            }
        }
    }
}

// Kirajzolódik a kapott panelre:
public void draw(Graphics gr) { //4
    gr.setColor(color);
    gr.fillOval(x,y,RADIUS,RADIUS);
}
}

```

BallPanel.java

A BallPanel-nek van egy labdákat tartalmazó konténere (//1); ebbe beteszünk három labdát (//2). //3 kirajzolja a panelt az összes labdával együtt; az újrarajzolást a labda elmozdulásakor kérjük. A start metódussal elindítjuk az összes szálat (//4); a setActive metódus meghívja az összes labda setActive metódusát (//5).

```

import javax.swing.*;
import java.util.*;
import java.awt.*;

class BallPanel extends JPanel {
    Vector balls = new Vector(); //1

    public BallPanel() { //2
        balls.add(new Ball(10,150,Color.RED,this));
        balls.add(new Ball(200,150,Color.BLUE,this));
        balls.add(new Ball(280,10,Color.YELLOW,this));
    }

    public void paintComponent(Graphics gr) { //3
        super.paintComponent(gr);
        for (int i = 0; i < balls.size(); i++) {
            ((Ball)balls.get(i)).draw(gr);
        }
    }

    public synchronized void start() { //4
        for (int i = 0; i < balls.size(); i++)
            ((Ball)balls.get(i)).start();
    }

    public synchronized void setActive(boolean active) { //5
        for (int i = 0; i < balls.size(); i++) {
            Ball ball = (Ball)balls.get(i);
            ball.setActive(active);
        }
    }
}

```

Balls.java

Az applet főosztályában létrehozzuk a BallPanel-t és rátesszük a tartalompanelre. Felülírjuk az applet start és stop metódusát, hogy a labdák ne mozogjanak, ha éppen nem látjuk őket.

```

import javax.swing.*;
import java.util.*;
import java.awt.*;

public class Balls extends JApplet {
    Container cp = getContentPane();
    BallPanel ballPanel;

    public void init() {
        cp.add(new JLabel("Labdák",JLabel.CENTER),"North");
        cp.add(ballPanel = new BallPanel());
        ballPanel.start();
    }
}

```

```

public void start() {
    ballPanel.setActive(true);
}

public void stop() {
    ballPanel.setActive(false);
}
}

```

Tesztkérdések

- 17.1. Mely állítások igazak? Jelölje be az összes jó választ!
- A többfeladatos operációs rendszer a folyamatokat párhuzamosan futtatja.
 - Egy programszál több folyamathoz is tartozhat.
 - A programszál vagy új, vagy futatható, vagy halott.
 - A démonszálnak mindenkor nagyobb a prioritása, mint egy „normális” szálnak.
- 17.2. Mely állítások igazak? Jelölje be az összes jó választ!
- A programszál is objektum a Javában, és egy objektumot futtat.
 - A szálobjektum osztálya leszármazottja a Thread osztálynak.
 - A Runnable interfész csak a start nevű metódust deklarálja.
 - A Thread osztálynak nincs egyparaméteres konstruktora.
- 17.3. Mely állítások igazak? Jelölje be az összes jó választ!
- Egy objektumot csak egyetlen programszál futtathat.
 - Az objektum monitora maga is objektum.
 - A wait és a notify metódusnak mindenkor közös, szinkronizált blokkban kell lefutnia.
 - Csak utasításblokkot lehet szinkronizálni, metódust nem.
- 17.4. Mely állítások igazak? Jelölje be az összes jó választ!
- A Thread osztály nem implementálja a Runnable interfészt.
 - Minden osztályban van wait és notify metódus.
 - A szinkronizáció a folyamatok, szálak futásának összehangolása.
 - A szálat a sleep metódussal el lehet altatni egy adott időre.

Feladatok

- 17.1. (A) Készítsen egy olyan keretet, amely 10 másodpercenként felbukkan a képernyőn, még akkor is, ha közben elrejtették! (*Felbukkan.jpx*)
- 17.2. (B) Tegyen egy lapra két szöveget! Írjon beléjük egy-egy szöveget, lassítva, egymástól független sebességgel!
(*LassítottIras.jpx*)

- 17.3. **(B)** Tegyen egy keretbe két forgó egyenest! A keret az egyenesek forgása közben mozogjon lassan balról jobbra a képernyőn! (*MindenMozog.jpx*)
- 17.4. **(B)** Készítsen különböző sebességgel forgó légcsavarokat!
(*Propellerek.jpx*)
- 17.5. **(C)** Az appletben tűnjön föl több forgó légcsavar is, s mindegyik alatt legyen egy "Állj" és egy "Indít" gomb – azok állítsák meg, illetve indításával a légcsavarok forgását! Ezenkívül görgetősávokkal szabályozni lehessen a légcsavarok forgási sebességét! (*forgaskavalkad.jpx*)
- 17.6. **(C)** Készítse el a 8. fejezet *Fenyujsagok.jpx* feladatának megoldását szálak segítségével! A feladat így szól:
Tegyen a keretbe egymás alá három fényújságot! A fényújság jobbról balra haladva folyamatosan lépett egy szöveget. A fényújság szövegét, a betűk színét és a sebességet (léptetési időt) inicializáláskor adjuk meg.
- Az első fényújság egy egyszerű fényújság legyen.
 - A második fényújságot egy hozzácsatolt görgetősávval lehessen gyorsítani, és minden írójék ki, hogy mekkora a pillanatnyi sebesség.
 - A harmadik fényújságnak szövegét, a színét és a sebességét is lehessen állítani.
(*Fenyujsagok.jpx*)
- 17.7. Egészítse ki TermelFogyaszt programot:
- **(B)** Legyen a raktárnak nyitvatartási ideje! (*termelfogyaszt2.jpx*)
 - **(B)** Legyen több termelő és több fogyasztó! Mindenkihez csak egy raktár tartozhat! (*termelfogyaszt3.jpx*)
 - **(C)** Legyen több termelő és több fogyasztó! Bárki bárhová termelhet, illetve bárholnan fogyaszthat! (*termelfogyaszt4.jpx*)
 - **(C)** Készítse el appletben! Hozzon létre egy grafikus felületet az események szemléltetésére! (*termelfogyaszt4.jpx*)

18. Nyomtatás

A fejezet pontjai:

1. A nyomtatás technikája
 2. Mintaprogram – PrintHello
 3. Printable interfész, PrinterJob osztály
 4. Oldalformázás – PageFormat osztály
 5. Megjelenítés és nyomtatás
-

Java programból a rendszer által felkínált nyomtatón kinyomtathatunk egy megadott objektumot. A nyomtatást mindenkor egy `PrinterJob`-objektum végzi; a nyomtatandó objektumtól kéri el az oldalakat, egy `Graphics` típusú felületen. Kész dialógusokkal állíthatjuk be a felhasználóval a nyomtató tulajdonságait, a nyomtatandó oldaltartományt és az oldalméretet.

A fejezetben egyszerű mintafeladatokon keresztül próbáljuk szóra bírni a nyomtatót: először csak a háttérből (vakon) fogunk nyomtatni; majd a képernyőn megjelenő képet is a nyomtatóra küldjük. A nyomtatáshoz szükséges interfések és osztályok a `java.awt.print` csomagban vannak – itt csak a legfontosabbakat fogjuk áttekinteni közülük.

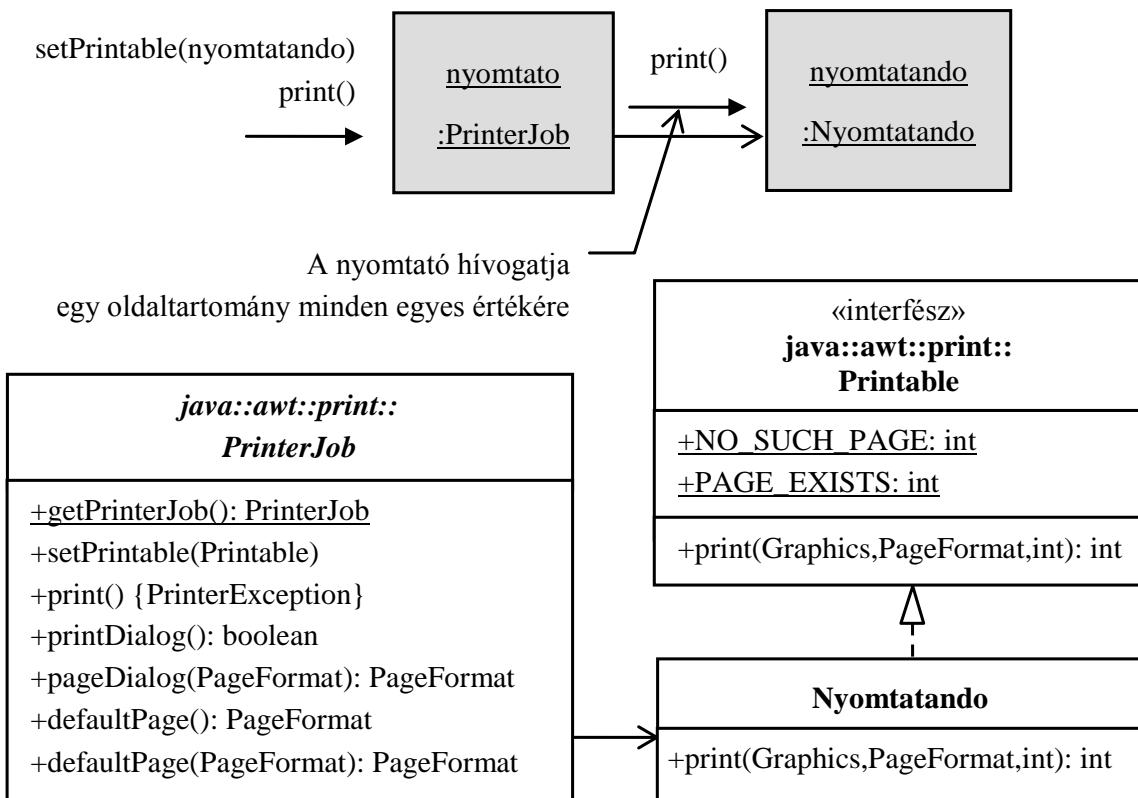
18.1. A nyomtatás technikája

A nyomtatást mindenkor egy `PrinterJob` (szó szerint nyomtatási munka, egyszerűen nyomtató) objektum végzi, mely valamilyen szabály szerint kinyomtat egy objektumot (18.1. ábra). Nyomtatóobjektumot a `PrinterJob` osztálytól kérhetünk:

```
PrinterJob nyomtato = PrinterJob.getPrinterJob();
```

A nyomtatóobjektumhoz hozzá kell rendelni a nyomtatandó objektumot, majd a `print` üzenettel elkezdhetjük a nyomtatást:

```
nyomtato.setPrintable(nyomtatando);  
nyomtato.print();
```



18.1. ábra. A nyomtatást a PrinterJob végzi

A nyomtatando objektum osztályának implementálnia kell a `Printable` interfészt; az egyébként csak egyetlen metódust definiál:

```
int print(Graphics gr, PageFormat pf, int pageIndex)
```

A `print` metódus a nyomtató objektumtól kapja a három paramétert:

- Grafikus objektum (`gr`): Erre kell rárajzolni a nyomtatandó oldalt (lapot). `gr` Graphics osztályú, és pontosan úgy használható, mintha egy komponensre rajzolnánk.
- Oldalformátum (`pf`, `page format`): Ebből kiolvasható az oldal mérete, a margók stb.
- Az elkészítendő oldal sorszáma (`pageIndex`): Ha a megadott sorszámú oldalt ki akar-juk nyomtatni, akkor a `print` visszatérési értéke `PAGE_EXISTS`. Ha a visszatérési érték egyszer `NO SUCH PAGE`, akkor onnan felfelé nincs nyomtatás. Csak folytonos oldal-tartományt lehet nyomtatni.

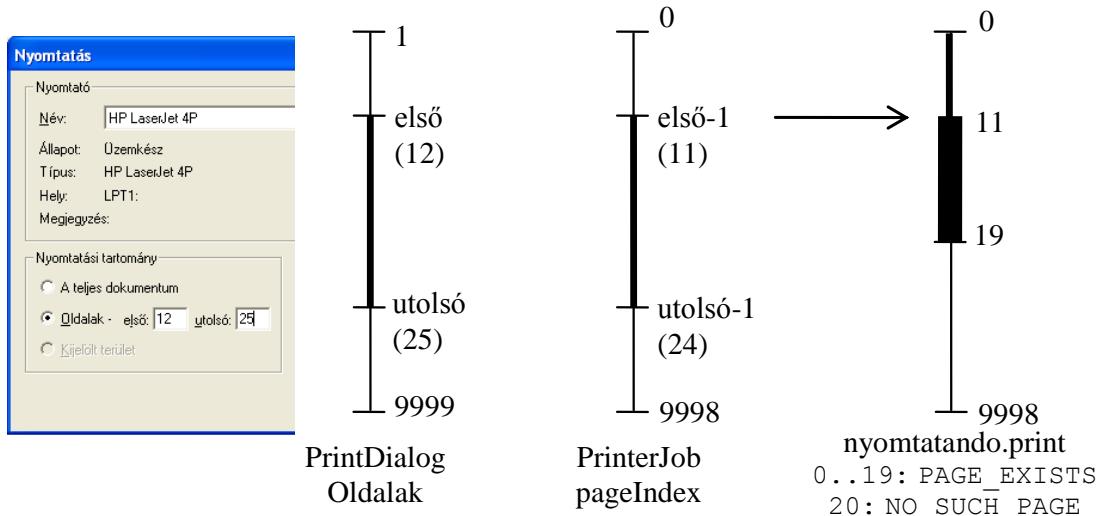
A nyomtatás úgy történik, hogy a nyomtató alapértelmezésben meghívja a nyomtatando objektum `print` metódusát sorban a `pageIndex=0..9998` oldaltartományra (ez a legbő-vebb tartomány). Ezt a tartományt leszükítheti:

- **a felhasználó**, ha felkínáljuk neki a nyomtatási dialógust (`printDialog`). A dialógus az 1..9999 oldalszámokat ajánlja fel, mert a felhasználó 1-től kezdi a számozást. A nyomtatóobjektum (`printerjob`) csak a dialógusban megadott oldalakra hívja meg a nyomtatandó objektum `print` metódusát (a `pageIndex`-nek egyetlen kisebb értéket ad át).
- **a programozó**, a `print` metódusban; ehhez el kell döntenie, hogy a paraméterben felkínált oldalszámú oldalt – amely egy 0-ról induló oldaltartomány sorszáma, – ki akarja-e nyomtatni vagy nem.

Végül a két intervallum közös részébe tartozó oldalak fognak kinyomtatódni. Az oldalak tartalmát a nyomtatandó objektum `print` metódusában kell megrajzolni.

A nyomtatóobjektum `printDialog` és `pageDialog` metódusa szabványos dialógusokat jelenít meg – a felhasználót rajtuk keresztül kérhetjük a nyomtatási paraméterek beállítására.

A 18.2. ábra egy konkrét nyomtatást szemléltet: a nyomtatási dialógus felkínálja az 1..9999 oldalakat. Ezt a felhasználó leszűkíti a 12..25 (első..utolsó) intervallumra. Ha nem kínálnánk fel a dialógust, akkor maradna az alapértelmezés: 1..9999. A `PrinterJob` levon 1-et az oldalszámokból, és átadja a tartományt (11..24) a `print` metódusnak. Tegyük fel, hogy a `print` metódus a 0..19 tartományt nyomtatná (0-19 értékekre igen-t mondott, a 20. oldalt visszautasította). A ténylegesen kinyomtatott tartomány ennek a két tartománynak a közös része lesz, a 11..19 oldal. Az oldalra természetesen olyan oldalszámot teszünk, amilyet akarunk (feltehetően a 12..20 számokat).



18.2. ábra. Az oldalszámok leképezése, szűkítése

A nyomtatóobjektum elvileg bármilyen sorrendben és akárhányszor meghívhatja a nyomtató objektum `print` metódusát, a programozónak ebbe nincs beleszólása. Elképzelhető, hogy egy oldal tényleges kinyomtatásához a `PrinterJob`-objektumnak többször is szüksége van ugyanarra az oldalra (bizonyos nyomtatók például sávonként nyomtatják ki a teljes oldalt). Ha a nyomtatási dialógusban a felhasználó fordította állította a nyomtatási sorrendet, akkor a nyomtató az indexeket visszafelé kínálja fel. A `print` metódusnak (vagyis a programozónak) csak azt kell eldöntenie, hogy kell-e nyomtatni az ilyen vagy olyan indexű oldalt, s ha kell, akkor mi legyen az oldal tartalma.

Megjegyzés: Az angol "page" szó jelentése nem egyértelmű: a szótárakban oldal, lap szerepel. A magyarban egy lap két oldalt jelent.

18.2. Mintaprogram – PrintHello

Feladat – PrintHello

Nyomtassuk ki a "Hello" szöveget!



Hello

Forráskód

```

import java.awt.print.*;                                //1
import java.awt.Graphics;

class Hello implements Printable {                      //2
    public int print(Graphics gr, PageFormat pf,
                      int pageIndex) throws PrinterException { //3
        if (pageIndex >= 1)                                //4
            return NO_SUCH_PAGE;
        gr.drawString("Hello",100,100);                    //5
        return PAGE_EXISTS;                               //6
    }
}

public class PrintHello {
    PrinterJob nyomtato = PrinterJob.getPrinterJob();   //7
    Printable nyomtatando = new Hello();                 //8

    public PrintHello() {
        nyomtato.setPrintable(nyomtatando);           //9
        try {
            nyomtato.print();                         //10
        } catch (PrinterException e) {                //11
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        new PrintHello();
    }
}

```

A program elemzése

- ◆ //1: A nyomtatáshoz importálnunk kell a `java.awt.print` csomagot.
- ◆ //2: A nyomtatandó oldalt a `Hello` objektum adja. A `Hello` osztálynak implementálnia kell a `Printable` interfészét, illetve annak `print` metódusát.
- ◆ //3: A `print` metódusnak három paramétere van. Ha nyomtatni akarjuk a `pageIndex` indexű oldalt, akkor azt rá kell rajzolni a `gr` grafikus objektumra. Az oldal formátuma `pf`. A metódus `PrinterException` kivételt dobhat, s azt kezelni kell.
- ◆ //4: Csak a 0 sorszámú oldalt nyomtatjuk, a többet nem. A `pageIndex>=1` értékeit a metódus a `Printable.NO_SUCH_PAGE` visszatérési értékkel utasítja vissza.
- ◆ //5: A grafikus felületre írjuk (rajzoljuk) a nyomtatandó szöveget. A grafikus felület (0,0) pontja a papír bal felső sarka, a margót is beleértve. A margóra írt dolgok nem jelennek meg az oldalon. Feltesszük, hogy a (100,100) pont a látható területen van.
- ◆ //6: A 0. oldal elkészítése után a metódus a `Printable.PAGE_EXISTS` értékkel tér vissza, jelezve, hogy ez az oldal nyomtatható.
- ◆ //7-8: A nyomtatáshoz létrehozunk egy `PrinterJob`-objektumot, majd létrehozzuk a nyomtatandó objektumot is.
- ◆ //9-10: A nyomtatót ráirányítjuk a nyomtatandó objektumra, és elindítjuk a nyomtatást.
- ◆ //11: Ha nyomtatás közben hiba történt, konzolra írjuk a kivétel metódusláncát.

18.3. **Printable** interfész, **PrinterJob** osztály

Printable interfész

Csomag: `java.awt.print`

Deklaráció: `public interface Printable`

A `PrinterJob` megköveteli, hogy az általa nyomtatandó objektum nyomtatható (`printable`) legyen, vagyis az objektum osztálya implementálja a `Printable` interfészét.

Konstansok

- ▶ `static int PAGE_EXISTS`
- ▶ `static int NO_SUCH_PAGE`

Van ilyen oldal, illetve nincs ilyen oldal.

Metódusok

- ▶ `int print(Graphics gr, PageFormat pf, int pageIndex)`

Elkészíti a kért `pageIndex` sorszámú oldalt a `Graphics` közege, `pf` oldalformátumban. A metódust a `PrinterJob`-objektum fogja meghívni, hogy elkészítesse a megadott oldalt a nyomtatáshoz. Ha a metóduson belül megszakad a nyomtatás, `PrinterException` kivétel keletkezik. A metódus visszatérési értéke `PAGE_EXISTS` vagy `NO_SUCH_PAGE` kell, hogy legyen aszerint, hogy az oldalt ki akarjuk-e nyomtatni

vagy sem. A nyomtatás az első NO SUCH PAGE érték visszaadásakor leáll. Az oldalon megjelenő konkrét oldalszám a pageIndex-ból számítható ki. Paraméterek:

- gr: A nyomtatandó oldal grafikus felülete;
- pf: Az oldal formátuma. A pf:PageFormat tárolja az oldal méretét, állását (fekvő vagy álló), valamint a margók nagyságát. Az oldalformátumot külön pont tárgyalja.
- pageIndex: A nyomtatandó oldal indexe.

● Vigyázat! Ha a visszatérési érték minden PAGE_EXISTS, akkor a nyomtatandó oldalak száma egyedül attól függ, hogy a felhasználó milyen tartományt ad meg a nyomtatódialógusban. Így akár 10 000 oldalt is kinyomtathatunk!

PrinterJob osztály

Csomag: java.awt.print Deklaráció: public abstract class PrinterJob
Közvetlen ős: java.lang.Object

A PrinterJob osztály a nyomtatás alaposztálya: vezérli a nyomtatást. PrinterJob-objektumot az osztály statikus getPrinterJob() metódusával hozhatunk létre.

Jellemzők

- ▶ String jobName
A nyomtatási feladat neve.
- ▶ int copies
A nyomtatandó példányszám.

Metódusok

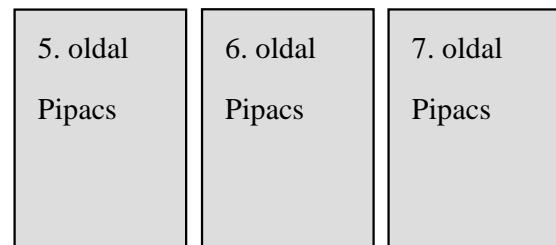
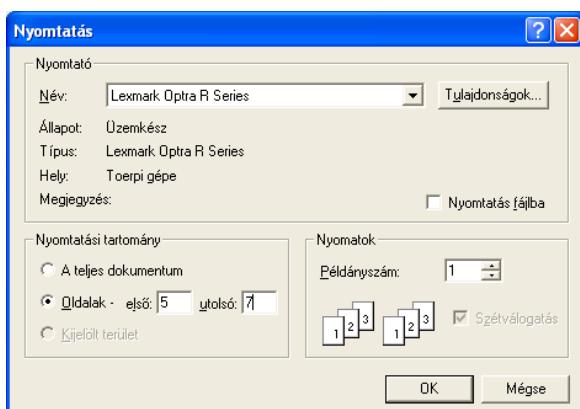
- ▶ static PrinterJob getPrinterJob()
Visszaad egy PrinterJob-objektumot.
- ▶ void setPrintable(Printable painter)
Az oldalak készítője painter lesz; osztályának implementálnia kell a Printable interfést (azzal lesz print metódusa). Az elkészítendő oldalak formátuma az alapértelmezés szerinti PageFormat.
- ▶ void setPrintable(Printable painter, PageFormat format)
Ugyanaz, mint az előző, de most beállítjuk az oldalformátumot is.
- ▶ void print() throws PrinterException
Kinyomtatja a setPrintable metódussal beállított objektumot. Az objektum print metódusát hívogatja 0-tól kezdődő oldalszámokkal mindaddig, amíg az a PAGE_EXISTS értékkel tér vissza. Amint NO SUCH PAGE lesz a visszatérési érték, leáll a nyomtatás.
- ▶ PageFormat defaultPage()
Visszaadja a PrinterJob-objektum aktuális oldalformátumát.

- ▶ `String getUserName()`
Visszaadja az operációs rendszer által tárolt felhasználói nevet.
- ▶ `void cancel()`
- ▶ `boolean isCancelled()`
Leállítja a nyomtatást, illetve lekérdezi, hogy a nyomtatás le van-e állítva.
- ▶ `boolean printDialog()`
Meghív egy natív (az operációs rendszer tulajdonában levő) dialógusablakot, s azon át a felhasználó megadhatja a nyomtató tulajdonságait, valamint a nyomtatandó oldalak intervallumát. Csak intervallumot lehet megadni; a legkisebb oldalszám 1 (nem 0!), a legnagyobb 9999. A `PrinterJob`-objektum nem kínálhatja fel a ténylegesen nyomtható oldalak intervallumát, hiszen azt csak a `print` metódusok meghívása után tudhatja. A `PrinterJob`-objektum csak a dialógusban megadott oldalakra hívja meg a nyomtatandó objektum `print` metódusát (a `pageIndex`-nek eggyel kisebb értéket ad át). A metódus visszatérési értéke `true`, ha a felhasználó megerősítette a dialógust; `false`, ha elvetette. Ha a nyomtatási dialógust nem jelenítjük meg, akkor a nyomtatandó intervallum alapértelmezése 1..9999.
- ▶ `PageFormat pageDialog(PageFormat page)`
Megjelenik egy dialógusablak a `page` oldalformátum adataival. A viszaadott érték a felhasználó által beállított oldalformátum. Ha a felhasználó elvetette a dialógust, akkor az eredeti `page` oldalformátumot adja vissza.

Nem illik a felhasználó papírját pazarolni! Nyomtatás előtt lehetőleg jelenítsük meg a nyomtatási dialógust! Adjuk meg a felhasználónak a lehetőséget, hogy az oldalaknak csak egy intervallumát nyomtassa ki, és ha úgy tartja kedve, akkor állományba nyomtasson vagy éppen ne nyomtasson ki semmit!

Feladat – PrintTol_Ig

Készítsünk egy 10 oldalas (lapos) dokumentumot, 1-től 10-ig tartó oldalszámozással! minden oldalon legyen rajta az oldalszám és a "Pipacs" szó, ahogyan az ábrán láttható. Nyomtassuk ki a dokumentumból a felhasználó által kérte oldalakat!



A képen látható: a felhasználó itt első oldalnak történetesen 5-öt, utolsónak 7-et ütött be.

Forráskód

```

import java.awt.print.*;
import java.awt.Graphics;

class EgyszeruDoksi implements Printable {
    public int print(Graphics gr, PageFormat pf, int pageIndex
                    throws PrinterException {
        if (pageIndex >= 10)                                //1
            return Printable.NO_SUCH_PAGE;
        gr.drawString((pageIndex+1)+" oldal",100,100);      //2
        gr.drawString("Pipacs",100,150);
        return Printable.PAGE_EXISTS;
    }
}

public class PrintTol_Ig {
    PrinterJob nyomtato = PrinterJob.getPrinterJob();

    public PrintTol_Ig() {
        nyomtato.setPrintable(new EgyszeruDoksi());
        if (nyomtato.printDialog())                         //3
            try {
                nyomtato.print();
            }
            catch (PrinterException e) {
                e.printStackTrace();
            }
    }
}

public static void main(String[] args) {
    new PrintTol_Ig();
}
}

```

A forráskód elemzése

- ◆ //1: Mivel 10 oldalunk van, indexük 0 és 9 közötti érték (az oldalindexek mindenkorban nullával kezdődnek). A 10. indexű oldalt tehát már nem nyomtatjuk.
- ◆ //2: Az oldalon levő szövegbe betesszük az oldalszámot. A printerJob a felhasználó által adott oldalszámból levon egyet, de ezt most kiigazítjuk, hiszen a felhasználó az oldalakat itt is 1-től számolja.
- ◆ //3: Kitesszük a nyomtatási dialógust. A felhasználó azon átírhatja az első (1) és utolsó (9999) oldal értékét. Ha itt az 5 és 7 értékeket adja meg, akkor az 5., a 6. és a 7. oldal fog kinyomtatódni. Ha a 9 és 30 értékeket adta volna meg, akkor a 9. és a 10. oldal nyomtatódna ki, mivel 11. oldal már nincsen.

18.4. Oldalformázás – PageFormat osztály

Egy oldal nyomtatása mindenkorban egy **oldalformátum (page format)** szerint történik. Az oldalformátum főbb tulajdonságai: az oldal állása (álló vagy fekvő), az oldal (papír) mérete és a margók. A `PrinterJob`-objektumnak mindenkorban van egy aktuális, `PageFormat` osztályú oldalformátuma (18.3. ábra).

Az oldalformátum pontban tárolja a méreteket, double típusú értékként. Átváltások:

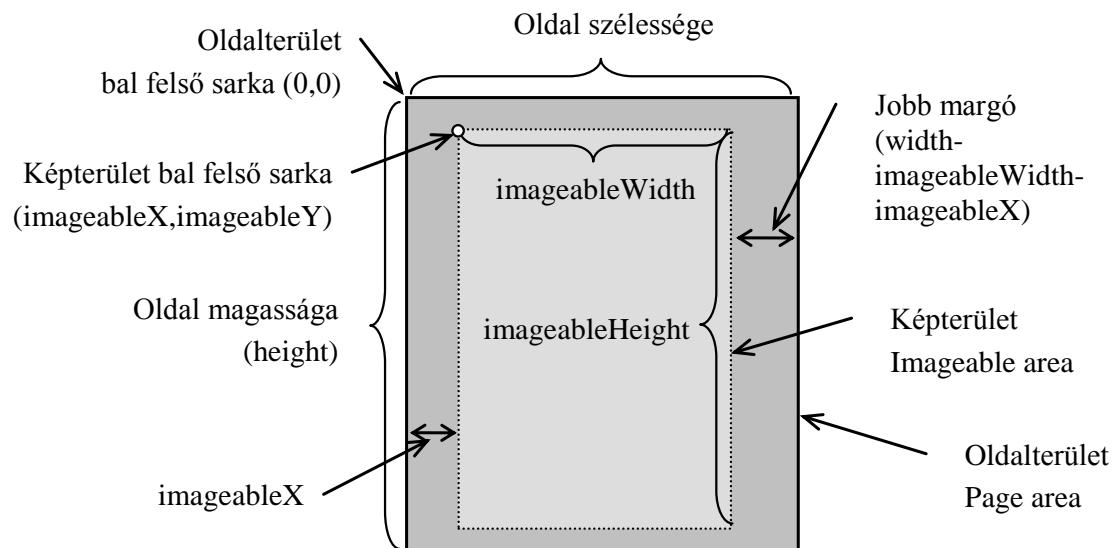
- 1 inch = 72 pont (1 inch, másnéven hüvelyk = 2,54 cm);
- 1 cm ~ 28,346 pont

Az oldal szélességét (`width`) és magasságát (`height`) a nyomtatási dialógusban lehet átalítni. Papírméretek például:

- A/4 papírméret: 21*29,7 cm, kb. 595*842 pont;
- A/5 papírméret: 14,8*21 cm, kb. 420* 595 pont.

A `Graphics` objektum (0,0) pontja a papír bal felső sarka. A képterület (imageable area) a margókkal „csonkított” rész, ez a grafikus objektum kivágási területe (clipping area). A margón kívülre eső részek nem látszanak – a margóra való nyomtatáshoz a kivágási területet ideiglenesen át kell állítani (`gr.setClip`). Ha azt akarjuk, hogy a (0,0) pont ne az oldalterület bal felső sarka legyen (hanem a margó által határolt látható képterületé), akkor az origót el kell tolni: `gr.translate`.

A felhasználó megváltoztathatja az oldalformátumot, ha felkínáljuk neki a `PrinterJob.pageDialog` metódus által adott szabványos dialógusablakot.



18.3. ábra. Az oldalformátum tulajdonságai

Csomag: `java.awt.print`

Deklaráció: `public class PageFormat`

Közvetlen ős: `java.lang.Object`

Fontosabb implementált interfések: `Cloneable`

Konstansok

- ▶ `static int PORTRAIT`
- ▶ `static int LANDSCAPE`

A kép álló, illetve fekvő.

Jellemzők

- ▶ `int orientation`
Az oldal állása: `PORTRAIT` vagy `LANDSCAPE`

Metódusok

- ▶ `double getWidth()`
- ▶ `double getHeight()`

Visszaadja az oldal szélességét, illetve magasságát.

- ▶ `double getImageableX()`
- ▶ `double getImageableY()`

Visszaadja a képterület bal felső sarkának (x,y) koordinátáit az oldal bal felső sarkához képest.

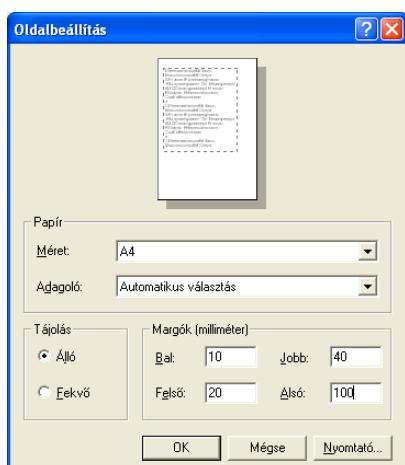
- ▶ `double getImageableWidth()`
- ▶ `double getImageableHeight()`

Visszaadja az oldal képterületének szélességét, illetve magasságát.

- Vigyázat! A visszaadott értékek mind `double` típusúak, a grafikus objektumon viszont `int` típusú értékkel pozicionálunk!

Feladat – PrintPage

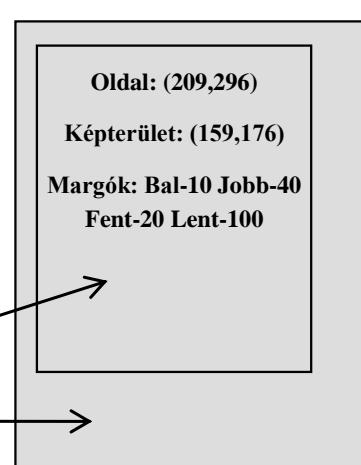
Nyomtassunk egy oldalt a következőképpen: a képterületet keretezzük be, a középére írjuk ki az oldal és a képterület méretét, valamint a margókat, mm-ben. Adjunk módot a felhasználónak arra, hogy nyomtatás előtt beállíthassa az oldal adatait!



Az itt megadott beállításokkal

képterület

oldalterület



Forráskód

```
import java.awt.print.*;
import java.awt.*;

class Page implements Printable {
    // Pont átszámítása mm-re:
    private int mm(int point) {
        return (int)(25.4*point/72);
    }

    public int print(Graphics gr, PageFormat pf,
        int pageIndex) throws PrinterException {
        if (pageIndex >= 1)
            return NO_SUCH_PAGE;

        // Oldal adatai pontokban:
        int width = (int)pf.getWidth();
        int height = (int)pf.getHeight();
        int imX = (int)(pf.getImageableX()+1);
        int imY = (int)(pf.getImageableY()+1);
        int imWidth = (int)(pf.getImageableWidth()-1);
        int imHeight = (int)(pf.getImageableHeight()-1);

        // Képterület bekeretezése:
        gr.translate(imX,imY);
        gr.drawRect(0,0,imWidth,imHeight);

        // Oldal és margók adatai:
        gr.setFont(new Font("SansSerif",Font.BOLD,20));           //4
        FontMetrics fm = gr.getFontMetrics();                         //5

        String str = "Oldal: ("+mm(width)+","+mm(height)+")";   //6
        gr.drawString(str,(imWidth-fm.stringWidth(str))/2,200);

        str = "Képterület: ("+mm(imWidth)+","+mm(imHeight)+")";
        gr.drawString(str,(imWidth-fm.stringWidth(str))/2,250);

        str = "Margók: " + "Bal-"+mm(imX)+" Jobb-"+
            mm(width-imX-imWidth)+" Fent-"+mm(imY)+"
            " Lent-"+mm(height-imY-imHeight);
        gr.drawString(str,(imWidth-fm.stringWidth(str))/2,300);

        return PAGE_EXISTS;
    }
}

public class PrintPage {
    PrinterJob pj = PrinterJob.getPrinterJob();
    PageFormat pf = pj.defaultPage();
```

```
public PrintPage() {
    pf = pj.pageDialog(pf); //7
    pj.setPrintable(new Page(),pf); //8

    try {
        pj.print();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) {
    new PrintPage();
}
```

A forráskód elemzése

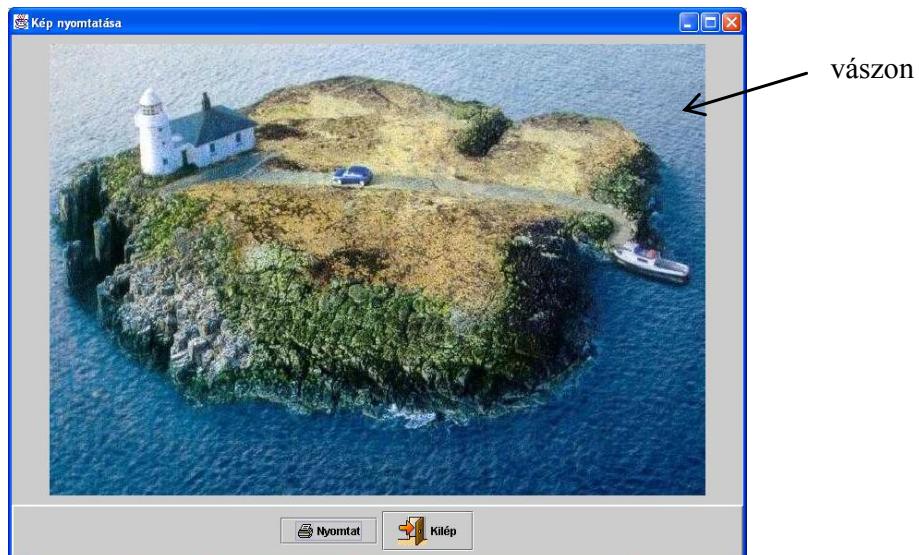
- ◆ //1: Az eredményt mm-ben kell kiírnunk – a függvény átszámítja a pontokat mm-re.
- ◆ //2: Kiszámítjuk az oldal adatait: (width,height) az oldal mérete, (imX,imY) a képterület bal felső sarka az oldalon, (imWidth,imHeight) pedig a képterület mérete. A PageFormat a méreteket double típusú értékként adja meg, a Graphics osztály metódusai viszont int értékekkel dolgoznak. Ezért az oldalformátum metódusainak visszatérési értékeit int típusúra konvertáljuk.
- ◆ //3: A (0,0) pontot eltoljuk a képterület bal felső sarkába, mert így könnyebb a rajzolás. Rajzolunk akkora téglalapot, amekkora bekeretezi a képterületet.
- ◆ //4: 20 pontos kövér SansSerif betűvel fogunk írni.
- ◆ //5: A FontMetrics osztály az Object közvetlen leszármazottja. Ez egy betűmérce szövegek adatainak kiszámításához. A betűmérce-objektumot az aktuális grafikus objektumtól kell elérni. Például az str szöveg hossza az fm betűmérce szerint: fm.stringWidth(str).
- ◆ //6: Az adatokat átszámítjuk milliméterre, és egyenként összeállítjuk a kiírandó szövegeket. A szövegeket egymás alá, középre igazítjuk – ez utóbbihoz elkérjük a betűmér-cétől a szöveg hosszát. Ha túl kicsi az oldalméret, akkor a szöveg egy része esetleg nem fér majd rá a kivágási területre.
- ◆ //7: Kitesszük a képernyőre az oldaldialógust a nyomtató aktuális oldalformátumával. Az újonnan beállított oldalformátumot szintén pf-ben tároljuk.
- ◆ //8: Beállítjuk a nyomtatandó objektumot a felhasználótól az imént elkerít oldalformátummal.

18.5. Megjelenítés és nyomtatás

Végül bemutatunk egy olyan feladatot, melyben ugyanazt a képet megjelenítjük a képernyön és a nyomtatóban. A rajzot egy közös metódusban fogjuk elkészíteni – ezt a metódust fogja meg-hívni a komponens paintComponent és paint metódusa is.

Feladat – PrintShow

Jelenítsünk meg egy képet a keret közepén! Gombnyomásra lehessen kinyomtatni a képet, illetve kilépni a programból.



Forráskód

```
import javax.swing.*;
import javax.swing.border.*;
import java.awt.print.*;
import java.awt.*;
import java.awt.event.*;

class Vaszon extends JPanel implements Printable {
    private Image image;

    public Vaszon() {
        image = Toolkit.getDefaultToolkit().createImage(
            "images/auto.jpg"); //1
        MediaTracker tr = new MediaTracker(this);
        tr.addImage(image, 0);
        try {
            tr.waitForID(0);
        }
        catch(InterruptedException e) {
        }
    }

    public void paintComponent(Graphics g) {
        g.drawImage(image, 0, 0, null);
    }

    public int print(Graphics g, PageFormat pf, int page) throws PrinterException {
        if (page > 0) return NO_SUCH_PAGE;
        Graphics2D g2 = (Graphics2D)g;
        g2.translate(pf.getImageableX(), pf.getImageableY());
        paintComponent(g2);
        return PAGE_EXISTS;
    }
}
```

```
void drawVaszon(Graphics gr) {                                //2
    int imWidth = image.getWidth(this);    // a kép szélessége
    int imHeight = image.getHeight(this);   // a kép magassága
    int w = gr.getClipBounds().width;
    int h = gr.getClipBounds().height;
    gr.drawImage(image, (w-imWidth)/2, (h-imHeight)/2, this);
}

public void paintComponent(Graphics gr) {
    super.paintComponent(gr);
    drawVaszon(gr);                                //3
}

public int print(Graphics gr, PageFormat pf, int pageIndex)
throws PrinterException {
    if (pageIndex >= 1)
        return NO_SUCH_PAGE;
    gr.translate((int)pf.getImageableX(),
                (int)pf.getImageableY());
    drawVaszon(gr);                                //4
    return PAGE_EXISTS;
}

class PrintShowFrame extends JFrame implements ActionListener {
    private Container cp;
    private Vaszon vaszon = new Vaszon();
    private JButton btNyomtat = new JButton("Nyomtat",
                                           new ImageIcon("icons/print.gif"));
    private JButton btKilep = new JButton("Kilép",
                                         new ImageIcon("icons/exit.gif"));
    private PrinterJob pj = PrinterJob.getPrinterJob();

    public PrintShowFrame() {
        setTitle("Kép nyomtatása");
        setBounds(50,50,800,600);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        cp = getContentPane();
        cp.setLayout(new BorderLayout());
        cp.add(vaszon,"Center");
        JPanel pnControl = new JPanel();
        pnControl.setBorder(new BevelBorder(BevelBorder.RAISED));
        cp.add(pnControl,"South");

        pnControl.add(btNyomtat);
        btNyomtat.addActionListener(this);

        pnControl.add(btKilep);
        btKilep.addActionListener(this);
    }
}
```

```

public void actionPerformed(ActionEvent evt) {
    if (evt.getSource() == btNyomtat) {
        pj.setPrintable(vaszon);
        try {
            pj.print();
        }
        catch (Exception ex) {
            System.out.println(ex);
        }
    }
    else if (evt.getSource() == btKilep)
        System.exit(0);
}
}

public class PrintShow {
    public static void main(String[] args) {
        new PrintShowFrame().show();
    }
}

```

A forráskód elemzése

A programban a `Vaszon` egy példányát fogjuk megjeleníteni, majd kinyomtatni.

- ◆ //1: A `Vaszon` konstruktőrben lemezről betöljtük a képet. A betöltést a `MediaTracker` végzi, és a program csak a betöltés befejeztével megy tovább. Vigyázni kell, hogy legyen ilyen kép a lemezen, különben a `MediaTracker` végtelen ciklusba esik!
- ◆ //2: A `drawVaszon` metódus a kivágási terület közepére teszi a képet. A képnek a nyomtatásban valószínűleg levágódik majd a két széle. Ahhoz, hogy az egész kép raférjen az oldalra, arányaiban hozzá kellene igazítani a kivágási területhez (és az ablakhoz is); de ez nagyban megnövelné a kód méretét.
- ◆ //3: A `panel paintComponent` metódusa meghívja az előbbi `drawVaszon` metódust.
- ◆ //4: A `print` metódus szintén a `drawVaszon` metódust hívja. Ezt megtehetjük, hiszen minden esetben `Graphics` objektumra nyomtatunk.

Tesztkérdések

- 18.1. A következők közül melyek elengedhetetlen feltételei a nyomtatásnak? Jelölje be az összes szükséges feltételt!
- A nyomtatandó objektum osztálya a `PrinterJob` osztály leszármazottja.
 - A nyomtatóobjektum osztályának implementálnia kell a `Printable` interfést.
 - A nyomtatóobjektumhoz hozzá kell rendelni a nyomtatandó objektumot.
 - Nyomtatás előtt kötelező megjeleníteni a `PrintDialog` dialógusablakot, hogy a felhasználó beállíthassa a nyomtatási paramétereket.

18.2. Jelölje be az összes igaz állítást!

- a) PrinterJob-objektumot a new operátorral hozhatunk létre.
- b) A nyomtatónak és a nyomtatandó objektumnak is van print metódusa.
- c) A nyomtatandó objektum print metódusa a Printable interfész implementációja.
- d) A nyomtatandó objektum print metódusa paraméterben kapja meg az oldalformátumot.

18.3. Jelölje be az összes igaz állítást!

- a) A print metódus Graphics objektumát ugyanúgy kell használni, mint a komponensek paintComponent metódusának Graphics objektumát.
- b) Csak a programozótól függ, hogy hányszor hívódjon meg a nyomtatandó objektum print metódusa.
- c) A Printable.print metódusnak boolean a visszatérési típusa.
- d) Programból el lehet érni, hogy a kinyomtatandó oldalakra írt oldalszámok csak páratlanok legyenek.

18.4. Alapértelmezésben mely állítások igazak a nyomtatandó objektum print metódusában? Jelölje be az összes igaz állítást!

- a) A PageFormat paraméter tárolja a papír méretét és a margókat.
- b) A margók megszabta oldalszélre rajzolt dolgok nem láthatóak.
- c) A Graphics objektum (0,0) pontja a papír bal felső sarka.
- d) Szintaktikailag szerepelnie kell egy return NO_SUCH_PAGE utasításnak.

18.5. Jelölje be az összes igaz állítást!

- a) A nyomtatási dialógus osztálya PrintDialog, és a new operátorral kell létrehozni.
- b) A nyomtatási dialógust a PrinterJob osztály printDialog metódusa teszi ki a képernyőre.
- c) A programozónak le kell vizsgálnia a felhasználó által a nyomtatódialógusban megadott oldaltartományt!
- d) Pontosan annyi oldal nyomtatódik ki, ahányat a felhasználó megadott a nyomtatódialógusban.

Feladatok

18.1. (A) Nyomtasson ki egyetlen oldalt egy, a képterületet betöltő ellipszessel!
(Ellipszis.java)

18.2. (B) Nyomtasson ki egyetlen oldalt egy nagy „Megállni tilos!” jelzéssel!
(PrintTilos.java)

18.3. (A) Nyomtasson ki egyetlen oldalt egy, a képterületet betöltő X-szel! Adjon lehetőséget a felhasználónak az oldalméret beállítására! (Iksz.java)

18.4. (A) Az 1. oldalon a bal felső sarokban szerepeljen az „1 dollár eladó”; a 2. oldalon a „2 dollár eladó” stb. Nyomtassuk ki a felhasználó által kért oldalintervallumot!
(PrintDollarElado.java)

18.5. (B) Nyomtassunk ki 3 oldalt!

- A képterület 4 sarkát jelöljük meg egy-egy kis tömör téglalappal úgy, hogy azok teljesen a képterületre essenek!
- Az oldalakra írjuk rá a sorszámukat (1, 2, 3). A szám nagy méretű és világosszürke legyen!
- Írjuk konzolra a `PrinterJob`-objektum tulajdonságait: nyomtatási munka neve, nyomtatandó példányszám, felhasználó neve, oldalformátum stb. (`PrintSzamok.java`)

18.6. (A) Adva van egy szövegeket tartalmazó vektor. Nyomtassunk ki egyetlen oldalt, s azon annyit jelenítsünk meg a vektor soraiból, amennyi kifér! A sorok 1-től kezdve legyenek beszámozva; a betűk 20 pontos Monospace típusúak legyenek; keretezzük be a margók által határolt területet! (`PrintSorok.java`)

18.7. (B) Egészítsük ki az előző feladatot: nyomtassunk ki több oldalt úgy, hogy a sorok számozása folytatódjon! Számozzuk az oldalakat is! (`PrintSorokFolyt.java`)

18.8. (B) Tegyen a képernyőre egy nagy „Megállni tilos!” jelzést, s az egy gomb lenyomására nyomtatódjék ki! (`PrintTilosGUI.jpx`)

18.9. (C) Tegyük lehetővé szöveges (`.txt` és `java` kiterjesztésű) állományok nézegetését és nyomtatását! A kinyomtatott oldalakat keretezzük be, és számozzuk be a soraikat! minden oldalnak legyen egy fejléce a kereten kívül; a fejlécen legyen rajta a kinyomtatott állomány teljes útvonala, valamint az oldalsorszám és az összes oldal száma (pl. 2/4. oldal)! A sorokat nem kell tördelni. (`PrintFile.jpx`)

18.10. (B) Készítsünk a felhasználónak olyan vásznat, amelyre egérrel rajzolhat! Tegyük lehetővé, hogy a rajz bármely állapotát kinyomtathassa! (`PrintRajz.jpx`)

18.11. (C) Jelenítsünk meg egy, az eltárolt `.jpg`- vagy `.gif`-állományok közül választható képet! Kérésre nyomtassuk is ki! A nyomató- és oldalbeállításokat bízzuk a felhasználóra! Az oldalformátum-dialógusban fekvő képet kínálunk fel! A kép a lehető legjobban illeszkedjen a nyomtatandó felületre! Nyomtatás előtt készítsünk egy nyomtatási képet! (`PrintImage.jpx`)

19. Hasznos osztályok

A fejezet pontjai:

1. Időpont – Date
 2. Környezet – Locale
 3. Időeltolás – TimeZone
 4. Naptár – GregorianCalendar
 5. Dátumformázás – DateFormat
 6. Számformázás – NumberFormat
 7. Megfigyelés – Observer, Observable
 8. Klónozás – Cloneable
 9. Rendszerjellemzők – System
 10. Külső program futtatása – Runtime
-

Ebben a fejezetben olyan osztályokról és interfésekkről lesz szó, amelyek „jól jöhetnek” az alkalmazások fejlesztésében. A Javában persze még nagyon sok további hasznos osztály van; igyekeztem azokat összegyűjteni, amelyeket egy átlagos programozó felhasználhat egy átlagos alkalmazásban.

19.1. Időpont – Date

Az **időpont** azonosítására az UTC (Universal Time Coordinated) szabványt használjuk. Az UTC az időpontot egy `long` típusú egész számmal azonosítja helytől, környezettől és időzónától (időeltolástól) függetlenül. Ezért ezt a számot **univerzális időnek** nevezzük. Az érték az eltelt ezredmásodperceket jelenti 1970. január 1. 00:00:00 óta, GMT (Greenwich Mean Time) idő szerint – Greenwichben 0 az időeltolás.

Az időzónát (időeltolást) is magába foglaló helyi időt a naptár testesíti meg.

A Javában az időpont lekérdezhető a `System` osztály statikus `currentTimeMillis` metódusával. A `Date` osztály példányai egy-egy időpontot tárolnak.

Megjegyzés: A számítógépek nem egészen pontosak – az UTC szabvány szerint évenként kb. 1 másodpercet hozzá kellene adni az időponthoz.

Aktuális időpont (`System.currentTimeMillis()`)

- `static long currentTimeMillis()`

A `System` osztály statikus metódusa. Visszaadja a számítógép által tárolt aktuális univerzális időt ezredmásodpercben.

Feladat – Ciklusmérő

Készítsünk egy metódust gépünk gyorsaságának tesztelésére! A metódus lemöri a paraméterként megadott ciklusszámú, üres for ciklus futási idejét ezredmásodpercben. Teszteljük a metódust 10 millió ciklussal!

Forráskód

```
public class CiklusMero {
    static long ciklusIdo(long ciklusSzam) {
        long kezdido = System.currentTimeMillis();
        for (long i=0; i<ciklusSzam; i++);
        long vegido = System.currentTimeMillis();
        return vegido-kezdido;
    }

    public static void main(String[] args) {
        // 10 millió ciklus végrehajtása ennyi időbe telik:
        System.out.println("10 millió ciklus ideje: "+
            ciklusIdo(10000000)+" ms");
    }
}
```

A program futása

| 10 millió ciklus ideje: 71 ms

Date osztály

Csomag: `java.util`

Deklaráció: `public class Date`

Közvetlen ős: `java.lang.Object`

Fontosabb implementált interfések: `Cloneable`, `Comparable`, `Serializable`

A `Date` osztály egy univerzális (UTC) időpontot tárol – ez az időpont a GMT szerinti, 1970. január 1. 00:00:00 óta eltelt idő, ezredmásodpercben.

Konstruktörök, metódusok

- `Date()`
- `Date(long time)`

Az objektumot az aktuális, illetve a megadott univerzális idővel inicializálja.

- ▶ `long getTime()`
- ▶ `void setTime(long time)`
Az objekum időpontjának lekérdezése, beállítása.
- ▶ `boolean after(Date when)`
- ▶ `boolean before(Date when)`
Megadja, hogy az időpont a paraméterben megadott időpont után van-e (after), illetve előtte van-e (before).
- ▶ `String toString()`
Visszaadja az időpont szöveges formátumát az alapértelmezés szerinti (a számítógépen beállított) időzónával (időeltolással), amerikai formátumban:
`dow mon dd hh:mm:ss zzz yyyy`
Rövidítések: `dow=day of week`, `mon=hónap`, `dd=nap`, `hh=óra`, `mm=perc`, `ss=másodperc`,
`zzz=időzóna`, `yyyy=év`.

Feladat – DateTest

Jelenítsük meg az aktuális időt egész (univerzális) és szöveges formában! Vonjunk le 2 órát ebből az időből, és jelenítsük meg azt is!

Forráskód

```
import java.util.Date;

public class DateTest {
    public static void main(String[] args) {
        Date d = new Date(); //1
        System.out.println("Pontos idő: "+d.getTime(), "+d); //2
        d.setTime(d.getTime()-2*60*60*1000); //3
        System.out.println("2 órával ezelőtt: "+
                           d.getTime(), "+d); //4
    }
}
```

A program egy lehetséges futása

```
Pontos idő: 1026836795488, Tue Jul 16 18:26:35 CEST 2002
2 órával ezelőtt: 1026829595488, Tue Jul 16 16:26:35 CEST 2002
```

A program elemzése

- ◆ //1: A `d` objektum az aktuális időpontot tartalmazza.
- ◆ //2: Kiírjuk a `d` által tartalmazott időpontot ezredmásodpercben, majd a `d` szöveges reprezentációját.
- ◆ //3: `d`-től elkerjük az időpontot, ahhoz hozzáadunk 2 órának megfelelő ezredmásodpercet, majd ezt az időpontot visszatöljük `d`-be.
- ◆ //4: Kiírjuk `d` új tartalmát, mint //2-ben.

A következő két utasítás majdnem egyenértékű (a `currentTimeMillis` gyorsabb):

```
long startTime = System.currentTimeMillis();
long startTime = new Date().getTime();
```

Időmérésre célszerűbb a `currentTimeMillis` metódust használni, mert az a gyorsabb. A `Date` objektum összeállítása időbe telik. A `Date` konstruktora a `currentTimeMillis` metódust hívja meg. A `Date`-et akkor használjuk, ha objektumra van szükségünk.

Megjegyzés: A `Date` osztály elnevezése zavaró – sokkal találóbb lenne a `Time`.

19.2. Környezet – Locale

Egy alaposabb szoftver alkalmazkodik a különböző országokban élő, különböző nemzetiségekhez tartozó emberek szokásaihoz. A környezettől függően más formában és más nyelven jelenítünk meg szövegeket, dátumokat, számokat és pénzt, más mértékegységet használunk, a tizedes helyére pontot vagy vesszőt teszünk stb.

Locale osztály

Csomag: `java.util`

Deklaráció: `public final class Locale`

Közvetlen ős: `java.lang.Object`

Fontosabb implementált interfések: `Cloneable`, `Serializable`

Egy `Locale` objektum egy **földrajzi, politikai, illetve kulturális környezetet reprezentál**. A `Locale` osztálynak a környezet azonosításán kívül nincs más feladata – nem tartalmazza az ezzel vagy azzal az országgal, illetve nyelvvel kapcsolatos konkrét szabályokat (például dátum, tizedes, mértékegység, pénznem megjelenítési formái). A konkrét szabályokat, nemzeti beállításokat a számítógépen erőforrás formájában lehet tárolni; ezekhez az erőforrásokhoz a környezeti objektumok vannak hozzárendelve. A környezetfüggő osztályok ezeken a környezeti objektumokon át érik el és használják a megfelelő erőforrásokat (állományokat). A `DateFormat` objektum például dátumot és időpontot formáz meg a `Locale` objektum alapján, a `NumberFormat` pedig számot. Az operációs rendszerben van egy alapértelmezés szerinti környezeti objektum; ez egy magyar gépen általában magyar; ezért a formázások, hacsak másnépp nem rendelkezünk, a magyar szokást követik.

Tulajdonságok

A következő táblázat példákkal illusztrálva mutatja be egy `Locale` objektum tulajdonságait. `country` az ország 2 karakteres, nagybetűs ISO 639-es azonosítója (nem teljesen következetes). `ISO3Country` az ország 3 karakteres, nagybetűs ISO3 azonosítója. A `displayCountry` a felhasználónak is érhető, beszédes azonosító. A nyelvet szintén háromféleképpen azonosít-

hatjuk. A nyelv ISO3-Language azonosítói kisbetűsek, sőt – amolyan kakukktojásként – a magyar displayLanguage is kisbetűvel kezdődik.

country	ISO3-Country	display-Country	language	ISO3-Language	display-Language
US	USA	United States	En	eng	English
AE	ARE	United Arab Emirates	Ar	Ara	Arabic
HU	HUN	Magyarország	Hu	Hun	magyar

A számítógépen elérhető összes környezetet a `Locale.getAvailableLocales` metódussal lehet lekérdezni, és a környezeti objektumokból aztán tetszés szerint lehet választani. Bizonyos környezeti objektumokat konstansként definiáltak az osztályban.

Konstansok

- ▶ CANADA, CANADA_FRENCH, ENGLISH, GERMAN, US, ...
Előre definiált környezeti objektumok. A magyar nincs köztük.

Konstruktörök, metódusok

- ▶ `Locale(String language, String country)`
Létrehoz egy nemzetközi környezetet azonosító objektumot.
- ▶ `static Locale[] getAvailableLocales()`
Lekérdezi a számítógépen elérhető összes környezeti objektumot.
- ▶ `static Locale getDefault()`
- ▶ `static void setDefault(Locale newLocale)`
Lekérdezi, illetve beállítja az alapértelmezett környezeti objektumot. Ha egy metódusban nem adunk meg környezeti objektumot, akkor ezt fogja alapul venni.
- ▶ `String getCountry()`
- ▶ `String getISO3Country()`
- ▶ `String getDisplayCountry()`
Lekérdezi a környezet országát, az ország ISO3 azonosítóját és bővebb formáját.
- ▶ `String getLanguage()`
- ▶ `String getISO3Language()`
- ▶ `String getDisplayLanguage()`
Lekérdezi a környezet nyelvét, a nyelv ISO3 azonosítóját, és bővebb formáját.
- ▶ `static String[] getISOCountries()`
- ▶ `static String[] getISOLanguages()`
Visszaadja a kétkarakteres ország-, illetve nyelvazonosítókat.
- ▶ `String toString()`
A `Locale` szöveges formátuma: `getCountry() + "_" + getLanguage()`

Feladat – LocaleTest

Keressük meg az elérhető környezeti objektumok között azokat, amelyeknek az ISO3 országazonosítója H betűvel kezdődik! Írjuk ki az ISO3 és a beszédes azonosítójukat! Ezután írjuk ki a számítógépen érvényben levő környezet országát és nyelvét!

Forráskód

```
import java.util.Locale;
public class LocaleTest {
    public static void main(String[] args) {
        System.out.println("H betűs országok: ");
        Locale[] nemzetek = Locale.getAvailableLocales();
        for (int i = 0; i < nemzetek.length; i++) {
            if (nemzetek[i].getISO3Country().startsWith("H"))
                System.out.print(nemzetek[i].getISO3Country()+"-"+nemzetek[i].getDisplayCountry(), " ");
        }
        System.out.print("\nAktuális ország és nyelv: ");
        Locale loc = Locale.getDefault();
        System.out.println(loc.getCountry()+" "+loc.getLanguage());
    }
}
```

A program futása

H betűs országok:	HKG-Hong Kong, HND-Honduras, HRV-Croatia, HUN-Magyarország,
	Aktuális ország és nyelv: HU, hu

19.3. Időeltolás – TimeZone

Az **időzóna** (TimeZone) objektum tárolja az időeltolást a GMT-hez képest, és a TimeZone osztály foglalja magába a napfelkelte és naplemente időpontját megadó szabályokat. A TimeZone osztály az időzóna-objektumok absztrakt őse. A leszármazott osztályokban absztrakt metódusok felülírásával lehet megadni a napfelkelte és naplemente napi időpontját és az időeltolást.

TimeZone osztály

Csomag: java.util

Deklaráció: public abstract class TimeZone

Közvetlen ős: java.lang.Object

Fontosabb implementált interfészek: Cloneable, Serializable

Tulajdonságok

- ◆ `String ID`: Az időzóna azonosítója, például: "Africa/Freetown", "Europe/Budapest" stb. (a hárombetűs azonosítók elavultak). Lásd a mellékelt táblázatot.
- ◆ `String displayName`: A felhasználó számára beszédes szöveg. Például Central European Time. Azonos eltolású időzónákra ugyanaz lehet.
- ◆ `int rawOffset`: Időeltolás a GMT-hez képest ezredmásodpercben. Ha az eltolás 1 óra, akkor `rawOffset` értéke: $1 * 60 * 60 * 1000$; ha -2 óra, akkor `rawOffset` értéke: $-2 * 60 * 60 * 1000$. Több, különböző azonosítójú vagy nevű időzónának is lehet ugyanakkora az időeltolása.

ID	displayName	Időeltolás
Africa/Freetown	Greenwich Mean Time	0 óra
Atlantic/Reykjavik	Greenwich Mean Time	0 óra
Europe/London	Greenwich Mean Time	0 óra
Atlantic/Canary	Western European Time	0 óra
Europe/Brussels	Central European Time	1 óra
Europe/Budapest	Central European Time	1 óra
Europe/Gibraltar	Central European Time	1 óra
Africa/Luanda	Western African Time	1 óra

Metódusok

- ▶ `static TimeZone getDefault()`
Visszaadja a számítógép alapértelmezés szerinti `TimeZone`-példányát. Az objektum azt az időzónát reprezentálja, amelyben a program éppen fut.
- ▶ `static TimeZone getTimeZone(String ID)`
Visszaadja az ID azonosítójú időzóna-objektumot.
- ▶ `String getID()`
Visszaadja ennek az időzónának az azonosítóját.
- ▶ `final String getDisplayName()`
▶ `final String getDisplayName(Locale local)`
Megadja az aktuális, illetve a `local` környezeti objektumhoz tartozó időzóna nevét.
- ▶ `static String[] getAvailableIDs()`
▶ `static String[] getAvailableIDs(int rawOffset)`
Visszaadja az elérhető időzóna-objektumok azonosítóit. A második esetben csak azokat adjva vissza, amelyekben az időeltolásnak `rawOffset` az értéke.
- ▶ `abstract int getRawOffset()`
▶ `abstract void setRawOffset(int offsetMillis)`
Lekérdezi, illetve beállítja az időeltolást. A leszármazott osztályokban felülírandó.
- ▶ `abstract boolean inDaylightTime(Date date)`
Megadja, hogy a `date` időpont a nyári vagy a téli időszámítás idejére esik-e. A visszaadott érték `true`, ha az időszámítás nyári. A leszármazott osztályokban felülírandó.

Feladat – TimeZoneTest

Írjuk ki a helyi időzóna adatait (az azonosítót és a beszédes nevet), az időpontot, valamint azt, hogy nappal van-e?

Forráskód

```
import java.util.*;
public class TimeZoneTest {
    public static void main(String[] args) {
        TimeZone tz = TimeZone.getDefault();
        System.out.println("\nHelyi időzóna");
        System.out.println("azonosítója : " + tz.getID());
        System.out.println("neve           : " + tz.getDisplayName());
        Date date = new Date();
        System.out.println("\nidőpont: " + date);
        if (tz.inDaylightTime(date))
            System.out.println("Nyári időszámítás van.");
    }
}
```

A program futása

```
Helyi időzóna
azonosítója : Europe/Budapest
neve         : Central European Time

Időpont: Tue Aug 06 06:12:21 CEST 2002
Nyári időszámítás van.
```

19.4. Naptár – GregorianCalendar

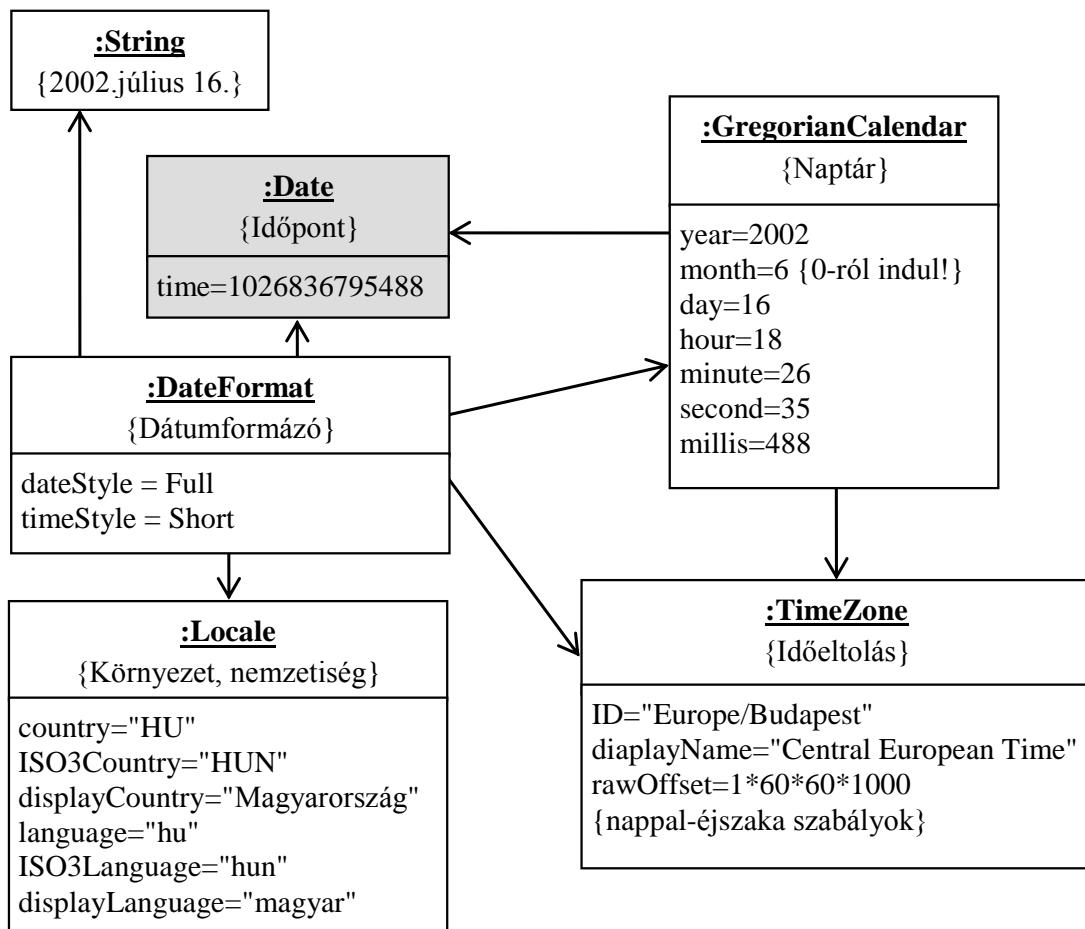
Az univerzális időpontot a különböző népek különböző **naptárrakkal tartják nyilván**. A legtöbb ember a világon a Gergely naptár szerint él, de vannak más naptárak is, például a kínai vagy a héber. Az időpontot a `Calendar` absztrakt osztály és annak leszármazottai értelmezik: az univerzális időből az időzóna (`TimeZone`) figyelembe vételével kiszámítják az év, hónap, nap, óra, perc, másodperc és ezredmásodperc adatokat. A naptár nem foglalkozik a dátum és időpont megjelenítésével – az a dátumformázó (`DateFormat`) objektum feladata.

A Java egyetlen konkrét naptárat definiál: a `GregorianCalendar` osztályt (Gergely-naptár).

A 19.1. ábra egy objektumdiagramot ábrázol az időt, a naptárat és a környezetet (a nemzeti jellegzetességeket) megtestesítő konkrét objektumokkal és azok lényeges kapcsolataival:

- ◆ `Date`: Univerzális időpontot szolgáltat más osztályoknak (központi szerepet játszik).
- ◆ `GregorianCalendar`: Az időpontból kiszámítja az év, hó, nap stb. adatot. Figyelembe veszi az időeltolást, hiszen más időzónában mást mutat a naptár.

- ◆ **TimeZone**: Az időeltolást tárolja. A helyi idő szerint működő osztályok használják.
- ◆ **DateFormat**: Megformázza az időpontot, beszámítva az időeltolást és a nemzetek dátumformázó szokásait. A formázáshoz felhasználja a naptárat.
- ◆ **Locale**: A környezetet, illetve nemzetiséget azonosítja.



19.1. ábra. Idő, naptár és környezeti objektumok

GregorianCalendar osztály

Csomag: `java.util`

Deklaráció: `public class GregorianCalendar`

Közvetlen ős: `java.util.Calendar`

Fontosabb implementált interfész: `Cloneable, Serializable`

A Calendartól örökolt mezőazonosító konstansok (field identifiers)

- YEAR // év
- MONTH // hónap, 0..11, 0-ról indul!
- DAY_OF_WEEK // a hétfő napja, 0..6
- DAY_OF_MONTH // a hónap napja, 1..31
- DAY_OF_YEAR // az év napja, 1..366
- WEEK_OF_YEAR // az év hete, 1..53
- HOUR // óra, 0..11
- HOUR_OF_DAY // óra, 0..23
- MINUTE // perc, 0..59
- SECOND // másodperc, 0..59
- MILLISECOND // ezredmásodperc, 0..999
- AM // délelőtt, 0
- PM // délután, 12

Ezeket a mezőazonosítókat a `get` és `set` metódusok használják. A konstansok melletti értékhatarok az általuk azonosított mezők értékhatarai.

Egyéb konstansok

- SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
 - JANUAR, FEBRUAR, MARCH, APRIL, MAY, JUNE, JULY, AUGUST...
- A `DAY_OF_WEEK`, illetve a `MONTH` mezők lehetséges értékei.

Konstruktörök, metódusok

- `GregorianCalendar()`
- `GregorianCalendar(TimeZone zone, Locale aLocale)`

Létrehoz egy Gergely-naptárhoz tartozó objektumot a számítógépen tárolt mostani időponttal. Az első esetben az időzóna és a nemzeti jellegzetességek az alapértelmezésnek megfelelők lesznek; második esetben ezeket is megadjuk.

- `GregorianCalendar(int year, int month, int day)`
- `GregorianCalendar(int year, int month, int day, int hour, int minute, int second)`

Létrehoz egy Gergely-naptárhoz tartozó objektumot a megadott adatokkal. Első esetben az `hour`, `minute` és `second` adatok értéke 0 lesz.

- `final int get(int field)`
- `final void set(int field, int value)`

A `field` azonosítójú mezőt (YEAR/MONTH/...) lekérdezi, illetve beállítja azt `value` értékre.

- `final Date getTime()`
- `final void setTime(Date date)`

A naptár időpontjának lekérdezése, illetve beállítása.

- `void add(int field, int amount)`

A naptár valamely mezőjéhez hozzáad egy értéket (az negatív is lehet). A naptár kezeli a túlcordulásokat.

- `boolean isLeapYear(int year)`

Megadja, hogy ez vagy az az év szökőév-e.

Feladat – Dátum, idő

Készítsünk olyan metódust, amely kiírja egy megadott időponthoz tartozó évet, hónapot, napot, azt, hogy a nap az év hányadik napja, a hét az év hányadik hete, azután az órát, percert, másodpercret és azt, hogy a megadott időpont délelőttre esik-e vagy délutánra! Írjuk ki először a pillanatnyi időnek megfelelő adatokat, majd a 10 percssel és 22 másodpercel későbbi időpont adatait!

Forráskód

```

import java.util.GregorianCalendar;
import java.util.Calendar;

public class DatumIdo {
    static void printCalendar(Calendar c) {
        System.out.println("Dátum: "+c.get(Calendar.YEAR)+"."+
                           (c.get(Calendar.MONTH)+1)+"."+
                           c.get(Calendar.DAY_OF_MONTH)+".");

        System.out.println("Az év "+
                           c.get(Calendar.DAY_OF_YEAR)+" napja");
        System.out.println("Az év "+
                           c.get(Calendar.WEEK_OF_YEAR)+" hete");

        System.out.print("Időpont: "+
                           c.get(Calendar.HOUR_OF_DAY)+":"+
                           c.get(Calendar.MINUTE)+":"+
                           c.get(Calendar.SECOND));
        System.out.println(c.get(Calendar.AM_PM)==Calendar.PM?
                           " Délután":" Délelőtt");
    }

    public static void main(String[] args) {
        GregorianCalendar naptar = new GregorianCalendar();
        System.out.println("Pontos idő:");
        printCalendar(naptar);

        // Előbbie igazítjuk az órát:
        naptar.add(Calendar.MINUTE,10);
        naptar.add(Calendar.SECOND,22);
        System.out.println("\n10 perc és 22 másodperckel később:");
        printCalendar(naptar);
    }
}

```

A program futása

Pontos idő:
Dátum: 2002.6.14.
Az év 195. napja
Az év 29. hete
Időpont: 6:38:53 Délelőtt
10 perc és 22 másodpercssel később:
Dátum: 2002.6.14.
Az év 195. napja
Az év 29. hete
Időpont: 6:49:15 Délelőtt

19.5. Dátumformázás – DateFormat

A DateFormat osztály segítségével egy **időpontot** (dátumot és időt) **lehet formázni** a megadott stílusban és nemzeti környezetben. A formázást a következőképpen végezzük el:

- először a DateFormat osztálytól elkérünk egy formázóobjektumot. A formázóobjektumnak beállíthatjuk a környezetét (`Locale`) és a stílustát (`FULL`=hosszú alak, `MEDIUM`=közepes alak stb.).
- a formázóobjektumot megkérjük az adott időpont formázására. A formázó által visszaadott szöveg az időpontot reprezentálja az adott környezeti objektumnak megfelelően és a kért stílusban (rövid, hosszú forma).

Megjegyzés: A Calendar leszármazottainak nem dolguk a formázás.

DateFormat osztály

Csomag: `java.text`

Deklaráció: `public abstract class DateFormat`

Közvetlen ős: `java.lang.Object`

Fontosabb implementált interfések: `Cloneable, Serializable`

Konstansok

- `static int FULL` // Teljes formátum, pl. 14 July 2002
- `static int LONG` // Hosszú formátum, pl. 14 July 2002
- `static int MEDIUM` // Közepes formátum, pl. 14-Jul-02
- `static int SHORT` // Rövid formátum, pl. 14/07/02

A dátum vagy az idő formázására vonatkozó stíluskonstansok. A különböző stílusokban elkészített szövegek nem feltétlenül térnek el egymástól. Az alapértelmezés szerinti stílus a `MEDIUM`.

Konstruktur, metódusok

- `DateFormat()`
Létrehoz egy dátumformázó objektumot az alapértelmezés szerinti stílussal és környezettel.
- `static DateFormat getTimeInstance()`
► `static DateFormat getTimeInstance(int style)`
► `static DateFormat getTimeInstance(int style, Locale aLocale)`

Visszaad egy **időformázó** objektumot a megadott (vagy az alapértelmezés szerinti) stílusban és környezetben. Az alapértelmezés szerinti stílus a `MEDIUM`.

- `static DateFormat getDateInstance()`
► `static DateFormat getDateInstance(int style)`
► `static DateFormat getDateInstance(int style, Locale aLocale)`

Visszaad egy **dátumformázó** objektumot a megadott (vagy az alapértelmezett) stílusban és környezetben. Az alapértelmezés szerinti stílus a `MEDIUM`.

- ▶ static DateFormat getDateTimeInstance()
 - ▶ static DateFormat getDateInstance(int dateStyle, int timeStyle)
 - ▶ static DateFormat getDateInstance(int dateStyle, int timeStyle, Locale aLocale)
- Visszaad egy **dátum+időformázó** objektumot a megadott (vagy az alapértelmezés szerinti) stílusban és környezetben. Az alapértelmezés szerinti stílus a MEDIUM.
- ▶ final String format(Date date)
- Visszaadja a megformázott date időpontot a beállított értékekkel.

Feladat – Nemzetek dátumai

Írjuk ki a teljes mai dátumot és időt helyi, angol, francia és olasz környezetben!
A dátumot hosszú formában adjuk meg, az időt rövidben!

Forráskód

```
import java.util.*;
import java.text.*;

public class NemzetekDatumai {
    static void datumIdoKiir(Date date, Locale locale) {
        System.out.print(locale.getDisplayCountry() + ": ");
        DateFormat df = DateFormat.getDateInstance(
            DateFormat.FULL, DateFormat.SHORT, locale);
        System.out.println(df.format(date));
    }

    public static void main(String[] args) {
        Locale[] nemzetek = {Locale.getDefault(), Locale.UK,
            Locale.FRANCE, Locale.ITALY};
        Date most = new Date();
        for (int i = 0; i < nemzetek.length; i++) {
            datumIdoKiir(most, nemzetek[i]);
        }
    }
}
```

A program futása

```
Magyarország: 2002. július 14. 18:52
United Kingdom: 14 July 2002 18:52
France: dimanche 14 juillet 2002 18:52
Italy: domenica 14 luglio 2002 18.52
```

Megjegyzések:

- A DateFormat utódja, a SimpleDateFormat osztály lehetőséget ad saját, felhasználói formátumok megadására. A formázás mintázási szöveg alapján megy, megadott szabályok szerint.
- Ha megelégszünk a DateFormat által adott idővel, akkor nincs is szükségünk naptárra.

19.6. Számformázás – NumberFormat

A **NumberFormat** osztály segítségével számokat lehet formázni. Az osztály három statikus metódust kínál számformázó objektumok gyártására:

- pénzformázó (`NumberFormat.getCurrencyInstance()`)
- tagolt számformázó (`NumberFormat.getNumberInstance()`)
- százalékos formázó (`NumberFormat.getPercentInstance()`)

A konkrét formázást a számformázó objektum végzi el.

NumberFormat osztály

Csomag: `java.text`

Deklaráció: `public abstract class NumberFormat`

Közvetlenős: `java.lang.Format`

Fontosabb implementált interfések: `Cloneable, Serializable`

Konstruktur, metódusok

- `static NumberFormat getXXXInstance()`
- `static NumberFormat getXXXInstance(Locale aLocale)`
- xxx lehet `Currency, Number vagy Percent` (`getCurrencyInstance, getNumberInstance` vagy `getPercentInstance`). Visszaad egy pénzt, számot vagy százalékot formázó objektumot az alapértelmezés szerinti (vagy a megadott) környezetben.
- `void setMaximumFractionDigits(int newValue)`
- `void setMaximumIntegerDigits(int newValue)`
- `void setMinimumFractionDigits(int newValue)`
- `void setMinimumIntegerDigits(int newValue)`
- Beállítja a formázandó szám egész-, illetve tizedesrészében szereplő számjegyek maximális, illetve minimális értékét (ha a kettő egyenlő, akkor a pontos értékét).
- `final String format(double number)`
- `final String format(long number)`
- Megformázza a number számot a beállított értékekkel.

Feladat – NumberFormatTest

Írjuk ki ugyanazt a pénz, szám, és százalékértéket először amerikai, majd alapértelmezés szerinti formában! A százalék pontosan 2 tizedesjegyen jelenjen meg!

Forráskód

```
import java.util.Locale;
import java.text.NumberFormat;

public class NumberFormatTest {
    static void lista(Locale locale) {
        System.out.println(locale.getDisplayCountry() + ":");

    }
}
```

```

NumberFormat nf; // számformázó objektum
nf = NumberFormat.getCurrencyInstance(locale);
System.out.print("Penz: "+nf.format(9250));

nf = NumberFormat.getNumberInstance(locale);
System.out.print("\tSzam: "+nf.format(19827.51));

nf = NumberFormat.getPercentInstance(locale);
nf.setMinimumFractionDigits(2);
nf.setMaximumFractionDigits(2);
System.out.println("\tSzazalek: "+nf.format(0.75));
}

public static void main(String[] args) {
    lista(Locale.US);
    lista(Locale.getDefault());
}
}

```

A program futása

United States:		
Penz: \$9,250.00	Szam: 19,827.51	Szazalek: 75.00%
Magyarország:		
Penz: Ft9 250	Szam: 19 827,51	Szazalek: 75,00%

19.7. Megfigyelés – Observer, Observable

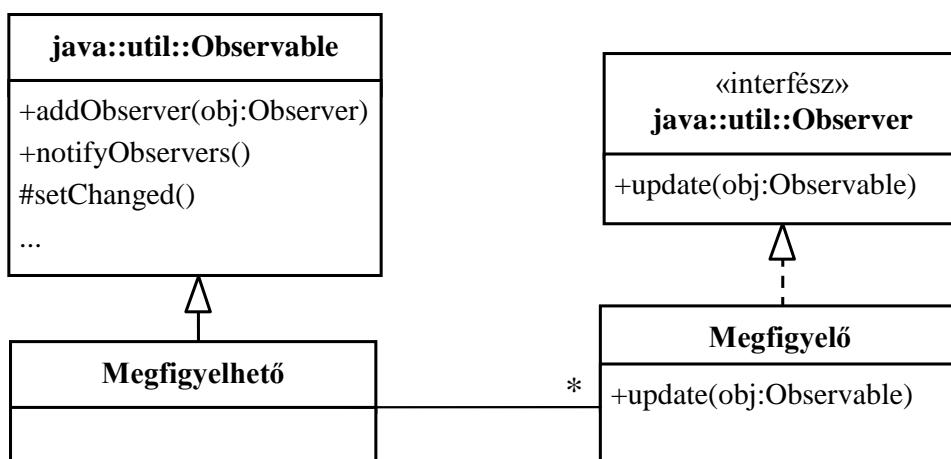
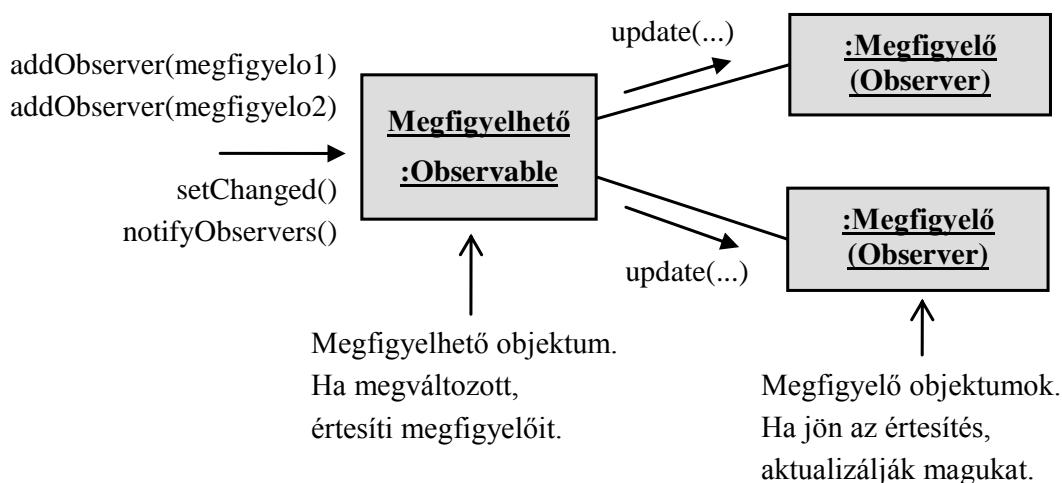
A megfigyelés egy tervezési minta (design pattern); e minta szerint egy **megfigyelhető** (másnépp: megfigyelt) objektumhoz **megfigyelő** objektumokat kapcsolunk. Amikor a megfigyelhető objektum megváltoztatja állapotát, akkor értesíti a megfigyelőit, s azok ennek megfelelően aktualizálhatják a saját állapotukat (19.2. ábra).

- **Megfigyelhető** (observable) objektum: Osztálya az **Observable** osztály leszármazottja. Megjegyezi a megfigyelőit (`addObserver`), és értesítést küldhet minden megfigyelőjének (`notifyObservers`): meghívja az `update` metódusukat.
- **Megfigyelő** (observer) objektum: Osztálya implementálja az **Observer** interfész, emiatt van `update` metódusa.

A megfigyelhető objektumok általában információhordozók, a megfigyelők pedig felhasználói interfések. A megfigyelési technikával elérhető, hogy ha egy adat változik, akkor az azt megjelenítő ablakok azonnal reagálhassanak erre a változásra.

A 19.2. ábra objektumdiagramján egy megfigyelhető objektum van és annak két megfigyelője. A megfigyelhetőre rá kell fűzni a megfigyelőket (`addObserver`). Ha a megfigyelhető objektum változik, akkor értesítést küld megfigyelőinek (`notifyObservers`): meghívja az `update` metódusukat. A megfigyelhető objektum általában maga hívja meg a `notifyObservers`

metódust. A `notifyObservers` metódusnak csak akkor van hatása, ha a megfigyelhető objektum változott (`hasChanged==true`).



19.2. ábra. A megfigyelhető és a megfigyelő objektumok

A megfigyelhető – Observable osztály

Csomag: java.util

Deklaráció: public class Observable

Közvetlen ős: java.lang.Object

Konstruktörök, metódusok

- ▶ void addObserver(Observer obj)

Az obj figyelő hozzákapcsolása ehhez a megfigyelhető objektumhoz.

- ▶ void deleteObserver(Observer obj)

- ▶ void deleteObservers()

A kapcsolat megszüntetése a megadott megfigyelővel vagy az összessel.

- ▶ int countObservers()

Visszaadja a figyelők számát.

- ▶ protected void setChanged()

- ▶ protected void clearChanged()

- ▶ boolean hasChanged()

A setChanged változott (changed) állapotba teszi a megfigyelhető objektumot – ekkor a hasChanged visszaadott értéke true lesz. A clearChanged törli ezt az állapotot.

- ▶ notifyObservers()

- ▶ notifyObservers(Object arg)

Ha hasChanged értéke true, akkor (és csak akkor!) az objektum összes megfigyelőjét értesíti a változásról, éspedig úgy, hogy meghívja az update metódusukat; ezután meghívódik a clearChanged metódus; a második esetben az objektum az arg objektumot is elküldi a megfigyelőknek. Ha tehát értesíteni akarjuk a megfigyelőket, akkor előbb meg kell hívunk a setChanged metódust!

A megfigyelő – Observer interfész

Csomag: java.util

Deklaráció: public interface Observer

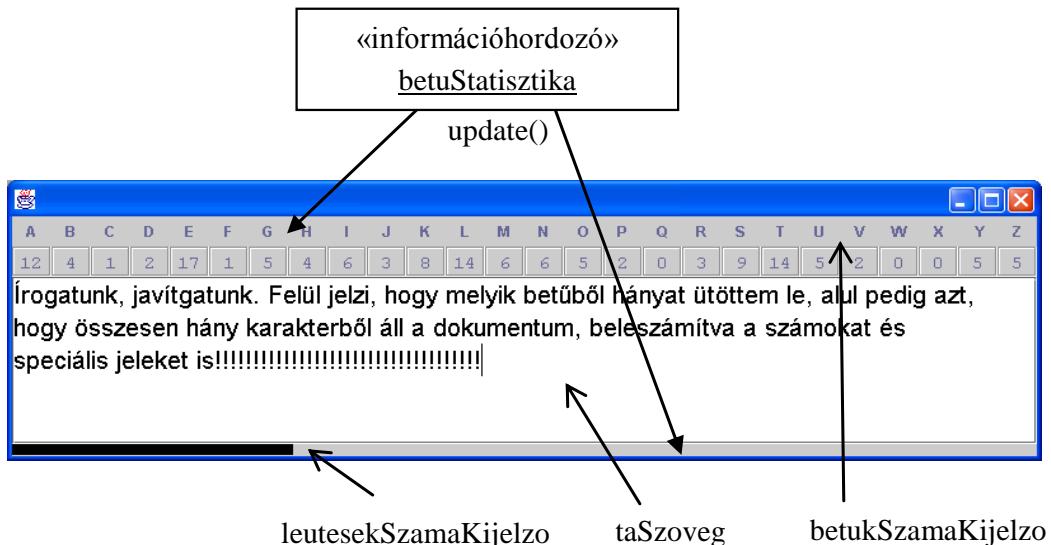
Metódus

- ▶ public void update(Observable obj, Object arg)

Ez a metódus mindenkor meghívódik, valahányszor változik a megfigyelhető objektum. Az arg objektumot a megfigyelhető objektum küldi – mint paramétert.

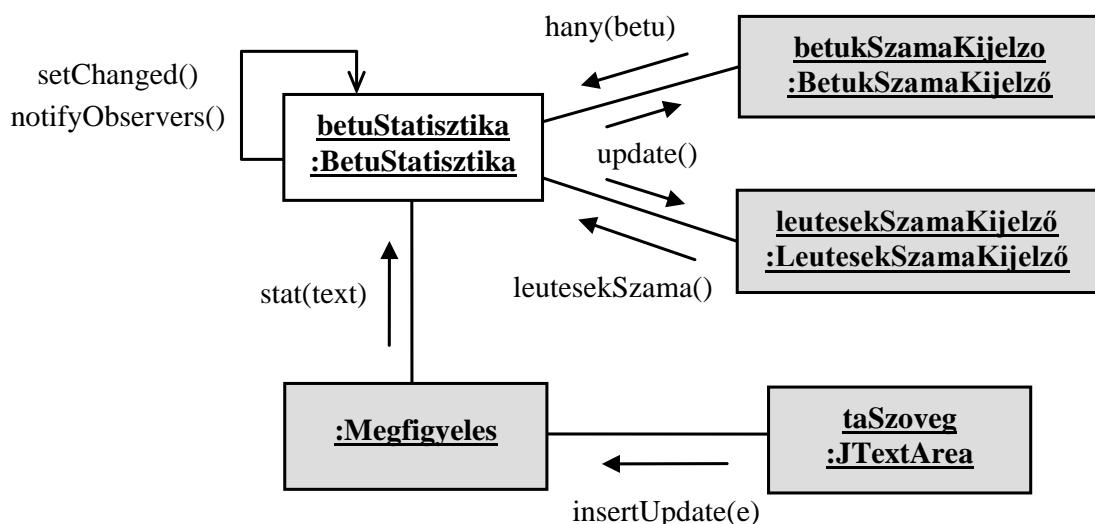
Feladat – Megfigyelés

Készítsük el a képen látható alkalmazást! Középen legyen egy szabadon szerkeszthető szövegterületet (lehet karaktert beszúrni, törlni és módosítani). Felül egy kijelző mutassa meg, hogy melyik betűből összesen hány van éppen a dokumentumban (csak az 'A' és 'Z' közötti betűk számítanak; a kis- és nagybetűket nem különböztetjük meg). Alul egy újabb kijelző egy vízszintesen növekedő, illetve összehúzódó fekete téglalappal mutassa a dokumentum karaktereinek pillanatnyi számát (egy betűnek 1 pont feleljen meg)!



A program terve

A program együttműködési diagramja a 19.3. ábrán látható. A betuStatisztika csupán adatokat tárol, a többi objektum a képernyőn megjelenő komponens. Az alkalmazás kerete a Megfigyeles. A taSzoveg eseményeit a keret kezeli (insertUpdate). Ha változik a taSzoveg tartalma, akkor a keret szól a betuStatisztiká-nak, hogy készítsen statisztikát. Az eleget tesz ennek a felszólításnak, egyszersmind szól a GUI megfigyelőinek, hogy aktualizálják magukat, mert megváltoztak az általuk megjelenítendő adatok. A két megfigyelő update metódusa kikéri a statisztikai adatokat a megfigyelt betuStatisztika-tól, a megfigyelők maguk pedig módosítják a kijelzést.



19.3. ábra. A Megfigyelés program együttműködési diagramja

Forráskód

Projekt: Megfigyeles
 Csomagok: db, gui

A forráskódnak csak a kritikus részeit adjuk meg. A mellékletben megvan a teljes forráskód.

BetuStatisztika.java

```
package db;
import java.util.*;

public class BetuStatisztika extends Observable {
    private int[] betukSzama;
    private int leutesekSzama;

    public void stat(String text) {
        // betukSzama és leutesekSzama kiszámítása text alapján
        // ...
        setChanged();                                         //1
        notifyObservers();
    }

    public int hany(char betu) {
        betu = Character.toUpperCase(betu);
        if (betu<'A' || betu>'Z')
            return 0;
        return betukSzama[betu-'A'];
    }

    public int leutesekSzama() {
        return leutesekSzama;
    }
}
```

BetukSzamaKijelzo.java

```
package gui;
import db.*;
// ...

public class BetukSzamaKijelzo extends JPanel
    implements Observer {
    private JLabel[] lbAdatok = new JLabel[26];

    public BetukSzamaKijelzo() {
        // panel összeállítása
        // ...
    }

    public void update(Observable obj, Object arg) {          //2
        // A címkek aktualizálása bs.hany() meghívásaival:
        if (!(obj instanceof BetuStatisztika))
            return;
        BetuStatisztika bs = (BetuStatisztika)obj;
        for (char c = 'A'; c <= 'Z'; c++)
            lbAdatok[c-'A'].setText(bs.hany(c)+"");
    }
}
```

LeutesekSzamaKijelzo.java (teljes kód)

```

package gui;
import db.*;
import java.util.*;
import javax.swing.*;
import java.awt.*;

public class LeutesekSzamaKijelzo extends JPanel
    implements Observer {
    int szel = 0;

    public void update(Observable obj, Object arg) {           //3
        if (!(obj instanceof BetuStatisztika))
            return;
        BetuStatisztika bs = (BetuStatisztika)obj;
        szel = bs.leutesekSzama();
        repaint();
    }
    public void paintComponent(Graphics gr) {
        super.paintComponent(gr);
        int mag = getHeight();
        if (szel>getWidth())
            szel = getWidth();

        gr.fillRect(0,0,szel-1,mag-1);
    }
}

```

Megfigyeles.java

```

import gui.*;
import db.BetuStatisztika;
// ...

public class Megfigyeles extends JFrame
    implements DocumentListener {                                //4
    JTextArea taSzoveg = new JTextArea("",5,20);
    BetuStatisztika betuStatisztika = new BetuStatisztika();
    LeutesekSzamaKijelzo leutesekSzamaKijelzo =
        new LeutesekSzamaKijelzo();
    BetukSzamaKijelzo betukSzamaKijelzo =
        new BetukSzamaKijelzo();

    public Megfigyeles() {
        // ... keret összeállítása
        betuStatisztika.addObserver(betukSzamaKijelzo);          //5
        betuStatisztika.addObserver(leutesekSzamaKijelzo);
        taSzoveg.getDocument().addDocumentListener(this);
        // ...
    }
}

```

```

public void insertUpdate(DocumentEvent e) {
    // ...
    betuStatisztika.stat(d.getText(0,d.getLength()));           //6
    // ...
}
// ...
}

```

A forráskód elemzése

A Megfigyeles keretben – //4-ben – létrehozzuk a megfelelő objektumokat. A betuStatisztika a megfigyelhető objektum; //5-ben ráakasztjuk a megfigyelőket. A taSzoveg fog változni; a DocumentListener eseménykezelőiben meghívjuk a betuStatisztika.stat metódust (//6). E metódus hatására megváltozik a betuStatisztika állapota, ezért //1-ben beállítjuk a változás tényét (setChanged), és értesítjük a megfigyelőket (notifyObservers). Ennek hatására meghívódik az összes megfigyelő update metódusa (//2 és //3).

19.8. Klónozás – Cloneable

Az objektumklónozás azt jelenti, hogy lemásolunk, duplikálunk egy objektumot. Ilyenkor az objektum adatai a konstruktor meghívása nélkül sorra átmásolódnak egy másik memóriaterületre. Nem minden világos, hogy pontosan mi értendő az objektum másolásán. A másolandó objektum nemcsak primitív adatokat tartalmazhat, hanem referenciakat, sőt konténereket is, a tartalmazott objektumok pedig további objektumokat, bonyolult szerkezeteket, és az sem lehetetlen, hogy két objektum megosztózik bizonyos más objektumokon. A másolás pontos technikáját az osztály tervezője adja meg.

► A klónozás meglehetősen kényes művelet. Egy rossz másolásból súlyos logikai és futási hibák adódhatnak.

Klónozáskor (az objektum másolatának elkészítésekor) a konstruktor meghívása nélkül meghívjuk a másolandó objektum `clone()` metódusát, s az visszaad egy másolt objektumot. A klónozás célja általában az, hogy teljesüljenek a következők (`obj` az eredeti objektum):

- a másolt objektum különbözzék az eredetitől (más memóriaterületet foglaljon le):
`obj.clone() != obj`
- a másolt objektum osztálya egyezzék meg az eredeti objektum osztályával:
`obj.clone().getClass() == obj.getClass()`
- a két objektum legyen egyenlő: `obj.clone().equals(obj) == true`

Ahhoz, hogy egy objektum klónozható legyen, az objektum osztályának implementálnia kell a **Cloneable** interfészét és tartalmaznia kell egy publikus `clone()` metódust.

Cloneable interfész

Csomag: `java.lang`

Deklaráció: `public interface Cloneable`

Az interfész jelölő szerepet játszik, nem definiál egyetlen metódust sem. A `Cloneable` interfész implementálása jelzi, hogy az osztály objektuma biztonságosan klónozható. Csak olyan objektum klónozható, amelynek osztálya vagy valamely őse implementálta a `Cloneable` interfészt. Ha egy olyan objektumnak küldjük a `clone()` üzenetet, amely nem klónozható (`not cloneable`), akkor `CloneNotSupportedException` (a klónozás nincs engedélyezve) kivétel keletkezik. Az `Object` osztály nem implementálja a `Cloneable` interfészt, és `clone()` metódusa védett.

Az `Object` osztály `clone` metódusa

- ▶ `protected native Object clone() throws CloneNotSupportedException`
Bájtonként átmásolja az objektumot; a másolt objektum lesz a visszatérési érték. Referencia típusú változók esetén csak a mutató értéke másolódik, a mutatott objektum nem (ez a sekély klónozás). A metódus védett, így a klónozandó objektum osztályában a programozónak újra kell írni a `clone()` metódust, `public` módosítóval (az elérhetőséget bővíteni lehet, csak szűkíteni nem). Korlátozó tényező, hogy a felülírt `clone()` visszatérési értéke is `Object` típusú kell, hogy legyen, akárcsak az őse.

A klónozandó objektum osztályában a programozónak újra kell írni a `clone()` metódust, `public` módosítóval. A felülírt metódusban általában meghívjuk az őt `Object` osztály `clone()` metódusát. Ez a `clone()` metódus nem kezeli a `CloneNotSupportedException` ellenőrzött kivételt (`throws`), ezért azzal is foglalkoznunk kell. Ha csak az őt `clone()` metódust akarjuk meghívni, két lehetőségünk van:

```
public Object clone() throws CloneNotSupportedException {
    return super.clone();
}
```

vagy

```
public Object clone() {
    try {
        return super.clone();
    }
    catch (CloneNotSupportedException e) {
        return this;
    }
}
```

Sekély klónozás – mély klónozás

Az egyszerű memóriamásolással végzett klónozást **sekély klónozásnak** (shallow cloning) nevezzük. A klónozás **mély** (deep cloning), ha a tartalmazott objektumokat is klónozzuk.

Ha a másolandó objektum csupán primitív adatmezőkből áll, akkor a klónozás egyszerű feladat: bájtról bájtra át kell másolni az objektumot. Bár a `clone()` metódust mindenképpen meg kell írni a maga osztályában (hiszen az eredeti védett), az átírt metódusban elegendő meghívni az ős `clone()` metódust.

Ha a másolandó objektum referencia típusú mezőt is tartalmaz, akkor a mező másolata ugyanaz a referencia lesz, mint az eredeti. Ha a mutatott objektumot is klónozni akarjuk (ez a mély klónozás), akkor az osztálynak egy összetettebb `clone()` metódust kell definiálnia.

A klónozhatóság vizsgálata

Egy objektum akkor klónozható, ha implementálja a `Cloneable` interfészt, vagyis a következő kifejezésnek `true` az értéke:

```
obj instanceof Cloneable
```

A Javában a tömb és a `Vector` klónozható. Fontos azonban tudni, hogy a tartalmazott objektumok nem másolódnak sem a tömbben, sem a vektorban.

Az API csak néhány osztályban implementálta a `Cloneable` interfészt. Klónozható osztály például a `Vector`, a `Rectangle`, a `Date`, a `Locale`, a `TimeZone`, a `GregorianCalendar`, a `DateFormat` és a `NumberFormat`. A komponensek, a csomagoló osztályok, a `String`, és a `StringBuffer` nem klónozható osztályok.

Feladat – Sekély klónozás (CloneSekely)

Készítsünk egy Gyerek osztályt három primitív adattal: magasság (int), súly (int) és jóság (boolean). Hozunk létre egy gyereket! Ha jó a gyerek, akkor klónozzuk!

Forráskód

```
class Gyerek implements Cloneable { //1
    int magassag=120, suly=35;
    boolean jo=true;

    public Object clone() throws CloneNotSupportedException { //2
        return super.clone(); //3
    }

    public String toString() {
        return magassag+" "+suly+" "+(jo?"JÓ":"Rossz");
    }
}
```

```

public class CloneSekely {
    public static void main (String args[]) throws
        CloneNotSupportedException {
        Gyerek eredeti = new Gyerek();
        Gyerek masolt = null;

        if (eredeti instanceof Cloneable && eredeti.jo)           //4
            masolt = (Gyerek)eredeti.clone();                      //5

        System.out.println("Eredeti: "+eredeti);
        if (masolt!=null)
            System.out.println("Másolt : "+masolt);
        else
            System.out.println("A gyerek nem klónozható, vagy "+
                "nem érdemes klónozni");
    }
}

```

A program futása

Eredeti: 120 35 Jó
Másolt : 120 35 Jó

A program elemzése

A Gyerek osztályban implementáljuk a `Cloneable` interfészt (//1), különben //5-ben a gyereket nem lehetne klónozni. //2-ben megadjuk a `clone()` metódust – a `CloneNotSupportedException` kivétellel most nem foglalkozunk, és ezt a metódus fejében közöljük is. A metódus minden össze az ős `clone()` meghívásából áll (//3), hiszen a gyereket (a gyerek primitív adatait) mi is bájtról bájtra akarjuk másolni. //4-ben megvizsgáljuk, hogy a másolandó (eredeti) objektum klónozható-e. Ha igen és a gyerek jó is, akkor //5-ben megtörténik a klónozás. A `clone()` visszatérési típusa `Object`, ezért a másolt objektumra rákényszerítjük annak valódi osztályát, a `Gyerek`-et.

Feladat – Primitív elemtípusú tömb klónozása (ClonePrimitivTomb)
Klónozzunk egy primitív (`int`) elemtípusú tömböt!

Forráskód

```

public class ClonePrimitivTomb {

    static void print(int[] t) {                                //1
        for (int i=0; i<t.length; i++)
            System.out.print(t[i]+" ");
        System.out.println();
    }
}

```

```

public static void main (String args[]) {
    int[] tomb = new int[5];
    for (int i=0; i<tomb.length; i++)
        tomb[i] = i;

    int[] masoltTomb = (int[])tomb.clone(); //2
    masoltTomb[0] = 9;
    print(tomb);
    print(masoltTomb);
}
}

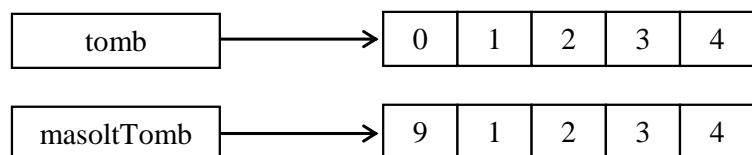
```

A program futása

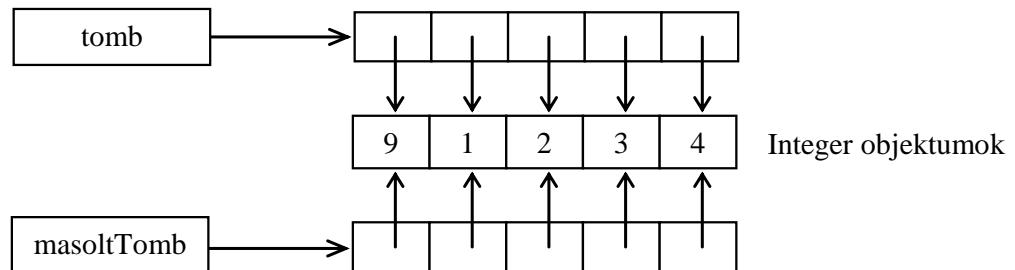
0	1	2	3	4
9	1	2	3	4

A program elemzése

Az //1-ben deklarált `print()` metódus egy `int` elemtípusú tömböt ír ki. A klónozás //2-ben zajlik; a tömb elemei értékről értékre átmásolódnak a klónozott tömbbe. Mivel a `tomb.clone()` visszatérési típusa `Object`, a tömbre rákényszerítjük a típusát. A program futásából egyértelműen látszik, hogy a két tömb elemei „külön életet élnek”, hiszen az egyik megváltoztatása (`masoltTomb[0]=9`) nem vonja maga után a másik változását (19.4. ábra).



19.4. ábra. Primitív elemtípusú tömb klónozása



19.5. ábra. Referencia elemtípusú tömb klónozása

Egy tömb csak sekély klónozással másolható. A 19.5. ábrán látható, hogy egy referencia elemtípusú tömb elemei megegyeznek a klónozott tömb elemeivel. Ha egy tömbben a referencia típusú elemeket is másolni akarjuk, akkor azt egyenként kell elvégeznünk.

Megjegyzés: A mellékletben talál egy feladatot mély klónozásra is, magyarázattal ellátva.
(*CloneMely.java*)

19.9. Rendszerjellemzők – System

A **System** osztály intézi az **operációs rendszerrel való kommunikációt**. Metódusaival többek között lekérdezhetők és beállíthatók a rendszer jellemzői.

System osztály

Csomag: `java.lang`

Deklaráció: `public final class System`

Közvetlen ős: `java.lang.Object`

Metódusok (rendszerjellemzők)

- ▶ `static Properties getProperties()`
- ▶ `static String getProperty(String key)`
- ▶ `static String getProperty(String key, String def)`
- ▶ `static void setProperties(Properties prop)`
- ▶ `static String setProperty(String key, String value)`

A metódusokkal lekérdezhetők, illetve beállíthatók a rendszer jellemzői (system properties). A `Properties` osztály rendszerjellemzők tárolására való.

Megjegyzés: Az összes rendszerjellemzőt megtekintheti, ha lefuttatja a következő programot.

Feladat – Rendszerjellemzők

Írjuk ki

- a Java home könyvtárat!
- a számítógép használójának a nevét!
- az osztályútvonalat (CLASSPATH)!
- az összes rendszerjellemzőt egyszerre!

Forráskód

```
import java.util.*;  
  
public class RendszerJellemzok {  
    public static void main(String[] args) {  
        System.out.print("Java home konyvtár: ");  
        System.out.println(System.getProperty("java.home"));  
    }  
}
```

```
System.out.print("Használó: ");
System.out.println(System.getProperty("user.name"));

System.out.print("Osztályútvonal: ");
System.out.println(System.getProperty("java.class.path"));

System.out.println("\nAz összes rendszerjellemző:");
Properties prop = System.getProperties();
StringTokenizer st=new StringTokenizer(prop.toString(),",");
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
}
```

A program futása

```
Java home konyvtár: J:\JBuilderX\jdk1.4\jre
Használó: Angster Erzsébet
Osztályútvonal:
C:\javaprog\javaprog_classes;C:\javaprog\lib\javalib.jar;...
Az összes rendszerjellemző:
{java.runtime.name=Java (TM) 2 Runtime Environment
 Standard Edition
sun.boot.library.path=J:\JBuilderX\jdk1.4\jre\bin
java.vm.version=1.4.2_01-b06
java.vm.vendor=Sun Microsystems Inc.
java.vendor.url=http://java.sun.com/
path.separator=;
java.vm.name=Java HotSpot (TM) Client VM
file.encoding.pkg=sun.io
user.country=HU
...
user.dir=C:\javaprog
java.runtime.version=1.4.2_01-b06
...
```

19.10. Külső program futtatása – Runtime

A Java alkalmazás külső programot **is futtathat** és kezeli a külső programtól származó adatokat. A külső program futtatásakor a következő dolgokat kell elvégezni:

1. Egy példányt hozunk létre a Runtime osztályból; az megadja az aktuális futási környezetet: `Runtime rt = Runtime.getRuntime();`
2. Meghívjuk a futási környezet `exec` metódusát a szükséges paraméterekkel: `rt.exec(cmdArray, envp, dir)`. Itt lehet megadni a külső program nevét és paramétereit (`cmdArray`), a szükséges környezeti változókat (`envp`), valamint az esetleges munkakönyvtárat (`dir`). A program teljes útvonalát meg kell adni, mert az `exec` nem használja az operációs rendszer `PATH` környezeti változóját. A program létezéséről

előzőleg meg kell győzödni! Az `exec` metódus ad egy folyamatobjektumot; az a `Process` absztrakt osztály valamely leszármazottjából való.

3. A `Process` objektumnak van egy beviteli folyama: `getInputStream()`, egy kiviteli folyama: `getOutputStream()` és egy hibafolyama: `getErrorStream()`; minden lekérdezhető és feldolgozható a hívó programban.

Megkötések:

- Csak olyan programokkal lehet input/output kapcsolatot teremteni, amelyek az operációs rendszer szabványos `in`, `out` és `err` eszközeit használják.
- Appletből nem lehet programot hívni.

Runtime osztály

Csomag: `java.lang`

Deklaráció: `public class Runtime`

Közvetlen ős: `java.lang.Object`

Az osztálynak nincs publikus konstruktora. Példányát a rendszer felügyeli.

Metódusok (külső program futása)

- `static Runtime getRuntime()`
Visszaadja az aktuális futási környezetet.
- `Process exec(String cmdarray[], String envp[], File dir) throws IOException`
- `Process exec(String cmdarray[], String envp[]) throws IOException`
- `Process exec(String cmdarray[]) throws IOException`
- `Process exec(String command, String envp[], File dir) throws IOException`
- `Process exec(String command, String envp[]) throws IOException`
- `Process exec(String command) throws IOException`

A `cmdarray` tömb tartalmazza a program útvonalát (0. elem) és az esetleges paramétereket (1., 2... elemek). Ha a programnak nincs paramétere, akkor az útvonalat `command`-ban lehet megadni. Az `envp` tömb tartalmazza a futáshoz szükséges környezeti változókat. `dir`-ben megadható egy munkakönyvtár.

Process absztrakt osztály

Csomag: `java.lang`

Deklaráció: `public abstract class Process`

Közvetlen ős: `java.lang.Object`

Metódusok

- `abstract OutputStream getOutputStream()`
- `abstract InputStream getInputStream()`
- `abstract InputStream getErrorStream()`

A folyamat kiviteli, beviteli és hibafolyama.

Feladat – Java program fordítása programból (Compile)

Hívjuk meg egy Java programból a `javac.exe` fordítóprogramot! Fordítsuk le egy megadott könyvtár megadott Java forráskódját! A hibalistát a környezet szabványos kimenetére írjuk! Ha a fordítás sikeres, akkor ezt írjuk ki a konzolra!

Forráskód (részlet)

```

String javacDir = ...; // javac.exe könyvtára
String classPath = ...; // CLASSPATH környezeti változó tartalma
String actualDir = ...; // a forráskód könyvtára
String actualFilename = ...; // a forráskód neve

public boolean compile() {
    boolean ok = false;
    //...
    String[] cmdarray = {javacDir+"javac.exe",
                         actualDir+actualFilename};
    String[] envp = {"CLASSPATH="+classPath};
    Runtime rt = Runtime.getRuntime();
    try {
        Process pr = rt.exec(cmdarray, envp);
        BufferedReader errors = new BufferedReader(
            new InputStreamReader(pr.getErrorStream()));
        String line;
        if ((line = errors.readLine()) == null) {
            System.out.println("A fordítás sikerült.");
            ok = true;
        }
        else {
            do {
                System.out.println(line);
            } while ((line = errors.readLine()) != null);
        }
    }
    catch (IOException e) {
        System.out.println(e);
    }
}

```

Tesztkérdések

19.1. Jelölje be azokat a szintaktikailag helyes utasításokat, amelyekben a `long` típusú `ido` változó értéke az aktuális univerzális időpont!

- a) `ido = Runtime.currentTimeMillis();`
- b) `ido = System.currentTimeMillis();`
- c) `ido = new Date();`
- d) `ido = new GregorianCalendar().getTime();`

19.2. Mely állítások igazak? Jelölje be az összes helyes állítást!

- a) Az UTC szabvány szerinti idő helyi időt ábrázol.
- b) A `Date` osztály az UTC szabvány szerinti univerzális időt ábrázolja.

- c) Egy `Locale` objektum tárolja az aktuális időt.
d) Egy `Locale` objektum egy környezetet azonosít.
- 19.3. Mely állítások igazak? Jelölje be az összes helyes állítást!
- A `TimeZone` osztály feladata az időeltolás mértékének tárolása, valamint az éjjel-nappal szabályok megadása.
 - A `GregorianCalendar`-ban be lehet állítani az aktuális univerzális időpontot.
 - A `GregorianCalendar` osztály `get(Calendar.HOUR_OF_DAY)` metódusának visszatérési értéke egyes-egyedül az eltárolt univerzális időtől függ.
 - Egy `Calendar` objektum megmondhatja, hogy ez vagy az az időpont a hét mely napjára esik.
- 19.4. Mely állítások igazak? Jelölje be az összes helyes állítást!
- A `DateFormat` osztály segítségével időpontot lehet formázni.
 - A `DateFormat` formázó objektumnak be lehet állítani a környezetét.
 - A `NumberFormat` osztály a `javax.swing` csomagban található.
 - A `NumberFormat` osztály segítségével megtehetjük, hogy számokat tagolunk egy adott `Locale` objektumtól függően.
- 19.5. Mely állítások igazak? Jelölje be az összes helyes állítást!
- Az `Object` osztályban van `clone()` metódus.
 - A Javában minden objektum klónozható.
 - Egy referencia elemtípusú tömb klónozásakor a klónozott elemek megegyeznek az eredetivel.
 - Az `Object` osztály `clone()` metódusa sekély klónozást végez.
- 19.6. Mely állítások igazak? Jelölje be az összes helyes állítást!
- A Javában egy külső program futtatásához a `Runtime` osztálytól el kell kérni az aktuális futási környezetet.
 - Futtatáskor egy folyamat (`Process`) objektum keletkezik.
 - A futtatandó külső programoknak nem adhatunk át paramétereket .
 - Appletből is lehet külső programot futtatni.

Feladatok

- 19.1. (A) Jelenítsen meg egy keretet, s abban egy gombot! Számolja, hogy mennyi ideig tartják nyomva a gombot! minden felengedés után írja ki, hogy a gomb eddig összesen mennyi ideig volt lenyomva! (*MennyitNyomod.java*)
- 19.2. Írja ki a számítógépen elérhető
- (A) összes nemzetközi környezet ISO országazonosítóját és beszédes nevét!
 - (B) összes nyelvet (például English, Hebrew stb.), de mindegyiket csak egyszer!
 - (B) összes időzóna azonosítóját (például Pacific/Api, Europe/Budapest)! (*NemzetkoziLista.java*)

- 19.3. (A) Írja ki az 1 órás eltolású időzónák nevét és azonosítóját!
(*EgyOraEltolas.java*)
- 19.4. (A) Állapítsa meg, hogy az 106836795488 időpont melyik évben, hónapban és napon volt! (*MelyikEvHoNap.java*)
- 19.5. (B) Készítsen olyan alkalmazást, mely megjeleníti az aktuális dátumot és időpontot! A nemzeti beállítást, az időzónát, a dátum stílusát, valamint a dátum betűinek a nagyságát egy-egy rendezett listából lehessen kiválasztani!
(*NemzetkoziDatum.java*)
- 19.6. (A) Először írja ki az összes elérhető kétkarakteres nyelv és országazonosítót! Ezután jelenítse meg a 19566.6 számot az összes elérhető környezet szerinti formában!
(*NyelvOrszagSzam.java*)
- 19.7. (B) Írja ki a
- 7566512.77 számot amerikai, francia és magyar környezetben!
 - 46.5 százalékot német, olasz és magyar környezetben!
 - 28 millió egységnnyi pénzt amerikai, svéd és magyar környezetben!
 - mai dátumot amerikai, román és magyar környezetben!
- (*Formazas.java*)
- 19.8. (C) Készítsen olyan havi naptárat, amelyben megválasztható az év és a hónap! A naptár fejlécében a megadott nyelvben használatos rövid névvel jelenjenek meg a hétfajai!
(*Naptar.java*)

Év:	2002	Hó:	Július			
Va	Hé	Ke	Sz	Cs	Pé	Sz
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

- 19.9. (A) Írja ki az operációs rendszer PATH környezeti változójának értékét! (*PathKiir.java*)
- 19.10. (B) Futtasson a programjából
- egy bájtkódöt!
 - egy JAR-állományt!
 - egy tetszőleges programot, például a rendszer kalkulátorát!
- (*Futtat.java*)
- 19.11. (C) Fordítson le egy Java alkalmazást a JVM-mel! (*CompileJava.jpx*)

I. OBJEKTUMORIENTÁLT TECHNIKÁK

1. Csomagolás, projektkezelés
2. Örökítés
3. Interfészek, belső osztályok
4. Kivételkezelés

II. GRAFIKUS FELHASZNÁLÓI INTERFÉSZ

5. A felhasználói interfész felépítése
6. Elrendezésmenedzserek
7. Eseményvezérelt programozás
8. Swing-komponensek
9. Grafika, képek
10. Alacsony szintű események
11. Belső eseménykezelés, komponensgyártás
12. Applet

III. ÁLLOMÁNYKEZELÉS

13. Állományok, bejegyzések
14. Folyamok
15. Közvetlen hozzáférésű állomány

IV. VEGYES TECHNOLÓGIÁK

16. Rekurzió
17. Többszálú programozás
18. Nyomtatás
19. Hasznos osztályok

V. ADATSZERKEZETEK, KOLLEKCIÓK

20. Klasszikus adatszerkezetek
21. Kolekció keretrendszer

V.

FELADATOK

FÜGGELÉK

- A tesztkérdések megoldása
Irodalomjegyzék
Tárgymutató



20. Klasszikus adatszerkezetek

A fejezet pontjai:

1. Az adatszerkezetek rendszerezése
 2. Absztrakt tárolók
 3. Tömb
 4. Tábla
 5. Verem
 6. Sor
 7. Fa
 8. Irányított gráf, hálózat
-

Ebben a fejezetben klasszikus adatmodellekről és azok viselkedéséről lesz szó; a klasszikus adatmodellek fontos szerepet játszanak a számítástechnikában, minden programozónak ismernie kell őket, és tudnia kell, hogy mikor melyiket alkalmazza. A szoftverfejlesztésben kulcskérdés lehet a konténerek típusának jó megválasztása. Az adatok, objektumok tárolásának és visszakeresésének módja, tárígyene és hatékonysága meghatározhatja, mennyire lesz áttekintő és elvezethető az elkészült szoftver.

A fejezetben klasszikus adatszerkezetekkel foglalkozunk – s nem csak olyanokkal, amelyeket a Java API is használ. A Java kollekció keretrendszerrel a következő fejezetben lesz szó; mint látni fogjuk, az lépten-nyomon használja az itt tárgyalt adatszerkezeteket.

20.1. Az adatszerkezetek rendszerezése

Egy szoftverrendszer kialakításakor **adat**-, és **eljárásszerkezeteket** (modelleket) kell készíteni. Az emberhez közelebb álló logikai modelleket a hardver lehetőségeit figyelembe véve le kell képezni a számítógépre. Az adatszerkezetek fizikai elemei a központi tárban, illetve külső tárolószolgáltatókban helyezkednek el, az eljárásszerkezeteknek gépi kódú utasítások felelnek meg.

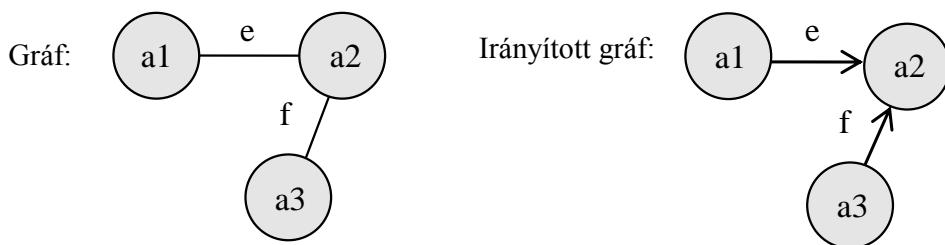
Bizonyos klasszikus adatszerkezetek és a rajtuk értelmezett eljárásszerkezetek elmélete már régen jelen van a számítástechnikában. Ismeretesek az adatok szerkezetének és működésének

szabályai, és a modellek is rendszerezve vannak. A programozónak, illetve a modellkészítőnek az a feladata, hogy felismerje ezeket a modelleket a maga készítendő rendszerében és helyesen alkalmazza őket – ehhez persze megfelelő szakmai ismeretek kellenek, absztraktiós készség és szoftverépítő tapasztalat.

Mindenekelőtt tisztázzuk a számítástudományban gyakran használatos gráf fogalmát:

Gráf

A **gráf** csúcsokból és a csúcsokat összekötő élekből álló véges, nem üres halmaz (20.1. ábra). Ha az e él az a_1 és a_2 csúcsokat köti össze, akkor azt mondjuk, hogy a_1 és a_2 illeszkedik e -re, vagy azt, hogy a_1 és a_2 szomszédos csúcsok. Az e él a rendezetlen (a_1, a_2) párral is jellemezhető. A gráfot irányítottnak nevezünk, ha az élekhez irányok is tartoznak. Az irányokat nyilakkal ábrázoljuk. Az irányított gráf e éle a rendezett (a_1, a_2) párral jellemezhető; a_1 a kezdőpont, a_2 a végpont.



20.1. ábra. Gráf

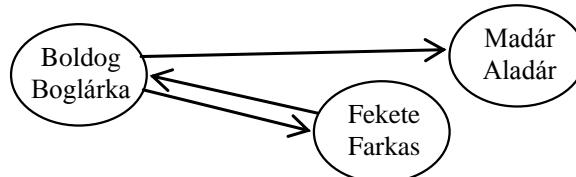
Adatszerkezet

Az **adatszerkezet** egymással kapcsolatban álló elemek, (csomópontok, adatok, objektumok) összessége. Hogy a csomópontok pontosan mit tartalmaznak, az a modell szempontjából lényegtelen. Az adatszerkezet gráffal ábrázolható: a csomópontok a gráf csúcsai, a kapcsolatok a gráf élei.

Vegyük például egy hallgatói névsort. Legyenek a csomópontokban a hallgatók, a rendszer szempontjából fontos tulajdonságaikkal. A hallgatók közötti kapcsolatoknak irányuk van, s ezt az irányt a nevek ábécésorrendje határozza meg. Ezek szerint Boldog Boglárka után Fekete Farkas következik, Fekete Farkast pedig Madár Aladár követi:



Ugyanezeket a hallgatókat egészen más szempont szerint is össze lehetne kötni, például a „szereti” kapcsolat alapján: Fekete Farkas szereti Boldog Boglárkát, Boldog Boglárka pedig szereti Fekete Farkast és Madár Aladárt:



Az adatszerkezetre az elemek közti kapcsolatokon túl a rajta értelmezett műveletek is jellemzők. A műveleteknek lényegében két fő csoportját különböztetjük meg:

- **Konstrukciós műveletek:** Az adatszerkezetet létrehozó és továbbépítő mechanizmusok.
- **Szelekciós műveletek:** Az adatszerkezetet lebontó, megszüntető, valamint az adatelemek eléréséről gondoskodó mechanizmusok.

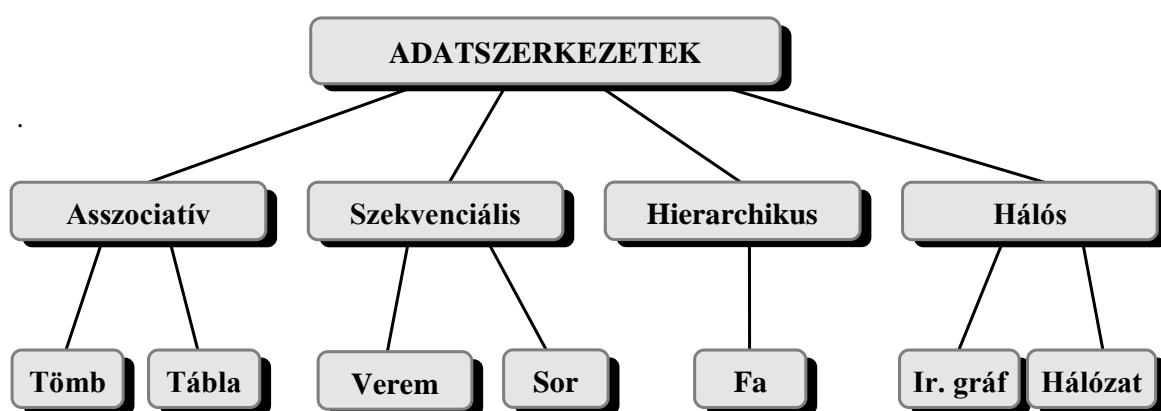
Minden adatszerkezethez meghatározott konstrukciós és szelekciós műveletekből álló készlet tartozik.

Az adatszerkezeten értelmezett műveletek (eljárások) szerkezetét leginkább az elemek közötti kapcsolatok határozzák meg.

A kapcsolatok alapján az adatszerkezetek az alábbi négy fő csoportba sorolhatók (20.2. ábra):

- **Asszociatív adatszerkezetek**
- **Szekvenciális adatszerkezetek**
- **Hierarchikus adatszerkezetek**
- **Hálós adatszerkezetek**

Az adatszerkezetek rendszerezése a 20.2. ábrán látható.

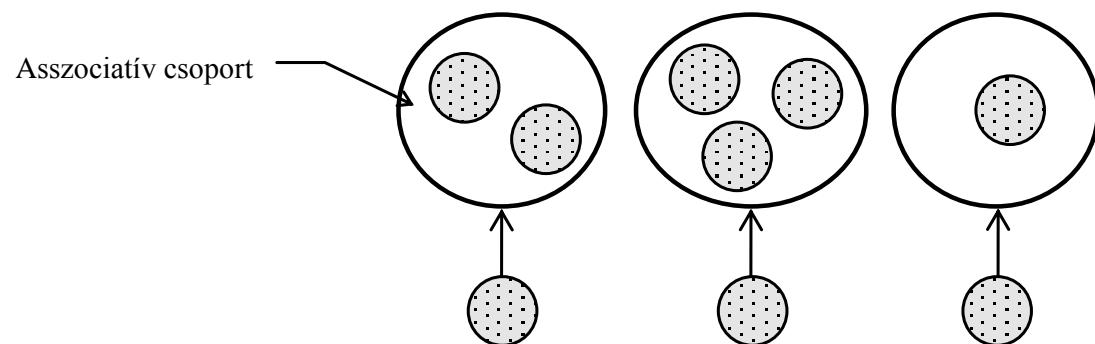


20.2. ábra. Az adatszerkezetek rendszerezése

Először jellemzzük majd a különféle csoportokat és felsoroljuk a hozzájuk tartozó legfontosabb adatszerkezeteket; az adatszerkezeteket és számítógépes megfelelőjüket külön pontok tárgyalják.

Az **asszociatív adatszerkezetben** az elemek közötti kapcsolatokat az elemek azonos tulajdonságértékei létesítik (20.3. ábra). Az elemeket e tulajdonságok alapján csoportosítjuk, így egy elemhez (tulajdonsághoz) egy csoport elemeit társítjuk. Az asszociatív szó jelentése: összekötő, társító. Az elemek közötti kapcsolatok az asszociatív adatszerkezetben a leglázábbak.

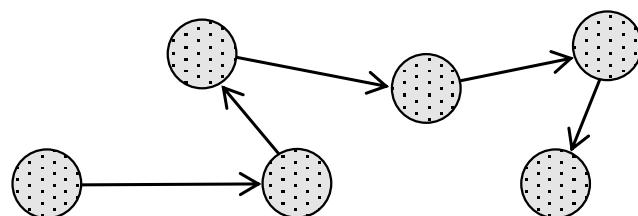
Asszociatív adatszerkezet a tömb és a tábla.



20.3. ábra. Asszociatív adatszerkezet

A **szekvenciális adatszerkezetben** az elemek logikailag egymás után helyezkednek el (20.4. ábra); minden elemet az adatszerkezet egy jól meghatározott eleme követ. Ez egy rendezési reláció – az egyik elem a másik előtt (mögött, alatt, felett stb.) helyezkedik el.

Szekvenciális adatszerkezet a verem és a sor.

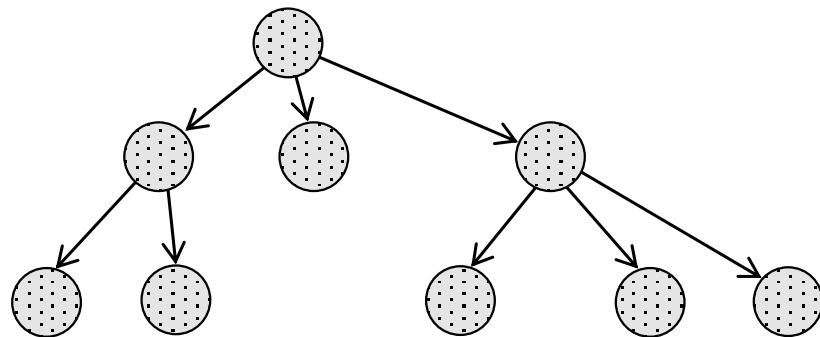


20.4. ábra. Szekvenciális adatszerkezet

A szekvenciális adatszerkezet olyan problémák megoldására használható, amelyekben az elemeket valamelyen szempont szerint sorban kell feldolgozni – akár az összeset, akár egy összefüggő részüket.

A **hierarchikus adatszerkezetben** a csomópontok hierarchikusan egymás alá vannak rendelve (20.5. ábra). Egy csomópont csak egy kapcsolatban lehet végpont.

Hierarchikus adatszerkezet a fa.

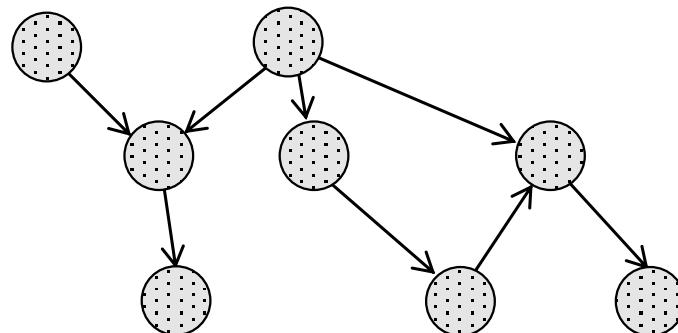


20.5. ábra. Hierarchikus adatszerkezet

A hierarchikus adatszerkezet olyan problémák megoldására alkalmazható, amelyekre a tulajdonosi viszony, lebontás, illetve részletezés jellemző.

A **hálós adatszerkezet** bármelyik csomópontja bármelyik csomóponttal kapcsolatban állhat (20.6. ábra). Ez az adatszerkezetek legbonyolultabb formája.

Hálós adatszerkezet az irányított gráf és a hálózat. A hálózat abban különbözik az irányított gráftól, hogy a kapcsolatokhoz valamelyen mérőszám is tartozik.



20.6. ábra. Hálós adatszerkezet

20.2. Absztrakt tárolók

Az adatszerkezet elméleti, logikai fogalom. Programozáskor az adatszerkezet elemeit le kell képezni a számítógép valamely külső vagy belső tárolójára, és hivatkozni kell rájuk: a kapcsolatok mentén be kell járni a csomópontokat. Az (a, b) rendezett párból az a -nak ismernie kell b -t. Az objektumok konkrét fizikai helyeken tárolódnak. Az egyik objektumból csak úgy lehet hivatkozni a másikra, ha az objektumokat tartalmazó fizikai tároló címezhető.

Nem címezhető tárolón csak szerkezet nélküli adathalmazokat tárolhatunk – ilyen esetben az egyetlen járható út a soros (fizikai sorrendben való) tárolás, illetve visszaolvasás.

A címezhető tárolókon két tárolási lehetőség kínálkozik:

- **egydimenziós tömb;**
- **láncolt lista;**

ezeket **absztrakt tárolóknak** (absztrakt társzerkezeteknek) nevezzük. Az absztrakt tároló lehet például a memóriában vagy lemezes egységen – bármilyen címezhető hardverelemen

Az adatszerkezetek leképezhetők absztrakt társzerkezetekre. A leképezés többszörös is lehet; a „végállomás” egy absztrakt társzerkezet.

Egydimenziós tömb

Az **egydimenziós tömb** olyan absztrakt tároló, amelyben indexeléssel közvetlenül címezhetők a tárolt objektumok (csomópontok, adatok).

A tömb fogalma kétféle értelemben is használatos: adatszerkezként és absztrakt tárolóként. A tömb mint adatszerkezet (az egy- és a többdimenziós is) leképezhető az egydimenziós tömbre mint absztrakt tárolóra.

Láncolt lista

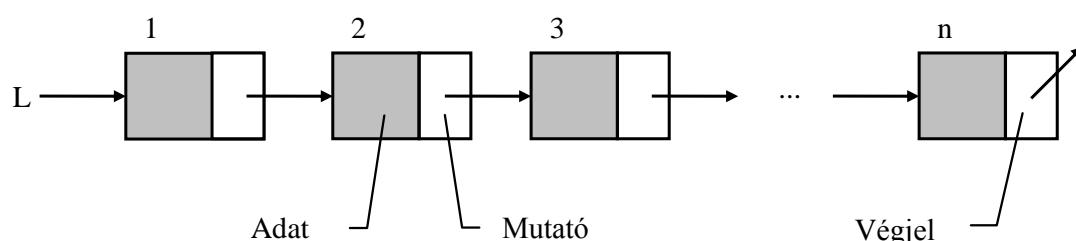
A **láncolt listára** jellemző, hogy a benne lévő adatelemek fel vannak fűzve, láncolva vannak (18.7. ábra): minden listaelem tartalmazza az öt követő elem mutatóját. A lista a mutatók révén elejtől a végéig bejárható. A listára új elemek kapcsolhatók, a már meglévők pedig lekapcsolhatók róla. A lista meghatározható az első vagy utolsó elemét kijelölő mutatóval (L). Az utolsó elem mutatója általában egy világosan felismerhető végjel; ez a végjel egy nem létező listaelemre mutat.

Egy listaelem két részből áll: **adatból** és **mutatóból**. A lista elemei fizikailag bárhol lehetnek, semmi sem köti meg a helyüket. Ábrázoláskor a logikailag egymást követő elemeket általában egymás mellé teszik.

A láncolt listának több változata is ismeretes:

- **Nyílt végű lista:** A lista utolsó vagy első eleme egy végjel, azt tehát nem követi listaelem. (20.7. ábra).
- **Cirkuláris (zárt) lista:** A lista utolsó eleme az első elemre mutat, vagyis az utolsó elem mutatója nem végjel. A cirkuláris lista mutatói körbeérnek.
- **Rendezett lista:** A lista elemei valamilyen szempont szerint jól meghatározott sorrendben követik egymást.
- **Kétirányú (szimmetrikus) lista:** A lista mutatói előre- és visszafelé mutatnak, s ezáltal az elemek előre és visszafelé is feldolgozhatók.
- **Fejelt lista:** A lista nem listaelemmel kezdődik, hanem egy fejjel (ez lehet üres), és abban egy mutató kijelöli az első (és/vagy utolsó) elemet.
- **Multilista (többszörös lista):** Olyan lista, amelynek az elemei újabb listák kiindulópontjai.

A láncolt lista nagy előnye, hogy egy listaelem beszúrásához, illetve törléséhez minden össze egy-két mutatót kell átállítanunk; a tömbben ugyanezeknek a műveleteknek az elvégzéséhez esetleg nagyon sok adatot kell megmozgatni.



20.7. ábra. Nyíltvégű láncolt lista

A lista programozásának érzékeltetésére most bemutatunk egy egyszerű láncolt listát:

Feladat –LinkedListTest

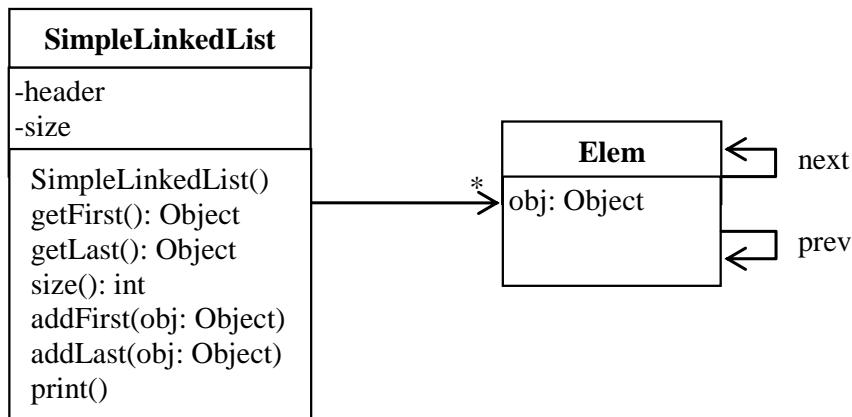
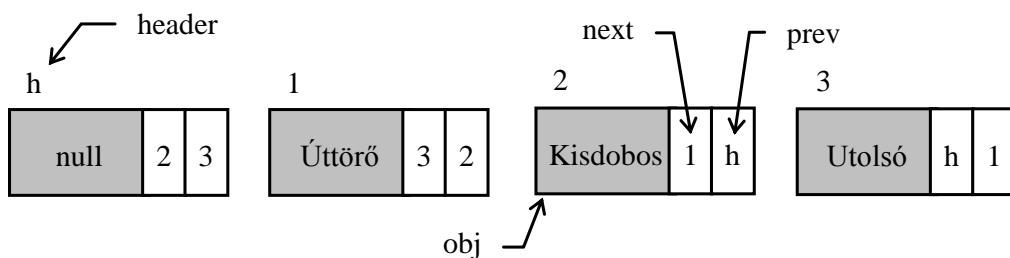
Készítsünk egyszerű fejelt, cirkuláris láncolt listát; legyenek meg benne a következő műveletek:

- beszúrás a lista elejére (addFirst)
- beszúrás a lista végére (addLast)
- az összes listaelem kiírása a konzolra (print)
- az első, illetve utolsó elem elkérése (getFirst, getLast)

Pédaként végezzük el a következő műveleteket, ilyen sorrendben:

```
list.addFirst("Úttörő");
list.addFirst("Kisdobos");
list.addLast("Utolsó");
list.addFirst("Első");
```

Először tehát betesszük a listába az "Úttörő"-t; ez lesz az első elem. Eléje beszúrjuk a "Kisdobos"-t, majd utolsóként betesszük az "Utolsó"-t. A 20.8. ábra mutatja, hogyan fest a lista a 3. művelet után. A sorszám és az elemek fizikai sorrendje azt tükrözi, hogy ez vagy az az elem hányadik lépésben került fel a listára; a mutatók már a logikai sorrendet adják meg: a next a következő elem sorszámát, a prev meg az előzőét. A lista feje (header) megmondja, hogy a logikailag első elem a 2-es sorszámú helyen van. A listaelemek láncolata tehát a 3. művelet után: "Kisdobos", "Úttörő", "Utolsó". Ezután az "Első" elemet kell majd beszúrnunk a lista elejére. Próbálja felvázolni azt az állapotot is!



20.8. ábra. Fejelt, cirkuláris láncolt lista

Forráskód

```

class SimpleLinkedList {
    // A listaelementet reprezentáló belső osztály:           //1
    class Elem {
        Object obj;
        Elem next, prev;

        public Elem(Object obj, Elem next, Elem prev) {
            this.obj = obj;
            this.next = next;
            this.prev = prev;
        }
    }
}

```

```
// LancoLista adatai, metódusai:  
private Elem header; //2  
private int size;  
  
public SimpleLinkedList() { //3  
    header = new Elem(null, null, null);  
    header.next = header.prev = header;  
    size = 0;  
}  
  
public Object getFirst() { //4  
    if (size == 0)  
        return null;  
    return header.next.obj;  
}  
  
public Object getLast() { //5  
    if (size == 0)  
        return null;  
    return header.prev.obj;  
}  
  
public int size() { //6  
    return size;  
}  
  
public void addFirst(Object obj) { //7  
    Elem newElem = new Elem(obj, header.next, header); //8  
    header.next.prev = newElem; //9  
    header.next = newElem; //10  
    size++; //11  
}  
  
public void addLast(Object obj) { //12  
    Elem newElem = new Elem(obj, header, header.prev); //13  
    header.prev.next = newElem; //14  
    header.prev = newElem; //15  
    size++; //16  
}  
  
public void print() { //17  
    Elem pointer = header.next;  
    while (pointer != header) {  
        System.out.println(pointer.obj);  
        pointer = pointer.next;  
    }  
}  
  
public class LinkedListTest {  
    public static void main(String[] args) { //18  
        SimpleLinkedList list = new SimpleLinkedList();  
        list.addFirst("Úttörő");  
        list.addFirst("Kisdobos");  
        list.addLast("Utolsó");  
        list.addFirst("Első");  
        list.print();  
    }  
}
```

```

        System.out.println("\nElső elem: "+list.getFirst());
        System.out.println("Utolsó elem: "+list.getLast());
    }
}

```

A forráskód elemzése

- ◆ //1: Az `Elem` osztály reprezentálja a listaelémeket. Tartalmazza magát a tárolt objektumot, valamint a következő (`next`) és előző (`previous`) objektum mutatóját. A konstruktur mindhárom értéket beállítja. A `SimpleLinkedList` osztályból bármely tulajdonság állítható, mivel az `Elem` osztály a `SimpleLinkedList` belső osztálya.
- ◆ //2: Fejelt listáról van szó – `header` a lista feje. `size` adja meg a lista elemeinek számát.
- ◆ //3: A konstruktur létrehozza a `headert`, vagyis a fejet, és beállítja a mutatóit. A fej következő és előző objektuma egyaránt maga a fej. A lista mérete kezdetben 0.
- ◆ //4: Ha a lista üres, akkor a `getFirst` metódus `null` értékkel tér vissza; egyébként a fejet követő listaelém objektumával (`header.next.obj`).
- ◆ //5: Ha a lista üres, akkor a `getLast` metódus `null` értékkel tér vissza; egyébként a fejet megelőző elem objektumával (`header.prev.obj`). Mivel a lista cirkuláris, azért az utolsó elem után a fej következik.
- ◆ //6: Visszaadja a lista aktuális méretét, vagyis a láncolt elemek számát (a fej nem számít közéjük!).
- ◆ //7: az `addFirst` metódus beszűr egy új elemet a lista elejére.
- ◆ //8: Először létrehozza az új elemet; annak objektuma a paraméterben megadott objektum lesz. `next` arra fog mutatni, amelyre eddig a `header` mutatott, `prev` értéke pedig a `header` lesz – ō lesz az első a sorban.
- ◆ //9: Az eddigi első elem előzője az új elem lesz.
- ◆ //10: A `header` rákövetkezője szintén az új elem lesz.
- ◆ //11: Eggyel megnöveljük a lista méretét.
- ◆ //12: az `addLast` metódus beszűr egy új elemet a lista végére.
- ◆ //13: Először létrehozza az új elemet úgy, hogy annak objektuma a paraméterben megadott lesz. `next` a fejre fog mutatni, `prev` pedig arra az eddigi utolsó elemre (`header.prev`) fog mutatni – ō lesz az utolsó a sorban.
- ◆ //14: Az eddigi utolsó elem rákövetkezője az új elem lesz.
- ◆ //15: A `header` előzőe szintén az új elem lesz.
- ◆ //16: Eggyel megnöveljük a lista méretét .
- ◆ //17: A `print` metódusban a pointer ciklusváltozóval sorban rámutatunk a lista elemeire, és egyenként kiírjuk őket a konzolra.
- ◆ //18: A `LinkedListTest` main metódusában létrehozzuk a `SimpleLinkedList` konténert, majd beledobáljuk a megadott szövegeket – hol a lista elejére, hol a végére. Végül kiírjuk a lista elemeit, majd az első és az utolsó elemet.

Megjegyzés: A láncolt listát a Javában a `java.util.LinkedList` osztály implementálja. Tanulmányozza az osztály forráskódját! A mi listánk egy kicsit más, és csak a legfontosabb metódusokat adtuk meg hozzá. Egészítse ki a `SimpleLinkedList` osztályt további metódusokkal!

A következő pontokban sorra vesszük az adatszerkezeteket.

20.3. Tömb

A tömb mint asszociatív adatszerkezet lehet **általános tömb** és **ritka mátrix**. Mindkettő lehet egy-, két-, illetve többdimenziós.

Általános tömb

Az **általános tömb** lényege az, hogy elemeit indexeken keresztül érjük el; ha a tömb többdimenziós, akkor az indexelést több „irányból” végezzük. A tömbök használatának az a hátránya, hogy a méretüket előre meg kell adni, és dimenzióinként meg vannak kötve az indexhatárok. Az általános tömb adatszerkezetet az absztrakt tárolók közül az egydimenziós tömbre képezzük le. Ha az adatszerkezet egydimenziós tömb, akkor az absztrakt tároló általában maga a tömb; indexelcsúszás lehetséges. A leképezést a programozási nyelv forratója végzi el.

Az általános tömbökről részletesen szó volt az I. kötetben.

Ritka mátrix

A ritka mátrixokat azért kell külön említenünk, mert azokat másként tároljuk, mint az általános tömböket.

Ritka mátrixnak nevezzük az olyan tömböt, amelynek a nagy része „kihasználatlan”.

Például:

	1.	2.	3.	4.	5.	6.	7.	8.	9.
1.	0	0	0	0	0	3	0	0	0
2.	0	21	0	0	0	0	0	0	0
3.	0	0	0	0	0	0	0	0	0
4.	0	0	99	0	0	0	0	7	0
5.	0	0	0	0	0	0	0	0	0

A ritka mátrixot rendkívül gazdaságtalan lenne a tömb absztrakt tárolóra leképezni – erre a célra a lista a jó tárolóeszköz; arra felfűzhetjük a „ritka adatokat”.

20.4. Tábla

A **tábla** asszociatív adatszerkezet; kulcsból és adatból álló párok az elemei. A kulcsok egyediek, és minden adat a kulcsán keresztül érhető el (20.9. ábra). A tábla leképezhető egy vagy több egydimenziós tömbre vagy láncolt listára.

A táblával kapcsolatos műveletekben központi kérdés a valamely kulcshoz tartozó adat minél gyorsabb megkeresése, és a tábla karbantartása. A táblához mint adatszerkezethez rendszerint a következők a kérések:

- ◆ Jegyezd meg ezt a kulcsot és a hozzá tartozó adatot!
- ◆ Kérém az ehhez a kulesshoz tartozó adatokat!
- ◆ Töröld ki azt az adatot, amelynek ez és ez a kulcsa!
- ◆ Módosítsd azt az adatot, amelynek ez és ez a kulcsa!
- ◆ Sorold fel az összes kulcsodat, hadd válasszak közülük!

adat	adat	adat	adat	adat	adat	adat
kulcs	kulcs	kulcs	kulcs	kulcs	kulcs	kulcs
Kérém azt az adatot, melynek kulcsa ...						



20.9. ábra. Tábla

A táblaműveletek hatékonysága erősen függ a tábla szervezésétől, vagyis az adatszerkezet(ek) megválasztásától és a konkrét megvalósítástól. Szinte minden műveletben van keresés: nem vihetünk be például olyan adatot, amelynek a kulcsa már szerepel a táblában. Ha a felhasználó nem bizonyosodik meg erről, akkor a táblának kell: vissza kell utasítania a duplikált kulcsot. Egy jó táblát elrontani nem lehet, legfeljebb értelmetlen adatokkal telepapolni.

A kulcsok és adatok tárolási módja szerint többféle táblát különböztetünk meg: soros tábla, önárendező tábla, rendezett tábla és hasítótábla. A következőkben ezeket vesszük sorra.

Soros tábla

Ez a táblák legegyszerűbb formája. A soros táblában a kulcsokat és az adatokat a felvitel sorrendjében tároljuk. A soros tábla leképezhető egydimenziós tömbre. A soros táblák feltűnően

jó hatásfokkal működnek, ha kicsi az elemszám. A táblában a rendezetlenség miatt nem lehet binárisan keresni, viszont a lehető legegyszerűbb a beszúrás, illetve a törlés. Olyankor célszerű használni, ha nem túl nagy az elemszám és nincs szükség rendezettségre.

Önárendező tábla

Ha a kulcsok közül bizonyosakat sokkal többször keresünk, mint a többit, akkor összességében sokkal hatékonyabb lesz a keresés, ha ezek a kulcsok a tábla elején vannak. Elég nehéz lenne azonban statisztikát vezetni a kulcskeresésről és annak megfelelő módszert kidolgozni a tábla folyamatos átrendezésére. Sokkal jobb ötlet a keresett kulcsot mindig a tábla elejére tenni, hiszen így a ritkán vagy sohasem keresett kulcsok előbb-utóbb a tábla végére szorulnak. Ha a táblát tömbre képezzük le, akkor persze nem sokat nyerünk a sok adatmozgatás miatt, de ha listában tároljuk, akkor elég a lista elejére át kapcsolunk a keresett adatot.

Rendezett tábla

A rendezett táblában a kulcsoknak van valamilyen meghatározott sorrendjük, és az adat–kulcs párok e szerint a sorrend szerint vannak rendezve. Rendezett táblában sokkal gyorsabban lehet keresni, mint rendezetlenben, a karbantartás azonban általában nehézkes, mert folyamatosan fenn kell tartani a rendezettséget.

Hasító (kulcstranszformációs) tábla

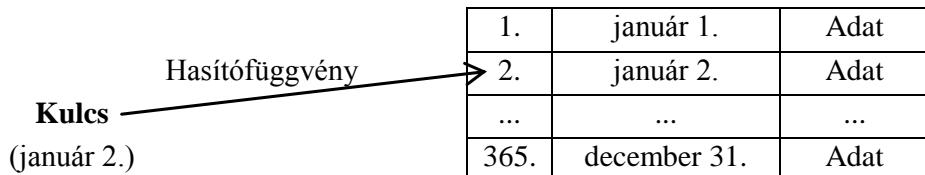
A táblákban valamilyen algoritmussal határozható meg az elemek helye. Soros táblában a beszúrás időpontja számít, rendezett táblában a kulcs „nagysága”.

A **hasítótáblában** (hash table) egy **hasítófüggvény** (hasítókód, transzformáció) adja meg a kulcsnak tartozó elem helyét. A hasítási technikának az a nehézsége, hogy több kulcsnak is tartozhat ugyanaz a hasítókód; ilyenkor több kulcs ugyanabba a csoportba kerül, csoportindexük ugyanaz. A hasítási technikával azonban közvetlenül érhetjük el a kulcsokat (s rajtuk keresztül adatokat), minimálisra csökkentve a hasonlítások számát.

A csoportok száma a hasítótábla kapacitása. A hasítófüggvény meghatározásában figyelembe kell venni a következőket:

- az elemszám (a táblába betett kulcsok száma) kevesebb legyen, mint a kapacitás;
- a kulcsok egyenletesen terüljenek szét;
- egy-egy asszociatív csoportban csak kevés számú elem legyen!

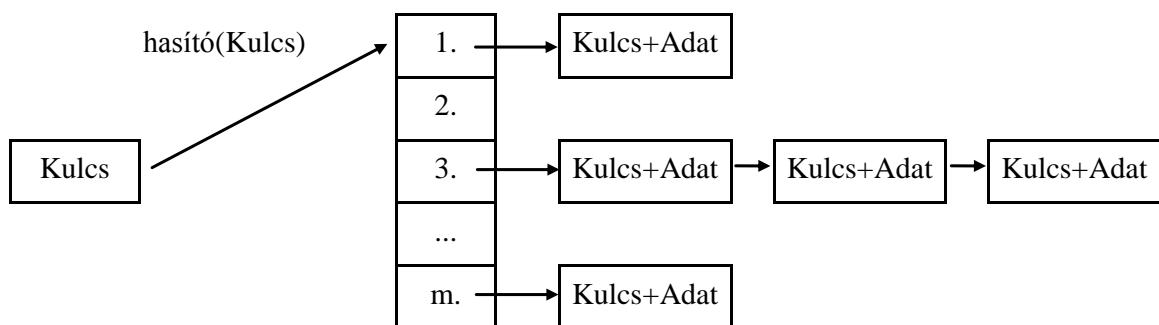
Előfordulhat, hogy a kulcs és a csoportindex között kölcsönösen egyértelmű megfeleltetést találunk; ekkor egy csoportban csak egy elem lesz. Ha például a kulcs a (hónap, nap) pár, akkor jó hasítófüggvény a dátum évbeli – 1 és 365 közé eső – sorszáma (20.10. ábra). A kulcsok értékei azonban általában nem ilyen szabályosak.



20.10. ábra. Hasítótába egyértelmű leképezéssel

Szinonimák

A hasítófüggvények általában visszafelé nem egyértelműek: előfordulhat, hogy két kulcsnak ugyanaz lesz a csoportindexe. Az azonos helyre leképezett kulcsokat szinonimáknak nevezzük, és azok elhelyezéséről külön kell gondoskodni. A szinonimaelhelyező algoritmust vagy függvényt külön meg kell határozni. Szokás a szinonimákat egy külön túlcordulási területre elhelyezni – fontos, hogy az elhelyezésük után könnyen felderíthetők legyenek. Jó megoldás, ha a csoportban egy lista tárolja a szinonimákat (20.11. ábra).



20.11. ábra. Hasítótábla szinonimákkal

Ha a kulcsok egész számok, akkor hasítófüggvények például a következők lehetnek:

- ◆ Maradékképzés: Ha a kulcsok száma n, melyek aránylag egyenletesen oszlanak szét, és a csoportindexek 0 és m-1 közé esnek, akkor a hasítófüggvény:

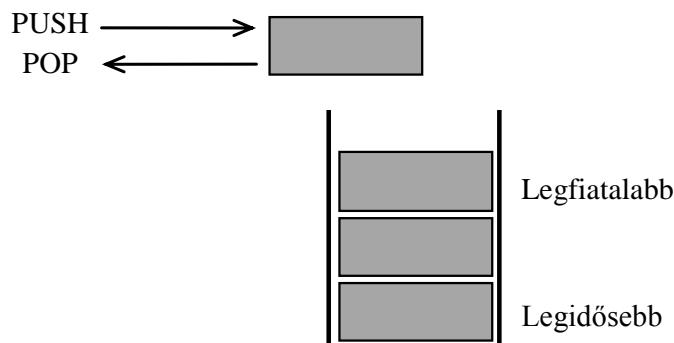
$$\text{hasító(kulcs)} = \text{kulcs \% m}$$

Ez a függvény prímszám méretű tábla esetén a leggazdaságosabb.
- ◆ Ha a kulcs egy egész szám: A kulcsot összeszorozzuk önmagával, és vesszük az eredmény középső jegyeit.

20.5. Verem

A **verem** (*stack*) olyan szekvenciális adatszerkezet, amelyből minden csatoljára betett elemet lehet kivenni (20.12. ábra). A verem szokásos elnevezése még: LIFO (Last In First Out) szerkezet. A verem műveletei:

- PUSH: Elem betétele a verembe – mindenkor a tetejére
- POP: Elem kivétele a veremből – mindenkor a tetejéről
- TOP: A legfelső elem lekérdezése – a verem változatlan marad



20.12. ábra. Verem

A verem tárolása

A verem egydimenziós tömbben és listában is tárolható. Ha a tömböt választjuk, akkor előre meg kell határoznunk a maximális elemszámot. Kell egy mutató, hogy a verem mindenkor tetejére, vagyis az első szabad helyre mutasson. Egy veremről mindenkor tudnunk kell, hogy üres-e vagy tele van, hiszen üres veremből nem lehet elemet kivenni, tele verembe pedig nem lehet betenni.

A verem alkalmazásai

Verem segítségével könnyedén megfordíthatjuk az elemek sorrendjét: az egyik veremről levett elemet betesszük egy másik verembe. A verem jól használható különböző visszatérési utak megjegyzésére is: odafelé sorba beletesszük az összes érintett város nevét, visszafelé a verem legfelső eleme szerinti város felé haladunk, s a levett nevet minden eldobjuk. Menjünk le például Budapestről Pécsre a következő útvonalon:

```

Push (Budapest) ;
Push (Székesfehérvár) ;
Push (Fonyód) ;
Push (Kaposvár) ;
Push (Pécs) ;

```

Visszafelé a Pop(Város) ismételt végrehajtásával visszajönnek az érintett városok, és ugyanazon az útvonalon hazatalálunk.

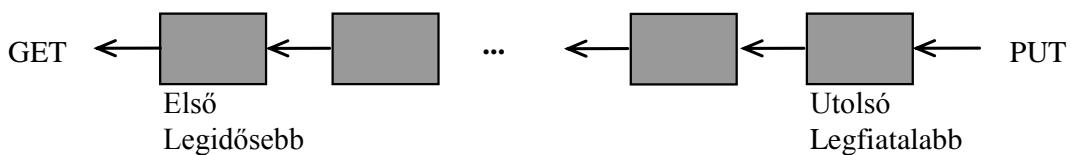
A metódusok hívásakor a program memóriavermet használ a visszatérési címek meghatározásához.

Megjegyzés: A verem jól használható visszaléptetéses algoritmusokban és veremautomaták alkalmazásakor is.

20.6. Sor

A **sor** (**queue**) olyan szekvenciális adatszerkezet, amelyből mindenkor legelsőnek betett elemet lehet kivenni. Angol szóhasználat szerint a sor FIFO (First In First Out) adatszerkezet (20.13. ábra). A sor műveletei:

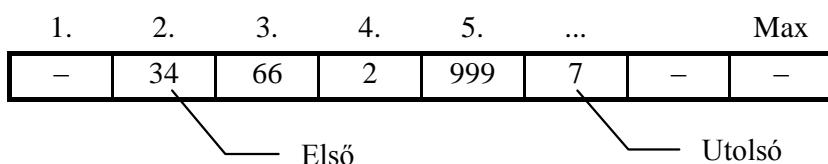
- PUT: Elem betétele a sorba – mindenkor a sor végére
 - GET: Elem eltávolítása a sorból – mindenkor a sor elejéről
 - FIRST: Az első elem lekérdezése – a sor változatlan marad



20.13. ábra. Sor

A sor tárolása

A sort egydimenziós tömbben és listában is tárolhatjuk. Vegyük a tömb esetét, és tegyük fel, hogy a tömböt az első helytől fogva elkezdjük tölteni. A sor a tömb vége felől bővül, az elejéről meg fogy. Ha az utolsó elem indexe elérte a maximumot, akkor kénytelenek vagyunk a tárolást a tömb elején folytatni (hacsak nem akarjuk a sor összes elemét minden új elem betéttele után lejjebb csúsztatni). Ezt a tárolást **ciklikus tárolásnak** nevezzük:



A sor alkalmazásai

Sorokkal általában olyan feladatokat oldunk meg, amelyekben az elemeket az érkezés sorrendjében kell feldolgozni. Az adminisztrátornak például abban a sorrendben kell elintéznie az aktákat, amelyben azok beérkeztek.

Sorként működnek a pufferek, például a billentyűzet- és a nyomtatópuffer. A nyomtatóra kiküldött karakterek nem állhatnak ki a sorból, először minden az előbb kiküldött karakter, dokumentum nyomtatódik ki.

20.7. Fa

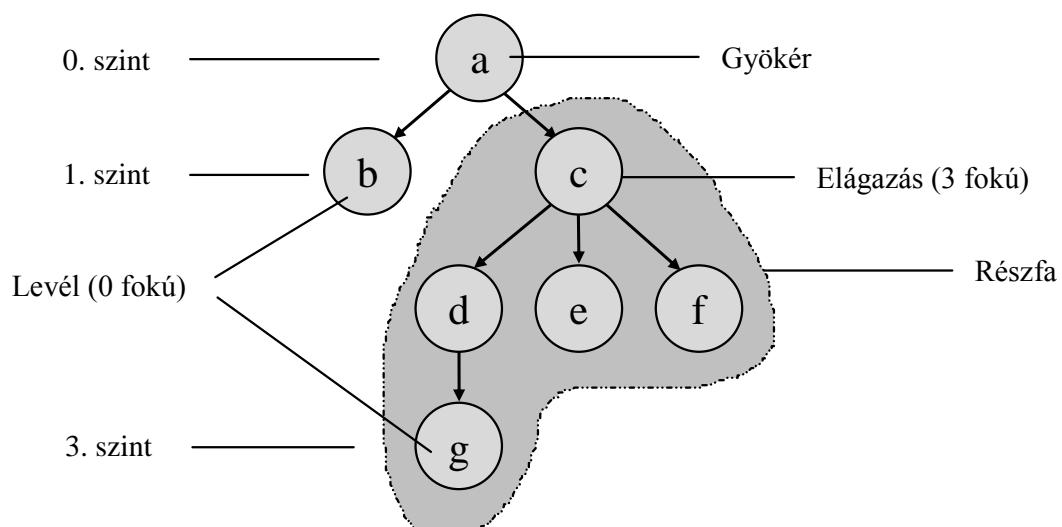
A **fa** olyan hierarchikus adatszerkezet (20.14. ábra), amely véges számú csomópontból áll és teljesíti a következő követelményeket:

- Két csomópont közötti kapcsolatban az egyik a kezdőpont, másik a végpont.
- Van a fának egy kitüntetett csomópontja, s az semelyik kapcsolatnak sem lehet a végpontja; ez a csomópont a fa gyökere.
- minden más csomópont pontosan egy kapcsolatban végpont.

Ez a fa meghatározása. A fának azonban van egy rekurzív **definíciója** is. A fa vagy üres, vagy:

- A fának van egy kitüntetett csomópontja, ez a gyökér.
- A gyökérhez 0 vagy több diszjunkt fa kapcsolódik. Ezek a gyökérhez tartozó részfák.

A fával kapcsolatos algoritmusok általában rekurzívak.



20.14. ábra. Fa

További definíciók:

- **Csomópont foka:** a csomóponthoz kapcsolt részfák száma
- **Fa foka:** A fában található legnagyobb fokszámú csomópont foka
- **Levél:** 0 fokú csomópont
- **Elágazás (átmenő csomópont):** > 0 fokú csomópont
- **Szülő (ős):** kapcsolat kezdőpontja (csak a levelek nem szülők)
- **Gyerek (leszármazott):** kapcsolat végpontja (csak a gyökér nem gyerek)
- **Szintszám:** a gyökértől mért távolság. A gyökér szintszáma 0. Ha egy csomópont szintszáma n , akkor a hozzá kapcsolódó csomópontok szintszáma $n + 1$.
- **Fa magassága:** a levelekhez vezető utak közül a leghosszabb, vagyis a maximális szintszám.
- **Rendezett fa:** ha az egy szülőhöz tartozó részfák sorrendje lényeges, vagyis azok rendezettek.
- **Kiegyensúlyozott fa:** olyan fa, melynek csomópontjai azonos fokúak; csak az utolsó szinten hiányozhatnak gyerekek. Így minden levél az utolsó két szint valamelyikén van. Megadott számú csomópont esetén az ilyen fának legkisebb a magassága.

Egy f fokú, m magasságú fában maximum $f^0 + f^1 + f^2 + \dots + f^m = (f^{m+1} - 1) / (f - 1)$ csomópont helyezhető el. A 20.14. ábrán szereplő fa kiegyensúlyozatlan, magassága 3, foka 3. A 3 magasságú, 3 fokú fára 40 csomópontot, a 3 magasságú, 2 fokú fára pedig 15 csomópontot lehetne „felenni”.

A fa bejárása

A fa csomópontjaiban adatokat, objektumokat tárolunk, és valamilyen rendszer szerint el is szeretnénk őket érni. Kereshetünk például megadott tulajdonságú objektumokat, vagy, mondjuk, ki szeretnénk listázni a fa összes objektumát. Egy általános fát lényegében a következő startégiák szerint járhatunk be:

- ◆ **Gyökérkezdő (preorder):** gyökér, majd a részfák bejárása sorban, például balról jobbra. A 20.14 ábrán szereplő általános fa gyökérkezdő bejárása:

$$\begin{array}{ccccccc} & & & & c & & \\ & & & & | & & \\ & & & & a & & \\ & & & & | & & \\ & & & & b & & \\ & & & & | & & \\ & & & & d & & \\ & & & & | & & \\ & & & & e & & \\ & & & & | & & \\ & & & & f & & \end{array}$$
- ◆ **Gyökérvégző (postorder):** részfák bejárása sorban, majd a gyökér. Az előző fa gyökér-végző bejárása lépésekkel kifejtve:

$$\begin{array}{ccccccc} & & & & c & & \\ & & & & | & & \\ & & & & a & & \\ & & & & | & & \\ & & & & b & & \\ & & & & | & & \\ & & & & d & & \\ & & & & | & & \\ & & & & e & & \\ & & & & | & & \\ & & & & f & & \\ & & & & | & & \\ & & & & c & & \\ & & & & | & & \\ & & & & a & & \end{array}$$
- ◆ **Gyökérközepű (inorder):** olyan fát érdemes így bejárni, amelyen egyértelműen eloszthatók a gyerekek. (Például a bináris fát; lásd következő cím alatt).

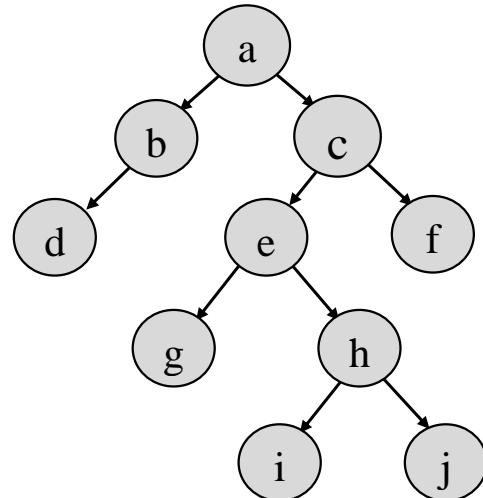
Bináris fa

A **bináris fa** gyökeréből legfeljebb két részfa ágazik: egy bal, és egy jobb oldali részfa. A bináris fa foka 2.

A bináris fa bejárása

Bináris fát a gyökérközepű stratégiával is be lehet járni, mert a szülő a gyerekek között helyezkedik el. A 20.15. ábrán látható bináris fa bejárásai tehát:

- ◆ **Gyökérkezdő (preorder):** gyökér, bal részfa, jobb részfa:
a b d c e g h i j f
- ◆ **Gyökérközepű (inorder):** bal részfa, gyökér, jobb részfa:
d b a g e i h j c f
- ◆ **Gyökérvégző (postorder):** bal részfa, jobb részfa, gyökér:
d b g i j h e f c a



20.15. ábra. A bináris fa bejárása

Rendezett bináris fa

Az adatok rendezésére, illetve keresésére használt fát **keresési fának** (search tree, kereső fa, rendező fa) szokás nevezni. Ilyen a **rendezett bináris fa**:

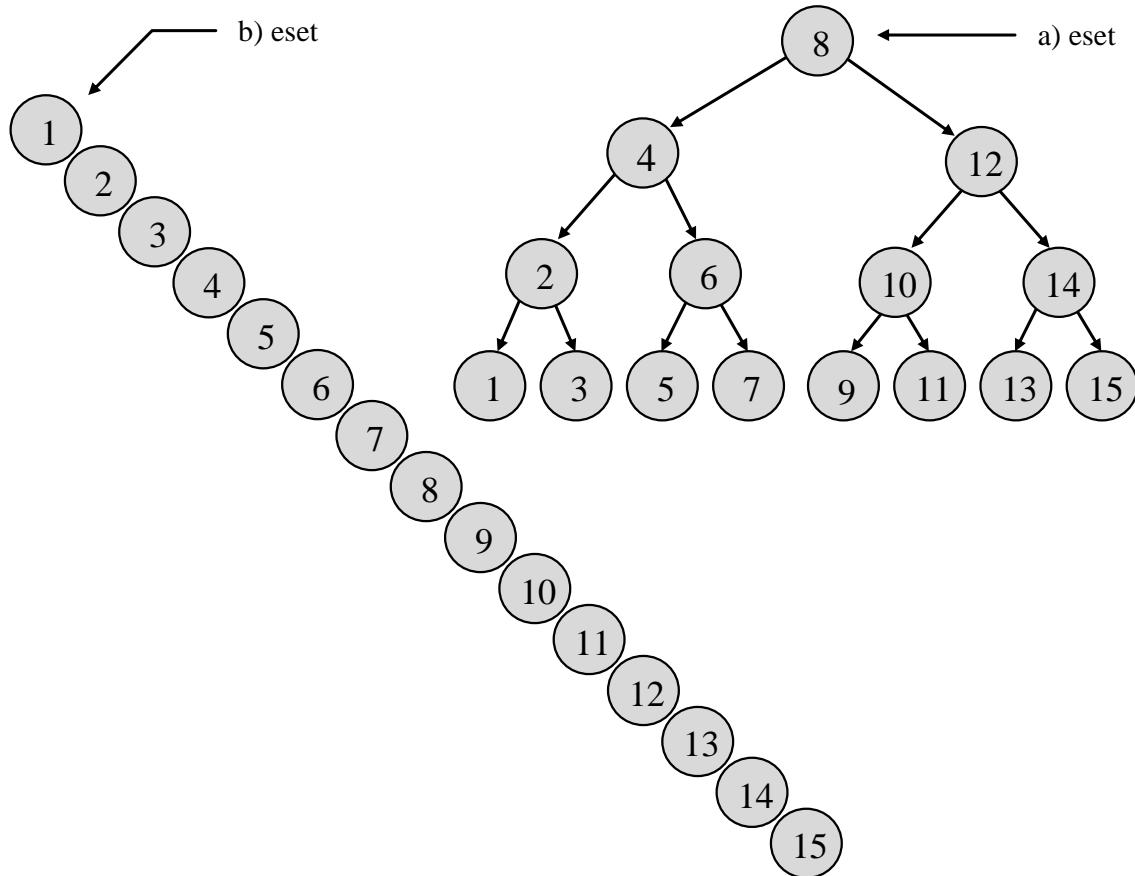
- a bal oldali részfa összes eleme kisebb, mint a szülő;
- a jobb oldali részfa összes eleme nagyobb, mint a szülő;
- egy n szintű fa elemeinek száma maximum $2^{n+1} - 1$;
- egy elemet maximum $n + 1$ lépésben megtalálunk.

Rendezett bináris fába például a következő algoritmus szerint szoktak **elemet beszúrni**: Elemet csak levélként lehet beszúrni. Az első elem a fa gyökere lesz. A későbbi elemeknek a gyökértől indulva keressük meg a helyét: ha az elem kisebb (vagy egyenlő), mint a csomópont, akkor balra megyünk, egyébként jobbra. Ha a kijelölt út nem folytatódik tovább, akkor egy újabb élet kiépítve az elemet levélként feltesszük a fára. Az elemkeresés ugyanígy megy. A keresendő elemet a fa csomópontjaihoz hasonlígtatva egyértelmű útvonalon haladunk lefelé. Ha ezen az útvonalon nem találjuk meg az elemet, akkor az nincs is rajta a fán.

Példaként építünk fel egy számokat tartalmazó bináris fát. A fára „akasztandó” számok legyenek az 1 és 15 közé eső számok. Az elemek beszúrási sorrendje legyen a következő:

- a) 8, 4, 12, 10, 9, 14, 2, 3, 6, 1, 15, 5, 7, 13, 11
- b) 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

Az eredmények a 20.16. ábrán láthatóak.

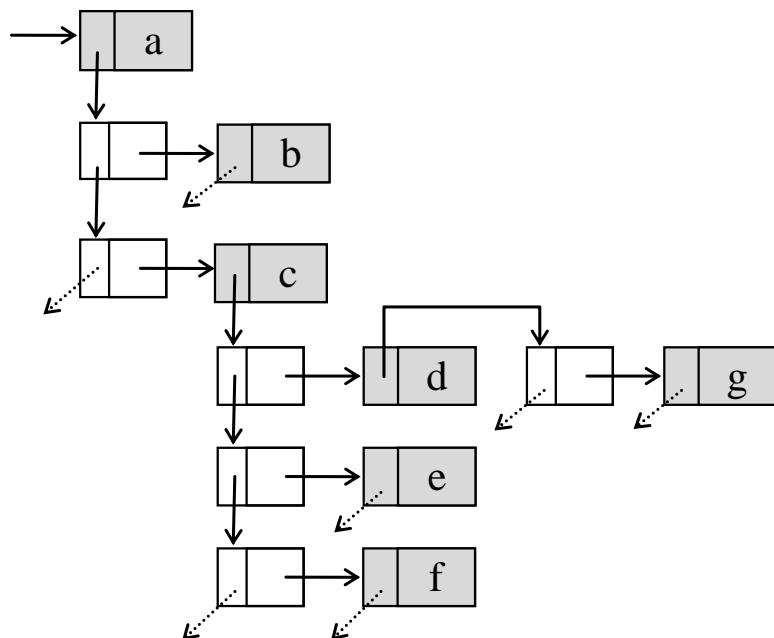


20.16. ábra. Kiegyensúlyozott és degenerált bináris fa

Talán észrevette az Olvasó, hogy az a) eset szép, **kiegyensúlyozott fája** nem egészen magától lett ilyen – a trükk az volt, hogy a beszúrandó számokat megfelelő sorrendbe állítottuk. b) esetben a számok teljesen rendezetlenül jöttek egymás után, ezért a fa igencsak torz lett, egészen lineárisára fajult. Az ilyen fát **degenerált fának** is nevezik. Látható tehát, hogy a fa alakja „szerencsétől függ”: a fa magassága optimális esetben 3, elfajlt esetben 14. A degenerált fában még annál is rosszabb hatásfokkal kereshetünk, mint egy lineáris listában, hiszen több hasonlítást végezünk, reménykedve az irányváltásban. A kiegyensúlyozott fában viszont legfeljebb 4 lépésekben megvan a keresett elem vagy kiderül, hogy az nincs az elemek között. Általánosítva, ha N az elemek száma, akkor kiegyensúlyozott fában a maximális lépések száma nagyságrendileg $\log_2 N$. A szemléletesség kedvéért: ha az elemek száma 1 milliárd, akkor kiegyensúlyozott fában 30 lépés elég bármelyik elem megtalálásához ($2^{30} = 1\ 073\ 741\ 824$), a lineáris listában viszont átlagosan 0,5 milliárd lépésre van szükség.

A fa tárolása

Egy általános fát legegyszerűbben multilistában tárolhatunk (20.17. ábra). minden csomóponthoz egy lista tartozik; annak első eleme az adat (az ábrán szürke), a többi pedig a kapcsolat (az ábrán fehér); annyi kapcsolati elem van, ahány fokú a csomópont. A kapcsolatok újabb csomópontokra, illetve listákra mutatnak. A 20.17. ábra multilistája a 20.14. ábrán levő fát tárolja. A bináris fa tárolása mindennek egy speciális esete: a bináris fa multilistájában minden csomópontnak két mutatója van: az egyik a bal oldali, a másik a jobb oldali részfára mutat.



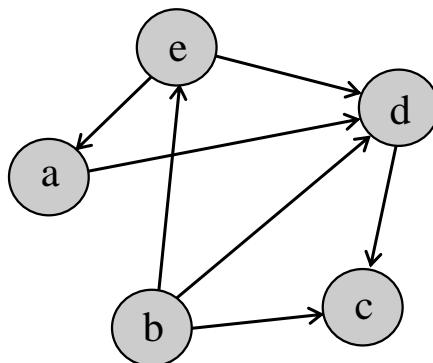
20.17. ábra. Fa tárolása multilistában

20.8. Irányított gráf, hálózat

A hálós adatszerkezetben bármely két csomópont kapcsolatban állhat egymással. Ha csupán a kapcsolat léte és iránya lényeges, akkor **irányított gráfról** beszélünk, s ha a kapcsolatokhoz valamilyen mérőszám is tartozik, akkor **hálózatról**.

Egy irányított gráfot, illetve hálózatot megadhatunk a csomópontok és a kapcsolatok felsorolásával (ha hálózatról van szó, akkor a mérőszámot is megadjuk). A 20.18. ábrán egy irányított gráfot adunk meg grafikusan és szövegesen. A kapcsolatokat valamelyen rendszer szerint kell felsorolni – példánkban a kapcsolatok kezdőpont és azon belül végpont szerint ren-

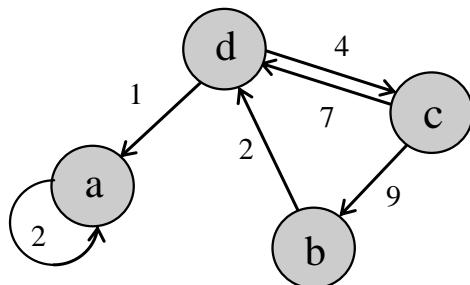
dezve. A 20.19. ábrán a hálózat kapcsolataihoz egész értékek tartoznak; ezeket az értékeket a gráf éleire írtunk, illetve a kapcsolatok utolsó adataiként adtunk meg.



Csomópontok: (a,b,c,d,e)

Kapcsolatok: (a,d),(b,c),(b,d),(b,e),(d,c),(e,a),(e,d)

20.18. ábra. Irányított gráf



Csomópontok: (a,b,c,d)

Kapcsolatok: (a,a,2),(b,d,2),(c,b,9),(c,d,7),(d,a,1),(d,c,4)

20.19. ábra. Hálózat

Nézzünk néhány példát a hálós adatmodellre:

- ◆ Adva van n falu. Tudjuk, hogy melyik faluból melyikbe jár autóbusz. Az autóbuszjárat lehet egyirányú is. Kérdés, hogy valamely adott faluból el lehet-e jutni autóbusszal egy másikba? Az ilyen jellegű feladatokban csak az számít, hogy van-e busz vagy nincs. Hogy mennyit kell utaznunk a célig, az most nem számít. Ez az adatmodell egy **irányított gráf**.

- ◆ A körülmények ugyanazok, mint az előző feladatban, de most nem elégünk meg akár milyen hosszú utazással. Arra vagyunk kíváncsiak, létezik-e az egyik faluból a másikba egy megadott útnál rövidebb út, illetve melyik út a legrövidebb a két falu között. Ebben az esetben már **hálózatról** beszélünk, hiszen a falvak közti úthosszakat is számításba kell venni.
- ◆ Van egy nőkből és férfiakból álló társaság. Az a feladat, hogy a társaság tagjait minél nagyobb létszámban össze kell házasítani. mindenki megmondja, hogy kiket szeret. Magát senki sem szeretheti. Ha két egyén között mindenki irányban létezik kapcsolat, akkor a pár összeadható. Ez az adatmodell egy **irányított gráf**.
- ◆ minden ugyanaz, mint az előző feladatban, de a társaság tagjai közül nem mindenki szeret ugyanolyan hévvel. A szeretetet 0-tól 5-ig lehet osztályozni. A feladat: úgy társíti a társaság tagjait, hogy az „össz-szeretet” a lehető legnagyobb legyen. A társaság tagjai **hálózatot** alkotnak.

Definíciók

- ◆ Egy hálós adatmodellben *útnak* nevezzük az a_0, a_1, \dots, a_m csomópontok sorozatát, ha az $(a_0, a_1), (a_1, a_2), \dots, (a_{m-1}, a_m)$

rendezett kapcsolatok fennállnak. Ekkor az út hossza m , kezdőpontja az a_0 , végpontja pedig az a_m csomópont. A 20.18. ábrán a b, e, d, és c csomópontok egy 3 hosszúságú utat határoznak meg. Egy útban egy csomópont többször is szerepelhet.

- ◆ Ha a hálós adatmodellben létezik olyan út, amelyen egy csomópontot legalább kétszer rajta van, akkor az a modell **ciklikus**, más szóval **hurkot tartalmazó**, különben **ciklus nélküli**, más szóval **hurokmentes**. A 20.18. ábra irányított gráfja hurokmentes, a 20.19. ábra hálózata viszont ciklikus, hiszen a d, c, b, d, c, b, d, c, b, ... útvonalon például a végtelenségig „járhatnánk”.

Hálós adatszerkezet tárolása

Ha a hálós adatszerkezetnek n csomópontja van, akkor a csomóponti adatok megadhatók egy n elemű vektorral, a kapcsolatok pedig egy $n \times n$ elemű mátrixszal; ha irányított gráfról van szó, akkor a mátrix a csomópontok „keresztezésébe” egy igen/nem adatot tartalmaz, ha hálózatról van szó, akkor egy mérőszámot.

Az irányított gráf kapcsolatainak mátrixa **szomszédossági mátrix**. Ez egy $n \times n$ logikai érték alkotta Boole-mátrix. Az (igen/nem) értékeket legtöbbször a 0 és az 1 reprezentálja.

A hálózat kapcsolatait leíró mátrixot **súlymátrixnak** nevezik; a mátrix elemei a kapcsolatokhoz tartozó értékek (súlyok).

A 20.18. ábra öt falu (a, b, c, d és e) közötti buszjáratokat szemléltet. A falvak közti kapcsolatokat a következő szomszédossági mátrix írja le:

	a	b	c	d	e
a	0	0	0	1	0
b	0	0	1	1	1
c	0	0	0	0	0
d	0	0	1	0	0
e	1	0	0	1	0

Tesztkérdések

- 20.1. Jelölje be az összes helyes állítást!
- a) Az adatszerkezetek gráfokkal ábrázolhatók.
 - b) Az elemek közötti kapcsolatok az asszociatív adatszerkezetekben a legerősebbek.
 - c) Hierarchikus adatszerkezetben az elemek közötti kapcsolatok sok-sok típusúak.
 - d) Hálós adatszerkezetben az elemek közötti kapcsolatok egy-sok típusúak.
- 20.2. Jelölje be az összes helyes állítást!
- a) A tömb szekvenciális adatszerkezet.
 - b) A verem szekvenciális adatszerkezet.
 - c) A fa asszociatív adatszerkezet.
 - d) Az irányított gráf hierarchikus adatszerkezet.
- 20.3. Jelölje be az összes helyes állítást!
- a) Nem címezhető tárolón csak sorosan (fizikai sorrendben) tárolhatók az adatok.
 - b) A láncolt lista olyan absztrakt tároló, amelyben a tárolt objektumok fizikailag egymás után helyezkednek el.
 - c) A láncolt listából egyszerűbb listaelementet törölni, mint egy egydimenziós tömbből.
 - d) A kétirányú láncolt listában minden elem tartalmazza az előző és következő elem mutatóját.
- 20.4. Jelölje be az összes helyes állítást!
- a) A ritka mátrixot egydimenziós tömbre szokás leképezni.
 - b) A tábla csak láncolt listára képezhető le.
 - c) A soros táblát akkor ajánlatos használni, ha nagy az elemszám.
 - d) A legjobb táblaszervezési megoldás az olyan hasítótábla, amelyben a kulcsok és a fizikai címek között kölcsönösen egyértelmű a leképezés.
- 20.5. Jelölje be az összes helyes állítást!
- a) A veremből minden a legelsőnek betett elemet lehet kivenni.
 - b) A nyomtatópuffer sorként működik.
 - c) A fának lehet két gyökere is.
 - d) A rendezett bináris fát adatok rendezésére és keresésére lehet használni.

Feladatok

- 20.1. **(B)** Készítsen egy `LinkedList` osztályt, s abban a következő metódusokkal egészítse ki a `SimpleLinkedList` osztály metódusait:
 - a) Visszaadja a megadott elemre következő, illetve az azt megelőző elemet. A lista vége után null értéket adjon vissza!
 - b) Visszaadja a megadott indexű elemet; az indexelés a láncolás sorrendjét kövesse.
 - c) Kitöröl a listából egy megadott indexű elemet.
(`LinkedList.jpx`)
- 20.2. Készítsen egy, a verem működését szimuláló `Stack` osztályt! (`Stack.jpx`)
 - a) **(B)** Az absztrakt tároló tömb legyen!
 - b) **(B)** Az absztrakt tároló láncolt lista legyen!
- 20.3. Készítsen egy, a sor működését szimuláló `Queue` osztályt! (`Queue.jpx`)
 - a) **(B)** Az absztrakt tároló tömb legyen!
 - b) **(B)** Az absztrakt tároló láncolt lista legyen!
- 20.4. **(C)** Készítse el a rendezett bináris fa működését szimuláló osztályt! Tegyen a fába számokat, majd jelenítse meg a fát! Ezután keressen megadott számokat a fán! Keresés-kor minden írja ki a lépések számát! (`RendBinFa.jpx`)

21. Kollekció keretrendszer

A fejezet pontjai:

1. A kollekció keretrendszer felépítése
 2. A Collection interfész és leszármazottai
 3. A HashSet osztály – hasítási technika
 4. A TreeSet osztály – Comparator
 5. Iterátor
 6. A List interfész implementációi
 7. A Map és a SortedMap interfész
 8. A Hashtable osztály
 9. A TreeMap osztály
-

A Java Kollekció keretrendszer (Java Collections Framework) egy interfészekből és kész osztályokból álló kollekció-, illetve konténergyűjtemény. Az interfések – a `Set`, a `SortedSet`, a `List` stb. – definiálják a kollekciók viselkedését és szabályrendszerét. Az interfésekkel osztályok implementálják – a `Set` interfészt például a `HashSet` osztály, a `List` interfészt a `Vector` stb. Az implementáló osztályok a hatékonyság érdekében lépten-nyomon használják a különböző adatszerkezeteket, például a hasítótáblát és a fát. Az egységes használat elősegítésére a keretrendszer tervezői az interfésekkel jól elkülönítették az implementációktól.

Ez a fejezet a kollekció keretrendszer szerkezetének és lehetőségeinek bemutatására szolgál. Egy programozónak gyakran lehet szüksége különböző konténerekre, egyszer olyanra, amely rendezetten tárolja az elemeket, másszor olyanra, amely gyorsan raktározza vagy gyorsan éri el őket. Más konténert kell használnia például akkor, ha a feladat megkívánja az elemek egyediségét, és mászt, ha nem.

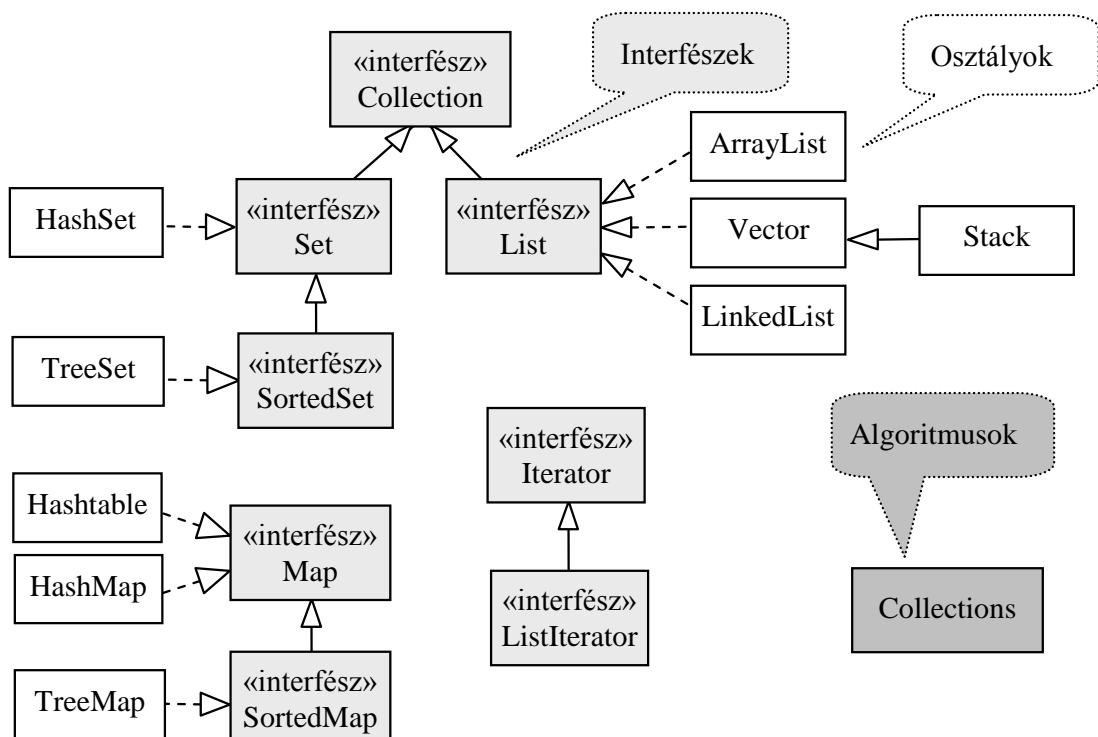
A kollekció keretrendszer minden olyan konténertípust tartalmaz, amelyre egy programozónak szüksége lehet egy átlagos program elkészítésében. Fontos, hogy a programozó jól válassza ki ki a feladathoz legjobban illő konténert, és helyesen alkalmazza azt. Ez a fejezet ebben igyekszik segítséget adni.

21.1. A kollekció keretrendszer felépítése

A Kollekció keretrendszer (Collections Framework) interfészeket és osztályokat definiál különböző konténerek (kollekciók és térképek) hatékony létrehozása céljából. A keretrendszer lényegesebb elemeit a 21.1. ábra mutatja; ezek az elemek három fő csoportba sorolhatók:

- **Interfészek:** Az interfészek definiálása jóvoltából a különböző implementációk egységesen és egymástól függetlenül kezelhetők. Az interfészek meghatározzák a különféle konténerek tulajdonságait és lehetséges műveleteit. Az ábrán az interfészeket sötét színnel jelöltük.
- **Osztályok:** Az interfészek implementációi; mindegyik a keretrendszer valamely interfészét implementálja. A különféle implementációk **sűrűn használják a különböző adatszerkezeteket**; például a fát és a hasítótáblát. Az implementációk a választott adatszerkezettől és annak megvalósításától függően eltérhetnek egymástól hatékonyságban, rugalmasságban és biztonságban.
- **Algoritmusok:** A Collections osztály egy algoritmusgyűjtemény; statikus metódusaival rendezni lehet a keretrendszer elemeit, keresni lehet közöttük stb. (a Collections osztályról már szó volt a könyv első kötetében).

A kollekció keretrendszer interfészei és osztályai a `java.util` csomagba kerültek.



21.1. ábra. A Java kollekció keretrendszer főbb elemei

Interfészek

A keretrendszer interfészeit fontos jól ismernünk, hiszen ők szabják meg a konkrét konténerek viselkedését. Az interfések három fő csoportba oszthatók: kollekciók, térképek és iterátorok.

A **kollekció** (collection) objektumokat tárol. minden kollekció implementálja a Collection interfészt. Ebbe a csoportba a következő interfések tartoznak:

- **Collection**: A kollekciók ösinterfész. Metódusai minden Java kollekcióról alkalmazhatók.
- **Set**: Halmaz. Olyan kollekció, amelynek egyediek az elemei (nem lehető bele két egyforma objektum), és az elemek között nincs sorrend.
- **SortedSet**: Rendezett halmaz. Olyan kollekció, amelynek egyediek az elemei (nem lehető bele két egyforma objektum), és az elemek meghatározott sorrendben követik egymást (az elemek rendezettek).
- **List**: Lista. Olyan kollekció, amelynek az elemei meghatározott sorrendben követik egymást, de nem egyediek. A lista elemei indexelhetők.

A **térkép** (map) bejegyzéseket, kulcs-érték párokat tárol. minden térkép implementálja a Map interfészt. A térkép egy kollekcióban tárolja a kulcsokat. Ebbe a csoportba tartoznak a következő interfések:

- **Map**: A térképek ösinterfész. Definiálja a kulcsokkal és a hozzá tartozó adatokkal kapcsolatos műveleteket.
- **SortedMap**: Olyan térkép, amelyben a kulcsok rendezve vannak.

Az **iterátor** a konténerek bejárását segíti. minden iterátor implementálja az Iterator interfészt. A keretrendszerben kétféle iterátorinterfész van:

- **Iterator**: Segítségével akkor is bejárhatjuk a kollekció objektumait, ha nem ismerjük a kollekció belső struktúráját. Az iterátorobjektum gondoskodik a kollekció bejárásának helyességéről.
- **ListIterator**: Segítségével a lista minden irányban bejárható.

Osztályok

A kollekció keretrendszer tervezői minden interfészhez implementáltak legalább egy osztályt. Tekintsük át röviden ezeket az osztályokat!

- ◆ **HashSet**: A Set interfész implementálja, elemei tehát egyediek és nincs közöttük sorrend. Az implementálás hasítótáblára támaszkodik, ezért fontos, hogy a HashSet objektumainak legyen hasítófüggvényük (`hashCode`).
- ◆ **TreeSet**: A SortedSet interfész implementálja, elemei tehát egyediek és határozott sorrendbe vannak állítva (rendezettek). A konténert kiegysúlyozott fa adatszerkezet implementálja.

- ◆ `ArrayList`, `Vector`: Ezek az osztályok a `List` interfészt implementálják, vagyis elemeik nem egyediek és meghatározott sorrendjük van. Az elemek indexelhetők. Belső implementációjuk újraméretezhető tömbre támaszkodik. A két osztály között az a különbség, hogy a `Vector` metódusai szinkronizálva vannak. Egy szinkronizált metódus mindenkor megszakítatlanul, egy menetben fut le, más szál nem szakíthatja meg a működését. Ezért a `Vector` használata lassúbb, de biztonságosabb.
- ◆ `LinkedList`: A `List` interfészt implementálja kétirányú, láncolt listával. Elemei nem egyediek. A lista objektumai láncolva vannak: minden objektum tartalmazza valamilyen módon a rá következő elem mutatóját. A `LinkedList` kollekció a mutatók révén mindenkor irányban bejárható.
- ◆ `Stack`: A `Vector` utódja; a verem adatszerkezetet implementálja: mindenkor csak a legutoljára betett elemet lehet kivenni belőle.
- ◆ `Hashtable`, `HashMap`: A `Map` interfészt implementálják hasítótáblával. Kulcs-érték párokat tárolnak, a kulcsok rendezetlenek. A két osztály között az a fő különbség, hogy a `Hashtable` metódusai szinkronizálva vannak.
- ◆ `TreeMap`: A `SortedMap` interfészt implementálja fa adatszerkezzel. Egy rendezett kulcsokat tartalmazó térkép.

Adataink, objektumaink tárolásához mindenkor ki kell választanunk a megoldandó feladathoz legjobban illő osztályt. A gondos programozó megválogatja konténereinek típusát, hogy gyors és jól karbantartató legyen a programja. **A konkrét osztály megválasztásában sokat segít a klasszikus adatszerkezetek ismerete.**

Mintaprogram

Feladat

Tegyük egy kollekcióba programozási nyelvek nevét! Írjuk ki sorrendben a nyelveket! Egy nyelv csak egyszer szerepeljen!

A kollekció elemei – a nyelvek neve – egyediek és rendezettek. A kollekció keretrendszer a `TreeSet` osztályt kínálja a feladat megoldására. Hozzuk létre a kollekciót, tegyük bele az elemeket, majd listázzuk ki őket! Figyelje meg, hogy hiába igyekszünk kétszer is betenni a "Java" nyelvet a konténerbe, az csak egyszer fog bekerülni, mert a `TreeSet` elemei egyediek. S a kollekció automatikusan végzi el a rendezést.

Forráskód

```
import java.util.*;  
  
public class Minta {  
  
    public static void main(String[] args) {  
        TreeSet nyelvek = new TreeSet();
```

```
nyelvek.add("Java");
nyelvek.add("PHP");
nyelvek.add("Delphi");
nyelvek.add("Java");

System.out.println("Nyelvek: "+nyelvek);
System.out.println("Nyelvek száma: "+nyelvek.size());
if (nyelvek.contains("Java"))
    System.out.println("A Java benne van.");
}
}
```

A program futása

```
| Nyelvek: [Delphi, Java, PHP]
| Nyelvek száma: 3
| A Java benne van.
```

21.2. A Collection interfész és leszármazottai

A Collection interfésznek és leszármazottainak osztálydiagramja a 21.2. ábrán látható. Vegyük sorra ezeket az interfészeket!

A Collection interfész

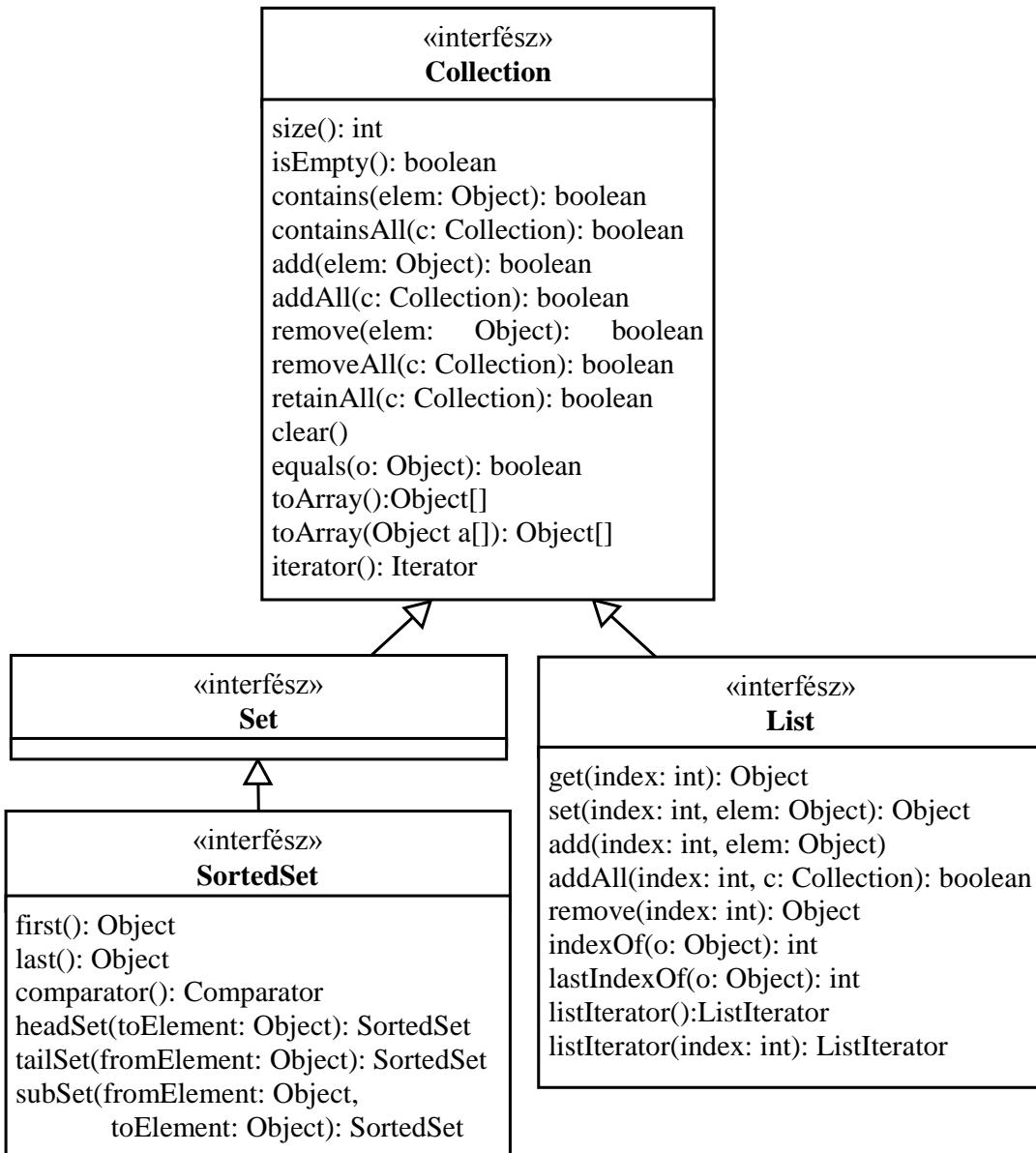
A **kollekció** objektumokat tárolhat. A **collection** interfész a kollekciók interfészeinek őse; tartalmazza a minden kollekcióban használható funkciókat. Az ő utódai a halmazok és a listák.

A Collection interfészt implementáló osztálynak kell, hogy legyen legalább két konstruktora: egy paraméter nélküli és egy egyparaméteres; a paraméter egy másik, tetszőleges kollekció.

A Collection interfésznek nincs közvetlen implementációja.

Metódusok

- ▶ **int size()**
Visszaadja a kollekció méretét, vagyis az éppen benne levő objektumok számát.
- ▶ **boolean isEmpty()**
Visszaadja, hogy a kollekció üres-e.
- ▶ **boolean contains(Object element)**
Megmondja, hogy a megadott elem benne van-e a kollekcióban.
- ▶ **boolean containsAll(Collection c)**
Megmondja, hogy a kollekció tartalmazza-e a paraméterben megadott kollekció összes elemét.



21.2. ábra. A Collection interfész és leszármazottai

- ▶ `boolean add (Object element)`
Betteszi a megadott elemet a kollekcióba; ha ez sikerült, akkor `true` a visszaadott érték.
- ▶ `boolean addAll (Collection c)`
Hozzáadja a kollekcióhoz a paraméterben megadott kollekció összes elemét (egyesítés).
A visszaadott érték `true`, ha a kollekció megváltozott.
- ▶ `boolean remove (Object element)`
Kiveszi a megadott elemet a kollekcióból; ha ez sikerült, akkor `true` a visszaadott érték.

- ▶ `boolean removeAll(Collection c)`
Kiveszi a kollekcióból a paraméterben megadott kollekció összes elemét (különbségképzés). A visszaadott érték `true`, ha a kollekció megváltozott.
- ▶ `boolean retainAll(Collection c)`
Kiveszi a kollekcióból azokat az elemeket, melyek a paraméterben megadott kollekcióban nincsenek benne (metszetképzés). A visszaadott érték `true`, ha a kollekció megváltozott.
- ▶ `void clear()`
Törli a kollekcióból az összes objektumot, s a méretét nullára állítja.
- ▶ `boolean equals(Object o)`
Visszaadja, hogy a kollekció egyenlő-e a megadott objektummal (kollekcióval). Két kollekció egyenlő, ha ugyanannyi elemük van, és elemeik rendre egyenlők.
- ▶ `Object[] toArray()`
- ▶ `toArray(Object a[]): Object[]`
Egy tömbben adja vissza a kollekció objektumait. A második esetben a visszaadott tömb dinamikus típusa a paraméterben megadott típus lesz.
- ▶ `Iterator iterator()`
Ad egy iterátorobjektumot a kollekció bejárására; lásd a 21.5. pontot.

Set interfész

A **Set** (halmaz) olyan kollekció, melynek objektumai egyediek és nincs közöttük semmiféle sorrend. A halmaznak nincs két olyan objektuma, amelyre `o1.equals(o2) == true` lenne. A **Set** nem definiál új metódusokat, bizonyos metódusait azonban másképp kell implementálni, mint az ös metódusokat. Az interfész formailag megegyezik a `Collection` interfésszel.

A **Set** interfészt a kollekció keretrendszerben a `HashSet` osztály implementálja.

Halmazműveletek

A `Collection` interfész szintaktikailag már definiálja a halmazműveleteket – például az `addAll`, a `removeAll` stb. metódusokkal – ; de a `Collection` ettől még nem feltétlenül fog úgy viselkedni, mint egy matematikai értelemben vett halmaz. A **Set** interfész nem definiál ugyan további metódusokat, de az öt implementáló osztálynak kötelessége gondoskodni az elemek egyediségéről – ha például egy halmazba beteszünk egy, valamelyik bent lévővel egyenlő elemet, akkor a halmaz nem változik. Halmazműveletek:

- ◆ **Egyesítés** (unió): Az `a` és `b` halmazok egyesítése: `a.addAll(b)`. Az eredményhalmaz tartalmaz minden olyan elemet, amely akár `a`-nak, akár `b`-nek eleme.
- ◆ **Metszetképzés** (közös rész): Az `a` és `b` halmazok metszete: `a.retainAll(b)`. Az eredményhalmaz tartalmaz minden olyan elemet, mely `a`-nak és `b`-nek is eleme.

- ◆ **Különbségképzés:** Az `a` és `b` halmazok különbsége: `a.removeAll(b)`. Az eredmény-halmaz tartalmaz minden olyan elemet, mely `a`-nak eleme, de `b`-nek nem eleme.
- ◆ **Komplementerképzés:** Az `a` halmaz komplementere: `alaphalmaz.removeAll(a)`. Az eredményhalmaz az alaphalmaz összes elemét tartalmazza, kivéve azokat, amelyek benne vannak `a`-ban.
- ◆ **Elem hozzáadása a halmazhoz:** Az `e` elemmel bővített `a` halmaz: `a.add(e)`. Ha `e` eddig még nem volt eleme az `a` halmaznak, akkor most eleme lesz, egyébként nem törtenik semmi.
- ◆ **Elemtípus vizsgálat:** `a.contains(e)` akkor `true`, ha ez `e` elem benne van az `a` halmazban.

A SortedSet interfész

A `SortedSet` rendezett halmaz. Olyan kollekció, amelynek objektumai egyediek és meghatározott sorrendben követik egymást (rendezettek). A `SortedSet` a `Set` interfész leszármazottja, és új metódusokat is definiál.

A `SortedSet` interfészt a kollekció keretrendszerben a `TreeSet` osztály implementálja.

Metódusok

- ▶ `Object first()`
- ▶ `Object last()`
Visszaadja a halmaz első (legkisebb), illetve utolsó (legnagyobb) elemét.
- ▶ `Comparator comparator()`
Lásd később, a 21.4. pontban.
- ▶ `SortedSet headSet(Object toElement)`
- ▶ `SortedSet tailSet(Object fromElement)`
- ▶ `SortedSet subSet(Object fromElement, Object toElement)`
Visszaadja a rendezett halmaz elejét, végét, illetve egy szakaszát. A részeket a tól-ig határokkal adhatjuk meg úgy, hogy az első elem a `fromElement`, az utolsó a `toElement` előtti elem. Alapértelmezés szerint a rész a halmaz elejtől indul, illetve a halmaz végéig ér.

A List interfész

A `List` (lista) olyan kollekció, amelynek elemei meghatározott sorrendben követik egymást, de nem egyediek. A lista elemei indexelhetők: az első elemének indexe 0, az utolsóé `size() - 1`. A listán kívüli indexelés `IndexOutOfBoundsException` kivételt kelt.

A `List` interfészt a kollekció keretrendszerben az `ArrayList`, a `Vector`, a `LinkedList` és a `Stack` osztály implementálja.

Metódusok

- ▶ `Object get(int index)`
Visszaadja az `index` indexű objektumot.
- ▶ `Object set(int index, Object element)`
Az `index` indexű objektumot átírja `element`-re és visszaadja a felülírt objektum referenciáját.
- ▶ `void add(int index, Object element)`
Az `element` objektumot beszúrja a listába az `index` indexű helyre.
- ▶ `boolean addAll(int index, Collection c)`
A `c` kollekció összes elemét beszúrja a listába az `index`-edik helytől kezdve.
- ▶ `Object remove(int index)`
Az `index` indexű objektumot lekapcsolja a listáról. A metódus visszatérési értéke a lekapcsolt objektum lesz.
- ▶ `int indexOf(Object obj)`
- ▶ `int lastIndexOf(Object obj)`
Megkeresi a listában `obj` objektummal egyenlő első, illetve utolsó elemet, és visszaadja annak indexét.
- ▶ `ListIterator listIterator()`
- ▶ `ListIterator listIterator(int index)`
Ad egy `ListIterator` objektumot a lista bejárására; a bejárás a lista elején, illetve a megadott indextől indul. Lásd később, a 21.6. pontban.

21.3. A HashSet osztály – hasítási technika

Hasítókód – hashCode

Az előző fejezetben, a Tábla pontban szó volt a hasítótáblákról és a hasítófüggvéniről. Most itt az alkalom, hogy átismétljük és alkalmazzuk ezt a fogalmat!

A hasítási technika (hashing) igen jó hatásfokkal használható rendezetlen elemhalmaz karbantartására. Az eltárolandó objektumok halmazát részekre hasítjuk (csoportosítjuk), és a részekhez egy-egy hasítókódot (egész számot) rendelünk. Az objektumokat tehát egyértelműen megfeleltetjük a csoportjuknak. **A hasítókód meghatározása a hasítótáblába betett objektum dolga.** A hasítás közvetlenné teszi az objektum elérését, tehát nagyon gyors. A hasítás után már csak a megfelelő – néhány elemű – csoportban kell keresni.

Mivel a hasítótáblába betett objektumok hasítókódjának tartományát nem lehet előre megjósolni, azért a csoportok indexét ajánlatos normalizálni: előre megadjuk a csoportok számát (ez a hasítótábla kapacitása), majd a tényleges csoportindexet így képezzük: `csoportindex = obj.hashCode() % kapacitás`. Ez azért jó, mert így a csoportindexek tartománya előre

ismeretes lesz, s ráadásul 0-val kezdődik. Igaz ugyan, hogy a csoportokban most különböző hasítókódú objektumok is bekerülnek, de az egyenlő objektumok így is egy csoportban lesznek.

A hasítófüggvény készítésének szabályai:

- Egy objektumnak következetesen minden ugyanazt a hasítókódot kell generálnia!
- Ha o1.equals(o2), akkor o1-nek és o2-nek ugyanazt a hasítókódot kell adnia! (De két különböző objektum is generálhatja ugyanazt a hasítókódot, vagyis egy csoporton belül nem feltétlenül egyenlők az objektumok.)

A hasítófüggvényt ajánlatos úgy meghatározni, hogy az objektumok egyenletesen osztódonak be a különféle csoportokba. A hasítókód értékének nagyságával – a normalizálás jóvoltából – nem kell törödni.

A hasítótábla tulajdonságai (21.3. ábra):

- **Kapacitás** (capacity): A hasítótábla csoportjainak a száma. Az objektum által előállított hasítókód a kapacitástól függően normalizálódik:
 $csoportindex = obj.hashCode() \% \text{kapacitás}$
- **Telítettség**: 0 és 1 közötti szám; megadja, hogy a hasítótábla mennyire van tele. Ha a hasítótáblán levő elemeknek elemszám a számuk, akkor
 $\text{telítettség} = \text{elemszám}/\text{kapacitás}$
- **Telítettségi küszöb** (load factor): A telítettség határértéke a hasítótáblában; ha az elemek száma eléri ezt az értéket, akkor a kapacitás megnövekszik, és a hasítótábla újraszerveződik.

Ritka, hogy egy csoportba több elem is bekerüljön, mert a kapacitás (a csoportok száma) minden nagyobb, mint az elemek száma.

n	E	o	H		t	a		
0		1	2	...	6	7	...	9

Csoportindex

```
Character kar = new Character('H');
kar.hashCode() == 72
csoportindex == kar.hashCode()%10 == 2
```

Kapacitás: 10
Elemszám: 6
Telítettség: 0.6
Telítettségi küszöb: 0.9

21.3. ábra. Hasítás

A 21.3. ábra szerint `Character` objektumokat teszünk a hasítótáblába. Az induló kapacitást 10-nek választjuk meg, a telítettségi küszöböt 0,9-nek. A `HashSet` osztály konstruktőrénél ezt így kellene megadni:

```
HashSet betuk = new HashSet(10, 0.9F);
```

Tegyük a hasítótáblába a "Hottentotta" szó karaktereit! Mivel a kollekció halmaz, azért egy karakter csak egyszer kerül bele; a karakterek tehát: "Hotena". A karakterek az unikód a hasítókódja (például `hashCode('H') == 72`), vagyis a betett karakterek hasítókódja rendre: H: 72, a: 97, e: 101, n: 110, o: 111, t: 116. A csoportindexet a 10-zel (kapacitás) való osztás maradéka adja, a 'H' tehát a 2-es csoportba kerül. Ily módon az 1-es csoportba két karakter is jut: az e és az o.

Az API-osztályok (mint például az `Integer`, a `Character` vagy a `String`) hasítófüggvénye minden felül van írva. Ezeket a függvényeket nem is tudjuk megváltoztatni, mert ezek az osztályok véglegesek (`final`). Érdekességeképpen nézzük meg ezeknek az osztályoknak a hasítókódját:

- ◆ `Integer`: maga a szám.
- ◆ `Character`: a karakter unikódja.
- ◆ `String`: $s[0]*31^{n-1}+s[1]*31^{n-2}+\dots+s[n-1]$
($s[i]$ az i-edik karakter, $^$ a hatványozás, n a `String` hossza)

Megjegyzések:

- A hasítás hatékonysága akkor érzékelhető igazán, ha a kollekcióba tett objektumok száma nagyon nagy. Ekkor ugyanis sokat számít, hogy nem kell az összes objektumot végigpásztázni, csak a hasítással keletkezett néhány elemű objektumhalmazt.
- Saját hasítófüggvény készítésére igen ritkán van szükség.

A `HashSet` osztály

Csomag: `java.util`

Deklaráció: `public class HashSet`

Közvetlen ős: `java.util.AbstractSet`

Fontosabb implementált interfések: `Cloneable`, `Serializable`, `Set`

A `HashSet` osztály a `Set` interfész implementációja, vagyis halmazként viselkedik. Mint a nevéből is látni való, az implementáció a hasítási technikára (hasítótáblára) támaszkodik. Az osztály a különféle műveleteket csaknem ugyanannyi idő alatt végzi el, mégpedig nagyon gyorsan. A halmazba `null` elemeket is be lehet tenni. Az iterátor megjósolhatatlan sorrendben járja be a halmazt, vagyis két kiírásban nem feltétlenül lesz ugyanaz a sorrend. A metódusok nincsenek szinkronizálva.

Alkalmazás: Olyan kollekciókkal használható, amelyben nem fontos a sorrend és az elemek nem lehetnek egyenlők. Nagy elemszám esetén nagyon hatékony!

Konstruktorok, metódusok

- ▶ `HashSet()`
- ▶ `HashSet(int initialCapacity)`
- ▶ `HashSet(int initialCapacity, float loadFactor)`
- ▶ `HashSet(Collection c)`

Létrehoz egy kollekciót (halmazt), az első három esetben üreset. Paraméterek:

- `c`: létrehozás után betölti a megadott kollekciót, mintha annak elemeit egyenként, az `add` metódussal tennénk be.
- `initialCapacity`: kezdeti kapacitás. Akkor jó, ha a várható elemek számának nagyjából a kétszerese. Alapértelmezés: 101.
- `loadFactor`: telítettségi arány = elemszám / kapacitás; értéke 0 és 1 között van. Ha a telítettség eléri ezt a kritikus értéket, akkor a hasítótábla kapacitása automatikusan bővül. Alapértelmezés: 0,75.

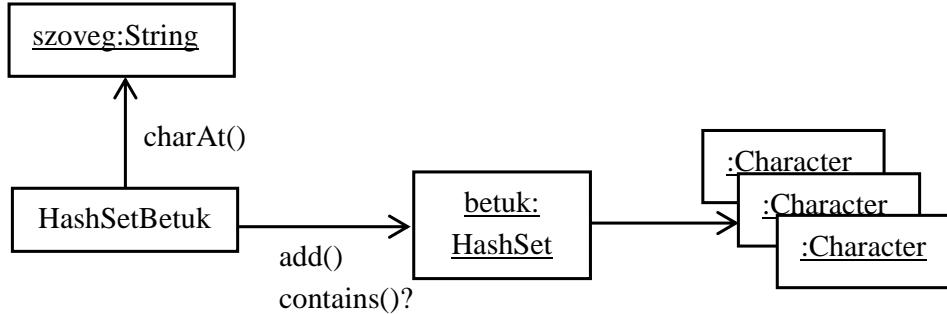
- ▶ `String toString()`

Ezt a metódust már az `AbstractCollection` ősosztályban implementálták: a kollekciót `[elem, elem, ...]` formára alakítja. Így a `System.out.println()` áttekinthető formában írja konzolra a kollekciót.

Példaként tegyük egy `HashSet`-objektumba karaktereket, ahogy azt a 21.3. ábrán láthattuk:

Feladat – HashSetBetuk

Kérjünk be konzolról egy karakterláncot! Írjuk ki a benne szereplő betűket ábécérendben (mindegyiket csak egyszer)!



21.4. ábra. A `HashSetBetuk` program együttműködési diagramja

A program terve

A program együttműködési diagramja a 21.4. ábrán látható. A fő osztály a `HashSetBetuk`. Bekérjük a szöveget, majd végigmegyünk a `szoveg` objektum karakterein (`charAt`). A karaktereket egyenként objektummá alakítjuk és bedobjuk a `betuk` halmazba (`add`).

Végül kiírjuk a halmaz karaktereit kétféle módszerrel:

- ◆ A `System.out.println()` metódus a halmaz `toString()` által visszaadott karakterláncát írja ki. Ebben az esetben a kiírás sorrendje megjósolhatatlan lesz.
- ◆ Végigmegyünk az összes létező unikódos karakteren, és megkérdezzük a `betuk` halmazt, hogy az éppen soron levő karakter benne van-e a halmazban (`contains`). Ekkor a kiírás rendezett lesz.

Forás kód

```
import extra.*;
import java.util.*;

public class HashSetBetuk {
    public static void main(String[] args) {
        HashSet betuk = new HashSet();
        String szoveg = Console.readLine("Szöveg= ");
        // Karakterek gyűjtése a betuk halmazban:
        for (int i=0; i<szoveg.length(); i++)
            betuk.add(new Character(szoveg.charAt(i)));

        // A betuk halmaz kiírása egyszerűen:
        System.out.println(betuk);

        // A betuk halmaz kiírása elemvizsgálattal:
        for (char c=0; c<Character.MAX_VALUE; c++)
            if (betuk.contains(new Character(c)))
                System.out.print(c);
        System.out.println();
    }
}
```

A program egy lehetséges futása:

Szöveg= Hottentotta [a, H, t, o, n, e] Haenot

21.4. A TreeSet osztály – Comparator

A TreeSet osztály a SortedSet interfész implementációja. A rendezettséghez az osztálynak bármely két objektumról tudnia kell, hogy melyik a kisebb, illetve hogy azok egyenlők-e. A kölcsönözött halmazban tehát csak összehasonlítható objektumokat lehet betenni. A Set-hez képest a SortedSet-ben megjelenik néhány új metódus: kikérhetjük például a halmaz legkisebb és legnagyobb elemét, valamint a rendezett halmaz elejét, egy szakaszát és végét – ezek szintén rendezett halmazok.

Első például vegyük az előző pontokban tárgyalta `HashSetBetuk.java` programot. Cseréljük ki az összes `HashSet` szót `TreeSet`-re! Ezt megtehetjük, hiszen a halmazokban `Character` típusú elemek vannak, és a `Character` osztály implementálta a `Comparable` interfészt. A

Character objektumok tehát összehasonlíthatók, vagyis rendezhetők is. Az eredmény a program futásán látható: a halmazok elemei rendezetten íródnak ki:

Szöveg= Hottentotta
[H, a, e, n, o, t]
Haenot

De általában nem lehet egy programban csak úgy, gondolkodás nélkül minden HashSet szót TreeSet-re cserélni! A TreeSet-be betett objektumoknak összehasonlíthatóknak kell lenniük, vagyis implementálniuk kell a Comparable interfész. A Character osztály implementálta ezt az interfész, a karaktereket tartalmazó kollekció tehát igenis lehet TreeSet.

TreeSet osztály

Csomag: java.util

Deklaráció: public class TreeSet

Közvetlen ös: `java.util.AbstractSet`

Fontosabb implementált interfészek: Cloneable, Serializable, SortedSet

A **TreeSet** osztály a **SortedSet** interfész implementációja, vagyis egy rendezett halmaz. Mint azt a neve is mutatja, az implementáció a kiegyensúlyozott fa adatszerkezetre támaszkodik. A halmazba tett objektumoknak összehasonlíthatónak kell lenniük. Két eset lehetséges:

- Az objektumok implementálják a Comparable interfészt – ez a **termézeszetes összehasonlítás**;
 - A TreeSet konstruktőrben megadunk egy összehasonlító (comparator) objektumot.

A kollekcionák persze egyszerre csak egyfélé rendezettsége lehet, de elemei „áttölthetők” egy más rendezettségű kollekcióba.

Alkalmazás: Olyan kollekciókra szokás alkalmazni, amelyekben nem lehet két egyenlő elem és fontos a rendezettség. Nagy elemszám esetén nagyon hatékony a keresés. A felvitel hatékonysága rosszabb, mint a HashSet osztályé.

Konstrukciók, metódusok

- ▶ `TreeSet()`
 - ▶ `TreeSet(Collection c)`
 - ▶ `TreeSet(SortedSet s)`
 - ▶ `TreeSet(Comparator comp)`

Létrehoz egy üres vagy nem üres kollekciót (rendezett halmazt). Paraméterek:

- `c, s`: létrehozás után betölti a megadott kollekciót, illetve rendezett halmazt.
 - `comp`: hasonlító objektum: implementálta a `Comparator` interfészt.

- #### ► Comparator comparator()

Visszaadja az aktuális hasonlító objektumot, illetve null-t, ha az osztály a természetes rendezettséget használja

A Comparator interfész

A Comparable interfész használatának nyilvánvaló korlátai vannak: 1. A betett objektumok osztályában csak egyfélé rendezettség adható meg; 2. A rendezettség „bele van égetve” az objektumba, vagyis az előzőleg rögzített rendezettséget nem lehet meg változtatni; 3. Az is elközelhető, hogy egy készen kapott osztály nem implementálta a Comparable interfészt.

A Comparator interfész egyetlen metódust definiál, a `compare`-t. Célja egyrészt az, hogy egy kollekciót természetes rendezettségen kívül más rendezettséget is adhasson, másrészt az, hogy a rendezettséget kívülről is meg lehessen adni.

A Comparator implementációja meghatározza két objektum sorrendjét. A belőle létrehozott példány egy, a TreeSet konstruktorának átadható **összehasonlító objektum**. A Comparator interfész természetesen névtelen osztályként is implementálhatjuk a TreeSet konstruktorának paraméterében.

Metódus

► `public int compare(Object a, Object b)`

A visszaadott érték 0, ha a és b egyenlő; negatív, ha a megelőzi b-t; egyébként pozitív.

A compare a Comparable interfész compareTo metódusához hasonlóan viselkedik.

A következőkben feladatokat oldunk meg: két feladatban (TreeSetLotto és TreeSetSzavak) a természetes hasonlításra hagyatkozunk; a harmadikban (TreeSetNevek) egyedi rendezettséget vezetünk be.

Feladat – TreeSetLotto

Írunk egy lottótippek-kírtékelő programot! Az öt nyerő számot „égeszük bele” a programba, a tippel (öt számot) a parancssori paraméter adja meg. Írjuk ki sorban:

- a nyerőszámokat;
- a tippelt számokat;
- az eltalált számok számát;
- azt, hogy mely számok találtak!

A halmaz elemei most Integer objektumok lesznek.

Forráskód

```
import java.util.*;

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

        // A nyerők halmaz összeállítása (kis elemszám esetén):
        TreeSet nyerok = new TreeSet(); //1
```

```
nyerok.add(new Integer(5));
nyerok.add(new Integer(13));
nyerok.add(new Integer(33));
nyerok.add(new Integer(34));
nyerok.add(new Integer(35));

// tipp összeállítása az args tömbből:
TreeSet tipp = new TreeSet(); //2
try {
    for (int i=0; i<args.length; i++)
        tipp.add(new Integer(args[i]));
}
catch (NumberFormatException ex) {
    System.out.println("A paraméter nem egész szám!");
    return;
}
catch (Exception ex) {
    System.out.println(ex.getMessage());
    return;
}

System.out.println("Nyerőszámok: "+nyerok); //3
System.out.println("Tipp: "+tipp);
TreeSet talalat = new TreeSet(nyerok);
talalatretainAll(tipp);
System.out.println("Találatok száma: "+talalat.size());
System.out.println("Találatok: "+talalat);
}
```

A program futása, ha a parancsor paraméter: 13 9 8 7 33

Nyerő számok: [5, 13, 33, 34, 35]
Tipp: [7, 8, 9, 13, 33]
Találatok száma: 2
Találatok: [13, 33]

A program elemzése

- ◆ //1: Összeállítjuk a nyerőszámokat. A primitív egészeket előbb becsomagoljuk: new Integer(13), s objektumként tesszük be őket a halmazba. Az öt elemet öt add utasítással tesszük a kollekcióba. Ez azonban nem szép megoldás: ha egymás után több elemet akarunk betenni egy halmazba, akkor sokkal jobb egy tömb inicializáló blokkjában megadni az elemeket, majd egy for ciklussal sorban betenni őket a halmazba. Ezzel a módszerrel állítjuk össze a tippeket az args tömbből (//2).
- ◆ //3: A találatokat úgy kapjuk meg, hogy vesszük a nyerok és a tipp halmaz metszetét (retainAll). A találatok számát a talalat halmaz size metódusa adja meg.

Feladat – TreeSetSzavak

Adva van egy mondat. Listázzuk ki a mondatban szereplő szavakat rendezetten, de mindeneket csak egyszer!

Forráskód

```
import java.util.*;  
  
public class TreeSetSzavak {  
    public static void main(String[] args) {  
        String szoveg = "Itt is, ott is, amott is.";  
  
        TreeSet szavak = new TreeSet();  
        StringTokenizer st =  
            new StringTokenizer(szoveg, " .,\t\n\r\f");  
        while (st.hasMoreTokens())  
            szavak.add(st.nextToken());  
  
        System.out.println(szavak);  
    }  
}
```

A program futása

```
| [Itt, amott, is, ott]
```

A program elemzése

A mondatot egy StringTokenizer objektummal szavakra bontjuk, s a szavakat sorban bedobjuk a szavak kölcsönözöttbe. Mivel a tárolás rendezett, azért a kiírás is a természetes rendezettséget, vagyis az ábécét követi. Látható, hogy az "is" szót háromszor is megpróbáltuk beleenni a halmaiba, de csak egyszer sikerült.

Feladat – TreeSetNevek

Tegyük be keresztnéveket egy konténerbe! Ne vigyük be egyforma neveket! Írjuk ki a neveket először névsor szerint, azután névhosszúság szerinti rendezettségen!

Forráskód

```
import java.util.*;  
  
public class TreeSetNevek {  
    public static void main(String[] args) {  
        TreeSet nevsor = new TreeSet(); //1  
  
        nevsor.add("Soma");  
        nevsor.add("Jeremiás");  
        nevsor.add("Janka");
```

```

// Kiírás névsorban:
System.out.println(nevsor);

TreeSet nevek = new TreeSet(new Comparator() { //2
    public int compare(Object a, Object b) {
        return ((String)a).length() - ((String)b).length();
    }
} );
nevek.addAll(nevsor);
System.out.println(nevek);
}
}

```

A program futása

[Janka, Jeremiás, Soma]
[Soma, Janka, Jeremiás]

A program elemzése

- ◆ //1-ben hasonlító objektum nélkül hozzuk létre a TreeSet-et, ezért a természetes rendezettség lép érvénybe. Mivel a String osztály compareTo metódusa lexikografikusan rendez, azért a nevek névsor szerint tárolódnak.
- ◆ //2-ben a TreeSet konstruktőrben megadtunk egy hasonlító objektumot. A Comparator interfész névtelen osztályként implementáljuk, és megírjuk compare metódusát: abban a szöveg hossza határozza meg a sorrendet. A kiírásban látszik is, hogy a legrövidebb név van legelöl.

21.5. Iterátor

Az **iterátor** egy, a kollekciótól elérhető objektum; arra szolgál, hogy a kollekció objektumait akkor is bejárassuk, ha nem ismerjük a kollekció belső struktúráját. Ennek a technikának az a nagy erénye, hogy **egy kollekcióhoz több iterátor is kapcsolható**, és azok egymástól függetlenül működtethetők.

Az iterátorobjektum gondoskodik a kollekció helyes bejárásáról. Rendezett kollekcióban a bejárás a megadott rendezettséget követi; nem rendezett kollekcióban a bejárás megjósolhatatlan.

A kollekció keretrendszer minden osztályának implementálnia kell a Collection interfészben megadott iterator metódust; az iterátort ezzel a metódussal lehet elérni a coll kollekciótól:

```
Iterator iter = coll.iterator();
```

Az iterátorobjektum osztálya implementálja az Iterator interfészét.

Az Iterator interfész

Az **Iterator** interfész három metódust definiál a kollekció bejárásához. Egy iterátorral csak egyszer járható be a kollekció; egy esetleges újabb bejárásrahoz újabb iterátorra van szükség.

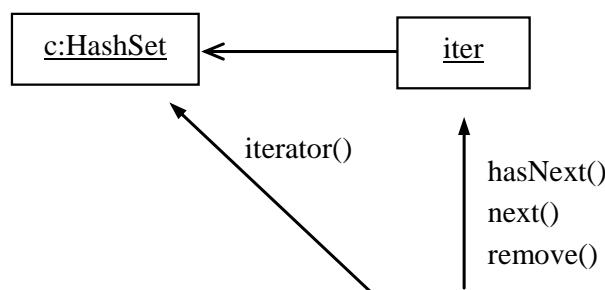
Metódusok

- ▶ `boolean hasNext()`
Visszaadja, hogy van-e még következő elem, vagyis hogy a `next` metódus eredményes lenne-e.
 - ▶ `Object next()`
Visszaadja a következő elemet. Az iterátor létrehozása után az első elemet adja. Ha nincs több elem, akkor `java.util.NoSuchElementException` keletkezik.
 - ▶ `void remove()`
Lekapcsolja az elemet a kollekcióról. A kollekció mérete eggyel kisebb lesz. A következő `next` az ezt követő elemet adja. A `remove` metódust nem lehet egymás után kétszer meghívni – előbb meg kell hívni a `next`-et!

Az alábbi példában egy tetszőleges kollekciót járunk be iterátor segítségével: a következő elemet minden az iterátortól kérjük a `next` metódussal.

```
// c osztálya implementálja valamelyik kollekciót
Collection c = new ...;
...
Iterator iter = c.iterator();      // c-től kapunk egy iterátort
while (iter.hasNext()) {          // ha van még elem
    Object current = iter.next(); // kérjük a következőt
    // (Objektum osztálya)current.üzenet;
}
```

Az iterátor által szolgáltatott objektum `Object` típusú, ezért bizonyos üzenetek küldése előtt rá kell kényszerítenünk igazi osztályát. Az iterátor és környezetének osztálydiagramja a 21.5. ábrán látható.



21.5. ábra. Az iterátor működése

Megjegyzések:

- A kollekció bejárása hasonlít a folyam olvasásához: azt is csak egyszer lehet végigolvasni, és egy-egy egység elolvasása után a mutató a következő elemre áll.
- Az iterátor azért tartalmazza a `remove` metódust, mert azzal nagyon hatékonyan törölheti az elemeket.

Feladat – Országok

Tegyük be országokat egy `HashSet` kollekcióba! Listázzuk ki őket először a `HashSet`-be beépített módon, majd egy iterátorobjektum segítségével!

Forráskód

```
import java.util.*;

public class Orszagok {
    public static void main(String[] args) {
        String[] nevek =
            {"USA", "Lengyelország", "Nepál", "Magyarország", "Kuba"};

        HashSet orszagok = new HashSet();
        for (int i = 0; i < nevek.length; i++)
            orszagok.add(nevek[i]);

        System.out.println(orszagok);

        // Kiirás iterátor segítségével:
        System.out.println("Országok:");
        Iterator iter = orszagok.iterator();
        while (iter.hasNext())
            System.out.println(iter.next());
    }
}
```

A program futása

```
[Lengyelország, USA, Magyarország, Nepál, Kuba]
Országok:
Lengyelország
USA
Magyarország
Nepál
Kuba
```

21.6. A List interfész implementációi

Az `ArrayList`, a `Vector` és a `LinkedList` osztály minden a `List` interfész implementációja, vagyis az elemek nem egyediek bennük, meghatározott sorrendben követik egymást, és indexelhetők. Ezek az osztályok csak hatékonyságban és az implementációban térnek el egymástól:

- **ArrayList**: A listát újraméretezhető tömbbel implementálja. Ha betelik a tömb, akkor felvesz egy nagyobb tömböt, és az objektumokat átmásolja abba.
- **Vector**: Mindössze abban különbözik az `ArrayList` osztálytól, hogy a `Vector` metódusai szinkronizálva vannak. Egy szinkronizált metódus minden egyrétegen fut le, más szál nem szakíthatja meg a futását. Emiatt a `Vector` lassúbb, de biztonságosabb.
- **LinkedList**: A `List` interfészét kétféleképpen implementálja. A lista objektumai össze vannak kapcsolva: minden objektum valamilyen módon tartalmazza a rá következő elem mutatóját. A lista a mutatók révén az elejtől a végéig bejárható. A láncolt listába sokkal hatékonyabb a felvitel, mint tömbös társaiba; az indexelés viszont lassúbb.

A `Stack` a `Vector` közvetlen utódja; ez implementálja a verem adatszerkezetet.

A `List` interfész `listIterator()` metódusa egy `ListIterator` objektumot ad: azzal oda és vissza be lehet járni a listát.

Konstruktörök, metódusok

- ▶ `ArrayList()`
- ▶ `Vector()`
- ▶ `LinkedList()`
- ▶ `Stack()`
- ▶ `...`
- ▶ `ArrayList(Collection c)`
- ▶ `...`

Mind a négy osztálynak van paraméter nélküli konstruktora. Ezenkívül megadható paraméterként egy másik kölcsönös hivatkozás (kivétel a `Stack`), az `ArrayList` és `Vector` konstruktoraiban az `InitialCapacity` stb.

A `ListIterator` interfész

Az `ListIterator` interfész az `Iterator` interfész leszármazottja; a lista kétféleképpen bejárásához definiál metódusokat. Az előre- és hátralépések tetszőleges sorrendben követhetik egymást. Bejárás közben új elemeket lehet beszűrni, már meglévőket kitörölni és átírni.

Metódusok

- ▶ `void add(Object obj)`
Beszűrja az `obj` objektumot a listába, az aktuális pozícióba.
- ▶ `set(Object obj)`
Az aktuális pozícióban levő elemet kicseréli a megadottal.

- ▶ boolean hasNext()
- ▶ boolean hasPrevious()

Visszaadja, hogy van-e még következő/előző elem, vagyis hogy a next/previous metódus eredményes lenne-e.

- ▶ Object next()
- ▶ int nextIndex()

Visszaadja a következő elemet, illetve annak az indexét. Az iterátor létrehozása után az első elemet, illetve annak az indexét (0) adja.

- ▶ Object previous()
- ▶ int previousIndex()

Visszaadja az előző elemet, illetve annak az indexét. Az iterátor létrehozása után az utolsó elemet, illetve annak az indexét adja.

- ▶ void remove()

Lekapcsolja a listáról a next vagy previous által legutóbb érintett elemet. A kollekció mérete eggyel kisebb lesz.

Feladat – LinkedListTest

Vigyük fel szövegeket egy láncolt listába! Ezután listázzuk ki a szövegeket

- a felvitel sorrendjében!
- visszafelé, s közben töröljük ki az A betűvel kezdődő szövegeket!
- s megint előlről!

Végül írjuk ki az utolsó szöveget!

Forráskód

```
import java.util.*;

public class LinkedListTest {
    public static void main(String[] args) {
        LinkedList lista = new LinkedList();
        lista.add("Gergő");
        lista.add("Anna");
        lista.add("Matyi");

        System.out.println("Listázás előre:");
        ListIterator iter = lista.listIterator();
        while (iter.hasNext())
            System.out.println(iter.next());

        System.out.println("\nListázás visszafelé, közben az "+
                           "A betűsök törlése:");
        while (iter.hasPrevious()) {
            String str = (String)iter.previous();
            System.out.println(str);
            if (str.startsWith("A"))
                iter.remove();
        }
    }
}
```

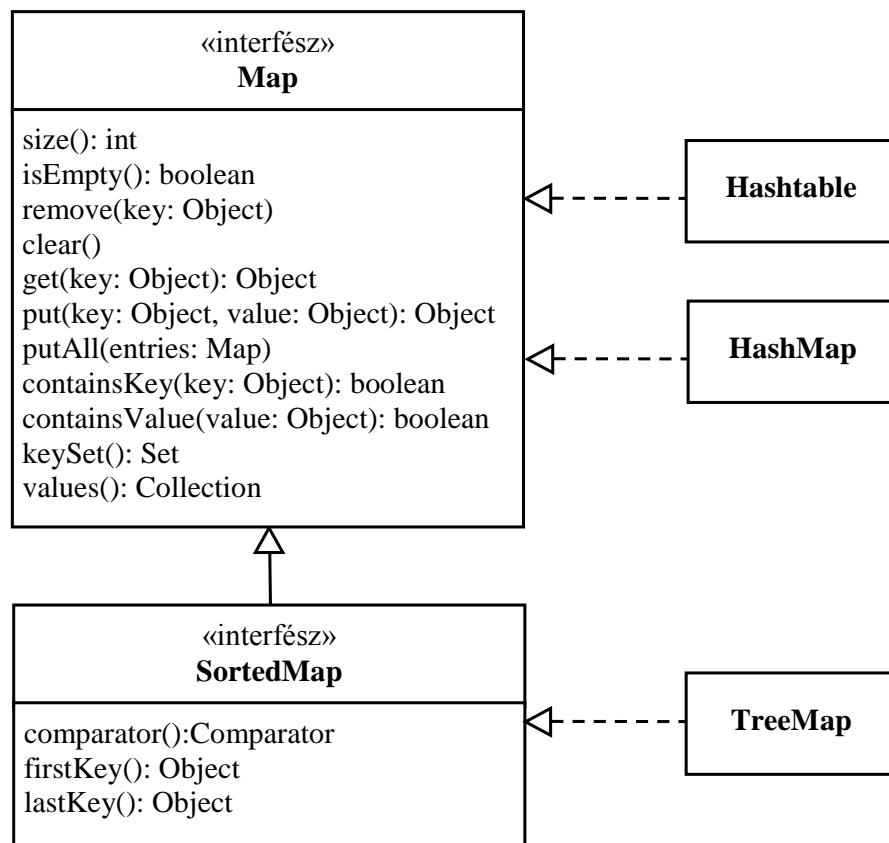
```

        System.out.println("\nLista előre: ");
        iter = lista.listIterator();
        while (iter.hasNext()) {
            System.out.println(iter.next());
        }

        System.out.println("\nUtolsó elem: "+
                           lista.get(lista.size()-1));
    } // main
}

```

21.7. A Map és a SortedMap interfész



21.6. ábra. Térképek

A **terkép (map)** asszociatív konténer (tároló): kulcs-érték párokat tárol. A kulcsot (key) és a hozzáartozó értéket (value) együtt bejegyzésnek (entry) nevezzük. A kulcsobjektum egyedi érték: aszerint megkereshetjük a hozzáartozó értékobjektumot. A kulcsokat egy kollekció (Collection) tárolja.

A Java két térképinterfész definiál:

- Map: implementációja a Hashtable és a HashMap osztály;
- SortedMap: implementációja a TreeMap osztály.

A SortedMap rendezetten tárolja a kulcsokat.

A Map és a SortedMap térképinterfész a 21.6. ábrán látható. Vegyük sorra ezeket az interfészeket!

A Map interfész

A térkép kulcs-érték párokat tárol. A **Map** interfész a térképek ösinterfésze: tartalmazza a minden térképre alkalmazható funkciókat. A Map interfész a Hashtable és a HashMap implementálja a kollekció keretrendszerben – ezek az osztályok HashSet-ben tárolják a kulcsokat, a kulcsoknak tehát nincs meghatározott sora.

Metódusok

- int size()
Visszaadja a bejegyzések számát.
- boolean isEmpty()
Visszaadja, hogy a térkép üres-e.
- Object remove(Object key)
Törli a key kulcsra tartozó bejegyzést (a kulcsot és az értéket is).
- void clear()
Törli a térképet (az összes bejegyzést).
- Object get(Object key)
Visszaadja a kulcsra tartozó értéket.
- Object put(Object key, Object value)
Kulcs-érték pár beszúrása a térképbe. Ha ilyen kulcs már szerepelt, akkor a kulcsra tartozó értéket (value) kicseréli az itt megadottat, és visszaadja a régi értékobjektumot. Ha nem volt még ilyen kulcs, akkor null a visszaadott érték.
- putAll(Map entries)
Hozzáadja ehhez a térképhez egy másik térkép összes bejegyzését.

- ▶ `boolean containsKey(Object key)`
A visszaadott érték `true`, ha a `key` kulcs jelen van a térképben.
- ▶ `boolean containsValue(Object value)`
A visszaadott érték `true`, ha a `value` objektum jelen van a térképben.
- ▶ `Set keySet()`
Visszaadja a térkép összes kulcsát egy halmazban.
- ▶ `Collection values()`
Visszaadja a térkép összes értékét egy kollekcióban.

A SortedMap interfész

A `SortedMap` olyan térkép, amelyben rendezettek a kulcsok. A `SortedMap` a `Map` interfész leszármazottja. A `SortedMap` interfészt a kollekció keretrendszerben a `TreeMap` implementálja; `TreeMap` `TreeSet`-ben tárolja a kulcsokat.

Metódusok

- ▶ `Comparator comparator()`
Visszaadja a hasonlító objektumot, vagy `null`-t, ha természetes a hasonlítás.
- ▶ `Object firstKey()`
- ▶ `Object lastKey()`
Visszaadja a térkép legkisebb, illetve legnagyobb kulcsát.

21.8. A Hashtable osztály

Csomag: `java.util`

Közvetlen ős: `java.util.Dictionary` Deklaráció: `public class Hashtable`

Fontosabb implementált interfések: `Cloneable`, `Serializable`, `Map`

A `Hashtable` osztály rendezetlen térkép, hasítótáblával implementálja a `Map` interfészt. A `Hashtable` típusú konténer kulcs-érték párokat tárol; a kulcsoknak nincs meghatározott sorrendjük benne.

Megjegyzések:

- A `Hashtable` osztály nevében a `t` kisbetű (\odot).
- A `HashMap` osztály nagyon hasonlít a `Hashtable` osztályhoz. Voltaképpen csak abban térnek el egymástól, hogy a `Hashtable` szinkronizálva van és `null` értékobjektumokat is megenged.

Konstruktörök

- ▶ Hashtable()
- ▶ Hashtable(Map entries)
- ▶ Hashtable(int initialCapacity)
- ▶ Hashtable(int initialCapacity, float loadFactor)

Létrehoz egy üres hasítótáblát. Paraméterek:

- entries: létrehozás után betölti a megadott térképet.
- initialCapacity: kezdeti kapacitás. Akkor jó, ha az elemek számának kb. kétszerese. Alapértelmezés: 101.
- loadFactor: telítettségi arány = elemszám / kapacitás; értéke 0 és 1 közé esik. Ha a telítettség eléri ezt a kritikus értéket, akkor a hasítótábla kapacitása automatikusan bővül. Alapértelmezés: 0,75.

Feladat – HashtableAuto

Vigyünk fel autók adatait! A rendszámot és a hozzá tartozó autómárkát egy-egy szövegben tároljuk. A rendszám egyedi adat. Ezután

- Írjuk ki az autók adatait a Hashtable.toString által felkínált formában!
- Írjuk ki külön a kulcsokat, majd külön az értékeket (autómárkákat)!
- Keressük meg a megadott rendszámú autót! Írjuk ki az autó márkJát.
- Cseréljünk ki egy megadott rendszámú autó márkJát!
- Írjuk ki egyenként, sorszámozva az autók adatait!

Forráskód

```
import java.util.*;

public class HashtableAuto {
    public static void main(String[] args) {
        Hashtable autok = new Hashtable(); //1

        autok.put("BIT-442", "Piros Romeo");
        autok.put("SIT-999", "Fehér Merci");
        autok.put("CAR-152", "Zöld Mazda");

        // A teljes térkép összes bejegyzésének kiírása:
        System.out.println(autok);

        // Kulcsok, értékek kiírása:
        System.out.println("Kulcsok: "+autok.keySet());
        System.out.println("Értékek: "+autok.values());

        // Adott kulcsú bejegyzés keresése:
        String rendszam = "CAR-152";
        if (autok.containsKey(rendszam))
            System.out.println(rendszam+" megvan, "+autok.get(rendszam));

        // Csere:
        autok.put(rendszam, "Fekete Trabant");
```

```
// Kiírás iterátorral:
Iterator iter = autok.keySet().iterator();
int n = 0;
while (iter.hasNext()) {
    Object key = iter.next();
    System.out.println(++n + ". " + autok.get(key));
}
}
```

A program futása

```
{CAR-152=Zöld Mazda, SIT-999=Fehér Merci, BIT-442=Piros Romeo}
Kulcsok: [CAR-152, SIT-999, BIT-442]
Értékek: [Zöld Mazda, Fehér Merci, Piros Romeo]
CAR-152 megvan, Zöld Mazda
1. Fekete Trabant
2. Fehér Merci
3. Piros Romeo
```

A program elemzése

Figyelje meg, hogy a kulcsok nincsenek rendezve: CAR előbb van, mint a BIT. Cserélje ki //1-ben a Hashtable osztályt TreeMap-re, s a kulcsok már rendezettek lesznek! Ekkor persze felmerülhet a kérdés: ha ebben a feladatban jobb lett volna a TreeMap, akkor miért van szükség a HashMap osztályra? A Hashtable előnye csak akkor mutatkozik meg, ha nagyon nagy az elemszám: mert akkor nagyon gyors a felvitel.

21.9. A TreeMap osztály

Csomag: java.util

Deklaráció: public class TreeMap

Közvetlen ős: java.util.AbstractMap

Fontosabb implementált interfések: Cloneable, Serializable, Map, SortedMap

A **TreeMap** osztály a SortedMap interfész implementációja, vagyis egy rendezett kulcsokat tartalmazó térkép. Mint azt az osztály neve is mutatja, fa adatszerkezzel implementálja a SortedMap-et. A kulcsoknak összehasonlíthatóknak kell lenniük.

Konstruktorok, metódusok

- TreeMap()
- TreeMap(Comparator c)

Létrehoz egy üres, rendezett térképet. Az első esetben a rendezettség természetes lesz, a második esetben a paramétere szerinti.

- TreeMap(Map m)
- TreeMap(SortedMap m)

Létrehoz egy rendezett térképet természetes rendezettséggel, és betölti az m térkép adatait.

Feladat – TreeMapUser

Egy internetszolgáltató tárolni akarja a felhasználók adatait: a felhasználó nevét, jelszavát és a legutóbbi látogatásának időpontját. A használati esetek a következők:

- Új felhasználó felvitelle
 - Felhasználók számának kiírása
 - Felhasználónevek listázása ábécé szerint
 - A jelszó és a legutóbbi látogatás kiírása a felhasználónév alapján
- A használati eseteket hajtsuk végre egymás után!

Forráskód

```

import java.util.*;
import java.text.DateFormat;

class User { //1
    private String userName; // felhasználónév
    private String password; // jelszó
    private Date lastVisited; // a legutóbbi belépés időpontja

    public User(String userName, String password) {
        this.userName = userName;
        this.password = password;
        this.lastVisited = new Date(); // mai dátum
    }
    public User(String userName) {
        this(userName, "");
    }
    public String getUserName() { return userName; }
    public String getPassword() { return password; }
    public String getLastVisited() {
        return DateFormat.getDateInstance().
            format(lastVisited);
    }
    public void setLastVisited() {
        lastVisited = new Date();
    }
    public String toString() {
        return userName+", "+password+", "+getLastVisited();
    }
}

public class TreeMapUser {
    public static void main(String[] args) {
        TreeMap users = new TreeMap(); //2

        users.put("Nagybuta", new User("NagyButa", "netuddmeg"));
        users.put("Bendegúz", new User("Bendegúz", "anyukád"));
        users.put("Erzsébet", new User("Erzsébet", "kimást"));
        users.put("Naspolya", new User("Naspolya", "eddmeg"));

        System.out.println("Felhasználók száma: " +
            users.size()); //3
    }
}

```

```

// Felhasználónevek listázása:
System.out.println("\nFelhasználónevek:");
Iterator iter = users.keySet().iterator();
while (iter.hasNext())
    System.out.println(iter.next()); //4

// Felhasználók adatai:
System.out.println("\nFelhasználók adatai:");
Collection values = users.values();
iter = values.iterator();
while (iter.hasNext())
    System.out.println(iter.next()); //5

// Megadott felhasználó jelszavának kiírása:
String userName = "Bendegúz";
User user = (User) users.get(userName); //6
if (user == null)
    System.out.println("\nNincs " + userName);
else {
    System.out.println("\n" + userName +
        " jelszava: " + user.getPassword() +
        ", legutóbbi látogatása: " + user.getLastVisited());
}
}
}
}

```

A program futása

Felhasználók száma: 4
Felhasználónevek:
Bendegúz
Erzsébet
Nagybuta
Naspolya
Felhasználók adatai:
Bendegúz, anyukád, 2004.07.31. 9:37:27
Erzsébet, kimást, 2004.07.31. 9:37:27
NagyButa, netuddmeg, 2004.07.31. 9:37:27
Naspolya, eddmeg, 2004.07.31. 9:37:27
Bendegúz jelszava: anyukád, legutóbbi látogatása: 2004.07.31. 9:37:27

A program elemzése

- ◆ //1: A felhasználónak készítünk egy osztályt. A `users` térkép értékei majd `User` típusú objektumok lesznek.
- ◆ //2: A felhasználókat rendezett térképben (`TreeMap`) tároljuk. Mindjárt létre is hozunk négy felhasználót, és betesszük őket a térképbe. A felhasználóobjektum kulcsa az objektum egyik adata, a `userName`.

- ◆ //3: A `users.size()` adja meg a térkép bejegyzéseinek a számát.
- ◆ //4: A felhasználónevek listázásához a térkép kulcskollekciójától (`users.keySet()`) kérünk egy iterátort, majd azon végighaladva kilistázzuk a kulcsokat (felhasználóneveket).
- ◆ //5: Most az értékek kollekciójától (`users.values()`) kérünk egy iterátort, és kilistázuk az értékeket, vagyis a `User` objektumokat.
- ◆ //6: Elkérjük a térképtől a megadott kulcsnak tartozó értéket (`users.get(userName)`) – egy felhasználóobjektumot –, és ha volt ilyen kulcs, akkor kiírjuk a felhasználóobjektum adatait.

Tesztkérdések

- 21.1. A következő pontokban egy-egy interfész-osztály párost talál. Jelölje be azokat a pontokat, amelyekben a jobb oldali osztály implementálja a bal oldali interfészt!
- Map - TreeSet
 - List - LinkedList
 - Collection - TreeSet
 - SortedMap - TreeMap
- 21.2. Jelölje be az összes helyes állítást!
- A kollekcióban lehet primitív adatokat is tárolni.
 - Minden kollekció inicializálható egy tetszőleges másik kollekcióval.
 - A térkép egy kulcsokat tároló kollekció.
 - Minden kollekció implementálja az `Iterator` interfészt.
- 21.3. Jelölje be az összes helyes állítást!
- A `Set`-nek minden implementációja rendezett.
 - A `TreeSet` implementálja a `Set` interfészt.
 - A `SortedSet` két objektuma lehet egyenlő.
 - A `List` két objektuma lehet egyenlő.
- 21.4. Jelölje be az összes helyes állítást!
- A `Set` elemei indexelhetők.
 - A `Vector` láncolt listával implementálja a `List` interfészt.
 - Egy `Iterator` objektum segítségével a kollekció elemei többször is bejárhatók.
 - A `Comparator` segítségével kívülről is megadható egy `TreeSet` rendezettsége.
- 21.5. Jelölje be az összes helyes állítást!
- A hasítókód meghatározása a hasítótáblába betett objektum feladata.
 - Két egyenlő objektumnak ugyanazt a hasítókódot kell generálnia.
 - A hasítótáblában két egyenlő objektum nem kerülhet ugyanabba a csoportba.
 - A hasítófüggvény megírásában ügyelni kell arra, hogy a kód értéke ne legyen nagyobb, mint a kapacitás.

Feladatok

- 21.1. **(A)** Kérjen be számokat a felhasználótól! Jelenítse meg a számokat rendezetten, mindegyiket csak egyszer! (*SzamGyujt.java*)
- 21.2. **(B)** A program paramétere adja meg, hogy hányas lottót játszunk (Ha például ötös lottót, akkor a felhasználónak 5 számot kell tippelnie). A program először kérje be a nyerő lottószámokat. Addig kérje, amíg be nem ütnek 5 különböző számot! Ezután dolgozza fel az asztalon megszámlálatlanul heverő lottószelvényeket! A szelvény bevitelekor írja ki a találatok számát, a program végén pedig készítsen statisztikát: hány darab volt a megfelelő találatokból! (*LottoSzelvenyek.java*)
- 21.3. **(A)** Megkérdezzük gyermekünket, hogy milyen ajándékokat kér karácsonyra. Üssük be a kívánságlistáját a számítógépbe! Végül írjuk ki az ajándékokat ábécérendben! Előfordulhat persze, hogy a gyermek egy ajándékot kétszer mond, de ha így történt, ha nem, minden ajándék csak egyszer szerepeljen a listán! (*Ajandekok.java*)
- 21.4. **(B)** Kérje be személyek adatait: név, cím stb. Két egyforma név nem szerepelhet a nyilvántartásban! Keresse meg a megadott névhez tartozó többi adatot! Írja ki az összes nevet! (*SzemelyNyilv.java*)

I. OBJEKTUMORIENTÁLT TECHNIKÁK

1. Csomagolás, projektkezelés
2. Örökítődés
3. Interfészek, belső osztályok
4. Kivételkezelés

II. GRAFIKUS FELHASZNÁLÓI INTERFÉSZ

5. A felhasználói interfész felépítése
6. Elrendezésmenedzserek
7. Eseményvezérelt programozás
8. Swing-komponensek
9. Grafika, képek
10. Alacsony szintű események
11. Belső eseménykezelés, komponensgyártás
12. Applet

III. ÁLLOMÁNYKEZELÉS

13. Állományok, bejegyzések
14. Folyamok
15. Közvetlen hozzáférésű állomány

IV. VEGYES TECHNOLÓGIÁK

16. Rekurzió
17. Többszálú programozás
18. Nyomtatás
19. Hasznos osztályok

V. ADATSZERKEZETEK, KOLLEKCIÓK

20. Klasszikus adatszerkezetek
21. Kölcsönös keretrendszer

FELADATOK

Fel

FÜGGELÉK

- A tesztkérdések megoldása
Irodalomjegyzék
Tárgymutató



Egy feladatot olyan mélységen kell megtervezni és dokumentálni, hogy az a célközönséget hozzásegítse a megoldás megértéséhez. Az ablak tulajdonosi hierarchiájának felrajzolása például a kezdő programozók dolgát könnyíti meg; a profiknak már csak ritkán van rá szükségük.

Ezzel a feladattal kb. 2 óra alatt lehet végezni. A szükséges részleteket nyugodtan nézze ki a könyvből!

1. feladat: Témák rögzítése

Feladatspecifikáció

Készítsen egy alkalmazást témák listában való felvitelére és a lista „kézi” rendeztésére. Használati esetek:

- Új téma felvitele: Bekérünk egy új témát. A téma a lista utolsó eleme lesz.
- Téma törlése: A kijelölt listaelem törlődik.
- Téma feljebb tolása: A kiválasztott elemet eggyel feljebb toljuk a listában.
- Téma lejjebb tolása: A kiválasztott elemet eggyel lejjebb toljuk a listában.

Ha egy funkciónak éppen nincs értelme, akkor a gomb legyen szürke!

Megszorítások:

- ◆ A listában mindenkor csak egy elem van kiválasztva; persze ha nincs elem, akkor egy sem.
- ◆ Magyar legyen a párbeszédablak! (Egyszerű: használja a `java.util.JOptionPanel` helyett az `extra.hu.HuOptionPane-t!`)

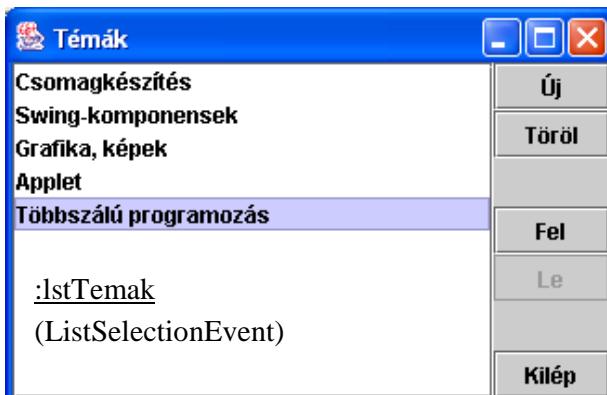
Programterv

A felhasználi interfész terve

:Temak

(ActionListener – btUj, btTorol, btFel, btLe, btKilep)

(ListSelectionListener – lstTemak)



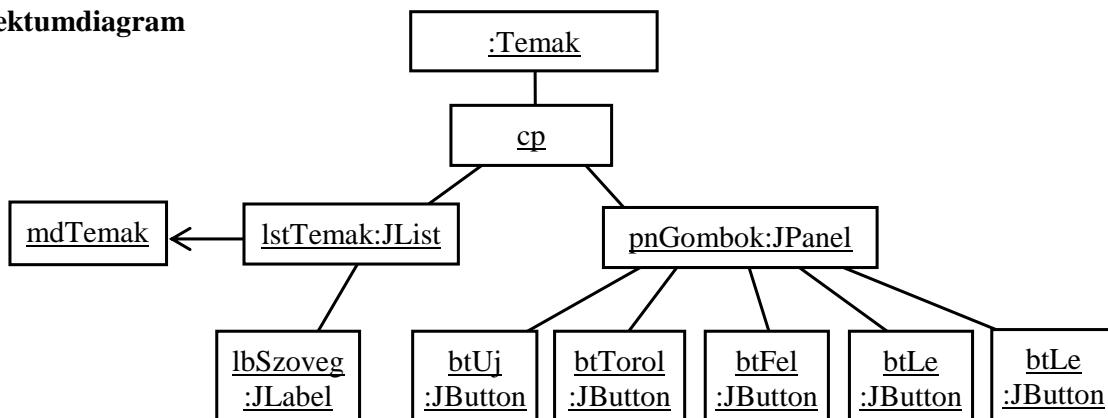
btUj (ActionEvent)

btTorol (ActionEvent)

btFel (ActionEvent)

btLe (ActionEvent)

btKilep (ActionEvent)

Objektumdiagram**A használati esetek megtervezése, átgondolása**

Ha egy gombot értelmetlenség lenne kiválasztani, akkor letiltjuk. A gombokat minden akció alkalmával megvizsgáljuk (enableUpdate)

A használati esetek a gombok lenyomásával indulnak. A gombok a következők:

- ◆ btUj: Megjelenik egy szövegbeviteli dialógus. Ha a bevitel sikeres, akkor a szöveget – utolsó elemként – betesszük a listába, sőt lesz a kiválasztott elem.
- ◆ btTöröl: Az éppen kijelölt szöveget töröljük a listából. Ha maradt elem, akkor a most törölt elem utáni elemet választjuk ki, ha nincs ilyen elem, akkor az előzőt.
- ◆ btFel: Ha van hely az elem felett (nem ő a legfelső), akkor az elemet kitöröljük és eggyel feljebb beszűrjuk.
- ◆ btLe: Ha van hely az elem alatt (nem ő a legalsó), akkor az elemet kitöröljük és eggyel lejjebb beszűrjuk.

Forráskód

Name: Temak
 Directory: C:/javaprogramok/Mintaprogramok/Feladatok
 Required libraries: extra.hu

Temak.java

```

import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Temak extends JFrame
    implements ActionListener, ListSelectionListener {
    Container cp = getContentPane();
    // A lista modellje:
    DefaultListModel mdTemak = new DefaultListModel();
    JList lstTemak; // a lista

```

```
    JButton btKilep;
    JButton btUj;
    JButton btTorol;
    JButton btFel;
    JButton btLe; // gombok

    public Temak() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLocation(100,100);
        setTitle("Témák");

        lstTemak = new JList(mdTemak); // lista a tartalompanelre
        cp.add(new JScrollPane(lstTemak)); // lehet görgetni
        lstTemak.setSelectionMode( // csak egyet lehet kiválasztani
            ListSelectionModel.SINGLE_SELECTION);

        JPanel pnGombok = new JPanel(); // gombok felpakolása
        pnGombok.setLayout(new GridLayout(0,1));
        pnGombok.add(btUj = new JButton("Új"));
        pnGombok.add(btTorol = new JButton("Töröl"));
        pnGombok.add(new JLabel(" "));
        pnGombok.add(btFel = new JButton("Fel"));
        pnGombok.add(btLe = new JButton("Le"));
        pnGombok.add(new JLabel(" "));
        pnGombok.add(btKilep = new JButton("Kilép"));
        cp.add(pnGombok, "East");

        btUj.addActionListener(this); // figyelők felfűzése
        btTorol.addActionListener(this);
        btFel.addActionListener(this);
        btLe.addActionListener(this);
        btKilep.addActionListener(this);
        lstTemak.addListSelectionListener(this);
        enableUpdate();

        pack();
        show();
    }

    // Engedélyezések beállítása:
    void enableUpdate() {
        int n = lstTemak.getSelectedIndex();
        // Ha fent van, nem lehet feljebb:
        btFel.setEnabled(n>0);
        // Ha lent van, nem lehet lejjebb:
        btLe.setEnabled(n<mdTemak.getSize()-1);
        btTorol.setEnabled(n>=0); // törlés eng., ha van kijelölés
    }

    // Ha mozognak a listában, újraértékeljük az engedélyezéseket:
    public void valueChanged(ListSelectionEvent e) {
        enableUpdate();
    }
}
```

```

// A használati eset indítása:
public void actionPerformed(ActionEvent e) {
    // Új téma:
    if (e.getSource() == btUj) {
        String tema = JOptionPane.showInputDialog(this, "Új téma");

        if (tema != null) // ha beütnek valamit, tároljuk
            mdTemak.addElement(tema);
        lstTemak.setSelectedValue(tema, true); // elem kiválasztása
    }

    // Kiválasztott téma törlése:
    else if (e.getSource() == btTorol) {
        int poz = lstTemak.getSelectedIndex(); // kiv. elem indexe
        if (poz == -1) // üres a lista
            return;
        mdTemak.remove(poz); // poz. téma törlése
        if (mdTemak.getSize() != 0) { // ha van még téma
            if (poz == mdTemak.getSize()) // ha utolsó volt
                poz--;
            lstTemak.setSelectedIndex(poz); // elem kijelölése
        }
    }

    // Téma mozgatása eggyel feljebb:
    else if (e.getSource() == btFel) {
        int poz = lstTemak.getSelectedIndex();
        if (poz > 0) {
            mdTemak.add(poz - 1, mdTemak.remove(poz));
            lstTemak.setSelectedIndex(poz - 1);
        }
    }
    // Téma mozgatása eggyel lejjebb:
    else if (e.getSource() == btLe) {
        int poz = lstTemak.getSelectedIndex();
        if (poz < mdTemak.getSize() - 1) {
            mdTemak.add(poz + 1, mdTemak.remove(poz));
            lstTemak.setSelectedIndex(poz + 1);
        }
    }
    // Kilépés:
    else if (e.getSource() == btKilep)
        System.exit(0);
}

public static void main(String[] args) {
    new Temak();
}
}

```

Feladat: Bővítse a programot! A témákat lehessen valamilyen néven elmenteni, majd újra betölteni! A bővítéshez kb. 1 óra szükséges.

Ezt a feladatot kb. 3 óra alatt lehet elkészíteni.

2. feladat: Csempetervező

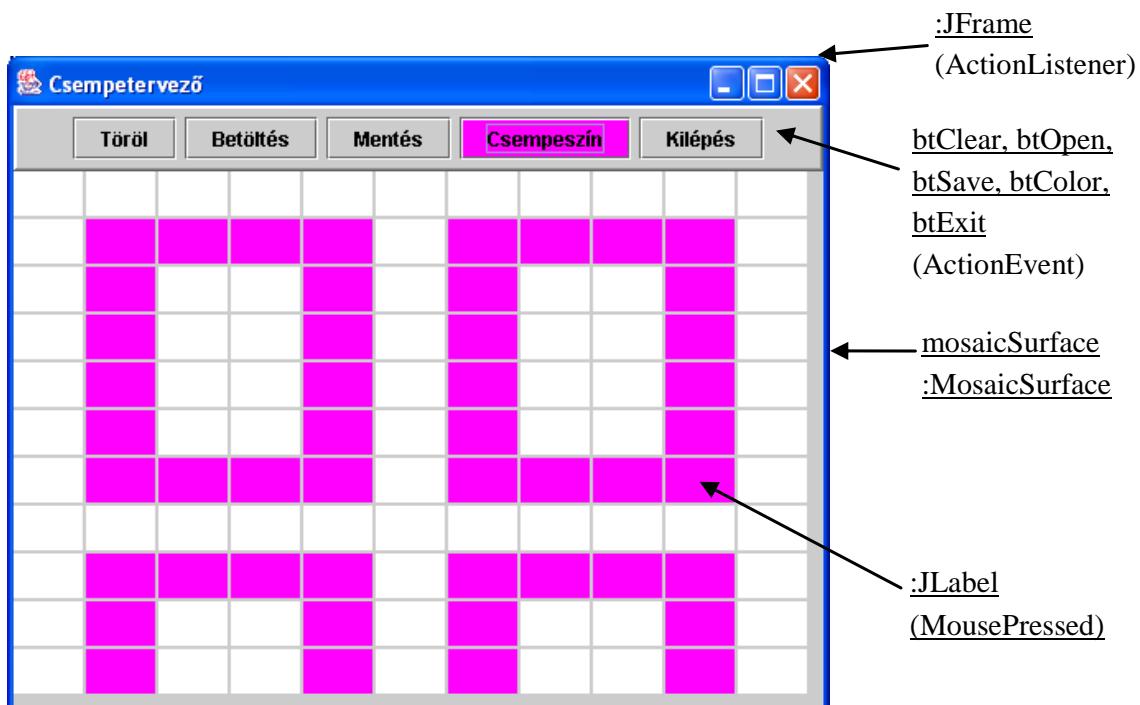
Feladatspecifikáció

Lakásfelújításkor a mesterek éppen csempeznek. Kétféle színű csempénk van, és szeretnénk őket valamilyen tetszetős mintában lerakni. Jó lenne, ha könnyedén készíthetnénk különböző variációkat, és este megmutathatnánk őket a család többi tagjának. Készítsünk egy egyszerű csempetervező programot!

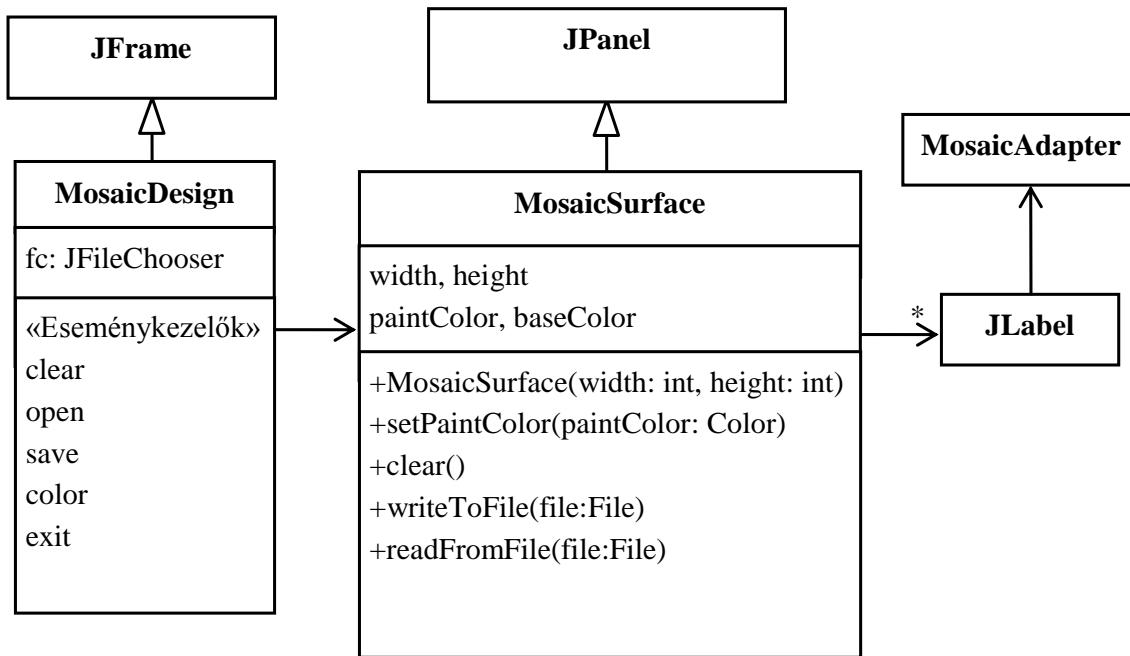
Programterv

A felhasználói interfész terve

Készítünk egy MosaicSurface (csempefelület) panelt; az készíti a mintát. A panelt megkérhetjük, hogy törölje le a felületét, töltse be a megadott állományban rögzített mintát stb. A parancsindító gombokat a keret tetejére tegyük, a csempetervező fölé, egy külön vezérlőpannelre.



Osztálydiagram



Osztályleírások

MosaicDesign

Keret, ő a fő osztály. Tartja a csempetervező panelt és vezérel. Eseménykezelők:

- ◆ `clear`, "Törlés": Letörli a csempetervező panelt (`mosaicSurface.clear`)
- ◆ `open`, "Betöltés": Állománykiválasztó párbeszédablakot jelenít meg. A kapott állományt átadja a csempefelületnek, hogy az töltse be belőle a mintát: `mosaicSurface.readFile`
- ◆ `save`, "Mentés": Állománykiválasztó párbeszédablakot jelenít meg. A kapott állományt odaadja a csempefelületnek, hogy az írja bele a mintát: `mosaicSurface.writeFile`
- ◆ `color`, "Csempeszín": Kér a felhasználótól egy színt – ehhez színválasztó párbeszédablakot kínál. A kapott színt átadja a csempetervezőnek: `mosaicSurface.setPaintColor`
- ◆ `exit`: "Kilépés": Kilép a programból.

MosaicSurface

Panel, rajta lehet megtervezni a csempét. Tartja a címkének (**JPanel**) vett csempéket. A felhasználó ezeken a csempéken kattint az egérrel. A címkéhez hozzáragasztunk egy

eseményadaptert, az majd kezeli az egérlenyomásokat: beszínezi a címkét az aktuális színnel. A MosaicAdaptert belső osztálynak vesszük, hogy ismerje az aktuális színt.

- ◆ **MosaicSurface(width: int, height: int)**

Létrehozza a csempekészítő felületet megadott szélességben és hosszúságban. Magára tesz width*height számú fehér „füles” címkét (az füleli az egérlenyomás eseményt).

- ◆ **setPaintColor(paintColor: Color)**

Beállítja a paintColort.

- ◆ **clear()**

Letörli a felületet. Bejárja a gyerekkomponenseit, és fehérre színezi őket.

- ◆ **writeToFile(file:File)**

Felírja a megadott állományba a csempék színobjektumait.

- ◆ **readFromFile(file:File)**

Visszaolvassa a megadott állományból a csempék színobjektumait. Akkor sincs baj, ha az olvasás nem sikerül: csak nem jelenik meg a minta.

Forráskód

Projekt

```
Name: MosaicDesign
Directory: C:/javaprog/Mintaprogramok/Feladatok
Required libraries: -
```

MosaicSurface.java

```
package mosaicdesign;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class MosaicSurface extends JPanel {
    int width = 20, height = 20; // alapértelmezés
    Color paintColor = Color.GRAY;
    Color baseColor = Color.WHITE;

    class MosaicAdapter extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
            JLabel lb = (JLabel) e.getSource();
            lb.setBackground(paintColor);
        }
    }
}
```

```
public MosaicSurface(int width, int height) {
    this.width = width;
    this.height = height;
    setLayout(new GridLayout(width, height, 2, 2));
    JLabel lb;
    for (int i = 0; i < width * height; i++) {
        lb = new JLabel(" ");
        lb.setBorder(BorderFactory.createEmptyBorder());
        lb.setBackground(Color.white);
        lb.setOpaque(true);
        lb.addMouseListener(new MosaicAdapter());
        add(lb);
    }
}

// minden csempe színe baseColor lesz:
public void clear() {
    Component[] labels = this.getComponents();
    for (int i = 0; i < labels.length; i++) {
        labels[i].setBackground(baseColor);
    }
}

// Aktuális csempeszín beállítása.
void setPaintColor(Color paintColor) {
    this.paintColor = paintColor;
}

// Saját szerializáció (kiírás):
public void writeToFile(File file) throws Exception {

    try {
        ObjectOutputStream out =
            new ObjectOutputStream(new FileOutputStream(file));
        for (int i = 0; i < this.getComponentCount(); i++) {
            out.writeObject(this.getComponent(i).getBackground());
        }
        out.close();
    }
    catch (Exception ex) {
        System.out.println(ex);
    }
}

// Saját szerializáció (beolvasás):
public void readFromFile(File file) {
    try {
        ObjectInputStream in =
            new ObjectInputStream(new FileInputStream(file));
        for (int i = 0; i < width * height; i++) {
            this.getComponent(i).setBackground(
                (Color) in.readObject());
        }
    }
}
```

```
        catch (Exception ex) {
            System.out.println(ex);
            // Valamennyit berakott, az jó nekünk.
        }
    }
}
```

MosaicDesign.java

```
package mosaicdesign;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import extra.util.*;
import java.io.*;

public class MosaicDesign extends JFrame implements
ActionListener {
    Container cp = getContentPane();
    JFileChooser fc = new JFileChooser();
    JPanel pnControl = new JPanel();
    JButton btClear = new JButton("Töröl");
    JButton btOpen = new JButton("Betöltés");
    JButton btSave = new JButton("Mentés");
    JButton btColor = new JButton("Csempeszín");
    JButton btExit = new JButton("Kilépés");
    MosaicSurface mosaicSurface = new MosaicSurface(11, 11);
    Color color = new Color(240, 220, 60);

    public MosaicDesign() {
        setTitle("Csempetervező");
        setSize(500, 400);
        fc.setCurrentDirectory(new File("."));
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pnControl.setBorder(BorderFactory.createRaisedBevelBorder());
        btClear.addActionListener(this);
        btOpen.addActionListener(this);
        btSave.addActionListener(this);
        btColor.addActionListener(this);
        btExit.addActionListener(this);
        cp.add(pnControl, BorderLayout.NORTH);
        pnControl.add(btClear);
        pnControl.add(btOpen);
        pnControl.add(btSave);
        pnControl.add(btColor);
        pnControl.add(btExit);
        cp.add(mosaicSurface);
        show();
        setLocationRelativeTo(null);
    }
}
```

```
public void actionPerformed(ActionEvent ae) {  
    if (ae.getSource() == btClear) {  
        mosaicSurface.clear();  
    }  
    else if (ae.getSource() == btOpen) {  
        if (fc.showOpenDialog(this) != JFileChooser.APPROVE_OPTION)  
            return;  
        File f = fc.getSelectedFile();  
        mosaicSurface.readFromFile(f);  
    }  
    if (ae.getSource() == btSave) {  
        if (fc.showSaveDialog(this) != JFileChooser.APPROVE_OPTION)  
            return;  
        File f = fc.getSelectedFile();  
        mosaicSurface.writeToFile(f);  
    }  
    else if (ae.getSource() == btColor) {  
        color = JColorChooser.showDialog(this,  
                                         "A következő csempe színe", color);  
        btColor.setBackground(color);  
        mosaicSurface.setPaintColor(color);  
    }  
    else if (ae.getSource() == btExit) {  
        System.exit(0);  
    }  
}  
  
public static void main(String[] args) {  
    MosaicDesign mosaicDesign = new MosaicDesign();  
}
```

Feladat: Fejlessze tovább a programot! Betöltéskor és elmentéskor a felhasználó, ha akarja, menthesse el az éppen szerkesztett panelt. Legyen a programban "Mentés másként" funkció is!

Tipp: Sok energiát takaríthat meg az extra.hu.FileManager felhasználásával!

3. Feladat – Címjegyzék

Feladatspecifikáció

Készítsünk címjegyzéket! Egy bejegyzéshez a következő adatok tartoznak majd: Név, Irányítószám, Város, Cím, Telefonszám.

A személyek adatait maradandóan (perzisztens módon) tároljuk. A már felvitt személyek neve kerüljön egy görgethető listába. Egyforma nevek is előfordulhatnak. Tegyük lehetővé:

- Új személyek adatainak fevitelét;
- Egy már felvitt személy adatainak módosítását. Bármelyik adatot lehessen módosítani;
- Egy már felvitt személy adatainak törlését. A törlés előtt kérjünk megerősítést.

Az adatok felviteléhez és módosításához használunk dialógusablakot!

Készítsünk az alkalmazáshoz segítség- és névjegydialogust is!

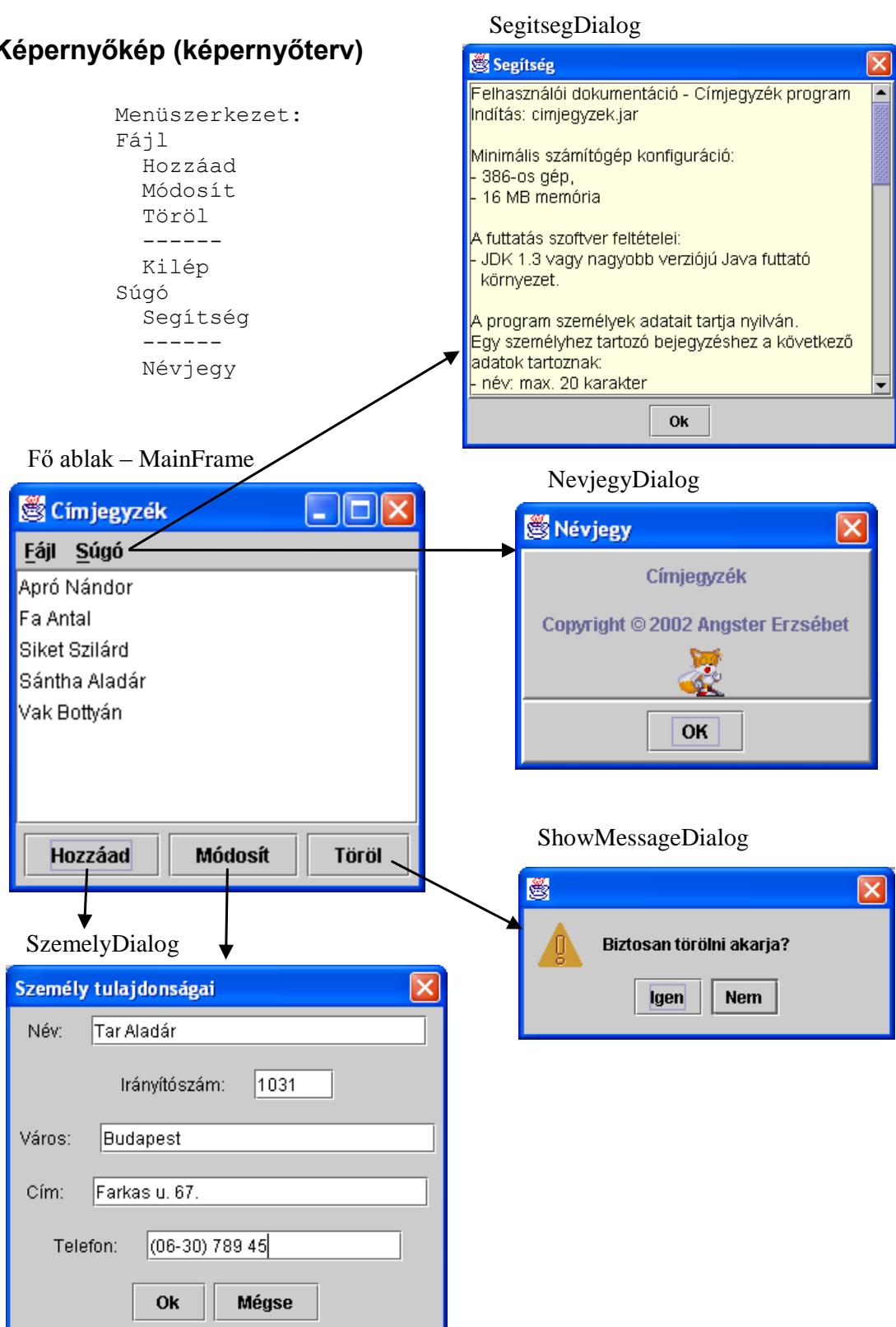
Programterv

Megtervezzük az alkalmazás fő ablakát, a használati eseteket és a használati esetek végrehajtásához szükséges további ablakokat (3.1. ábra). A fő ablakban névsorba szedve megjelenítjük az felvitt személyek nevét. A listát görgethetjük, nézegethetjük. Használati esetek:

- ◆ *Új személy hozzáadása, adatainak felvitele.* Megjelenik a "Személy tulajdonságai" dialógusablak (SzemelyDialog).
- ◆ *A kijelölt személy törlése.* Megjelenik egy megerősítést kérő dialógusablak (ShowMessageDialog). Ha megerősítést kapunk, akkor a személyt töröljük a nyilvántartásból.
- ◆ *A kijelölt személy módosítása.* Megjelenik a Személy tulajdonságai dialógusablak (SzemelyDialog). Bármelyik adatot lehet majd módosítani.
- ◆ *Segítsékkérés.* Megjelenik a Segítség dialógusablak (SegitsegDialog).
- ◆ *Névjegykérés.* Megjelenik a Névjegy dialógusablak (NevjegyDialog).
- ◆ *Kilépés.* Vége a programnak.

Megjegyzés: Amit itt közlünk, az nem profi megoldás; csak az alaptechnikák gyakorlására szolgál.

Képernyőkép (képernyőterv)



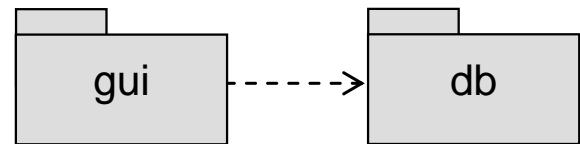
3.1. ábra. A Címjegyzék alkalmazás felhasználói interfész terve

Az osztályokat két csomagban helyezzük el, a következőképpen:

```

gui
  Start
  MainFrame
  SzemelyDialog
  SegitsegDialog
  NevjegyDialog
db
  Szemely
  Szemelyek
  FixedLengthIO
  KeyEntry

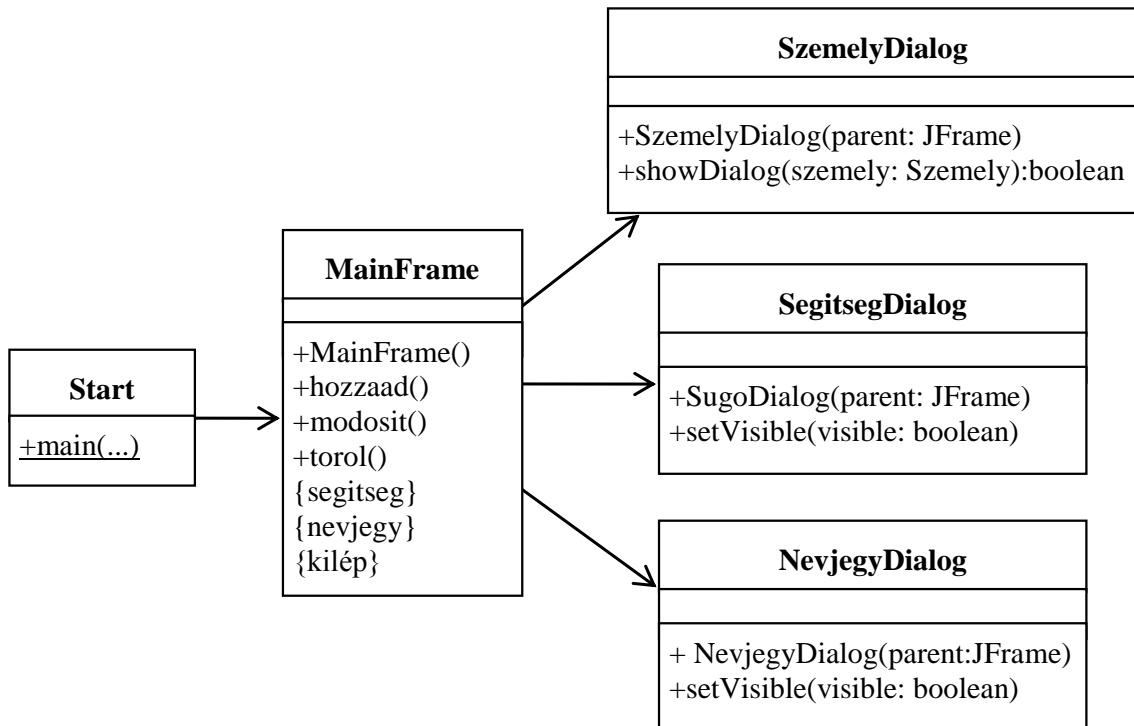
```



A `gui` csomagban lesznek a felhasználói interfésszel kapcsolatos osztályok; a `db` (némi túlzás-sal database) csomagban pedig az adatok tárolásával kapcsolatosak. A `gui` osztályai használják a `db` osztályait, de azok nem használják őket – a `db` tehát a `gui`-től független csomag.

A `gui` csomag

A `gui` csomag osztálydiagramját a 3.2. ábra mutatja.



3.2. ábra. A `gui` csomag osztálydiagramja

Osztályleírások

Start

Az alkalmazás indító osztálya. Létrehoz egy MainFrame osztályú objektumot.

MainFrame

Az alkalmazás fő ablaka; ezen át működtethetők a fő használati esetek. Bizonyos használati esetekhez – a hozzaad, modosit és torol esethet – jól megnevezhető metódus is tartozik, más használati esetekhez nem: azokat azonnal „elintézzük” az eseménykezelőben. Ez utóbbi használati eseteket megjegyzésben tüntetjük fel a terven – ilyenek a segítség, névjegy és a kilép.

SegitsegDialog

Az alkalmazás segítség (help) dialógusa. Nem modális, vagyis munka közben nyugodtan tanulmányozhatjuk. Létrehozás után a setVisible(true) metódussal lehet láthatóvá tenni. Az Ok gomb lenyomására a dialógus eltűnik (de nem szűnik meg). A segítség szövegét az "adatok/segitseg.txt" egyszerű szöveges állományból hozza be, a segítség szövege tehát könyen szerkeszthető.

NevjegyDialog

Az alkalmazás modális névjegydialogusa. Létrehozás után a setVisible(true) metódussal lehet láthatóvá tenni. Az Ok gomb lenyomására a dialógus eltűnik (de nem szűnik meg).

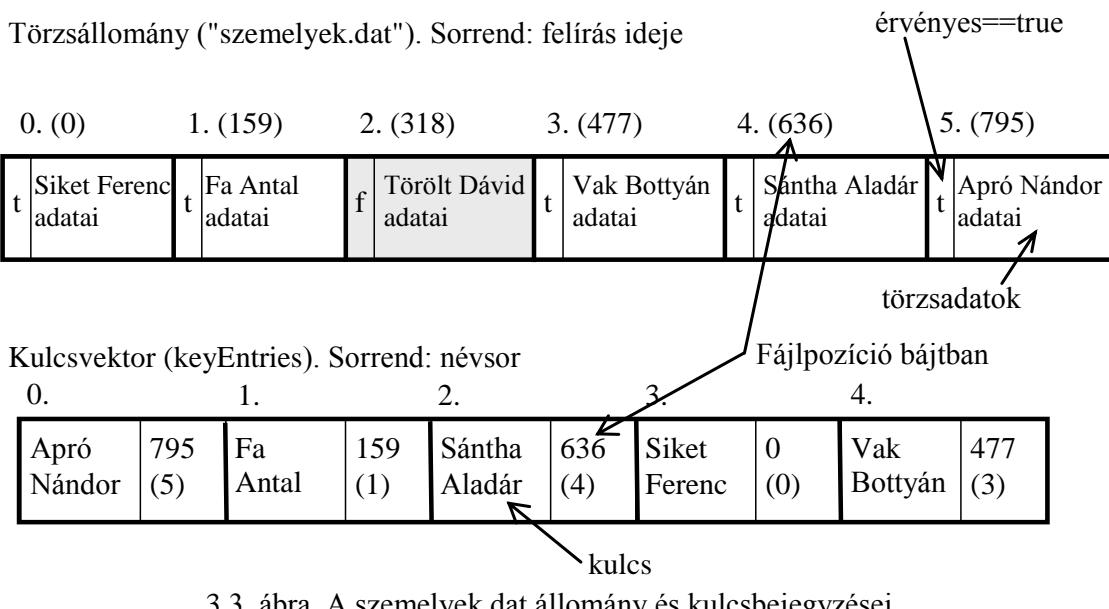
SzemelyDialog

Modális dialógus, feladata a szemely objektum kitöltése, illetve módosítása. A showDialog kiteszi a paraméterben megadott szemely objektum adatait a dialógusablakba. A mezőket szerkeszteni lehet. Ok lenyomására true lesz a visszatérési érték, és a szemely objektum módosul; Mégse lenyomására a visszatérési érték false lesz, és a szemely objektum változatlan marad.

db csomag

Az adatok tárolására indexelt szekvenciális állományt használunk (3.3. ábra). A személyek törzsadatait egy **közvetlen elérésű állományba** fogjuk felírni, minden az utolsó helyre. Egy személy adatai csak akkor érvényesek, ha első adata érvényes==true. Egy személy adatainak törlésekor ezt az értéket false-ra állítjuk majd. A memoriában külön nyilvántartunk egy kulcsbejegyzéset tartalmazó vektort, hogy egy-egy személy kereséséhez ne kelljen az egész állományt végigolvasnunk. minden kulcsbejegyzésban lesz egy kulcs és egy állománypozíció – a kulcshoz tartozó törzsadatok helye. Például:

Sántha Aladár kulcsvektorbeli indexe 2, állománypozíciója 636 (= 4 * 159). Az állománypozíció úgy adódik, hogy az egy személyhez tartozó adatok hossza 159 (érvényes:1, név:40, irszám:8, város:40, cím:40, telefon:30 bájt), és Sántha Aladár a törzsfájlban a 4. személy.



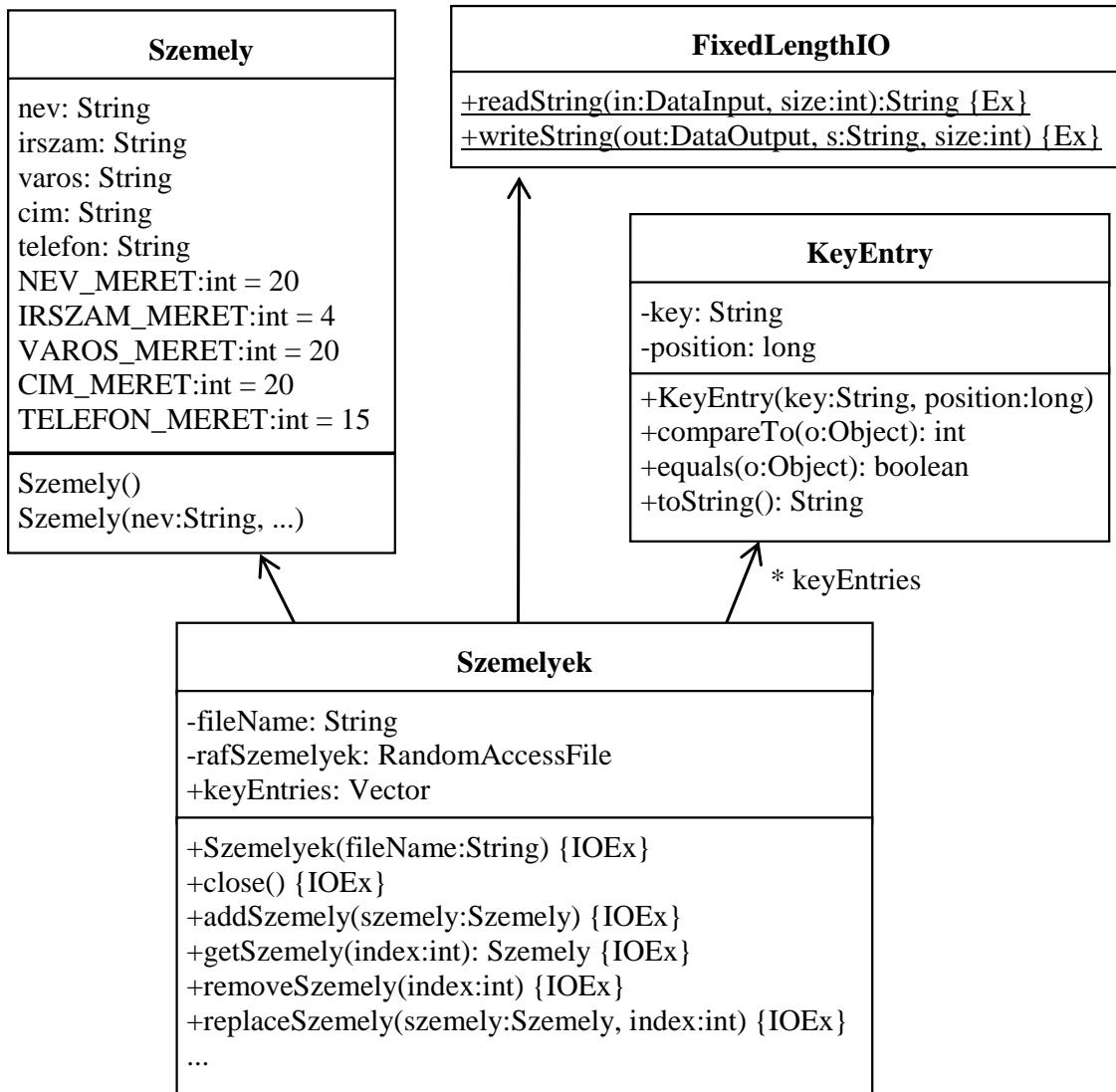
A név szerint rendezett kulcsvektorban könnyen megtaláljuk a keresett személyt; ezután már csak ki kell nyúlnunk érte a lemezre. Törléskor a személyt egyszerűen kitöröljük a kulcsvektorból, a törzsadat érvényes változóját pedig false-ra állítjuk. Az érvényesség karbantartásával a törzsállomány könnyen helyreállítható a kulcsvektor elvesztése vagy elromlása esetén. A törölt objektumok „ottmaradásukkal” igen erősen megduzzasztják az állományt; ezért a program befejezésekor minden besűrítjük: végigolvassuk, és csak az érvényes rekordokat hagyjuk meg.

A db csomag osztálydiagramját a 3.4. ábra mutatja.

Osztályleírások

FixedLengthIO

Általános célú osztály. Feladata egy rögzített hosszúságú String beolvasása, illetve kiírása az állományba. Erre azért van szükség, mert a közvetlen elérésű állományban a felvitt adatoknak egyforma hossznak kell lenniük – csak így számítható ki a tárolt adat pozíciója.



3.4. ábra. A db csomag osztálydiagramja

KeyEntry

Általános célú, kulcsbejegyzést tároló osztály. key a kulcsbejegyzés kulcsa; position a kulcsshoz tartozó törzsadat pozíciója (bájtsorszáma) a lemezen. Két kulcsbejegyzés akkor egyenlő, ha kulcsuk és pozíciójuk is egyenlő. A kulcsbejegyzések a kulcs szerint vannak rendezve.

Szemely

A címjegyzékben nyilvántartandó személyek adatai. Az itt megadott mezők kerülnek ki a lemezre, mégpedig a statikus MERET adatok szerinti hosszúságban. Ezzel a lemezre mentett személyi adatok minden egyenlő hosszúak lesznek, és ki lehet számítani a személyi adatok pozícióját.

Szemelyek

Ez az osztály intézi a személyek adatainak elmentését és a visszaolvasásukat. Használójának nem kell túl sokat tudnia a működéséről – elég, ha megadja az adatokat tartalmazó állomány nevét és összeállítja a Szemely objektumot; a többi elvégzi az addSzemely, a removeSzemely és a replaceSzemely. Ezek a metódusok mind potenciális „IOException-féshetők”, hiszen mindegyik a lemezen matat. A kivételeket az osztály sehol sem kezeli, minden továbbadja őket a hívónak (`throws`). Így a kivétel típusának megfelelően ízlésünk szerint jeleníthetjük meg az információt, illetve a teendőket a felhasználói interfészen.

Forráskód

A forráskód nagyon terjedelmes, ezért nem adjuk itt meg, csak a mellékletben (bőven elláttuk megjegyzésekkel). Ízelítőként nézzünk meg a KeyEntry osztály kódját!

Projekt

Name: Cimjegyzek
 Directory: C:/javaprogramok/Mintaprogramok/Feladatok
 Required libraries: -

KeyEntry.java

Kulcs osztály egy egyedhez. Tartalmazza a kulcsot és az egyed pozíóját a lemezen:

```
package db;
import java.util.*;

// Egy kulcsbejegyzés:
public class KeyEntry implements Comparable {
    public String key;
    public long position;

    public KeyEntry(String key, long position) {
        this.key = key;
        this.position = position;
    }

    public int compareTo(Object o) {
        return key.compareTo(((KeyEntry)o).key);
    }

    public boolean equals(Object o) {
        return key.equals(((KeyEntry)o).key) &&
            position == ((KeyEntry)o).position;
    }

    public String toString() {
        return key;
    }
}
```

4. Esettanulmányok

Az **OOTPJava2/Esettanulmanyok** könyvtárban megtalálja a következő feladatok teljes feladatspecifikációját, a hozzájuk tartozó fejlesztői és felhasználói dokumentációt, és forráskódot.

- ◆ Tilitoli: A Tili-toli logikai játékot szimulálja a számítógépen!
- ◆ KisDraw: Egyszerű szövegszerkesztő program!
- ◆ KissDraw: Egyszerű rajzoló program!
- ◆ Mikro: A mikrohullámú sütöt szimulációja számítógépen!

Használja a javalib csomagjait, osztályait! Nagyon hasznos komponensek (minták) a következők:

- ◆ az extra.util csomag FileManager osztálya felhasználható bármely Savable objektum lemezre való mentésére és lemezről való betöltésére. Szükség esetén kommunikál is a felhasználóval (mentés másként, el akarja-e menteni stb.).
- ◆ az extra.hu csomag HuOptionPane osztálya magyar dialógusokat definiál a javax.swing.JOptionPane osztály mintájára. A HuFileChooser magyarul „beszélő” JFileChooser, a HuFileManager pedig magyar FileManager.

Ezeknek az osztályoknak, komponenseknek a dokumentációját megtalálja a könyv mellékletében. **Tanulmányozza a Esettanulmányok könyvtárát!**

További feladatötletek

- 4.1. Készítse el az operációs rendszerben is használatos kalkulátort!
- 4.2. Készítsen egy kiss (Keep It Simple and Stupid) Java környezetet. Ebben lehessen Java forráskódot betölteni, kimenteni, fordítani és futtatni. A fordítás eredménye egy alsó, kimeneti ablakban jelenjen meg. Ha a felhasználó rákattint a szövegszerkesztőben egy hibasorra, akkor a kurzor álljon rá a javítandó sorra!
- 4.3. Készítsen egy tetszőleges kártyajátékot szimuláló programot!

I. OBJEKTUMORIENTÁLT TECHNIKÁK

1. Csomagolás, projektkezelés
2. Örökítés
3. Interfészek, belső osztályok
4. Kivételkezelés

II. GRAFIKUS FELHASZNÁLÓI INTERFÉSZ

5. A felhasználói interfész felépítése
6. Elrendezésmenedzserek
7. Eseményvezérelt programozás
8. Swing-komponensek
9. Grafika, képek
10. Alacsony szintű események
11. Belső eseménykezelés, komponensgyártás
12. Applet

III. ÁLLOMÁNYKEZELÉS

13. Állományok, bejegyzések
14. Folyamok
15. Közvetlen hozzáférésű állomány

IV. VEGYES TECHNOLÓGIÁK

16. Rekurzió
17. Többszálú programozás
18. Nyomtatás
19. Hasznos osztályok

V. ADATSZERKEZETEK, KOLLEKCIÓK

20. Klasszikus adatszerkezetek
21. Kölcsönös keretrendszer

FELADATOK



FÜGGELÉK

- A tesztkérdések megoldása
Irodalomjegyzék
Tárgymutató

Függ

A tesztkérdések megoldása

1. Csomagolás, projektkezelés

1. cd	2. b	3. acd	4. d	5. bc	6. ab	7. b	8. a	9. bcd
-------	------	--------	------	-------	-------	------	------	--------

2. Öröklődés

1. bd	2. c	3. bc	4. acd	5. abc	6. a	7. c	8. bcd	9. bd	10. ad
11. cd									

3. Interfészek, belső osztályok

1. cd	2. acd	3. acd	4. d	5. ab	6. d
-------	--------	--------	------	-------	------

4. Kivételkezelés

1. cd	2. abc	3. ad	4. ab	5. b	6. ad
-------	--------	-------	-------	------	-------

5. A felhasználói interfész felépítése

1. abd	2. b	3. d	4. abd	5. bc	6. d	7. b
--------	------	------	--------	-------	------	------

6. Elrendezésmenedzserek

1. abc	2. c	3. cd	4. ac	5. c	6. d
--------	------	-------	-------	------	------

7. Eseményvezérelt programozás

1. ab	2. bd	3. abc	4. c	5. ad	6. ad	7. bc
-------	-------	--------	------	-------	-------	-------

8. Swing-komponensek

1. c	2. acd	3. ac	4. ac	5. bd	6. acd	7. bcd	8. b	9. d
------	--------	-------	-------	-------	--------	--------	------	------

9. Grafika, képek

1. ad	2. d	3. b	4. bcd	5. b	6. bd
-------	------	------	--------	------	-------

10. Alacsony szintű események

1. ad	2. c	3. bd	4. cd	5. b	6. c	7. bd	8. abc
-------	------	-------	-------	------	------	-------	--------

11. Belső eseménykezelés, komponensgyártás

1. ac	2. cd	3. d	4. abd
-------	-------	------	--------

12. Applet

1. c	2. abc	3. d	4. d	5. acd
------	--------	------	------	--------

13. Állományok, bejegyzések

1. bd	2. bcd	3. cd	4. b
-------	--------	-------	------

14. Folyamok

1. c	2. abd	3. ad	4. bd	5. b	6. abc
------	--------	-------	-------	------	--------

15. Közvetlen hozzáférésű állományok

1. bcd	2. c	3. bc	4. abc	5. b
--------	------	-------	--------	------

16. Rekurzió

1. bc	2. ad	3. bcd
-------	-------	--------

17. Többszálú programozás

1. ac	2. ab	3. abc	4. bcd
-------	-------	--------	--------

18. Nyomtatás

1. c	2. bcd	3. ad	4. abc	5. b
------	--------	-------	--------	------

19. Hasznos osztályok

1. b	2. bd	3. abd	4. abd	5. acd	6. ab
------	-------	--------	--------	--------	-------

20. Klasszikus adatszerkezetek

1. a	2. b	3. acd	4. d	5. bd
------	------	--------	------	-------

21. Kölleció keretrendszer

1. bcd	2. b	3. bd	4. d	5. ab
--------	------	-------	------	-------

Irodalomjegyzék

Magyarul:

- [1] Angster Erzsébet: Az objektumorientált tervezés és programozás alapjai (UML, Turbo Pascal, C++), 1997
- [2] Jamsa, Kris: Java
Kossuth könyvkiadó, 1996, fordítás
- [3] Nyékyné G. Judit (szerk.) et al.: Java 2 Útikalauz programozóknak
ELTE TTK Hallgatói Alapítvány, 2000
- [4] Vég Csaba - dr. Juhász István: Java - start!
Logos 2000, 1999

Angolul:

- [5] Booch, Grady: The Unified Modeling Language User Guide
Addison-Wesley, 1998
- [6] Bruce Eckel: Thinking in Java
Prentice Hall, 1999
- [7] Cay S. Horstmann, Gary Cornell: Core Java 1.2
Sun MicroSystems Press, 1999
- [8] Eriksson, Hans-Erik - Penker, Magnus: UML Toolkit
Wiley, 1998
- [9] Fowler, Martin: UML Distilled
Addison-Wesley, 1997
- [10] Gamma, Erich - Helm, Richard - Johnson, Ralph - Vlissides, John:
Design patterns
Addison-Wesley, 1997
- [11] Jacobson, Ivar: The Unified Software Development Process
Addison-Wesley, 1999
- [12] Khalid A. Mughal, Rolf W. Rasmussen: A Programmer's Guide to Java
Certification
Addison-Wesley, 2000

- [13] Kruchten, Philippe: The Rational Unified Process - An Introduction
Addison-Wesley, 1999
- [14] Rumbaugh, James: OMT Insights
SIGS Books, 1996
- [15] Rumbaugh, James: The Unified Modeling Language Reference Manuel
Addison-Wesley, 1999
- [16] Y. Daniel Liang: Introduction to Java Programming
Que E&T, 1999

Tárgymutató

A, Á

ablak, 230
AbstractButton, 187
AbstractDocument, 207,
208
absztrakt metódus, 57
absztrakt osztály, 57
absztrakt tárolók, 516
adatfolyam, 378, 394
adatmodell, 511
adatszerkezet, 512
asszociatív, 514
hálós, 515
hierarchikus, 515
konstrukciós műveletek,
513
szekvenciális, 514
szelekciós műveletek,
513
adatszerkezetek, 511
alignmentX, 130
alignmentY, 130
állomány, 353
szöveges állomány, 387,
388
véletlen elérésű
állomány, 415
állománymutató, 418
állományműveletek, 363
állománypozíció, 418
általános tömb, 521
applet, 325, 453
Applet osztály, 334
appletnéző, 332
appletviewer, 332
archive, 330
Archive Builder, 22, 24
ArrayList, 540, 557
AudioClip, 338

automatikus típuskonverzió
felfelé, 45
AWT, 105
osztályhierarchia, 110
AWTEvent, 162
AWT-szál, 441

B

background, 129
bájtfolyam, 377, 381
bejárhatóság, 278
belői eseménykezelés, 307
belői osztály, 75
betű, 124
billentyűesemény, 285
bináris fa, 529
biztonság, 346
blokkolt szál, 440
border, 111
Border, 130
BorderFactory, 130
BorderLayout, 112, 148
böngésző, 326, 330
BufferedInputStream, 398
BufferedOutputStream, 398
BufferedReader, 400
BufferedWriter, 400
ButtonGroup, 113, 198

C

Calendar, 484
catch, 96
célútvonal, 5
címke, 184
cirkuláris lista, 517
class path, 12
Cloneable, 497
code, 330
codebase, 330

Collection, 539
Collection interfész, 541
Collections Framework, 538
Color, 112, 124, 129
Comparator interfész, 549
ComponentEvent, 275
Container, 112, 133
controller, 204
country, 480
currentTimeMillis, 478
Cursor, 129

Cs

csmag, 3, 583
csomagolás, 3
csomagszintű, 10

D

DataInput, 395
DataInputStream, 396
DataOutput, 395
DataOutputStream, 396
Date, 477
DateFormat, 488
dátumformázás, 488
DefaultListModel, 217
démonszál, 441
dialógusablak, 182, 231
Dimension, 120, 130
dinamikus kötés, 50
dinamikus metóduskerés, 51
dinamikus típus, 43
direkt szervezés, 416
downcasting, 46

E, É

egéresemény, 293
egydimenziós tömb, 516

eljárásmóddell, 511
 elrendezésmenedzser, 141
 enabled, 130
 Error, 86
 esemény, 160
 alacsony szintű, 163, 273
 életútja, 307
 engedélyezése, 311
 magas szintű, 163
 eseményadapter, 175
 eseménydelegációs modell, 166
 eseményfigyelő, 169
 eseményforrás, 169
 eseménymaszk, 311
 eseményvezérelt program, 113
 eseményvezérelt
 programozás, 157
 esettanulmanyok, 588
 EventObject, 161
 Exception, 86
 extends, 33
 extra.util.FileManager, 588
 extra.util.HuFileManager, 588

F

fa, 527
 fa bejárása, 528
 fa magassága, 528
 fájdialógus, 367
 felhasználói felület, 169
 felhasználói interfész, 105, 571
 figyelő, 160
 File, 357
 file pointer, 418
 FileInputStream, 383
 FilenameFilter, 365
 FileOutputStream, 384
 FileReader, 390
 FileWriter, 390
 FilterInputStream, 396
 FilterOutputStream, 396
 finally, 96
 FlowLayout, 112, 144
 FocusEvent, 277, 280
 fok, 528
 fókuszbirtokos, 277
 fókuszesemény, 277
 fókuszlánc, 277

folyam, 377
 adatfolyam, 394
 bájtfolyam, 381
 beviteli, 378
 célhely, 378
 forráshely, 378
 karakterfolyam, 387
 kiviteli, 378
 mutató, 378
 objektumfolyam, 402
 pufferező folyam, 398
 szűrő, 378
 folyamat, 439
 Font, 112, 126, 129
 fordítás, 17
 foreground, 129
 forrásobjektum, 160
 forrásútvonal, 5, 12
 fő szál, 441
 futás alatti kötés, 50
 futtatás, 17
 futtatható szál, 440

G

görgetőlap, 210
 görgetősav, 221
 gráf, 512
 grafika, 251
 grafikus felhasználói
 interfész, 105
 Graphics, 113, 256, 460
 GregorianCalendar, 484
 GridLayout, 112, 146
 GUI, 105

Gy

gyorsrendezés, 436

H

halmaz, 543
 halott szál, 440
 hálózat, 531
 hanglejátszás, 338
 Hanoi tornyai, 433
 hashCode, 545
 HashSet osztály, 545
 Hashtable, 540
 Hashtable osztály, 561
 hasítási technika, 545

hasítókód, 545
 hasítótábla, 523
 használati eset, 572
 határmenti elrendezés, 148
 hiba, 85
 homokozó, 326
 HTML, 325
 HTML-kód, 329

I, Í

időeltolás, 482
 időpont, 477
 időzítő, 241
 Image, 263
 import, 7
 indexelt szekvenciális
 szervezés, 417
 inheritance, 32
 InputEvent, 287
 InputStream, 381
 InputStreamReader, 389
 instanceof, 45
 interface, 69
 interfész, 14, 69
 io csomag, 353
 irányított gráf, 531
 ISO3, 480
 Iterator, 539, 555
 iterátor, 554

J

JApplet, 334
 JAR, 3
 JAR fájl, 19
 Java Plug-In, 331
 javalib, 4
 JBuilder, 333
 JButton, 111, 188
 JCheckBox, 111, 194
 JCheckBoxMenuItem, 228
 JColorChooser, 235
 JComboBox, 111, 200
 JComponent, 111, 128
 JDialog, 112, 182, 231
 jellemző, 119
 jelölőmező, 194
 JFileChooser, 367
 JFrame, 112, 136
 JLabel, 111, 184
 JList, 111, 215
 JMenu, 111, 227

JMenuBar, 111, 225, 226
 JMenuItem, 111, 227
 JOptionPane, 217, 236
 JPanel, 112, 150
 JRadioButton, 111, 197
 JRE, 331
 JScrollPane, 111, 221
 JScrollPane, 111, 210
 JTextArea, 111, 211
 JTextField, 111, 206
 JTextField, 111, 208
 JVM, 25, 347, 445
 JWindow, 112, 230

K

karakterfolyam, 378, 387
 képek, 263
 képernyőkép, 582
 keresőfa, 529
 keret, 338
 késői kötés, 51
 kétirányú lista, 517
 KeyEvent, 285, 288
 kiegyensúlyozott fa, 528
 kimeneti útvonal, 12
 kiterjesztés, 32
 kivétel, 85
 ellenőrzött, 89
 kiváltása, 89
 lekezelése, 96
 nem ellenőrzött, 89
 saját, 99
 továbbadása, 94
 klónozás, 497
 kollekció, 541
 kollekció keretrendszer, 537, 538
 kombinált lista, 200
 komponens
 tulajdonosi hierarchia, 106
 komponensesemény, 275
 komponensgyártás, 316
 konstrukciós műveletek, 513
 konstruktorkok láncolása, 54
 konténer, 133
 koordinátarendszer, 122
 könyvtárstruktúra, 5
 környezet, 480
 közvetlen hozzáférés, 418
 külső program futtatása, 503

L

láncolt lista, 516
 language, 481
 láthatóság, 62
 LayoutManager, 143
 levél, 528
 LIFO, 525
 LinkedList, 540, 557
 List, 539
 List interfész, 544, 556
 lista, 215, 544
 ListIterator, 539
 ListIterator interfész, 557
 Locale, 480

M

Map, 539
 Map interfész, 559
 MediaTracker, 265
 megfigyelés, 491
 megfigyelhető objektum, 491
 megfigyelő objektum, 491
 mély klónozás, 499
 menüsor, 225
 méret, 119
 modális, 231
 modell, 203
 monitor, 441, 447
 MouseEvent, 293, 294
 MVC modell, 203

N

naptár, 484
 névtelen osztály, 79
 névtelen tömb, 82
 nézet, 203
 notify, 446
 NumberFormat, 490

Ny

nyíltvégű lista, 517
 nyomógomb, 188
 nyomtatás, 459

O,Ó

Object osztály, 448

ObjectInputStream, 404
 ObjectOutputStream, 404
 objektumfolyam, 378, 402
 Observable, 492
 Observer, 493
 Oldalformátum, 460
 Oldalformázás, 467
 osztály, 16
 osztályhierarchia diagram, 33
 osztályleírás, 584
 osztályútvonal, 12
 output path, 12
 OutputStream, 382
 OutputStreamWriter, 389
 overriding, 50

Ö,Ő

öröklés
 egyszeres, 33
 többszörös, 33
 öröklődés, 31
 ős osztály, 32

P

package, 7
 PageDialog, 465
 PageFormat, 467
 paraméter
 applet, 345
 Point, 120
 polimorfizmus, 56
 Polygon, 113, 260
 pont, 119
 Pop, 525
 Printable, 463
 printDialog, 465
 PrinterJob, 459, 464
 PrintWriter, 393
 Prioritás, 441
 privát, 10
 Process, 504
 processEvent, 309
 programszál, 439
 állapota, 440
 programterv, 581
 projekt, 11
 projektkezelés, 3, 11
 property, 119
 publikus, 10
 pufferező folyam, 398

Push, 525

Q

queue, 526
quicksort, 436

R

rácsos elrendezés, 146
rádiógomb, 197
rajzolás, 251
RandomAccessFile, 418
Reader, 387
Rectangle, 120
referencia
 dinamikus, 43
 értékkadási kompatibilitás,
 43
 statikus, 43
rekurzió, 427
 közvetett rekurzió, 430
 közvetlen rekurzió, 430
rekurzív feladat, 428
rendezett bináris fa, 529
rendezett fa, 528
rendezett halmaz, 544
rendezett lista, 517
rendszerhiba, 86
rendszerjellemzők, 502
requestFocusEnabled, 130
ritka mátrix, 521
Runnable, 444
Runtime, 503
runtime binding, 50

S

sekély klónozás, 499
Set, 539
Set interfész, 543
sokszög, 260
sor, 526
sorfolytonos elrendezés, 144
soros hozzáférés, 417
soros szervezés, 416
SortedMap interfész, 559
SortedSet, 539
SortedSet interfész, 544
source path, 12
specializálás, 32
stack, 431, 525

Stack, 540, 557
Statikus kötés, 52
statikus típus, 43
súlymátrix, 533
super, 53
Swing, 105
 osztályhierarchia, 110
SwingConstants, 183
System, 502
SystemColor, 126

Sz

szálescsoport, 441
számformázás, 490
szegély, 111
szekvenciális hozzáférés,
 417
szelekciós műveletek, 513
szerIALIZÁció, 408
szerIALÍLÁS, 405, 408
szimmetrikus lista, 517
szín, 124
szinkronizáció, 441, 446
szintszám, 528
szomszédossági mátrix, 533
szöveges állomány, 387,
 388
szövegmező, 208
szövegterület, 211
szülő, 528
szűrés, 365

T

tábla, 522
téglalap, 119
Telítettségi küszöb, 546
teljes indukció, 427
this, 53
Thread, 441, 444
throw, 89
Throwable, 86
throws, 94
Timer, 241
TimeZone, 482
Típuskényszerítés lefelé, 46
Toolkit, 123
toolTipText, 130
többszálú programozás, 439
tömb, 521
TreeMap, 540
TreeMap osztály, 563

TreeSet, 539
TreeSet osztály, 549
try, 96

U, Ú

upcasting, 45
utód osztály, 32
útvonal, 355

V

Vector, 557
védett, 10
véletlen elérésű állomány,
 415
véletlen szervezés, 416
verem, 431, 525
vezérlő, 204
vezérlő komponens, 106
view, 203
visible, 130

W

wait, 446
Window, 135
Writer, 388

Z

zárt lista, 517