COMPUTERBOOKS

Együtt könnyebb a programozás

Java

Benkő Tiborné
Tóth Bertalan

Együtt könnyebb a programozás

Java

LEKTOR
Kuzmina Jekatyerina

COMPUTERBOOKS
BUDAPEST, 2006

Első kiadás, 2005. november

ISBN : 963 618 337 6

©Kiadó: ComputerBooks Kiadó Kft.
1126 Budapest, Tartsay Vilmos u. 12.
Telefon: 375-1564, Tel/Fax: 375-3591
E-mail: info@computerbooks.hu
http://www.computerbooks.hu
Felelős kiadó: ComputerBooks Kft Ügyvezetője
Borítóterv: Székely Edith

Nyomdai munkák: Pressman Nyomdaipari Bt.

Tartalomjegyzék

Előszó

A Java nyelv 2005. május 23-án ünnepelte 10. születésnapját. A 10 esztendő alatt ez az egyszerű programozási nyelv a világ egyik felét meghatározó technológiává nőtte ki magát. Ahhoz azonban, hogy a valós igényeknek megfelelő megoldások szülessenek, nem elegendő az a professzionális osztálygyűjtemény, amely a Java 2 platformokon keresztül (J2SE, J2EE) rendelkezésünkre áll. Minden technológiai megoldás feltételezi a Java nyelv alapjainak és alapvető osztályainak (Java API) készségszintű ismeretét.

A legtöbb magyar nyelven megjelent Java szakkönyv nagyvonalúan kezeli ezeket a részeket, és a különböző Java technológiákra helyezi a hangsúlyt. Mások a Java nyelvet csupán eszköznek tekintik ahhoz, hogy az objektum-orientált programozást magyarázzák, megint mások pedig az UML leírónyelvvel kapcsolják össze. Az eredmény – amit az oktatási tapasztalatunk is igazol –, hogy a gazdag szakkönyvkínálat ellenére a kezdő programozók nehezen sajátítják el a Java nyelvet.

Az „***Együtt könnyebb a programozás – Java***" könyv az alapoktól indulva, a szöveges és grafikus felületű alkalmazásokon át, a HTML-oldalakba ágyazott kisalkalmazások fejlesztéséig viszi el az Olvasót. Innen kezdve azonban „átengedjük a terepet" a már említett műveknek.

A könyv fejezetei az „*Együtt könnyebb a programozás*" sorozat filozófiájának megfelelően, egy rövid elméleti összefoglalót követően, részletesen magyarázott példák sorával segíti az ismeretek elsajátítását. A CD-mellékleten a könyv példáin túlmenően egy tudásfelmérő alkalmazás, és néhány Java kérdéssor is megtalálható.

Reményeink szerint a könyv feldolgozását követően az Olvasó magabiztosan készít grafikus felületű Java alkalmazásokat és appleteket. Ehhez kívánunk sok sikert!

A szerzők

Bevezetés a Java világba

Könyvünk első fejezetében egy kirándulásra hívjuk az Olvasót. A kirándulás során – a részletekben való elmélyülés nélkül – betekintést nyerhetnek a Java múltjába és jelenébe. A fejezet végén már programot is készítünk, bevezetve ezzel a könyv további fejezeteit.

Az előzményekről röviden

Az 1990-es évek elején a *Sun Microsystems* cégen belül elindult egy kisebb projekt azzal a céllal, hogy a cég részesedést szerezzen a felhasználói elektronikai piac ún. „okos" (*smart*), processzorral vezérelt, programozható készülékek területéből. E készülékcsalád jellegzetes képviselője a kábel-TV társaságok által használt eszköz volt. Az ilyen berendezések programozásához olyan architektúra-független technológiára volt szükség, amely lehetővé tette a kész programok hálózaton keresztüli letöltését és megbízható futtatását.

A fejlesztők kezdetben a C++ nyelvet használták a projekthez, azonban ezt alkalmatlannak találták a célkitűzéseik maradéktalan megvalósítására. Mivel más, létező nyelvvel sem boldogultak, új nyelvet terveztek maguknak. Alapul a C++ nyelvet vették, kigyomlálva belőle a bonyolultnak, nem megbízhatónak tartott szerkezeteket, hozzáadva az innen-onnan átvett, hasznosnak tűnő ötleteket, nyelvi elemeket.

A felhasználói elektronikai berendezések piaca azonban lassabban nőtt a vártnál. A projekt és a kidolgozott *Oak* (tölgy) nyelv eltűnt volna a világ szeme elől, ha eközben az Internet nem indul rohamos fejlődésnek. A *Netscape* cég munkatársai észrevették, hogy az Internet a programozható elektronikai berendezésekhez hasonló körülményeket teremt, és hasonló igényeket támaszt egy új programozási technológiával szemben. Az *Oak* nyelv *Java* néven 1995-ben (1995. május 23-án) indult világhódító útjára, mint a böngészőben futtatható kisalkalmazások (appletek) fejlesztőeszköze.

A Java alapvető tulajdonságai

A előzményekben már említettük, hogy az Internet és a World Wide Web fejlődése a programtermékek fejlesztésének és terjesztésének újszerű megközelítését igényli. A Java nyelv fejlesztésénél szem előtt tartották/tartják ezeket az igényeket.

Egyszerű

Az egyszerűség azt jelenti, hogy a programozók gyorsan elsajátíthatják a Java nyelv alapjait, és néhány objektum megismerése után már programot is írhatnak. A nyelv kidolgozásánál figyelembe vették, hogy a programozók többsége ismeri a C++ nyel-

vet, így amennyire csak lehetett alkalmazták a C++ nyelv elemeit, azonban kihagyták a bonyolult, nehezen kézben tartható megoldásokat (operátor-átdefiniálás, többszörös öröklés stb.) Mindezeken túlmenően az automatikus szemétgyűjtés (*garbage collection*) bevezetése a memória egyszerűbb, hatékonyabb kezelését eredményezte.

Objektum-orientált

A Java a születésétől fogva tiszta (*pure*) objektum-orientált nyelv; így a legegyszerűbb alkalmazás esetén is osztályt és metódust kell definiálnunk. Bebizonyosodott, hogy az elosztott, kiszolgáló-ügyfél rendszerek problémáira az objektum-orientált paradigma adja a legjobb választ, az egybeépítés (*encapsulation*), öröklés (*inheritance, subclassing*) és a sokalakúság (*polymorphism*) felhasználásával. A Java fejlesztői környezetben (JDK) gazdag objektumtár áll a programozók rendelkezésére, melynek objektumai egyszerűsítik és gyorsítják a fejlesztés folyamatát.

Megbízható és biztonságos

A Javát nagy megbízhatóságú alkalmazások fejlesztésére hozták létre. A kitűzött célt a szigorú és erősen típusos fordítóprogrammal valamint a futás közbeni dinamikus ellenőrzéssel érték el. Nem szabad azonban megfeledkeznünk a Java memóriakezeléséről sem, melynek köszönhetően nagyon sok programozási hiba kiküszöbölhető. Ennek legfontosabb elemei a dinamikus helyfoglalású objektumok alkalmazása, az ellenőrzött tömbkezelés, valamint a már nem használt tárterületek automatikus felszabadítása (*garbage collection*).

Mivel Java nyelv az elosztott (*distributed*) rendszereken futó programok fejlesztését célozta, nem csoda, hogy az első helyen a biztonság (*security*) kérdése áll. Magába a nyelvbe és a futtató környezetbe beépített védelmi eszközöknek köszönhetően olyan alkalmazások készíthetők, amelyeket nem lehet kívülről „megtámadni". A hálózatos környezetben futó Java programok védettek a vírust hordozó; illetve a fájlrendszert támadó azonosítatlan kód fertőzésével szemben.

Platformfüggetlen és hordozható

A Javát a hálózaton keresztül terjesztett alkalmazások fejlesztésére hozták létre. Az ilyen hálózatokban fontos igény, hogy a programkód különböző felépítésű hardvereken, eltérő operációs rendszerek alatt működjön. A platformfüggetlenség érdekében a Java fordítóprogramja ún. bájtkódot készít, ellentétben a hagyományos nyelvek fordítói által létrehozott gépi kóddal. Ezt a bájtkódot aztán a különböző platformokra telepített (platformfüggő) alkalmazás, az ún. Java Virtuális Gép (JVM - Java Virtual Machine) értelmezi, futtatja.

A programkód platformfüggetlensége csupán egyik (lényeges) összetevője a hordozhatóságnak, míg a másik összetevőt magában a nyelvben kell keresnünk. A C++

nyelvtől eltérően az alapvető (primitív) adattípusok mérete, valamint a rajta végezhető műveletek szigorúan kötöttek, és szintén függetlenek a hardvertől, a szoftverkörnyezettől. A hordozhatóságot a Java fordítóprogram és a JVM futtatókörnyezet Java-interpretere együttesen biztosítja.

Többszálú

A legtöbb hálózati alkalmazással szemben igényként jelentkezik, hogy több műveletet legyen képes végrehajtani egyidejűleg. A Java a szemaforok mechanizmusával, a folyamatok futásának szinkronizálásával valamint a ***Thread*** osztállyal támogatja a könnyűsúlyú folyamatok (szálak – *threads*) kezelését. A Java virtuális gép a szálak futtatásához egy prioritáson alapuló preemptív ütemezőt tartalmaz.

Interpretált és dinamikus

Egyaránt a dinamikuság jellemzi a Java memóriakezelését, illetve futás közben a hivatkozott osztályok elérését. Javában az objektumok és a tömbök dinamikusan jönnek létre a **new** operátor használatakor, azonban a megszüntetésükről a futtatórendszer gondoskodik a szemétgyűjtési algoritmus háttérben történő futtatásával.

A bájtkód interpretált végrehajtása és a dinamikus szerkesztés nagy mértékben gyorsítja a fejlesztés folyamatát. A hivatkozások feloldását futás közben az osztálybetöltő (*class-loader*) végzi, amely szükség esetén hálózaton keresztül is képes kódot letölteni. Az interpretált végrehajtás nagy hátránya, hogy a programok sokkal – becslések szerint 5-10-szer lassabban futnak, mint a gépi kódú megfelelőik. Bár a speciálisan optimalizált bájtkód könnyen gépi kóddá alakítható, a virtuális gépen történő végrehajtás sebessége nem veheti fel a versenyt a gépi utasítások sebességével. Mindehhez még hozzáadódik a memória-gazdálkodási modellel járó szemétgyűjtési algoritmus futásához szükséges idő.

A futási sebesség az alkalmazások jelentős részénél – a gyakori felhasználói közreműködést, vagy hálózati kommunikációt igénylő programoknál – nem a legfontosabb követelmény. Más esetekben a Java alkalmazásokból hívott gépi kódú (*native*) függvényeket alkalmazhatunk, azonban ezzel elvesztjük a platformfüggetlenséget.

Egyre több virtuális gép támogatja az úgynevezett „röptében fordítást" (*JIT - just-in-time*). A gépi kóddá való fordítás történhet az egyes osztályok betöltésekor – a szükséges ellenőrzések elvégzése után –, vagy végrehajtás közben. Napjainkban leginkább a második eset a jellemző, hiszen futás közben az is eldőlhet, hogy érdemes-e az adott programrészt lefordítani, mert gyakran használjuk (például ciklusmag).

A Java technológiák

Már láttuk, hogy a Java elektronikus berendezések programozási eszközeként indult, majd a webböngészők programozási nyelveként kezdett terjedni – megjelentek a kisalkalmazások, az *appletek*. Később világossá vált, hogy Javában teljes értékű alkalmazások (*applications*) is készíthetők. A programok grafikus elemeit komponensek formájában kezdték előállítani – megjelent a *JavaBeans*. Ezzel a Java belépett az elosztott rendszerek és a közbenső programelemek fejlesztésének világába, amely szoros kapcsolatban áll a CORBA technológiával. Csak egy kis lépés maradt a kiszolgálók programozásáig, amit meg is léptek, amikor megjelentek a *servletek* (szerveren futó kisalkalmazások), valamint az *EJB* (*Enterprise JavaBeans*). A kiszolgálók adatbázisokkal működnek együtt, amihez szükséges megoldásokat a *JDBC* (*Java DataBase Connection*) biztosítja. Napjainkban a Java alkalmazásának egyik fontos területe a mobil eszközök (telefonok, kéziszámítógépek - *PDA*-k) programozása, valamint a webalapú vállalati alkalmazások (*web services*) fejlesztése.

A felsorolást még sokáig lehetne folytatni, azonban már ennyiből is látszik, hogy jogosan beszélünk a Javáról, mint technológiáról. Könyvünkben a Java nyelv megismertetése mellett, a felsorolás első két elemére, az alkalmazásokra és az appletekre helyezzük a hangsúlyt, hiszen a további technológiák szintén ezeken az alapokon nyugszanak.

A Java 2 platform

A Java platform az a szoftverkörnyezet, amelyben a Java nyelven fejlesztett, majd bájtkóddá fordított programok futnak. A Java platform két alapvető részre bontható:

- Java virtuális gép (*Java Virtual Machine, JVM*);
- Java alkalmazás-programozási felület (Java *Application Programming Interface, Java API*).

Mivel a kezdetekhez képest mind a virtuális gép, mind pedig a Java API 1998 végén lényegesen átalakult, a napjainkban használt Java platformot ***Java 2*** platformnak nevezzük. (A *Java 1* platform elemei ugyan még elérhetők, azonban nem ajánlott az érvénytelenített (*deprecated*) megoldásokat alkalmazni.)

A ***Sun*** cég *Java 2 Platform Standard Edition* (*J2SE*) néven teszi elérhetővé azt a minimális platformot, amelyen a fejlesztők Java nyelven készített appleteket és alkalmazásokat futtathatnak. Ez a platform képezi az alapját a *Java 2 Platform Enterprise Edition* (*J2EE*) és a *Java Web Services* technológiáknak is.

Két alapvető programtermék tartozik a J2SE platformhoz, a J2SE futtatókörnyezet (*Runtime Environment - JRE*) és a J2SE fejlesztőkészlet (*Development Kit - JDK*).

- A ***JRE***-ben megtalálható a Java API, a Java virtuális gép, valamint más olyan programelemek, melyek szükségesek a Java nyelven készített appletek és alkalmazások futtatásához.
- A ***JDK*** magában foglalja a JRE-t, és egy sor olyan (parancssor) programot, amelyek nélkülözhetetlenek az appletek és az alkalmazások fejlesztéséhez.

Felhívjuk a figyelmet arra, hogy a JDK semmilyen integrált fejlesztői környezetet (*Integrated Development Environment - IDE*) sem tartalmaz, ezeket külön kell beszereznünk.

A könyvünkben szereplő példaprogramok fejlesztéséhez a *Java 2 Platform Standard Edition 5.0 JDK*-t használtuk.

Java fejlesztőeszközök

A fellelhető Java fejlesztőeszközök igyekeznek mind több Java technológia alkalmazásához segítséget nyújtani, így ezek ára meglehetősen magas. A legtöbb eszköznek azonban létezik korlátozott ideig használható próbaverziója, vagy valamilyen más módon lebutított változata. Mielőtt azonban áttekintenénk ezeket a tekintélyes méretű alkalmazásokat, nézzük meg azt a megoldást, amely Javával való ismerkedés kezdetén minden igényt kielégít!

Először a ***Sun*** cég honlapjáról le kell töltenünk a legfrissebb ***JDK***-t (például JDK 5.0) (*http://java.sun.com/j2se/downloads/*). A JDK letöltése ingyenes, mindössze el kell fogadnunk a licencszerződést (*Accept License Agreement*), és választanunk kell a különböző számítógépes platformok közül (*Windows*, *Linux*, *Solaris*). A sikeres letöltés után el kell végeznünk a telepítést, elfogadva a felajánlott beállításokat.

Mivel a fejlesztés folyamatában fontos szerepet kap a szövegszerkesztő, érdemes egy olyat keresni, amelyik ismeri a JDK fordítási (*javac*) és futtatási (*java*, *appletviewer*) parancsait. Egy ilyen szerkesztő például a ***TextPad*** *shareware* program, amelyet a *http://www.textpad.com/* címről tölthetünk le. A *TextPad* telepítése után a ***Tools*** menű utolsó három menüpontját használhatjuk a fordításhoz, illetve a futtatáshoz.

Az alábbiakban a teljesség igénye nélkül összefoglaltuk a kisebb – nagyobb Java integrált fejlesztői környezeteket. Felhívjuk a figyelmet arra, hogy ezek egy része Javában készült, így csak a ***JRE*** telepítése után futtathatók. (A JRE-t szintén ingyen tölthetjük le a *Sun* cég honlapjáról.)

Eszköz	Webcím	R	Megjegyzés
BlueJ	bluej.org	-	szabadszoftver
Eclipse	eclipse.org	-	szabadszoftver
IntelliJ™ IDEA 5.0.1	www.jetbrains.com/idea/	-	30-napos próbaverzió
JBuilder 2005 Foundation	www.borland.com/downloads/	igen	szabadszoftver
JBuilder 2006 Enterprise	www.borland.com/downloads/	igen	30-napos próbaverzió, majd Foundation
JCreator	www.jcreator.com	igen	30-napos próbaverzió és szabadszoftver
JDeveloper 10*g*	www.oracle.com/technology/	igen	szabadszoftver
NetBeans IDE	netbeans.org	-	szabadszoftver
Sun Java™ Studio Creator	www.sun.com/download/	igen	30-napos próbaverzió
Sun Java™ Studio Enterprise	www.sun.com/download/	igen	90-napos próbaverzió
WebSphere Software	www.ibm.com	igen	90-napos próbaverzió

A táblázat fejlécében az **R - Regisztráció**.

Programfejlesztés a JDK segítségével

A JDK segítségével való fejlesztés első lépése a forráskód előállítása. Ezt tetszőleges olyan szövegszerkesztővel elvégezhetjük, amely kezeli tiszta (formázás nélküli) szöveget (például a *Jegyzettömb* a *Windows*-ban).

Mivel a Java teljesen objektum-orientált nyelv, már a legegyszerűbb program is tartalmaz egy nyilvános elérésű (**public**) osztályt. A **class** szó után álló osztálynevet (a kis- és nagybetűk figyelembevételével) a forrásállomány neveként kell használnunk, melynek kiterjesztés ***.java***.

Szöveges felületű Java alkalmazás

A Java nyelv lehetőségeinek megismerését kiválóan segítik a szöveges felhasználói felületű programok, hisz ezek írásakor nem kell foglalkoznunk a grafikus felület elemeivel. Az első Java programunkat *TextApplication.java* állomány tartalmazza. A létrehozott osztály kötelezően rendelkezik egy ***main()*** nevű függvénnyel (metódussal), amely kijelöli az alkalmazás belépési pontját. Ez a metódus **void** típusú, nyilvános elérésű (**public**) és statikus (**static**) kell, hogy legyen. A statikus metódusok az osztállyal együtt léteznek, így a hívásukhoz nincs szükség objektumra (az osztály példányára.) Általánosan is elmondhatjuk, hogy minden Java alkalmazás rendelkezik egy ilyen ***main()*** metódussal.

```
public class TextApplication
{
   public static void main (String[] args)
   {
      System.out.println("\nJava 2 Platform Standard Edition 5.0");
   }
}
```

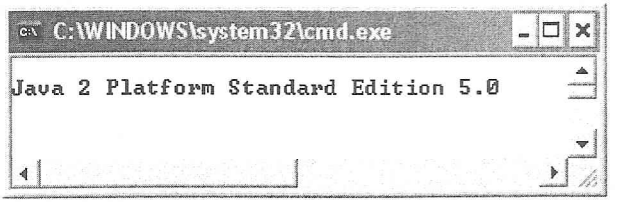
A fejlesztés következő lépése a fordítás, amelyet a JDK ***javac*** parancsával, parancssorban végzünk:

```
javac TextApplication.java
```

A fordítás eredményeként keletkező bájtkódot a *TextApplication.class* állomány tartalmazza. Az alkalmazást a JDK ***java*** parancsával futtathatjuk (a *.class* kiterjesztés megadása nélkül):

```
java TextApplication
```

A futás eredményeként a szöveges ablakban megjelenik a kiírt szöveg:



Grafikus felületű Java kisalkalmazás (applet)

A program bonyolultságát tekintve nem a grafikus felületű alkalmazások, hanem az appletek következnek, melyek mindig grafikus felülettel rendelkeznek. Az appleteknél hiányzik a ***main()*** metódus, viszont megtalálhatjuk a ***paint()*** metódust, amely a grafikus megjelenítésért felelős.

```
import java.awt.*;
import java.applet.*;

public class GraphApplet extends Applet
{
    public void paint (Graphics g)
    {
        g.drawString("\nJava 2 Platform Standard Edition 5.0",12,23);
    }
}
```

Az **import** utasításokkal megmondjuk, hogy a Java API mely csomagjaiból kívánunk osztályokat használni. Az osztályok teljes nevének megadásával a importálás feleslegessé válik:

```
public class GraphApplet extends java.applet.Applet
{
    public void paint (java.awt.Graphics g)
    {
        g.drawString("\nJava 2 Platform Standard Edition 5.0",12,23);
    }
}
```

A példában szereplő *GraphApplet* osztályt a *GraphApplet.java* állomány tárolja, így a fordítás parancssora:

```
javac GraphApplet.java
```

A bevezetőben már említettük, hogy az appleteket általában a webböngésző alkalmazás futtatja. Az applet indításhoz (és paraméterezéséhez) szükséges adatokat ***HTML***-állományban kell megadnunk. A példában szereplő *GraphApplet.html* fájl tartalma:

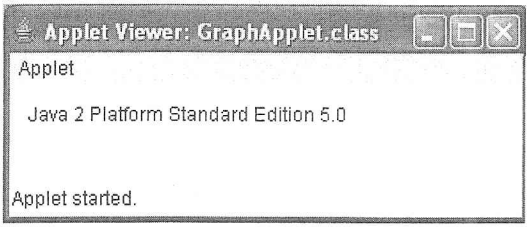
```
<HTML><HEAD><TITLE>Java applet futtatása</TITLE></HEAD>
<BODY>
<applet
        code=GraphApplet.class
        name=GraphApplet
        width=320 height=60 >
</applet>
</BODY> </HTML>
```

Az appletet korlátozott módon az ***Applet Viewer*** programmal is megjeleníthetjük,

```
appletviwer GraphApplet.html
```

azonban teljes funkcionalitása csak a böngészőben érhető el.

Futtatás appletviewer segítségével



Futtatás böngészőben

Grafikus felületű Java alkalmazás

A grafikus felületű Java alkalmazások a ***main()*** és a ***paint()*** metódust egyaránt tartalmazzák. A ***main()*** metódus annyiban bővült, hogy benne hozzuk létre ***Frame*** osztályból származtatott *GraphApplication* osztály példányát (**new**), majd pedig megjelenítjük azt.

```
import java.awt.*;
public class GraphApplication extends Frame
{
   public static void main(String[] args)
   {
      GraphApplication application = new GraphApplication();
      application.setSize(320,120);
```

```
        application.setTitle("Egyszerű Java alkalmazás");
        application.setVisible(true);
    }
    public void paint (Graphics g)
    {
        g.drawString("\nJava 2 Platform Standard Edition 5.0",12,53);
    }
}
```

A *GraphApplication.java* állományban tárolt alkalmazás fordítása és futtatása a már ismertetett módon történik:

```
javac GraphApplication.java

java GraphApplication
```

Az alkalmazás ablaka most grafikus felülettel rendelkezik:

(Megjegyezzük, hogy a példából hiányzik az alkalmazásból való kilépést biztosító kódrészlet, így a megállításához drasztikus módszert – például a *Windows Feladatkezelőjét* – kell használnunk.)

1. A Java alkalmazások szerkezete

Az objektum-orientált Java nyelven készített programok szerkezetét és utasításait egyszerű példákon keresztül mutatjuk be a további fejezetekben.

1.1 Egyszerű Java alkalmazás

Minden Java program osztályból (**class**) épül fel, amely az objektum-orientált nyelv fontos velejárója.

Írjunk programot, amely szöveget ír ki a képernyőre! (*Elso*)

A program listája:

```
// Mintafeladat a java program felépítésére
public class Elso{
        public static void main(String[] args) {
          System.out.println("Java programozasa nem nehez!");
        }
```

A program futásának eredménye:

```
Java programozasa nem nehez!
```

A Java program magyarázata:

- // jelekkel kezdődő sor a sor végéig megjegyzést tartalmaz.
- A **class** kulcsszó kijelöli azt az osztályt, melynek neve a példában *Elso*. A Java programot az osztály nevével azonos néven és *.java* kiterjesztéssel kell tárolni. Jelen esetben a programunkat az *Elso.java* állomány tartalmazza.
- Az osztály előtt elhelyezkedő **public** kulcsszó jelöli az osztály hozzáférhetőségét. A példában ez mindenki számára használható osztályt jelent, azaz nyilvános. A hozzáférés további lehetőségeiről később esik szó.
- A kapcsos nyitó { és a kapcsos csukó } zárójel fogja közre az osztály törzsét.

Az *Elso* osztály egyetlen függvényt (Java szóhasználattal élve metódust) tartalmaz. Ez a ***main()*** (fő) függvény az alkalmazás indításakor hívódik meg. A ***main()*** függvény jellemzői:

- A hozzáférés nyilvános: **public**.
- Nem változtatható: **static**.
- Nincs visszatérési értéke: **void**.
- Paraméterlistáját kerek zárójelbe kell tenni.
- Paramétere: ***String[] args***, azaz több szövegtípusú indítási argumentuma lehet.

– Szintén kapcsos nyitó { és a kapcsos csukó } zárójel fogja közre a ***main***() metódus törzsét, mely a Java program futtatásakor kerül végrehajtásra.

A ***main***() függvény egyetlen – *System.out.println* – utasítást tartalmaz, amelynek segítségével szöveges üzenetet írunk ki (ez a képernyőn, szöveges ablakban jelenik meg). A kiírandó szöveget idézőjelek között adjuk meg *println()* függvény paraméterlistáján, amely a szöveg kiírása után sort emel.

```
System.out.println("Java programozasa nem nehez!");
```

1.2 Szöveg írása a képernyőre

A Java alkalmazásokban a képernyőre való íráshoz a *System.out.println()* metódust használjuk.

Írjunk programot, amely szöveget ír ki a képernyőre, és a szöveg közben, valamint a végénél is sort emel! Az első és utolsó sornál szóközzel pozícionál, míg a második sorban tabulátorral ('\t'). Az utasításokat lássuk el megjegyzésekkel! (*Masodik*)

```
// Szöveg írása több sorban
public class Masodik{
       public static void main(String[] args) {
         System.out.println
          ("          Nem nehez\n\tJava nyelven\n        programozni!");
       }
       /* ez a kiírás
          több sorban
          folytatódik
       */
  }
```

A program futásának eredménye:

```
         Nem nehez
       Java nyelven
       programozni!
```

A /* és a */ jelek között a megjegyzést több sorban adhatjuk meg.
A *println()* függvény paraméterlistáján az idézőjelek között megadott szövegben szerepelhet szóköz, új sor (\n) és a tabulátor (\t) is.

Írjunk programot, amely több sorban ír szöveget, használja a *println()* és a *print()* függvényeket! (*Szoveg1*)

A *print()* metódus nem emel sort a szöveg kiírása után. A paraméter nélküli *println()* egy üres sort emel.

```
public class Szoveg1 { // Szöveg írása több sorban
      public static void main(String[] args) {
        System.out.println("Elso sor");
        System.out.println("\tMasodik sor");
        System.out.println("\t\tHarmadik sor");
        System.out.print("Negyedik ");System.out.println("sor");
        System.out.println(); // ures sort emel
      }
}
```

A program futásának eredménye:

```
Elso sor
        Masodik sor
                Harmadik sor
Negyedik sor
```

1.3 Az osztályok dokumentálása

A JDK ***javadoc*** segédprogramja képes arra, hogy a Java forrásállományból automatikusan elkészítse az osztály HTML-formátumú dokumentációját. Ehhez persze megfelelő formában kell megjegyzéseket elhelyeznünk a fájlban:

```
/**
 * Az ilyen megjegyzéseket a javadoc program dolgozza fel
 * lehet használni benne HTML-elemeket is, illetve
 * speciális vezérlőszavakat, melyek a @ jellel kezdődnek.
 */
```

A ***javadoc*** program leírásának áttanulmányozása után, készítsük el az *Elso* programunk dokumentált változatát, illetve a dokumentációját! (*Harmadik*)

A megoldásban külön dokumentáljuk az osztályt (programot) és a ***main()*** metódust. A leírás előállításához az alábbi parancssort használjuk:

```
...\java\bin\javadoc Harmadik.java -author
```

A programunk *Harmadik.java* forrásállománya:

```
/**
 * Minden java alkalmazás az osztály <b>public static void</b> tipusú
 * <i><b>main</b>()</i> metódussal indul el.
 * @author ComputerBooks
 */
public class Harmadik {
      /**
       * Az alkalmazás belépési pontja.
       *
       * @param args a parancssorban megadott szöveges paraméterek
       * tömbje.
       * @return nem ad vissza semmit
       * @see java.lang.String
       */
```

```
    public static void main(String[] args)
    {
        System.out.println("Java programozasa nem nehez!");
    }
}
```

A leírás több HTML-állományt is tartalmaz, hiszen a Java API dokumentációjával megegyező formátumú az eredmény. A *Harmadik.html* állomány megnyitásakor, a böngészőben megjelenő tartalom lényegesebb részei:

Class Harmadik

```
java.lang.Object
  └Harmadik
```

```
public class Harmadik
extends java.lang.Object
```

Minden java alkalmazás az osztály **public static void** tipusú ***main()*** metódussal indul el.

Author:
ComputerBooks

Method Detail

main

```
public static void main(java.lang.String[] args)
```

Az alkalmazás belépési pontja.

Parameters:
`args` - a parancssorban megadott szöveges paraméterek tömbje.
See Also:
`String`

---

2. Alaptípusok, változók és konstansok

A Java nyelvű programban a használni kívánt neveket mindig deklarálnunk kell, ezáltal adjuk meg a fordítónak a név tulajdonságait. A Java betűérzékeny, így a kis- és a nagybetűvel írt nevek különböző memóriaterületet azonosítanak.

2.1 A Java nyelv primitív típusai

A Java nyelv alaptípusait táblázatban foglaljuk össze:

Típus	*Méret*	*Értékkészlet*
boolean	8 bit	true vagy false
byte	8 bit	-128 ÷ 127
short	16 bit	-32,768 ÷ 32,767
char	16 bit	16-bites Unicode karakterek.
int	32 bit	-2,147,483,648 ÷ 2,147,483,647
long	64 bit	-9223372036854775808 ÷ 9223372036854775807
float	32 bit	
double	64 bit	

Megjegyzés a típusokhoz:

`boolean`	Logikai értéket tárol, melynek értéke igaz (**true**), vagy hamis (**false**) lehet.
`char`	Karaktert tárol, mivel a **char** típus 16 bitet foglal el, ezért a Java a Unicode kódolási szabványt támogatja.
`byte, short, int, long`	A tárolási helynek megfelelő előjeles (pozitív, negatív) egész számokat tárolnak.
`float, double`	Lebegőpontos számokat tárol (mantissza és karakterisztika bontásban).

A Java programok osztálydefiníciókból állnak. Javasolt, hogy az osztályok nevét nagybetűvel kezdjük. Az osztályok változókat (mezőket) és metódusokat tartalmaznak. A metódusok (azaz függvények, ill. eljárások) tárolják azokat az utasításokat, melyek végrehajtásával különböző feladatokat oldhatunk meg. Ezekről később lesz szó. A Java utasításainak megismeréséhez csak a ***main***() függvényt használjuk.

Nézzünk néhány példát a változódeklarációra! A deklarációban szereplő neveket betűvel kell kezdeni, ezt követően betű és szám, valamint _ (aláhúzásjel) szerepelhet benne. A deklarációt valamilyen Java típus vezeti be.

Például:

```
char csillag;
int i, j2, k_3;
double a, b2b, szorzo_1;
```

A Java nyelv lehetővé teszi a deklarált változó kezdőértékkel való ellátását a deklarációban. (A kezdőérték nélküli változók területe törlődik, nullaértékű bájtokkal töltődik fel.)

```
public class Deklaracio
{
  public static void main(String[] args)
  {
       char csillag = '*';
       int i, j2, k_3 = 3;
       double a, b2b, szorzo_1 = 12.56;
  }
 }
```

2.2 Szabványos bement és kimenet

Röviden foglalkozunk a szabványos bemenettel és kimenettel, melyeket a feladatokban használunk. A be-, ill. kivitelre a Java adatfolyam (*stream*) osztályokat használ. Ebben a fejezetben csak a billentyűzeten és a képernyőn keresztül történő kommunikációt mutatjuk be.

Felhívjuk a figyelmet arra, hogy az alkalmazott megoldásokkal csak szöveget tudunk megjeleníteni, illetve beolvasni. Más típusú adatok esetén valamilyen módon el kell végeznünk a szöveggé, vagy a szövegből történő átalakítást.

2.2.1 Képernyőn való megjelenítés

Karaktersorozatot jeleníthetünk meg a *System.out.println()*, ill. a *System.out.print()* metódusokkal, melyek argumentumaként a kiírni kívánt szöveget kell megadnunk. A szövegkonstanst idézőjelek közé kell tenni. Ha a szöveget több darabban adjuk meg, vagy változók értékét szeretnénk kiíratni, akkor az összeadásjellel (+) fűzhetjük össze az egyes részeket szöveggé:

```
System.out.println("szoveg" + valtozo + "szoveg" + valtozo);
```

Írjunk programot, amely egész, valós és karakter típusú változóknak ad értéket, és kiíratja szövegesen! (*Kiir1*)

```
// Képernyőre való írás
public class Kiir1{
       public static void main(String[] args) {
                int i;
```

```
                double a;
                char c;
                i = 13;
                a = 2.56;
                c = '*';
                System.out.println("egesz adat: " + i +" valos adat: "
                                    + a + " karakter: " + c);       }
}
```

A program futásának eredménye:

```
egesz adat: 13 valos adat: 2.56 karakter: *
```

2.2.2 Beolvasás billentyűzetről

A billentyűzetről történő beolvasás bonyolultabb. Az ***InputStreamReader*** adatfolyamosztályból a ***System.in*** paraméterrel létrehozunk egy objektumpéldányt, amelyet a ***BufferedReader*** konstruktora paraméterként vár. A beolvasáshoz használt objektum neve legyen *be*, melyet az alábbi utasítással állíthatunk elő:

```
BufferedReader be= new BufferedReader(new InputStreamReader(System.in));
```

A ***BufferedReader*** típusú *be* objektum ***readLine()*** metódusával beolvashatjuk a begépelt és *<Enter>* billentyűvel lezárt szövegsort, amelyet aztán megfelelő típusúvá kell alakítanunk.

Írjunk programot, amely egész és valós típusú változóknak billentyűzetről ad értéket, valamint megjeleníti a beolvasott adatokat! (*Olvas1*)

```
// Változók beolvasása
import java.io.*;
public class Olvas{
      public static void main(String[] args) throws IOException  {

            BufferedReader be=new BufferedReader
                              (new InputStreamReader(System.in));
            System.out.print("i = ");
            int i =Integer.valueOf(be.readLine()).intValue();
            System.out.print("a = ");
            double a =Double.valueOf(be.readLine()).doubleValue();
            System.out.println("egesz adat: " + i +
                               "  valos adat: " + a );
      }
}
```

A program futásának eredménye:

```
i = 12
a = 134.56
egesz adat: 12  valos adat: 134.56
```

2.2.3 Karakter beolvasása billentyűzetről

Egyetlen karakter beolvasására használhatjuk a *System.in.read()* metódust. A karaktert egész típusú változóba olvassuk be, mely értékének karakterré a (**char**) típuskonverzióval alakítjuk át.

```
int kar1 = System.in.read();
char ckar1 = (char)kar1;
```

Írjunk programot, amely két karaktert olvas be billentyűzetről, valamint megjeleníti a beolvasott karaktereket és a hozzájuk tartozó kódokat! (*Olvas2*)

```
// Karakter beolvasása
import java.io.*;
public class Olvas2{
       public static void main(String[] args) throws IOException  {
        int kar1, kar2;
        char ckar1, ckar2;
        System.out.print("2 karakter = ");
        kar1 = System.in.read();
        ckar1 = (char)kar1;
        kar2 = System.in.read();
        ckar2 = (char)kar2;
      System.out.println("Beolvasott karakterek: " + ckar1 + ckar2);
      System.out.println("Karakter: " + ckar1 + " kodja: " + kar1);
      System.out.println("Karakter: " + ckar2 + " kodja: " + kar2);
    }
 }
```

A program futásának eredménye:

```
2 karakter = a2
Beolvasott karakterek: a2
Karakter: a kodja: 97
Karakter: 2 kodja: 50
```

2.3 Konstansok a Java nyelvben

A Java nyelvben a konstansok definiálására a **final** kulcsszót használjuk.

Definiáljunk konstansokat a **final** használatával, néhánynak írassuk ki az értékét! (*Konstansok*)

Megjegyzés:
A konstans tárolók csak olvashatók a programban, értéküket nem lehet megváltoztatni. A konstansok azonosítóját általában nagybetűkkel adjuk meg.

```
// Konstansok
public class Konstansok{
        public static void main(String[] args) {
                final char UJSOR = '\n';
                final int MODOSITO = 12;
                final float SZORZO = 1.5e-2F;
                final double NORMALO = 150;
                int k;
                double sz;
                k = 32 + MODOSITO;
                sz = 12542.5 * SZORZO/NORMALO;
                System.out.println(" k: " + k + UJSOR
                                        + " sz: " + sz + UJSOR);
                System.out.print(UJSOR);
        }
}
```

A program futásának eredménye:

```
k: 44
sz: 1.254249971965328
```

2.4 Értékadás

A változók általában értékadás során kapnak értéket, melynek általános alakja:

változó = érték;

Az értékadás operátorának (=) bal és jobb oldalán egyaránt szerepelhetnek kifejezések, melyek azonban különböznek egymástól. A bal oldalon szereplő kifejezés azt a *területet* jelöli ki a memóriában, ahová a jobb oldalon megadott kifejezés *értékét* be kell tölteni.

Nézzünk néhány példát az értékadásra!

```
int x;
x = 7;
x = x + 5;
```

`x = 7` az *x* változó kijelöli azt tárterületet, ahová a jobb oldalon megadott konstans értékét (7) be kell másolni.

`x = x + 5;` az értékadás során az *x* változó az egyenlőségjel mindkét oldalán szerepel. A bal oldalon álló *x* változó kijelöli azt a memóriaterületet, ahová a futtató rendszer behelyezi a jobb oldalon álló kifejezés értékét. Vagyis az értékadás előtt az *x* pillanatnyi értékét (7) összeadja az 5 konstans értékkel (elvégzi az összeadást), majd az eredményt az *x*-ben tárolja. Ezáltal az *x* korábbi 7 értéke felülíródik 12-vel.

```
// Adattipusok bemutatása, értékadás
public class Adattipusok{
        public static void main(String[] args) {
                boolean igaz, hamis;
                char karakter, csillag, szamjegy;
                byte b_szam;
                short s_szam;
                int i_szam;
                long l_szam;
                float f_szam;
                double d_szam;
                igaz = true;
                hamis = false;
                karakter = 'a';
                csillag = '*';
                szamjegy = '5';
                b_szam = 127;
                s_szam = 32767;
                i_szam = 2147483;
                l_szam = 217452345;
                f_szam = 1.237F;
                d_szam = 3.123E4;
        }
}
```

2.5 Java függvények (metódusok) hívása

Néhány általános szabályt meg kell ismernünk a függvényekkel (metódusokkal) kapcsolatosan, mielőtt rátérnénk a matematikai függvények használatára. (A könyvünkben a metódus és a függvény szavakat szinonimaként használjuk.)

Minden metódusnak van neve, és vannak paraméterei, melyeken keresztül megkapja azokat az adatokat, melyekkel dolgozik a függvény törzsében. A paraméterek kerek zárójelben vannak, a kerek nyitó és záró zárójelet még akkor is ki kell tenni, ha a metódus nem rendelkezik paraméterrel. A metódus törzsét a nyitó és csukó kapcsos zárójel fogja közre.

A függvény jellemzője a visszatérési értéke, melynek típusát a függvényfejben, a neve előtt kell megadni. A visszatérési értéket a **return** utasításban adjuk meg. Az elmondottakat jól szemlélteti az alábbi kis metódus:

```
int fgnev(int a)
{  int b;
   . . .
   return b;
}
```

A metódus hívásához a nevét, valamint az egyes paramétereknek megfelelő típusú argumentumok listáját használjuk. A metódusnév után a kerek zárójelpár megadása akkor is kötelező, ha a függvény nem rendelkezik paraméterrel. A fenti függvény hívását az alábbiak szerint végezhetjük el:

```
int x, y=12;
x = fgenv(y);
```

2.5.1 Matematikai függvények

A *java.lang* csomag osztályait, így a *Math* osztályt sem kell importálnunk. Az alábbiakban felsorolunk néhány fontos matematikai függvényt, a visszatérési érték típusának jelzésével. (A teljes táblázat megtekinthető az F2. függelékben.) A matematikai függvények aktiválásánál a *Math.* névvel jelezzük, hogy a *Math* osztály statikus metódusait használjuk. Néhány, gyakran használt matematikai függvény:

Használat	*Típusa*	*Metódus*
abszolút érték képzése	**double**	abs(**double** x)
abszolút érték képzése	**float**	abs(**float** x)
abszolút érték képzése	**int**	abs(**int** x)
szög koszinusza	**double**	cos(**double** x)
szög szinusza	**double**	sin(**double** x)
szög tangense	**double**	tan(**double** x)
természetes alapú logaritmus	**double**	log(**double** x)
e^x	**double**	exp(**double** x)
hatványozás x^y	**double**	pow(**double** x, **double** y)
négyzetgyök	**double**	sqrt(**double** x)
véletlen szám 0>= és <1	**double**	random()
kerekít a legközelebbi **long** értékre	**long**	round(**double** x)
kerekít a legközelebbi **int** értékre	**int**	round(**double** x)

A *Math.E* a természetes alap és a *Math.PI* pedig π értékét adja meg konstansként. Példa néhány függvény hívására:

gyökvonás : `y = Math.sqrt(x);` vagy `y = Math.pow(x,1.0/2);`

koszinusz : `y = Math.cos(x);`

szinusz : `y = Math.sin(x);`

tangens `y = Math.tan(x);`

hatványozás : a^n
```
y = Math.pow(a,n); vagy
y = Math.exp(Math.log(x)*n));
```

exp függvény : e^x `y = Math.exp(x);`

természetes log. : ln x `y = Math.log(x);`

Írjunk programot, amelyben kiszámítjuk az egységsugarú kör területét és kerületét! (*Math1*)

A π értékét saját konstanssal is megadhatjuk, de használhatjuk az előre definiált ***Math.PI*** konstanst is.

```
        final double PI = 3.141592654;


public class Math1{
        public static void main(String[] args) {
                double sugar = 1, Korter, Korker;
                Korter = Math.pow(sugar,2) * Math.PI;
                Korker = 2 * sugar * Math.PI;
                System.out.println("Kor terulete: " + Korter);
                System.out.println("Kor kerulete: " + Korker);
        }
 }
```

A program futásának eredménye:

```
Kor terulete: 3.141592653589793
Kor kerulete: 6.283185307179586
```

Írjunk programot, amelyben néhány fontos matematikai függvényt használunk. (*Mathpr*)

```
// Matematikai függvények használata
public class Mathpr{
        public static void main(String[] args) {
                double adat = 4;
                double gyok, negyzet, kobgyok1, kobgyok2;
                double a = 3, b = 4, szog = 90, c;
                gyok = Math.sqrt(adat);                    // gyökvonás
                negyzet = Math.pow(adat,2);        // négyzetre emelés
                kobgyok = Math.exp(Math.log(adat)/3.0);    // köbgyök
                kobgyok2 = Math.pow(adat, 1.0/3.0);        // köbgyök
                System.out.println("Adat: " + adat);
                System.out.println("\nGyok: " + gyok);
                System.out.println("Negyzet: " + negyzet);
                System.out.println("1. Kobgyok: " + kobgyok1);
                System.out.println("2. Kobgyok: " + kobgyok2);
        }
 }
```

A program futásának eredménye:

```
Adat: 4.0

Gyok: 2.0
Negyzet: 16.0
1. Kobgyok: 1.5874010519681994
2. Kobgyok: 1.5874010519681994
```

3. Operátorok és kifejezések

A Java nyelvben a legnépesebb utasításcsoportot a pontosvesszővel lezárt kifejezések alkotják. A kifejezések egyetlen operandusból, vagy operandusok és műveleti jelek (operátorok) kombinációjából épülnek fel.

Az operandusok a Java nyelv azon elemei, amelyeken az operátorok fejtik ki a hatásukat. Az operandus lehet konstans érték, azonosító, sztring, metódushívás, tömbindex kifejezés, tagkiválasztó kifejezés (ezek az ún. elsődleges kifejezések) és tetszőleges összetett kifejezés, amely zárójelezett vagy zárójel nélküli, operátorokkal összekapcsolt további operandusokból áll. Ezeket az operátorokat elsődleges (*primary*) kifejezéseknek nevezzük.

Az operátorokat több szempont alapján lehet csoportosítani. Az egyoperandusú (*unary*) operátorok esetén a kifejezés általános alakja:

```
op operandus    vagy    operandus op
```

Az első esetben, amikor az operátor (*op*) megelőzi az operandust, előrevetett (*prefix*), míg a második esetben hátravetett (*postfix*) alakról beszélünk.

`int j;`	egész típusú *j* deklarálva,
`j = 4;`	értékadás,
`j++;`	*j* értékének növelése (*postfix*),
`--j;`	*j* értékének csökkentése (*prefix*),
`(double)j`	*j* értékének valóssá alakítása.

Az operátorok többsége két operandussal rendelkezik, ezek a kétoperandusú (*binary*) operátorok:

```
operandus1 op operandus2
```

Például: `j + 2, j << 3, j += 3` stb.

3.1 Aritmetikai operátorok

Az aritmetikai operátorok felsorolása a műveletek precedenciája szerint történt, az azonos szintű precedencia azonos sorszám alatt jelenik meg.

1. * szorzás
   / osztás
   % maradékképzés
2. \+ összeadás
   – kivonás

3.2 Precedencia és asszociativitás

A bonyolultabb kifejezések az elsőbbségi (*precedencia*) szabályok szerint kerülnek kiértékelésre. Ezek a szabályok meghatározzák a kifejezésekben szereplő műveletek kiértékelési sorrendjét.

Egyes operátorok közötti elsőbbségi kapcsolatot táblázatban foglaltuk össze, ahol a táblázat sorai az azonos precedenciával rendelkező operátorokat tartalmazzák. Minden sorban külön jeleztük az azonos precedenciájú operátorokat tartalmazó kifejezésben a kiértékelés irányát, amit *asszociativitásnak* nevezünk. A táblázat első sora tartalmazza a legnagyobb precedenciával rendelkező **new** műveletet.

Precedencia	Asszociativitás	Műveleti jel	Megnevezés
1	nincs	`new`	objektum-létrehozás
2	nincs	`new`	tömblétrehozás
3	balról-jobbra	`.`	mezőelérő művelet
4	nincs	`()`	metódushívás
5	nincs	`[]`	tömbelérő művelet
6	nincs	`++, --`	postfix műveletek
7	jobbról-balra	`- , +, !, ~, ++, --, (típus)`	egyoperandusú műveletek
8	balról-jobbra	`*, /, %`	multiplikatív műveletek
9	balról-jobbra	`+, -`	additív műveletek
10	balról-jobbra	`<<, >>, >>>`	biteltoló műveletek
11	balról-jobbra	`<, >, <=, >=, instanceof`	összehasonlító műveletek
12	balról-jobbra	`==, !=`	azonosságvizsgálat
13	balról-jobbra	`&`	bitenkénti logikai ÉS
14	balról-jobbra	`^`	bitenkénti logikai kizáró VAGY
15	balról-jobbra	`|`	bitenkénti logikai VAGY
16	balról-jobbra	`&&`	logikai ÉS
17	balról-jobbra	`||`	logikai VAGY
18	jobbról-balra	`?:`	feltételes művelet
19	jobbról-balra	`=, *=, /=, %=, -=, <<=, >>=, >>>=, &=, ^=, |=`	értékadó műveletek

3.2.1 A precedencia-szabály

Ha egy kifejezésben különböző precedenciájú műveletek szerepelnek, akkor mindig a magasabb precedenciával rendelkező operátort tartalmazó részkifejezés értékelődik ki először (*Prec1*).

Az $a+b*c-d*e$ és az $a+(b*c)-(d*e)$ kifejezések kiértékelési sorrendje megegyezik:

```
1. b * c    -> 30
2. d * e    -> 18
3. a + b * c = a + 30   -> 33
4. a + b * c - d * e = 33 - 18 -> 15
```

```
// Precedencia-szabály
public class Prec1{
        public static void main(String[] args) {
                int a = 3, b = 5, c = 6, d = 9, e = 2, ered;
                ered = a + b * c - d * e;
                System.out.println("Eredmeny: " + ered);
        }
}
```

A program futásának eredménye:

```
Eredmeny: 15
```

A kiértékelés sorrendje a matematikából ismert zárójelek segítségével megváltoztatható. Megjegyezzük, hogy a Java nyelvben csak a kerek zárójel `()` használható, bármilyen mélységű zárójelezést is hajtunk végre (*Prec2.*)

Az *(a + b) * (c - d) * e* kifejezés kiértékelésének lépései:

```
1. (a + b) -> 8
5. (c - d) -> -3
6. 8 * -3 -> -24
7. -24 * 2 -> -48
```

```
// Precedencia szabály
public class Prec2{
        public static void main(String[] args) {
                int a = 3, b = 5, c = 6, d = 9, e = 2, ered;
                ered = (a + b) * (c - d) * e;
                System.out.println("Eredmeny: " + ered);
        }
}
```

A program futásának eredménye:

```
Eredmeny: -48
```

Ha azonos precedenciájú műveletek szerepelnek egy aritmetikai kifejezésben, akkor a balról-jobbra szabály lép életbe.

A példában egyértelmű, hogy először a b * c szorzás kerül végrehajtásra a precedencia-szabály miatt. Mi történik a továbbiakban, mivel a * és az / azonos precedenciájú?

```
a + b * c / d * e;
```

A *d* operandus jobb (*) és bal (/) oldalán azonos precedenciájú művelet szerepel, ekkor a balról-jobbra szabály alapján a *d* operandus a tőle balra álló művelethez tartozik, azaz az / (osztás) műveleti jelhez. Ezért *b*c* szorzat után először a *d* operandussal való osztás, majd az *e* operandussal való szorzás hajtódik végre. Matematikai képletre átírva ez jobban látszik:

$$a + \frac{b \cdot c}{d} \cdot e$$

Ha azonban a feladat az alábbi képlet programozása, akkor az ezt leíró utasítást kétféleképpen is megadhatjuk (*Prec3*).

$$a + \frac{b \cdot c}{d \cdot e}$$

Tekintsük meg a *Prec3* mintafeladat utasításait és eredményét! Ha a nevezőben szorzat áll, azt kétféleképpen programozhatjuk:

1. Zárójelbe tesszük a nevezőt, így mindenképpen a szorzattal osztunk:

```
a + b * c / (d * e);
```

2. Az is jó megoldás, ha a szorzat mindkét operandusával osztunk:

```
a + b * c / d / e;
```

```
// Precedencia szabály
public class Prec3{
       public static void main(String[] args) {
           double a = 3, b = 5, c = 12, d = 10, e = 2, ered1, ered2;
           ered1 = a + b * c / (d * e);
           ered2 = a + b * c / d / e;
           System.out.println("Eredmeny1: " + ered1);
           System.out.println("Eredmeny2: " + ered2);
       }
}
```

A program futásának eredménye:

```
Eredmeny1: 6.0
Eredmeny2: 6.0
```

3.2.2 Az asszociativitás szabály

Az asszociativitás határozza meg, hogy az adott precedenciaszinten található műveleteket balról-jobbra vagy jobbról-balra haladva kell elvégezni.

Az értékadó utasítások csoportjában a kiértékelés jobbról-balra halad, ami lehetővé teszi a Java nyelvben több változó együttes értékadását:

```
a = b = c = 0;  azonos az  a = (b = (c = 0));
```

3.3 Értékadó operátorok

Írjunk programot, amely bemutatja az értékadás szabályait! (*Ertekado*)

```
// Értékadó operátorok
public class Ertekado{
      public static void main(String[] args) {
                double a, b, c = 6, ered1, ered2;
                a = (b = 6)/4 * 3;
                System.out.println("a: " + a);
                System.out.println("b: " + b);
                ered1 = ered2 = 2;
                ered1 /= a + 2 - b;
                ered2 += 4 * c - 3;
          System.out.println("Eredmeny1: " + ered1);
          System.out.println("Eredmeny2: " + ered2);
      }
}
```

A program futásának eredménye:

```
a: 4.5
b: 6.0
Eredmeny1: 4.0
Eredmeny2: 23.0
```

Megjegyzések:

- `a = ( b = 5)/4 * 2;` kifejezésben az *a* és a *b* is értéket kap.
- A jobbról-balra szabály lehetővé teszi több változó ugyanazon értékre való állítását: `ered1 = ered2 = 2;`
- Ha egy változó a jobb és a bal oldalon is szerepel, akkor a kifejezés tömörebben is felírható úgy, hogy a műveleti jelet az egyenlőség elé tesszük (szóköz nélkül). Ezek az összetett értékadás műveletei.

  `ered1 = ered1/(b + 2 - d);` → `ered1 /= b + 2 - d;`

  Lehet még: `+=, -=, *=, %=, &=, |=, ^=, <<=, >>>=, >>=`

3.4 Léptető (inkrementáló/dekrementáló) operátorok

Egy numerikus változó eggyel való növelése, illetve csökkentése hagyományos formája helyett érdemes a léptető operátort alkalmazni:

`i = i+1;` és `i += 1;` helyett `i++;` vagy `++i;`
`i = i-1;` és `i -= 1;` helyett `i--;` vagy `--i;`
`++i; --i;` prefixes alakok.
`i++; i--;` postfixes alakok.

Vigyázzunk, ha a *prefix* operátort kifejezésben használjuk, akkor a léptetés a kifejezés kiértékelése előtt megy végbe! A postfixes alaknál a léptetés a kifejezés kiértékelése után következik be, így a kifejezésben például az *i++* értékeként az *i* eredeti értéke szerepel.

```
int k = 1, m = 7, n;
n = ++k - m--;  // k 2, m 6 és n -5 lesz
```

Írjunk programot, amely bemutatja több egész típusú változó 1-gyel való léptetését! (*Lepteto*)

```
// Léptetés
public class Lepteto{
      public static void main(String[] args) {
            int i = 3, j = 6, ered1, ered2, ered3, ered4;
            System.out.println("i: " + i);
            System.out.println("j: " + j);
            ered1 = i++ + ++j;
            System.out.println("Muvelet vegrehajtaskor: ");
            System.out.println("j: " + j);
            System.out.println("ered1: " + ered1);
            System.out.println("Muvelet utan: ");
            System.out.println("i: " + i);
            System.out.println("j: " + j);
            System.out.println("----------------");
            ered2 = ++i + j++;
            System.out.println("i: " + i);
            System.out.println("ered2: " + ered2);
            System.out.println("Muvelet utan: ");
            System.out.println("i: " + i);
            System.out.println("j: " + j);
            System.out.println("----------------");
            ered3 = i-- + j--;
            System.out.println("ered3: " + ered3);
            System.out.println("Muvelet utan: ");
            System.out.println("i: " + i);
            System.out.println("j: " + j);
            System.out.println("----------------");
            ered4 = --i + --j;
            System.out.println("i: " + i);
```

```
                System.out.println("j: " + j);
                System.out.println("ered4: " + ered4);
        }
}
```

A program futásának eredménye:

```
i: 3
j: 6
Muvelet vegrehajtaskor:
j: 7
ered1: 10
Muvelet utan:
i: 4
j: 7
----------------
i: 5
ered2: 12
Muvelet utan:
i: 5
j: 8
----------------
ered3: 13
Muvelet utan:
i: 4
j: 7
----------------
i: 3
j: 6
ered4: 9
```

3.5 Bitműveletek

Egész típusú változókon bitenként végezhetünk logikai műveleteket:

- ~ 1-es komplemens (bitenkénti tagadás),
- & bitenkénti ÉS,
- | bitenkénti VAGY,
- ^ bitenkénti kizáró VAGY.

Írjunk programot, amelyben **int** típusú változókon végzünk *VAGY* (|) *ÉS* (&) bitenkénti logikai műveleteket! (*Bitmuv*)

```
// Bitműveletek
public class Bitmuv{
        public static void main(String[] args) {
                int i = 0x004, j1 = 0x0002, j2 = 0x007, Vagy, Es;
                Vagy = i | j1;
                System.out.println("i: " + i);
```

```
                System.out.println("j1: " + j1);
                System.out.println("VAGY: " + Vagy);
                System.out.println();
                Es = i & j2;
                System.out.println("i: " + i);
                System.out.println("j2: " + j2);
                System.out.println("ES: " + Es);
        }
}
```

A program futásának eredménye:

```
i: 4
j1: 2
VAGY: 6

i: 4
j2: 7
ES: 4
```

0004 | 0002 = 0006

```
  0000 0000 0000 0100
| 0000 0000 0001 0010
  0000 0000 0011 0110   → 6
```

0004 & 0007 = 0004

```
  0000 0000 0000 0100
& 0000 0000 0000 0111
  0000 0000 0000 0100   → 4
```

Írjunk programot, amely megállapítja, hogy egy előjel nélküli egész szám n-edik bitje (0<=0 n <=31) 0 vagy 1 értékű! (*Bitmuv2*)

```
// bitművelet
import java.io.*;
public class Bitmuv2{
        public static void main(String[] args) throws IOException  {

                BufferedReader be = new BufferedReader
                                        (new InputStreamReader(System.in));
                System.out.print("Elojel nelkuli pozitiv szam = ");
                int szam =Integer.valueOf(be.readLine()).intValue();
                System.out.print("Vizsgalando bit(2..31) = ");
                int n =Integer.valueOf(be.readLine()).intValue();
                int maszk = 1;

                maszk <<=n;
                if ((szam & maszk)>0)
                System.out.println(n + ". helyen: van 1 bit");
                else
                System.out.println(n + ". helyen: van 0 bit");
        }
}
```

A program futásának eredményei:

```
Elojel nelkuli pozitiv szam = 8
Vizsgalando bit(2..31) = 3
3. helyen: van 1 bit

Elojel nelkuli pozitiv szam = 8
Vizsgalando bit(2..31) = 2
2. helyen: van 0 bit
```

Írjunk programot, amelyben **int** típusú értékeken bemutatja a *KIZÁRÓ VAGY* művelettel való kódolást, és a kódolt adat visszaállítását! (*Kodolo*)

```
// Kódolás
public class Kodolo{
        public static void main(String[] args) {
                int kod = 123456, adat = 1974;
                System.out.println("Kodolando adat: " + adat);
                adat = adat ^ kod;
                System.out.println("Kodolt adat: " + adat);
                adat = adat ^ kod;
                System.out.println("Visszakodolas (^): " + adat);
        }
 }
```

A program futásának eredménye:

```
Kodolando adat: 1974
Kodolt adat: 124406
Visszakodolas (^): 1974
```

3.5.1 Biteltoló műveletek

Egész szám bitjeinek *n* lépéssel történő balra tolása (*szám*<<*n*) a szám 2^n értékkel való szorzását eredményezi.

Egész szám bitjeinek *n* lépéssel történő jobbra tolása (*szám*>>*n*)a szám 2^n értékkel elvégzett egész osztásának felel meg.

Megjegyezzük, hogy egy egész szám 2^n-nel való szorzásának/osztásának ez a leggyorsabb módja. A Java nyelv >>> operátora szintén jobbra tolja a biteket, azonban a legfelső helyiértékű bitet nullázza (így az **int** vagy **long** típusú eredmény nem lesz előjelhelyes).

Írjunk programot, amely egy **int** típusú értéket kettővel jobbra, illetve balra léptet! (*Bitlepteto*)

```
// Bitek léptetése
public class Bitlepteto{
        public static void main(String[] args) {
                int a = 12;
                System.out.println("Adat: " + a);
                System.out.println(a + " >> 2 = " + (a >> 2));
                System.out.println(a + " << 2 = " + (a << 2));
        }

 }
```

A program futásának eredménye:

```
Adat: 12
12 >> 2 = 3
12 << 2 = 48
```

3.6 Konstansok használata

A **final** kulcsszó után adjuk meg egyenként a konstansokat nagybetűvel írva.

Írjunk programot, amelyben a gömb felszínének és térfogatának számítása során a π értékét beépített konstansként használjuk, valamint a kiírandó szöveget a ***String*** osztály segítségével tároljuk! Használjuk a ***pow()*** matematikai függvényt a hatványozásra! (*Konstans*)

```
// Konstans és matematikai függvény használata
public class Konstans{
        public static void main(String[] args) {
                final String SZOVEG1 = "A gomb felszine : ";
                final String SZOVEG2 = "A gomb terfogata: ";
                double sugar = 1, gombfelsz, gombterf;

                gombfelsz = 4 * Math.pow(sugar, 2)* Math.PI;
                gombterf = 4 * Math.pow(sugar, 3)* Math.PI/3;
                System.out.println("A gomb sugara: " + sugar);
                System.out.println(SZOVEG1 + gombfelsz);
                System.out.println(SZOVEG2 + gombterf);
        }
 }
```

A program futásának eredménye:

```
A gomb sugara: 1.0
A gomb felszine : 12.566370614359172
A gomb terfogata: 4.1887902047863905
```

4. A Java nyelv utasításai

A Java nyelv utasításait többféleképpen is csoportosíthatjuk. A szelekciós utasítások az egyszerű, illetve többszörös elágazást lehetővé tevő **if** és **switch** utasításokat jelentik. Az iterációs utasítások csoportja a Java nyelv ciklusutasításait (**while**, **for**, **do while**) tartalmazza.

4.1 Utasítások és blokkok

Tetszőleges kifejezés utasítássá válik, ha pontosvesszőt (;) helyezünk mögé – ez a kifejezés-utasítás:

```
--k;  i++;
```

Egyetlenegy pontosvesszőből áll az üres utasítás, amely utasítást helyettesít, amennyiben mást nem akarunk végrehajtani azon a helyen.

Kapcsos zárójelekkel { } több utasítást egyetlen utasítássá vonhatunk össze. A szelekciós és iterációs utasítások alaphelyzetben csupán egy tevékenységet (utasítás) hajtanak végre. A kapcsos zárójelekkel összefogott utasítások azonban egy utasításnak látszanak számukra. Ha elfelejtjük blokkba tenni a megfelelő utasításokat, akkor általában csak az első hajtódik végre.

4.2 Szelekciós utasítások

A szelekciós utasításokkal válogatást végezhetünk valamilyen feltétel alapján. Többfajta szelekciós utasítás létezik: egyágú, kétágú, többágú. A szelekciókat egymásba is ágyazhatjuk.

4.2.1 Relációs műveletek és logikai operátorok

A szelekciós utasításokban relációs műveletek használunk a feltételek megfogalmazásához. A relációs műveletben két aritmetikai kifejezés relációs műveleti jellel van összekapcsolva.:

`kifejezés1 > kifejezés2`	akkor igaz, ha a *kifejezés1* nagyobb, mint a *kifejezés2*
`kifejezés1 >= kifejezés2`	akkor igaz, ha a *kifejezés1* nagyobb, vagy egyenlő, mint a *kifejezés2*
`kifejezés1 < kifejezés2`	akkor igaz, ha a *kifejezés1* kisebb, mint a *kifejezés2*
`kifejezés1 <= kifejezés2`	akkor igaz, ha a *kifejezés1* kisebb, vagy egyenlő, mint a *kifejezés2*
`kifejezés1 == kifejezés2`	akkor igaz, ha a *kifejezés1* egyenlő a *kifejezés2*-vel
`kifejezés1 != kifejezés2`	akkor igaz, ha a *kifejezés1* nem egyenlő a *kifejezés2*-vel

Logikai operátorokat logikai műveletek során használjuk, azonban relációkat is összekapcsolhatnak. A logikai operátorokat a precedencia sorrendjében mutatjuk be:

tagadás **negáció**	`! kifejezés`	A *kifejezés* logikai értékét megfordítja.
logikai ÉS **konjukció**	`kifejezés1 && kifejezés2`	Ha mindkét logikai kifejezés igaz, akkor a művelet eredménye igaz, ha bármelyik kifejezés hamis, a művelet eredménye hamis.
logikai VAGY **diszjunkció**	`kifejezés1 || kifejezés2`	Ha bármelyik logikai kifejezés igaz, akkor a művelet eredménye igaz. A művelet akkor hamis, ha mindkét logikai kifejezés hamis.

Példák a relációs műveletekre és a logikai operátorok használatára:

`x % 2 == 0`	akkor igaz, ha az *x* egész változó 2-es osztási maradéka (modulusa) zérus, azaz 2-vel osztható, tehát az *x* páros szám,
`x % 3 == 0`	akkor igaz, ha az *x* egész változó 3-as modulusa zérus, azaz az x hárommal osztható,
`x1 > 0 && x2 > 0 && x3 > 0`	igaz, ha *x1*, *x2* és *x3* mindegyike nagyobb nullánál, vagyis mind a három szám pozitív,
`x >= 0 && x % 2 == 1`	akkor igaz, ha az x nagyobb nullánál és az x változó 2-es modulusa 1, azaz az x pozitív, páratlan szám.

4.2.2 A feltételes operátor

A feltételes operátor (?:) három operandussal rendelkezik:

```
feltétel_kif  ? igaz_kif : hamis_kif
```

Ha a *feltétel_kif* igaz (**true**), akkor az *igaz_kif* értéke adja a feltételes kifejezés értékét, ellenkező esetben a kettőspont (:) után álló *hamis_kif*.

Írjunk programot, amelyben feltételes operátorokat használunk két változó közül a kisebb és a nagyobb értékű kiválasztására! (*Feltetelop*)

```
public class Feltetelop{
        public static void main(String[] args) {
                int x = 12, y = 2, min, max;
                min = x < y ? x : y;
                max = x > y ? x : y;
                System.out.println("Legkisebb ertek (min) : " + min);
                System.out.println("Legnagyobb ertek (max): " + max);
        }
}
```

A program futásának eredménye:

```
Legkisebb ertek (min) : 2
Legnagyobb ertek (max): 12
```

4.2.3 Az if utasítás

Feltételes utasításnak három formája van.

- Az egyszerű **if** utasítás:

```
if (feltétel_kif)
      utasítás
```

  Ha a feltétel igaz (**true**), akkor az *utasítás* végrehajtódik.

- Az **if-else** utasításnál a feltétel teljesülése esetén az *utasítás1* hajtódik végre, különben pedig az *utasítás2*.

```
if (feltétel_kif)
      utasítás1
else
      utasítás2
```

- A láncolt **if-else-if** utasításnál tetszőleges számú feltételt sorba láncolhatunk, mely sort az **else** ág zárja. Ha egyik feltétel sem teljesül, akkor az **else** ág hajtódik végre:

```
if (feltétel_kif1)
      utasítás1
else if (feltétel_kif2)
      utasítás2
      . . .
else
      utasításn
```

Írjunk programot, amely egy egész számról kijelzi, hogy páros vagy páratlan, majd pedig megnézi a 100-hoz való viszonyát! (*Feltetel1*)

```
// Feltételes utasítás
public class Feltetel1{
      public static void main(String[] args) {
            int x;
             BufferedReader be = new BufferedReader
                                 (new InputStreamReader(System.in));
             System.out.print("Adat = ");
             x =Integer.valueOf(be.readLine()).intValue();
             if (x % 2 == 0)
               System.out.println("Paros szam: " + x);
             else
               System.out.println("Paratlan szam: " + x);
```

```
            if (x > 100)
              System.out.println("Nagyobb mint 100: " + x);
            else
              System.out.println("Kisebb vagy egyenlo mint 100: " + x);
        }
 }
```

A program futásának eredménye:

```
Adat = 10
Paros szam: 10
Kisebb vagy egyenlo, mint 100: 10
```

Írjunk programot, amely egy egész számról kijelzi, hogy pozitív, negatív vagy zérus! (*Feltetel2*)

```
// Feltételes utasítás
public class Feltetel2{
       public static void main(String[] args) {
                int x;
                BufferedReader be = new BufferedReader
                                    (new InputStreamReader(System.in));
                System.out.print("Adat = ");
                x =Integer.valueOf(be.readLine()).intValue();

                if (x > 0)
                  System.out.println(x + " pozitiv szam");
                else if (x < 0)
                  System.out.println(x + " negativ szam");
                else
                  System.out.println(x + " zerus");
       }
 }
```

A program futásának eredménye:

```
Adat = -4
-4 negativ szam
```

Írjunk programot, amely három valós pozitív számnak kiszámítja az átlagát, hibát jelez, ha a kívánt (pozitív) feltétel nem teljesül! (*Feltetel3*)

```
// Feltételes utasítás használata
public class Feltetel2{
       public static void main(String[] args) throws IOException {
                double x1, x2, x3, atlag;
                  BufferedReader be = new BufferedReader
                                      (new InputStreamReader(System.in));
                  System.out.println("Harom pozitiv szam atlaga");
                  System.out.print("1. adat : ");
                  x1 =Double.valueOf(be.readLine()).doubleValue();
                  System.out.print("2. adat : ");
                  x2  =Double.valueOf(be.readLine()).doubleValue();
```

```
                System.out.print("3. adat : ");
                x3 =Double.valueOf(be.readLine()).doubleValue();
                if (x1 > 0 && x2 > 0 && x3 > 0) {
                  atlag = (x1 + x2 + x3)/3;
                  System.out.println("Atlag: " + atlag);
                }
                else System.out.println("Hibas adat!");
        }
}
```

A program futásainak eredménye:

```
Harom pozitiv szam atlaga
1. adat : 1
2. adat : 2
3. adat : 3
Atlag: 2.0

Harom pozitiv szam atlag
1. adat : 1
2. adat : -2
3. adat : 3
Hibas adat!
```

Írjunk programot, amely beolvas egy pozitív egész számot, megvizsgálja, ha pozitív, akkor négyzetgyököt von belőle, ha negatív, akkor négyzetre emeli! (*Feltetel4*)

```
// Feltételes utasítás használata
public class Feltetel4{
        public static void main(String[] args) throws IOException  {
                double adat, eredmeny;
                 BufferedReader be = new BufferedReader
                     (new InputStreamReader(System.in));
                System.out.print("Adat = ");
                adat =Integer.valueOf(be.readLine()).intValue();
                if (adat > 0) {
                  eredmeny = Math.sqrt(adat);
                  System.out.println("Gyoke: " + eredmeny);
                }
                else {
                  eredmeny = Math.pow(adat,2);
                  System.out.println("Negyzete: " + eredmeny);
                }
        }
}
```

A program futásainak eredménye:

```
Adat = 16
Gyoke: 4.0

Adat = -3
Negyzete: 9.0
```

Írjunk programot, amely beolvas egy pozitív egész számot, megvizsgálja, hogy páros vagy páratlan, valamint osztható-e 3-mal! (*Feltetel5*)

```
// Feltételes utasítás
public class Feltetel5{
       public static void main(String[] args) throws IOException{
       int x;
       BufferedReader be=
       new BufferedReader(new InputStreamReader(System.in));
              System.out.print("Pozitiv szam = ");
              x =Integer.valueOf(be.readLine()).intValue();
              if (x > 0 && x % 2 == 1) {
                System.out.print(x + " paratlan ");
                if (x % 3 == 0)
                     System.out.println("oszthato 3-mal");
                else
                     System.out.println("nem oszthato 3-mal");
              }
              else if (x >= 0) {
                System.out.print(x + " paros ");
                if (x % 3 == 0)
                     System.out.println("oszthato 3-mal");
                else
                     System.out.println("nem oszthato 3-mal");
              }
              else
                  System.out.println(x + " hibas adat!");
          }
       }
```

A program futásainak eredménye:

```
Pozitiv szam = 15
15 paratlan oszthato 3-mal

Pozitiv szam = 11
11 paratlan nem oszthato 3-mal

Pozitiv szam = 30
30 paros oszthato 3-mal

Pozitiv szam = 14
14 paros nem oszthato 3-mal
```

Írjunk programot, amely beolvassa egy dolgozat pontszámát, és a ponthatárok alapján leosztályozza azt! (*Feltetel6*)

Ponthatár	*Érdemjegy*
0 - 50	1
51 - 65	2
66 - 75	3
76 - 85	4
86 - 100	5

```
public class Feltetel6 {
      public static void main(String[] args) throws IOException {
           int pontszam, erdemjegy = 0;
           BufferedReader be =
           new BufferedReader(new InputStreamReader(System.in));
           System.out.print("Pontszam: ");
           pontszam =Integer.valueOf(be.readLine()).intValue();
           if (pontszam > 0 && pontszam <= 100)
           {
              if (pontszam >=0 && pontszam <= 50)
                 erdemjegy = 1;
              else if (pontszam >= 51 && pontszam <= 65)
                 erdemjegy = 2;
              else if (pontszam >= 66 && pontszam <= 75)
                 erdemjegy = 3;
              else if (pontszam >= 76 && pontszam <= 85)
                 erdemjegy = 4;
              else // ekkor a pontszam >= 86 és pontszam <= 100
                 erdemjegy = 5;
              System.out.println("Erdemjegy: " + erdemjegy);
           }
           else
              System.out.println("Hibas adat!");
    }
}
```

A program futásainak eredménye:

```
Pontszam: 78
Erdemjegy: 4

Pontszam: 99
Erdemjegy: 5
```

Írjunk programot, amely beolvassa a víz hőmérsékletét, és a hőmérséklet alapján szövegesen megjeleníti a víz halmazállapotát! (*Feltetel7*)

A számítási algoritmus:

fok <= 0 szilárd (jég), 0 < fok <100 folyékony (víz), fok >= 100 légnemű (gőz).

```
public class Feltetel7 {
      public static void main(String[] args) throws IOException {
          BufferedReader be =
          new BufferedReader(new InputStreamReader(System.in));
          System.out.print("Viz homerseklete = ");
          int fok =Integer.valueOf(be.readLine()).intValue();
          if (fok <= 0)
              System.out.println("Halmazallapota: szilard (jeg)");
          else if (fok > 0 && fok < 100)
              System.out.println("Halmazallapota: folyekony (viz)");
          else // ekkor a fok >= 100
              System.out.println("Halmazallapota: legnemu (goz)");
      }
}
```

A program futásainak eredménye:

```
Viz homerseklete = 12
Halmazallapota: folyekony (viz)

Viz homerseklete = -2
Halmazallapota: szilard (jeg)

Viz homerseklete = 100
Halmazallapota: legnemu (goz)
```

4.2.4 A switch utasítás

A vezérlés egy *kifejezés* értékétől függően több lehetséges utasítás valamelyikére kerül a **switch** típusú elágaztatásban. Az utasítás általános alakja:

```
switch (kifejezés) {
   case konstans_kifejezés:
         utasítások
   case konstans_kifejezés:
         utasítások
    . . .
   default:
         utasítások
}
```

- A **switch** utasítás kiértékeli a *kifejezést,* és arra a **case** címkére adja át a vezérlést, amelyben a *konstans_kifejezés* értéke megegyezik a *kifejezés* értékével. A **default** címkével megjelölt utasítással folytatódik a program futása, ha egyikkel sem egyezik meg.
- A konstans kifejezés csak egész jellegű (**int, long, char** stb. lehet).
- Az adott címkéhez tartozó programrészlet végrehajtása után a **break** utasítással léphetünk ki a **switch** utasításból.

Írjunk programot, amelyben egy karakter értéke meghatározza, hogy összeget (+) vagy különbséget (-) számolunk! Szövegesen jelenítsük meg az eredményt! (*Switch1*)

```
// Választható művelet
public class Switch1{
       public static void main(String[] args) {
              char jel;
              double a, b, ered;
              jel = '+';  a = 3; b = 7;
              switch( jel) {
                 case '+':
                    ered = a+b;
                    System.out.println(a + " + " + b + " = " + ered);
                    break;
```

```
                case '-':
                    ered = a-b;
                    System.out.println(a + " - " + b + " = " + ered);
                    break;
                default:
                    System.out.println("Hibas jel: " + jel);
            }
        }
}
```

A program futásainak eredménye:

```
3.0 + 7.0 = 10.0
```

Írjunk programot, amely beolvassa egy dolgozat pontszámát és a ponthatárok alapján leosztályozza azt! Az osztályzatot számmal és szövegesen egyaránt jelenítsük meg! (*Switch2*)

```
public class Switch2{
    public static void main(String[] args) throws IOException {
        int pontszam, erdemjegy;
        BufferedReader be= new BufferedReader(
                            new InputStreamReader(System.in));
        System.out.print("Pontszam = ");
        pontszam =Integer.valueOf(be.readLine()).intValue();
        if (pontszam >= 0 && pontszam <=100) {
          if (pontszam >= 86)
              erdemjegy = 5;
          else if (pontszam >= 76)
              erdemjegy = 4;
          else if (pontszam >= 66)
              erdemjegy = 3;
          else if (pontszam >= 51)
              erdemjegy = 2;
          else
              erdemjegy = 1;
          System.out.print("Erdemjegy: " + erdemjegy);
          switch (erdemjegy) {
             case 1:  System.out.println(" (elegtelen)");
                      break;
             case 2:  System.out.println(" (elegseges)");
                      break;
             case 3:  System.out.println(" (kozepes)");
                      break;
             case 4:  System.out.println(" (jo)");
                      break;
             case 5:  System.out.println(" (jeles)");
          }
        }
        else
           System.out.println("Hibas adat ");
    }
}
```

A program futásainak eredménye:

```
Pontszam = 78
Erdemjegy: 4 (jo)

Pontszam = 97
Erdemjegy: 5 (jeles)
```

4.3 Ciklusutasítások

A programozási nyelvekben bizonyos utasítások automatikus ismétlését biztosító programszerkezetet iterációnak vagy ciklusnak (*loop*) nevezzük. Ez az ismétlés mindaddig tart, amíg az ismétlési feltétel igaznak bizonyul.

4.3.1 A while ciklus

A **while** ciklus mindaddig ismétli a ciklus törzsében lévő utasításokat, míg a vizsgált *kifejezés* (vezérlőfeltétel) értéke igaz (**true**). Az utasítás általános alakja:

```
while (feltétel_kif)
    utasítás;  // a ciklus törzse
```

Ez egy ún. elől tesztelt ciklus. Nem biztos, hogy a ciklusmag egyáltalán lefut.

Írjunk programot, amely a 0-tól 9-ig futó **while** ciklusban kiszámítja a páratlan számok szorzatát és a páros számok összegét! (*While1*)

```
// while utasítás használata
public class While1{
       public static void main(String[] args) {
          int osszeg = 0, szorzat = 1;
          int i = 0;
          while (i < 10) {
               if (i % 2 == 0)
                  osszeg += i;
               else
                  szorzat *= i;
               i++;
          }
          System.out.println("Paros szamok osszege: " + osszeg);
          System.out.println("Paratlan szamok szorzata: " + szorzat);
       }
}
```

A program futásának eredménye:

```
Paros szamok osszege: 20
Paratlan szamok szorzata: 945
```

Készítsünk programot, amely adott számú kockadobást szimulál, és megszámlálja az egyes eredmények előfordulását! (*While2*)

```
import java.io.*;
public class While2{
      public static void main(String[] args) throws IOException{
           int dobas, dobasok_szama, db_1 = 0, db_2= 0,
               db_3=0, db_4=0, db_5=0,db_6=0;
           BufferedReader be = new BufferedReader
                                   (new InputStreamReader(System.in));
           System.out.print("Dobasok szama: ");
           dobasok_szama = Integer.valueOf(be.readLine()).intValue();
           int i = 0;
           while (i < dobasok_szama) {
               dobas = (int)(Math.random() * 6) + 1;
               switch( dobas) {
                     case 1: db_1++; break;
                     case 2: db_2++; break;
                     case 3: db_3++; break;
                     case 4: db_4++; break;
                     case 5: db_5++; break;
                     case 6: db_6++; break;
                }
               i++;
           }
           System.out.println("1 dobasa: " + db_1);
           System.out.println("2 dobasa: " + db_2);
           System.out.println("3 dobasa: " + db_3);
           System.out.println("4 dobasa: " + db_4);
           System.out.println("5 dobasa: " + db_5);
           System.out.println("6 dobasa: " + db_6);
      }
}
```

A program futásának eredménye:

```
Dobasok szama: 100
1 dobasa: 16
2 dobasa: 19
3 dobasa: 9
4 dobasa: 22
5 dobasa: 18
6 dobasa: 16
```

4.3.2 A for ciklus

A **for** utasítást általában akkor használjuk, ha a ciklus törzsében megadott utasításokat egy számsorozat elemein végigfuttatva kívánjuk végrehajtani. A **for** utasítás általános formája:

```
for(inicializáló_kif; feltétel_kif; léptető_kif)
   utasítás;     // a ciklus törzse
```

A **for** ciklus a **while** utasítás speciális megfogalmazásának tekinthető. Felhívjuk a figyelmet arra, hogy az inicializáló kifejezés változódefiníciót is tartalmazhat – ekkor a ciklusváltozó csak a cikluson belül használható.

Írjunk programot, amely a 0-tól 9-ig futó **for** ciklusban kiszámítja a páratlan számok szorzatát és a szorzat gyökét, valamint a páros számok összegét és az átlagát! (*For1*)

```
// for utasítás használata
public class For1{
    public static void main(String[] args) {
       int db = 0;
       int osszeg = 0, szorzat = 1;
       double gyok, atlag;

       for (int i = 0; i<10; i++)  // az i csak a ciklusban érhető el
          if (i % 2 == 0) {
             osszeg += i;
             db++;
          }
          else
             szorzat *= i;

       atlag = (double)osszeg/db;
       gyok = Math.sqrt(szorzat);
       System.out.println("Paros szamok osszege: " + osszeg);
       System.out.println("Paros szamok atlaga: " + atlag);
       System.out.println("Paratlan szamok szorzata: " + szorzat);
       System.out.println("Paratlan szamok szorzatanak gyoke: "+gyok);
    }
}
```

A program futásának eredménye:

```
Paros szamok osszege: 20
Paros szamok atlaga: 4.0
Paratlan szamok szorzata: 945
Paratlan szamok szorzatanak gyoke: 30.740852297878796
```

4.3.3 A do-while ciklus

A **do-while** ciklus törzse egyszer, a feltétel tesztelése nélkül, mindig végrehajtódik (hátul tesztelő ciklus). A ciklus addig működik, amíg a *feltételes kifejezés* igaz (**true**). Az utasítás általános alakja:

```
do
     utasítás  // a ciklus törzse
while (feltétel_kif);
```

Írjunk programot, amely a valós alapot pozitív, illetve negatív egész kitevőre emeli! (*Dowhile1*)

Az összefüggés, amelynek alapján negatív kitevőjű hatványt számoljuk:

$$a^{-n} = \frac{1}{a^n}$$

- Először kiszámítjuk az alapnak a pozitív hatványát úgy, hogy a **do-while** ciklus feltételében a kitevő abszolút értékét vizsgáljuk. Az abszolút érték meghatározása az *Math.abs()* matematikai függvényt használjuk.
- Ezután megvizsgáljuk a kitevőt, amennyiben negatív, úgy a hatványértéknek a reciprokát képezzük.

```
// a do-while ciklus használata
import java.io.*;
public class Dowhile1{
    public static void main(String[] args) throws IOException{
          double alap, hatvany = 1;
          int kitevo, i;
          BufferedReader be=new BufferedReader
                                (new InputStreamReader(System.in));
          System.out.println("Valos szamot emeljunk egesz kitevore");
          System.out.print("Alap  = ");
          alap =Integer.valueOf(be.readLine()).intValue();
          System.out.print("Kitevo  = ");
          kitevo =Integer.valueOf(be.readLine()).intValue();
          if (kitevo == 0)
              System.out.println("Hatvany erteke: " + hatvany);
          else  {
              i = 1; hatvany = alap;
              do {
                 hatvany *= alap;
                 i++;
              } while (i < Math.abs(kitevo));
              if (kitevo < 0)
                  hatvany = 1./hatvany;
              System.out.println("Hatvany erteke: " + hatvany);
          }
    }
}
```

A program futásainak eredménye:

```
Valos szamot emeljunk egesz kitevore
Alap  = 3
Kitevo  = 2
Hatvany erteke: 9.0

Valos szamot emeljunk egesz kitevore
Alap  = 2
Kitevo  = -3
Hatvany erteke: 0.125
```

```
Valos szamot emeljunk egesz kitevore
Alap   = 3
Kitevo  = 0
Hatvany erteke: 1.0
```

Írjunk programot, amely egy pozitív, 2-nél nagyobb számnak megkeresi az osztóit, illetve kijelzi, ha a szám prímszám! (*Dowhile2*)

Megjegyzés: A megoldásban **do-while** ciklussal biztosítjuk, hogy megfelelő adatot olvassunk be!

```
// a do-while ciklus használata
import java.io.*;
public class Dowhile2{
     public static void main(String[] args) throws IOException{
          int szam, fele, db = 0;
          BufferedReader be = new BufferedReader
                                   (new InputStreamReader(System.in));
          do {
               System.out.print("Pozitiv egesz szam (>2): ");
               szam =Integer.valueOf(be.readLine()).intValue();
          } while (szam <= 2);

          if (szam > 2) {
               fele = szam/2;
               System.out.println("Osztoi: ");
               for(int oszto = 2; oszto <= fele; oszto++)
                  if (szam % oszto == 0) {
                       System.out.println(oszto);
                       db++;
                  }
          }
          if (db == 0)
               System.out.println("Primszam");
     }
}
```

A program futásainak eredménye:

```
Adat = 18
Pozitiv egesz szam (>2): 18
Osztoi:
2
3
6
9

Adat = 13
Pozitiv egesz szam (>2): 13
Osztoi:
Primszam
```

4.3.4 A break és a continue utasítások

A Java nyelvből hiányzik a *goto* utasítás. Bizonyos esetekben azonban, elsősorban az összetett utasításokból, kiléphetünk a **break** utasítás segítségével. A **break** címke nélküli változatával az aktuális **switch** vagy ciklusblokkból (**for**, **while**, **do-while**) ugorhatunk ki, azaz a vezérlés a blokk utáni első utasításra kerül. Amennyiben a **break** után címke is szerepel, akkor a vezérlés a megadott címke által kijelölt utasításblokk utáni utasításra adódik át.

A **continue** utasítás hatására a ciklus blokkjában a **continue** utáni utasítások nem kerülnek végrehajtásra, és a ciklus a következő iterációt hajtja végre. Így: a **continue** utasítással feltételtől függően iterációkat hagyhatunk ki. A ciklusokban szereplő **countinue** esetén is használhatunk címkét.

Írjunk programot, amely végtelen **while** ciklusban a nulla érték megadásáig olvas egész számokat! Páratlan és hárommal osztható szám beolvasásakor **continue** utasítással hagyjuk ki az aktuális ciklusléptést! Határozzuk meg a páros adatok számát, valamint a beolvasott összes adat számát! Számítsuk ki a beolvasott adatok összegét, a páratlan és a hárommal oszthatók kivételével! (*While3*)

```
// break és continue használata while ciklusban
public class While3{
     public static void main(String[] args) throws IOException{
          int i, n = 0, adat, osszeg = 0, paros = 0;
          BufferedReader be = new BufferedReader
                              (new InputStreamReader(System.in));
           while(true) {
              System.out.print("Adat = ");
              adat =Integer.valueOf(be.readLine()).intValue();
              n++; // a beolvasott adatok száma
              if (adat % 2 != 0 && adat % 3 == 0) {
                 System.out.println("Paratlan es 3-mal oszthato: "
                                    + adat + "-> continue");
                 continue;
              }
              if (adat %2 == 0) // a páros adatok számlálása
                 paros++;
              if (adat == 0) {
                 System.out.println("zerus adat -> break");
                 break;  // kilépés a végtelen ciklusból
              }
              osszeg += adat;
           }
           System.out.println("Adatok szama: " + n);
           System.out.println("Parosak szama: " + paros);
           System.out.println("Osszeg: " + osszeg);
     }
}
```

A program futásának eredménye:

```
Adat = 3
Paratlan es 3-mal oszthato: 3-> continue
Adat = 12
Adat = 15
Paratlan es 3-mal oszthato: 15-> continue
Adat = 6
Adat = 32
Adat = 21
Paratlan es 3-mal oszthato: 21-> continue
Adat = 0
zerus adat -> break
Adatok szama: 7
Parosak szama: 4
Osszeg: 50
```

Megjegyzések a megoldáshoz:

- A páratlan és hárommal osztható szám beolvasásakor a **continue** utasítás a következő iterációt hajtja végre, a **continue** mögötti utasítások ekkor kimaradnak.
- A páros számokat számláljuk.
- Ha a beolvasott adat 0, akkor a **break** utasítással lépünk ki a végtelen ciklusból!

Írjunk programot, amely végtelen **for**(; ;) ciklusban addig olvas egész számokat, míg egymás után 2 db zérust nem olvasott! A második 0 adatot ne számoljuk bele az adatok darabszámába! Számláljuk meg az adatok darabszámát, és számítsuk ki a beolvasott pozitív adatok szorzatát! (*For2*)

```
// break és continue for utasításban
public class For2{
     public static void main(String[] args) throws IOException  {
          int i, osszes_db = 0, adat, szorzat = 1, nulla = 0;
          BufferedReader be= new BufferedReader
                              (new InputStreamReader(System.in));

          for(;;) {  // végtelen for ciklus
             osszes_db++;
             System.out.print("Adat = ");
             adat =Integer.valueOf(be.readLine()).intValue();

             if (adat < 0)
                 continue;
             else if (adat > 0)
                 szorzat *= adat;

             if (adat == 0)
                 nulla++;
             else
                 nulla = 0;
```

```
            if (nulla == 2) {
                osszes_db--;
                break;
            }
        }
        System.out.println("Osszes adatok szama: " + osszes_db);
        System.out.println("Pozitiv adatok szorzata: " + szorzat);
    }
}
```

A program futásának eredménye:

```
Adat = 3
Adat = -4
Adat = 2
Adat = 0
Adat = 6
Adat = 0
Adat = 0
Osszes adatok szama: 6
Pozitiv adatok szorzata: 36
```

Készítsünk programot, amelyben cikluson és **switch** utasításon kívül használjuk a címkézett **break** utasítást! (*Break1*)

```
public class Break1 {
    public static void main(String[] args) {
        boolean kilep = true;
        ablokk: {
            System.out.println("A blokk eleje");
            bblokk: {
                System.out.println("B blokk eleje");
                cblokk: {
                    System.out.println("C blokk eleje");
                    if (kilep)
                        break bblokk;
                    System.out.println("C blokk vege");
                }
                System.out.println("B blokk vege");
            }
            System.out.println("A blokk vege");
        }
    }
}
```

A program futásának eredménye,
*ha a kilép változó értéke **true**:*

```
A blokk eleje
B blokk eleje
C blokk eleje
A blokk vege
```

*ha a kilép változó értéke **false**:*

```
A blokk eleje
B blokk eleje
C blokk eleje
C blokk vege
B blokk vege
A blokk vege
```

Készítsünk Java alkalmazást, amelyik megkeresi az első, 10 és 20 közé eső Pitagoraszi számhármast! (*Break2*)

```
public class Break2 {
    public static void main(String[] args) {
       int a=0, b=0;
       double c=0;
       boolean talalt= false;

          megvan:  // a break utasítással a külső ciklusból is kilépünk
          for (a=10; a<20; a++)
             for (b=a+1; b<20; b++) {
                c = Math.sqrt(a*a+b*b);
                if ((c-(long)c)<1e-6) {
                    talalt = true;
                    break megvan;
                }
             }

          if (talalt)
              System.out.println(""+ a +", "+ b +", "+ c);
          else
              System.out.println("nincs Pitagoraszi szamharmas");
    }
}
```

A program futásának eredménye:

```
12, 16, 20.0
```

Írjunk alkalmazást, amelyben egy **for** ciklusba ágyazott **switch** utasításban használjuk a **break** és **continue** utasításokat! (*Break3*)

```
public class Break3 {
     public static void main(String[] args) {
       ciklus: for (int a=0; a<7; a++) {
                  switch (a) {
                      case 2:
                           System.out.println("switch break: "+a);
                           break;
                      case 1: case 3:
                           System.out.println("for continue: "+a);
                           continue ciklus; // mint a continue;
                      case 5:
                           System.out.println("for break: "+a);
                           break ciklus; // kilépünk a ciklusból
                      default:
                           System.out.println("switch default: "+a);
                  }
                  System.out.println(""+a);
            }
     }
}
```

A program futásának eredménye:

```
switch default: 0
0
for continue: 1
switch break: 2
2
for continue: 3
switch default: 4
4
for break: 5
```

Jelenítsük meg a 7x7-es szorzótábla főátló alatti részét! (*Continue1*)

```
public class Continue1 {
    public static void main(String[] args) {
          final int N = 7;

          kulso:  // a külső ciklus újabb iterációját indítjuk
          for (int i=1; i <= N; i++) {
            belso:
            for (int j=1; j <= N; j++) {
                 if (j > i) {
                      System.out.println();
                      continue kulso;
                 }
                System.out.print("\t"+ i*j);
            }
          }

       System.out.println();
    }
}
```

A program futásának eredménye:

```
1
2       4
3       6       9
4       8       12      16
5       10      15      20      25
6       12      18      24      30      36
7       14      21      28      35      42      49
```

A strukturált programozás elveit követve, oldjuk meg a *Break2* és a *Continue1* feladatokat a **break**, illetve a **continue** utasítások használata nélkül! (*Brerak2No*, *Continue1No*)

```
// a Break2 feladat break nélküli megoldása
public class Break2No {
     public static void main(String[] args) {
       int a=0, b=0;
```

```
        double c=0;
        boolean talalt= false;

           for (a=10; a<20 && !talalt; a++)
             for (b=a+1; b<20  && !talalt; b++) {
                c = Math.sqrt(a*a+b*b);
                if ((c-(long)c)<1e-6)
                   talalt = true;
             }

           if (talalt)
               System.out.println(""+ (a-1) +", "+ (b-1) +", "+ c);
           else
               System.out.println("nincs Pitagoraszi szamharmas");
     }
}

// a Continue1 feladat continue nélküli megoldása
public class Continue1No {
    public static void main(String[] args) {
        final int N = 7;
        boolean kulso;

        for (int i=1; i <= N; i++) {
          kulso = false;
          for (int j=1; j <= N && !kulso; j++) {
             if (j<=i)
                System.out.print("\t"+ i*j);
             else {
                System.out.println();
                kulso = true;
             }
          }
        }
       System.out.println();
    }
}
```

A program futásának eredménye:

```
        1
        2       4
        3       6       9
        4       8       12      16
        5       10      15      20      25
        6       12      18      24      30      36
        7       14      21      28      35      42      49
```

5. Tömbök használata

Különböző feladatokat mutatunk be az egy- és többdimenziós tömbök kezelésére.

5.1 Tömbök definíciója

A tömbök egynél több értéket is tárolhatnak, az elemek száma rögzített. A tömb tehát adott számú, azonos típusú elemet tartalmazó adattípus. A tömb indexe 0-val kezdődik A tömb nélkülözhetetlen a programozás során, hiszen nagyban megkönnyíti több azonos adat kezelését.

A Java nyelvben a tömböket dinamikusan, a **new** operátor segítségével kell létrehozni, és a tömbök mérete lekérdezhető a ***length*** adatmezővel. A Java teljes körű indexhatár-ellenőrzést végez, és kivétellel jelzi, ha az index kisebb, mint 0, vagy nagyobb vagy egyenlő, mint a ***length*** mező értéke. A tömb területének felszabadításáról a *garbage collection* (szemétgyűjtő) folyamat gondoskodik, melynek átadhatjuk a tömb területét, ha a tömb nevéhez a **null** értéket rendeljük. A tömb elemeit, más nyelvekhez hasonlóan, az indexelés operátorával *[ ]* érjük el.

5.2 Egydimenziós tömbök

Az egyetlen kiterjedéssel rendelkező tömböket szokás vektoroknak nevezni. Egydimenziós tömbhivatkozások definíciója:

```
típus tömbnév[];
típus[] tömbnév;
```

```
public class EgyDimTombok {
    public static void main(String[] args)
    {
        int[] x;              // tömbhivatkozás létrehozása
        double  y[];          // tömbhivatkozás létrehozása
        x = new int[12];      // a 12-elemű egész tömb létrehozása
        y = new double[23];   // a 23-elemű valós tömb létrehozása

        // a tömbök feltöltése
        for (int i=0; i<x.length; i++)
            x[i] = i*i;
        for (int i=0; i<y.length; i++)
            y[i] = 1.0/(i*i+1.0);

        // nincs szükségünk többé a tömbökre
        x = null;
        y = null;
    }
}
```

Egydimenziós tömb létrehozása

A Java nyelvben a tömböt a **new** utasítással a program futása közben hozzuk létre:

```
new elemtípus [méret]
```

Az egész típusú *méret* nem lehet negatív. A tömb *mérete* az ***Integer.MAX_VALUE*** értékét nem haladhatja meg. Például:

```
int[] x = new int[15];
```

Mivel a tömb indexe 0-val kezdődik, ezért **hibás** a deklarációban beírt értékre, mint indexre hivatkozni:

```
x[15] = 5;    // hibás értékadás
```

Az *x* tömbnek valóban 15 darab eleme van, ezért az index csak 0-tól 14-ig változhat.

A Java nyelv lehetővé teszi, hogy a tömböket a definiálásuk során konstans értékekkel inicializáljuk. Ekkor a tömböt a fordítóprogram hozza létre a szükséges mérettel. Ez az értékadás eltér az egyszerű változók esetén használt megoldástól:

```
típus [] tömbnév = { vesszővel tagolt konstansok };
```

Néhány példa az inicializálásra:

```
int[] x = { 22, 81, -41, 0, 36 };
double[] y = { 11.56, 32.6, -13.2, 0.03};
```

Írjunk programot, amely 1-10 közötti véletlen számmal feltölt egy ötelemű, egész típusú tömböt! Számítsuk ki az elemek összegét és átlagát! (*Tomb1*)

```
// Tomb használata
public class Tomb1{
     public static void main(String[] args) {
          int osszeg = 0;
          double atlag;
          int[] x = new int[5];
          System.out.println("A tomb elemei: ");
          for(int i = 0; i < x.length; i++) {
             x[i] = (int)(10*Math.random()+1);
             System.out.println(x[i]);
          }
          for(int i = 0; i < x.length; i++)
              osszeg += x[i];
          atlag = (double)osszeg/x.length;
          System.out.println("A tomb elemeinek osszege: " + osszeg);
          System.out.println("A tomb elemeinek atlaga : " + atlag);
     }
}
```

A program futásának eredménye:

```
A tomb elemei:
8
6
3
8
4
A tomb elemeinek osszege: 29
A tomb elemeinek atlaga : 5.8
```

Írjunk programot, amely 1-10 közötti véletlen számmal feltölt egy 8-elemű, egész típusú tömböt! Számláljuk meg a tömb páros és páratlan elemeinek számát! (*Tomb2*)

```
// Tömb használata
public class Tomb2{
     public static void main(String[] args) {
          int paros_db = 0, paratlan_db = 0;
          int[] y = new int[8];
          System.out.println("A tomb elemei: ");
          for(int i = 0; i < y.length; i++) {
              y[i] = (int)(10*Math.random()+1);
              System.out.println(y[i]);
          }
          for(int i = 0; i < y.length; i++)
              if (y[i] %2 ==0)
                  paros_db++;
              else
                  paratlan_db++;
          System.out.println("A tomb paros elemeinek szama : " +
                             paros_db);
          System.out.println("A tomb paratlan elemeinek szama: " +
                             paratlan_db);
     }
}
```

A program futásának eredménye:

```
A tomb elemei:
5
2
10
8
10
3
2
6
A tomb paros elemeinek szama    : 6
A tomb paratlan elemeinek szama: 2
```

Írjunk programot, amely *MERET* darabszámú egész típusú adatot olvas be! Csak a konstans *index* tömbben megadott elemeket jelenítsük meg, és számítsuk ki a szorzatukat! (*Tomb3*)

```
// Tömb használata
import java.io.*;
public class Tomb3{
    public static void main(String[] args) throws IOException {
        final int INDEX_DB = 4;
        final int MERET = 8;
        final int index[] = { 2, 4, 5,7 };
        int szorzat = 1;
        int[] x = new int[MERET];
        BufferedReader be = new BufferedReader
                            (new InputStreamReader(System.in));
        System.out.println("Adatok szama:" + MERET);
        for (int i = 0; i<MERET; i++) {
            System.out.print("adat = ");
            x[i] =Integer.valueOf(be.readLine()).intValue();
        }
        System.out.println("Osszeszorzando elemek");
        System.out.println("Index  Adat");
        for (int i = 0; i<INDEX_DB; i++) {
             szorzat *= x[index[i]];
             System.out.println(index[i] + "    " + x[index[i]]);
        }
        System.out.println("Szorzat: " + szorzat);
    }
}
```

A program futásának eredménye:

```
Adatok szama:8                  Osszeszorzando elemek
adat = 1                        Index  Adat
adat = 2                        2   3
adat = 3                        4   5
adat = 4                        5   6
adat = 5                        7   8
adat = 6                        Szorzat: 720
adat = 7
adat = 8
```

Készítsünk alkalmazást, amely 1-10 közötti véletlen számmal feltölt egy 10-elemű, egész tömböt! Keressük meg a tömb legnagyobb és a legkisebb elemét! (*Tomb4*)

```
// Tömb használata
public class Tomb4{
     public static void main(String[] args) {
          int[] adat= new int[10];
          System.out.println("A tomb elemei :");
```

```
        for(int i = 0; i < adat.length; i++) {
            adat[i] = (int)(10*Math.random()+1);
            System.out.println(adat[i]);
        }
        int min_adat = adat[0], max_adat = adat[0];
        for(int i = 1; i<adat.length; i++) {
            if (adat[i] < min_adat)
                min_adat = adat[i];
            if (adat[i] > max_adat)
                max_adat = adat[i];
        }
        System.out.println("A tomb legkisebb eleme : " + min_adat);
        System.out.println("A tomb legnagyobb eleme: " + max_adat);
    }
}
```

A program futásának eredménye:

```
A tomb elemei :
8
7
3
7
9
8
4
8
5
5
A tomb legkisebb adata :3
A tomb legnagyobb adata:9
```

Írjunk programot, amely beolvassa az elemszámot, majd pedig a tömb elemeit! Számítsuk ki a pozitív adatok összegét és a negatív adatok átlagát! Kiíratás előtt vizsgáljuk meg az eredményeket! (*Tomb5*)

```
// Tömb használata
import java.io.*;
public class Tomb5{
    public static void main(String[] args) throws IOException {
        int xdb, osszeg = 0, atlag_db = 0;
        double atlag = 0;
        BufferedReader be = new BufferedReader
                            (new InputStreamReader(System.in));
        System.out.print("Adatok szama: ");
        xdb =Integer.valueOf(be.readLine()).intValue();
        int[] x = new int[xdb];
        for (int i = 0; i<xdb; i++) {
            System.out.print(i + ".elem: ");
            x[i] =Integer.valueOf(be.readLine()).intValue();
        }
```

```
            for (int i = 0; i<xdb; i++) {
                if (x[i] > 0)
                    osszeg += x[i];
                if (x[i] < 0) {
                    atlag += x[i];
                    atlag_db++;
                }
            }
            if (osszeg > 0)
                System.out.println("Pozitiv adatok osszege: "+osszeg);
            if (atlag_db > 0) {
                atlag = (double)atlag/atlag_db;
                System.out.println("Negativ adatok atlaga: " + atlag);
            }
    }
}
```

A program futásának eredménye:

```
Adatok szama: 5
0.elem: 2
1.elem: 3
2.elem: -4
3.elem: 6
4.elem: -3
Pozitiv adatok osszege: 11
Negativ adatok atlaga: -3.5
```

5.3 Többdimenziós tömbök

A Java nyelvben is használhatunk többdimenziós tömböket, melyek általános definíciója:

elemtípus `[][]…[]` *tömbnév;*

vagy

elemtípus tömbnév `[][]…[];`

Nézzük meg a kétdimenziós tömbök általános definícióját:

elemtípus tömbnév `[`*méret1*`][`*méret2*`];`

ahol az első dimenzió (*méret1*) a tömb sorainak, míg a második (*méret2*) a tömb oszlopainak számát határozza meg. Mátrixok tárolására a kétdimenziós tömböket használjuk:

```
int matrix[][] = { { 2, 4 },
                   { 3, 5 } };
```

A kétdimenziós tömb első eleme *matrix[0][0]* – az indexelés a 0. sorral és a 0. oszloppal kezdődik. A kezdőértékkel való feltöltés megfelel az alábbi értékadásoknak:

```
matrix[0][0] = 2; matrix[0][1] = 4;
matrix[1][0] = 3; matrix[1][1] = 5;
```

A kétdimenziós tömb valójában sorvektorok vektora. A tömböt szintén dinamikusan hozzuk létre.

```
// 5x5-ös egységmátrix előállítása
int m[][] = new int[5][5];
for (int i=0; i<m.length; i++)
    for (int j=0; j<m[0].length; j++)
        m[i][j] = i == j ? 1 : 0;
```

A példában az *m.length* kifejezés az első dimenzió (sorok), míg az *m[0].length* a második dimenzió (oszlopok) számát adja.

Készítsünk alkalmazást, amely megjeleníti a konstans értékekkel feltöltött kétdimenziós tömb elemeit! (*Tomb2dimp*)

```
// Kétdimenziós tömb használata
public class Tomb2dimp{
    public static void main(String[] args) {
      int[][] matrix = { { 2, 4, 5, -6},
                         { 3, 5, 0,  7} };
      System.out.println("A matrix tomb tartalma:");
      for (int i = 0; i < matrix.length; i++) {
          for (int j = 0; j < matrix[0].length; j++)
              System.out.print(matrix[i][j] + "  ");
          System.out.println();
      }
    }
}
```

A program futásának eredménye:

```
A matrix tomb tartalma:
2  4  5  -6
3  5  0  7
```

Egy egész típusú 4×4-es mátrixot töltsünk fel úgy, hogy a felső háromszög 1-es, a főátló 2-es és az alsó háromszög 3-as értékeket vegyen fel. Számítsuk ki a főátlóbeli elemek szorzatát és a főátló alatti elemek összegét! (*Tomb2dim*)

Megjegyzések:

- A mátrix sorait az *i*, az oszlopait pedig a *j* ciklusváltozóval érjük el a programban: `x[i][j]`
- Egy szimmetrikus mátrix főátlóbeli elemeinek sor- és oszlopindexe megegyezik: `i == j`

```
public class Tomb2dim{
    public static void main(String[] args) {
        final int MERET = 4;
        int osszeg = 0, szorzat = 1;
```

```
        int[][] x = new int[MERET][MERET];
        for (int i = 0; i < MERET; i++)
           for (int j = 0; j < MERET; j++)
               if (i < j )
                   x[i][j] = 1;
               else if (i == j)
                   x[i][j] = 2;
               else
                   x[i][j] = 3;

         System.out.println("x tomb tartalma:");
         for (int i = 0; i < MERET; i++) {
            for (int j = 0; j < MERET; j++)
                System.out.print(x[i][j] + "  ");
            System.out.println();
         }

         for (int i = 0; i < MERET; i++)
            for (int j = 0; j < MERET; j++) {
               if ( i > j)
                   osszeg += x[i][j];
               else if ( i == j)
                   szorzat *= x[i][j];
         }
         System.out.println("\nSzorzat: " + szorzat);
         System.out.println("Osszeg : " +osszeg);
    }
}
```

A program futásának eredménye:

```
x tomb tartalma:
2  1  1  1
3  2  1  1
3  3  2  1
3  3  3  2

Szorzat: 16
Osszeg : 18
```

Készítsünk egy 6x6-os alsó háromszög mátrix tárolására alkalmas tömböt, töltsük fel 1 és 2005 közötti véletlen számokkal, és jelenítsük meg az eredményt! (*Tomb2dah*)

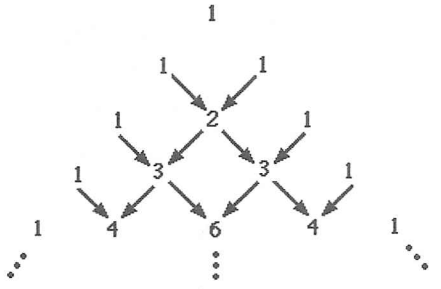
```
public class Tomb2dah{
    public static void main(String[] args) {

       long[][] ahm = new long[6][]; // a sorok 6-elemű vektora
       // minden sor hossza más és más
       for (int i = 0; i < ahm.length; i++)
          ahm[i] = new  long[i+1];
```

```
        // a tömb feltöltése
        for (int i = 0; i < ahm.length; i++)
            for (int j = 0; j < ahm[i].length; j++)
                ahm[i][j] = (long) (Math.random()*2005+1);

        // a tömb megjelenítése
        System.out.println("Az ahm tomb tartalma:");
        for (int i = 0; i < ahm.length; i++) {
            for (int j = 0; j < ahm[i].length; j++)
                System.out.print(ahm[i][j] + "\t");
            System.out.println();
        }
    }
}
```

A program futásának eredménye:

```
Az ahm tomb tartalma:
1494
812     1103
1356    1977    1666
649     93      925     371
222     2003    1393    1757    497
210     1313    11      632     164     1834
```

Használjuk az alsó háromszög mátrixot a Pascal-háromszög első 7 sorának tárolására! *(PascalHaromszog)*



Az ábrán a Pascal-háromszög első 5 sora látható. A jelzett nyilak mentén haladva programozzuk be az elemek rekurzív előállítását.

Mint ismeretes, a háromszög elemei a binomiális együtthatókat adják:

$$\binom{n}{k} = \frac{n!}{k!\cdot(n-k)!}$$

```
// a Pascal-háromszög előállítása
public class PascalHaromszog {
    public static void main(String[] args) {
        final int SOROK=7;
        int [][] p = new int [SOROK][]; // a sorok vektora
        p[0] = new int[1];
        p[0][0] = 1;                    // 0. sor
        p[1] = new int[2];
        p[1][0] = p[1][1] = 1;          // 1. sor
```

```
        for (int sor=2; sor<SOROK; sor++) {  // a további sorok
            p[sor] = new int[sor+1];
            p[sor][0] = p[sor][sor] = 1;
            for (int oszlop=1; oszlop < sor; oszlop++)
                p[sor][oszlop] = p[sor-1][oszlop-1]+p[sor-1][oszlop];
        }
        // a p tömb megjelenítése
        System.out.println("A Pascal-haromszog elso "+SOROK+" sora:");
        for (int i = 0; i < p.length; i++) {
            for (int j = 0; j < p[i].length; j++)
                System.out.print(p[i][j] + "\t");
            System.out.println();
        }
    }
}
```

A program futásának eredménye:

```
A Pascal-haromszog elso 7 sora:
1
1       1
1       2       1
1       3       3       1
1       4       6       4       1
1       5       10      10      5       1
1       6       15      20      15      6       1
```

6. Metódusok

A Java programozáshoz nélkülözhetetlen a metódusok (függvények) használata, amely az osztály feladatait részekre bontja. Természetesen az osztályban definiált metódusok egymást is hívhatják.

6.1 A metódusok definiálása

A metódus definiálásának általános formája:

```
[típusmódosítók] típus metódusnév([paraméterlista])
{
        // a metódus törzse
        lokális definíciók
        utasítások
        [ return [kifejezés]; ]
}
```

A leggyakrabban előforduló típusmódosítók:

- **public** - nyilvános, mindenki számára hozzáférhető.
- **private** - privát,
- **protected** - a származtatott osztály hozzáfér.
- **static** - statikus, objektumpéldány nélkül is hívható.

A metódus (visszatérési) típusa az alábbiak közül kerülhet ki:

- A 8 primitív típus valamelyike: **byte**, **short**, **int**, **long**, **char**, **float**, **double** és **boolean**.
- Egy referencia azaz objektum vagy tömbtípus.
- Ha nincs visszatérési érték, akkor ezt a **void** típussal jelezzük.

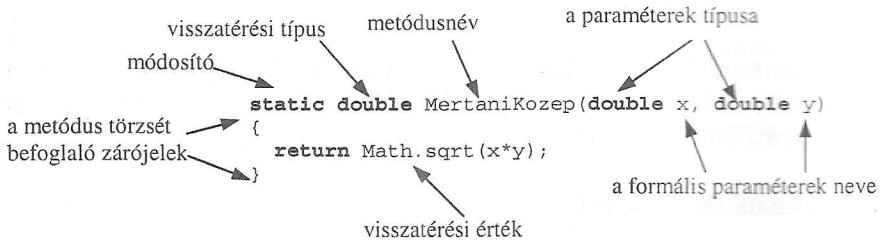
A **return** utasításnak két fontos szerepe van:

- hatására a hívott metódus visszatér a hívás helyére,
- az utasításban szereplő *kifejezés*, mint függvényérték (visszatérési érték) jelenik meg.

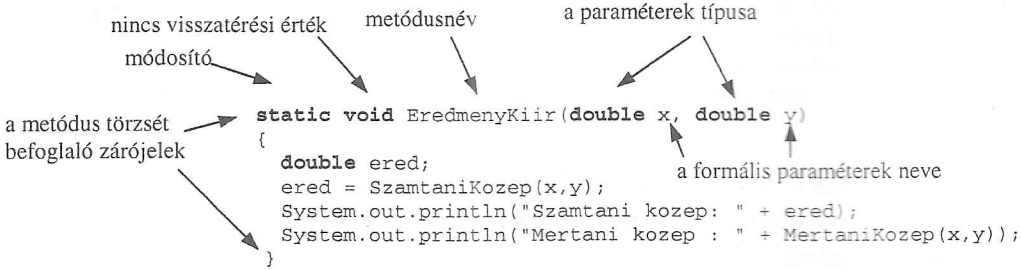
A Java nyelv metódusait az alábbiak szerint is osztályozhatjuk:

1. Ha a metódus rendelkezik visszatérési *típussal* (*például* **double**), akkor (más nyelvekben) függvénynek hívjuk. A függvény egyetlen visszatérési értékének típusa megegyezik a metódus neve előtt megadott *típussal*. A *paraméterlistán* átadott paraméterekkel műveletet végez, és egyetlen értéket ad vissza a **return** segítségével. Az elmondottaknak megfelelő metódus (függvény) aktiválható az értékadó utasí-

tás jobb oldalán, vagy például a *System.out.println()* metódus argumentumaként. Ekkor a metódushívás, mint kifejezés jelenik meg.



2. A visszatérési értékkel nem rendelkező (**void** *típusú*) metódust, más nyelveken eljárásnak nevezzük. Az ilyen metódusok hívását utasításként használjuk.



A metódusok hívása

A metódus hívásakor a metódus nevét és azt követően kerek zárójelek között, a vesszővel tagolt aktuális paramétereket (argumentumokat) kell megadnunk. A zárójelek használata akkor is kötelező, ha semmilyen paraméterrel sem rendelkezik a metódus. Az aktuális paraméterek típusának és számának egyeznie kell a metódusfejben szereplő megfelelő formális paraméterek típusával és számával. A fenti metódusok hívása:

```
double mk = MertaniKozep(12, 23);
EredmenyKiir(12, 23);
```

Formális és aktuális paraméterek

A metódusfejben szereplő paramétereket ***formális paramétereknek*** nevezzük. Minden formális paramétereknek van típusa, és vesszővel választjuk el őket egymástól. A metódusok tetszőleges számú paraméterrel rendelkezhetnek.

A metódusok hívása során használt argumentumokat ***aktuális paramétereknek*** nevezzük. Mint említettük, az aktuális paraméterek típusának és sorrendjének összhangban kell lennie a formális paraméterlistával.

Megjegyezzük, hogy a statikus (**static**) metódusokhoz, akkor is hozzáférhetünk, ha az osztálynak egyetlen példánya sem létezik – osztályszintű metódus. Ez magyarázza, hogy az eddigiekben miért használtunk statikus ***main()*** függvényt a példaprogramokban.

6.2 Különböző típusú paraméterek

Ebben a fejezetben olyan feladatokat mutatunk be, amelyekben a metódusok különféle típusú paramétereket kezelnek.

6.2.1 Primitív típusú paraméterek

Írjunk Java programot, amelyben két valós érték számtani és mértani közepét külön metódusok számítják ki, az eredményeket pedig egy harmadik metódus jeleníti meg! (*MetodusO*)

A mintaprogramban szereplő metódusok feladata és megoldása:

Írjunk *SzamtaniKozep()* metódust, amely az argumentumlistán megadott két valós értékből kiszámítja a számtani közepet, és azt függvényértékként adja vissza!

```
static double SzamtaniKozep(double x, double y){
        double ered;
        ered = (x + y)/2;
        return ered;
}
```

A metódus törzsében deklaráltunk egy ***lokális***, valós *ered* változót, melyben tároljuk az *x* és *y* formális paraméterben átadott adatok számtani közepét. Ezt követően a **return** utasításban szereplő *ered* változó értékét függvényértékként adjuk vissza.

Írjunk *MertaniKozep()* metódust, amely az argumentumlistán megadott két valós számnak kiszámítja a mértani közepét, és azt függvényértékként adja vissza!

Mivel az algoritmus nagyon egyszerű, nem szükséges lokális, segédváltozót deklarálni a művelet végrehajtásához, hanem közvetlenül a **return** utasításban is kiszámíthatjuk az *x* és az *y* mértani közepét.

```
static double MertaniKozep(double x, double y){
   return Math.sqrt(x * y);
}
```

Írjunk *EredmenyKiir()* metódust, amely aktiválja a *SzamtaniKozep()* és a *MertaniKozep()* metódusokat, és megjeleníti az eredményt.

```
    static void EredmenyKiir(double x, double y) {
        double ered;
        ered = SzamtaniKozep(x,y);
        System.out.println("Szamtani kozep: " + ered);
        System.out.println("Mertani kozep : " + MertaniKozep(x,y));
    }
```

A metódusok hívása:

1. Függvénymetódus aktiválható az értékadás jobb oldalán:

   ```
   ered = SzamtaniKozep(x,y);
   ```

   A példában az *ered* változóba kerül a függvényérték.

2. Függvénymetódus közvetlenül is hívható, például a ***println()*** argumentumában:

   ```
   System.out.println("Mertani kozep : " + MertaniKozep(x,y));
   ```

   A *MertaniKozep()* függvény által visszaadott érték átadódik a ***println()*** metódusnak, amely megjeleníti az eredményt.

3. Az *EredmenyKiir*() eljárásmetódusnak két **double** típusú paramétere van:

   ```
   static void EredmenyKiir(double x, double y)
   ```

A ***main()*** függvényben deklarált *adat1* és *adat2* változók értékei lesznek az aktuális paraméterek. Az eljárás hívása a nevével és az aktuális paramétereivel történik.

```
double adat1 = 1, adat2 = 2;
EredmenyKiir(adat1,adat2);
```

A feladat teljes megoldása:

```
// Metódusok tervezése
public class Metodus0{
        static double SzamtaniKozep(double x, double y){
          double ered;
          ered = (x + y)/2;
          return ered;
        }
        static double MertaniKozep(double x, double y){
          return Math.sqrt(x * y);
        }
        static void EredmenyKiir(double x, double y){
          double ered;
          ered = SzamtaniKozep(x,y);
          System.out.println("Szamtani kozep: " + ered);
          System.out.println("Mertani kozep : " + MertaniKozep(x,y));
        }
        public static void main(String[] args) {
          double adat1 = 1, adat2 = 2;
          EredmenyKiir(adat1,adat2);
        }
}
```

A program futásának eredménye:

```
Szamtani kozep: 1.5
Mertani kozep : 1.4142135623730951
```

Lokális változók használata

A metódusokban akkor van szükség ***lokális változók*** deklarálására, ha egy bonyolultabb algoritmus programozását egyszerűbbé teszik. Ezek a változók csak az őket deklaráló blokk végéig élnek. Minden alkalommal a blokk elején megszületnek, és a memóriaterületük felszabadul, ha a vezérlés kikerül a blokkból. Természetesen ezt követően már nem lehet rájuk hivatkozni.

Megjegyezzük, hogy a ***main()*** metódusban deklarált lokális változók is a programfutás végéig élnek.

Írjunk programot és tervezzünk két metódust! Az *Osztalyoz()* függvény a zárthelyin elért pontszámot alakítsa osztályzattá, és a *SzovegesOsztalyzat()* függvény pedig szövegesen írja ki az érdemjegyet! (*Metodus1*)

```
import java.io.*;
public class Metodus1{
    public static void main(String[] args) throws IOException {
        int zhpont, erdemjegy;
        BufferedReader be = new BufferedReader
                                (new InputStreamReader(System.in));
        do {
            System.out.print("zh pontszama = ");
            zhpont =Integer.valueOf(be.readLine()).intValue();
        } while ( zhpont <0 || zhpont > 100);
        erdemjegy = Osztalyoz(zhpont);
        SzovegesOsztalyzat(erdemjegy);
    }
    static int Osztalyoz(int pontszam)
    {
        int erdemjegy;
        if (pontszam >= 0 && pontszam <=100) {
           if (pontszam >= 86) erdemjegy = 5;
           else if (pontszam >= 76) erdemjegy = 4;
           else if (pontszam >= 66) erdemjegy = 3;
           else if (pontszam >= 51) erdemjegy = 2;
           else erdemjegy = 1;
        }
        else {
            System.out.println("Hibas adat ");
            erdemjegy = -1;
        }
        return erdemjegy;
    }
```

```
    static void SzovegesOsztalyzat(int jegy)
    {
        if( jegy > 0 && jegy <= 5) {
            System.out.print("Erdemjegy: " + jegy);
            switch( jegy) {
                case 1: System.out.println(" (elegtelen)");
                        break;
                case 2: System.out.println(" (elegseges)");
                        break;
                case 3: System.out.println(" (kozepes)");
                        break;
                case 4: System.out.println(" (jo)");
                        break;
                case 5: System.out.println(" (jeles)");
            }
        }
        else
            System.out.println("Hibas adat ");
    }
}
```

A program futásának eredménye:

```
zh pontszama = 97
Erdemjegy: 5 (jeles)
```

6.2.2 Tömbtípusú paraméterek

A metódusoknak tömbparamétere is lehet, azonban jelölni kell tömb dimenzióinak számát, például:

```
static int MinKeres(int a[])
```

A *MinKeres()* függvény formális paramétere egy egész típusú, egydimenziós tömb. A `[]` üres zárójelekkel jelezzük a tömb dimenzióját. Kétdimenziós tömbparaméter esetén a `[][]` jelölést kell alkalmaznunk.

Írjunk programot, amely metódusokkal keresi meg a paraméterlistán átadott tömb legkisebb és legnagyobb elemét! (*Metodust1*)

```
public class Metodust1{ // Tömbparaméter használata
    public static void main(String[] args) {
        final int db = 10;
        int min_adat, max_adat;
        int[] adat= new int[db];
        for (int i = 0; i < db; i++)
            adat[i] = (int)(10*Math.random()+1);
        Kiir(adat);
        min_adat = MinKeres(adat);
        max_adat = MaxKeres(adat);
        System.out.println("\nLegkisebb adat: " + min_adat);
        System.out.println("Legnagyobb adat: " + max_adat);
    }
```

```
    static int MinKeres(int a[]) {
           int min_adat = a[0];
           for (int i = 1; i<a.length; i++)
              if (a[i] < min_adat)
                 min_adat = a[i];
           return min_adat;
    }
    static int MaxKeres(int a[]) {
           int max_adat = a[0];
           for (int i = 1; i<a.length; i++)
              if (a[i] > max_adat)
                max_adat = a[i];
            return max_adat;
    }
    static void Kiir(int a[]) {
            System.out.println("Adatok:");
            for (int i = 0; i<a.length; i++)
               System.out.print(a[i] + "  ");
            System.out.println();
    }
}
```

A *MinKeres()* metódus a paraméterlistán átadott *a* tömb elemei közül megkeresi a legkisebbet. A *min_adat* lokális változó először felveszi a tömb 0. elemét, majd ciklusban keresünk 1-től a tömb méreténél kisebb, legnagyobb indexig. Ha találunk a *min_adat*-nál kisebbet, akkor felülírjuk a *min_adat* előző tartalmát. Végül a *min_adat* lesz a metódus visszatérési értéke.

```
    static int MinKeres(int a[]) {
           int min_adat = a[0];
           for (int i = 1; i<a.length; i++)
              if (a[i] < min_adat)
                 min_adat = a[i];
           return min_adat;
    }
```

A *MaxKeres()* metódus a paraméterlistán átadott *a* tömb elemei közül megkeresi a legnagyobbat. A *max_adat* lokális változó először felveszi a tömb 0. elemét, majd ciklusban keresünk 1-től a tömb méreténél kisebb, legnagyobb indexig. Ha találunk a *max_adat*-nál nagyobbat, akkor felülírjuk a *max_adat* előző tartalmát. Végül a *max_adat* lesz a metódus visszatérési értéke.

```
    static int MaxKeres(int a[]) {
           int max_adat = a[0];
           for (int i = 1; i<a.length; i++)
              if (a[i] > max_adat)
                max_adat = a[i];
            return max_adat;
    }
```

A *Kiir()* metódus a paraméterlistán átadott *a* tömb elemeit két szóközzel tagolva, egy sorban jeleníti meg az *Adatok* felirat alatt. Az utolsó elem kiírása után sort emel.

```
static void Kiir(int a[]) {
        System.out.println("Adatok:");
        for (int i = 0; i<a.length; i++)
           System.out.print(a[i] + "  ");
        System.out.println();
}
```

A program futásának eredménye:

```
Adatok:
3  3  3  9  8  10  5  6  4  9

Legkisebb adat: 3
Legnagyobb adat: 10
```

Írjunk programot, amely metódussal feltölt egy tömböt egész adatokkal, véletlenszám-generátor segítségével! Ezt követően metódus felhasználásával kiszámítja a tömb elemeinek összegét és átlagát! (*Metodust2*)

```
// Tömbparaméter használata
public class Metodust2{
      public static void main(String[] args) {
            int osszege;
            double atlaga;
            int[] adat = new int[10];
            Feltolt(adat);
            Kiir(adat);
            osszege = Osszeg(adat);
            atlaga = Atlag(adat);
            System.out.println("\nAdatok osszege : " + osszege);
            System.out.println("Adatok atlaga  : " + atlaga);
      }
      static void Feltolt(int a[]) {
            for (int i = 0; i < a.length; i++)
                a[i] = (int)(10*Math.random()+1);
      }
      static void Kiir(int a[]) {
            System.out.println("Adatok:");
            for (int i = 0; i < a.length; i++)
                System.out.print(a[i] + "  ");
            System.out.println();
      }
      static int Osszeg(int a[]) {
            int ossz = 0;
            for (int i = 0; i < a.length; i++)
                ossz += a[i];
            return ossz;
      }
```

```
    static double Atlag(int a[]) {
            int ossz = Osszeg(a);
            return (double)ossz/a.length;
    }
}
```

A program futásának eredménye:

```
Adatok:
7  3  8  2  1  9  5  3  7  6

Adatok osszege : 51
Adatok atlaga  : 5.1
```

Írjunk programot, amely metódusok segítségével kockadobást szimulál! A szimulációhoz használjunk véletlen számokat, készítsünk és jelenítsünk meg statisztikát az eredményekről! (*Metodust3*)

```
// Tömbparaméter használata
public class Metodust3
{
   // a Kockadobas() és Megjelenit() metódusok helye!
   public static void main(String[] args) {
           int [] k = new int[7];
           Kockadobas(k,1000);
           Megjelenit(k);
   }
}
```

A *Kockadobas()* metódus a *kocka* tömbben összegzi a *dobas* darab kockadobás eredményét.

```
static void Kockadobas(int kocka[], int dobas){
     int index;
     for (int i = 1; i < 7; i++)
         kocka[i] = 0;
     for (int i = 1; i <= dobas; i++) {
         index = (int)(Math.random()*6 + 1);
         kocka[index]++;
     }
}
```

A *Megjelenit()* metódus megjeleníti a paraméterlistán megadott *kocka* tömb elemeinek indexét és tartalmát.

```
static void Megjelenit(int kocka[]) {
     int ossz_db = 0;
     for (int i = 1; i <= 6; i++) {
        System.out.println(i + "  " + kocka[i]);
        ossz_db += kocka[i];
     }
     System.out.println("-------");
     System.out.println("  " + ossz_db);
}
```

A program futásának eredménye:

```
1   175
2   160
3   161
4   173
5   181
6   150
-------
  1000
```

6.3 Metódusok túlterhelése

A Java nyelvben a különböző paraméterlistájú metódusokat azonos névvel is elláthatjuk. Ezt a metódusok ***túlterhelése*** (*overloading*) teszi lehetővé. Híváskor az a metódus aktiválódik, mely aktuális paraméterlistájának lenyomata (típussora) megegyezik a formális paraméterekével.

Írjunk programot, amely azonos nevű metódusokkal két különböző típusú adatot ad össze! (*OverLoad1*)

```
// Metódusok túlterhelése
public class Metodus2
{
        static double Osszeg(double w1, double w2) {
           return w1 + w2;
        }
        static double Osszeg(int w1, double w2) {
           return w1 + w2;
        }
        static int Osszeg(int w1, int w2) {
           return w1 + w2;
        }
        static double Osszeg(double w1, int w2) {
           return w1 + w2;
        }
        public static void main(String[] args) {
           int a = 2, b = 4, ered1;
           double x = 3.2, y = 4.3, ered2, ered3, ered4;
           ered1 = Osszeg(a, b);
           System.out.println("Osszeg1: " + ered1);
           ered2 = Osszeg(a, x);
           System.out.println("Osszeg2: " + ered2);
           ered3 = Osszeg(y, b);
           System.out.println("Osszeg3: " + ered3);
           ered4 = Osszeg(x, y);
           System.out.println("Osszeg4: " + ered4);
      }
}
```

A program futásának eredménye:

```
Osszeg1: 6
Osszeg2: 5.2
Osszeg3: 8.3
Osszeg4: 7.5
```

Írjunk programot, amely azonos nevű metódusokkal kiszámítja egy kocka felszínét és térfogatát valós és egész típusú élekre! (*OverLoad2*)

```
// Metódusok túlterhelése
public class OverLoad2
{
      static double KockaFelszin(double a) {
         return 6 * a *a;
      }
      static double KockaTerfogat(double a) {
         return a * a * a;
      }
      static int KockaFelszin(int a) {
         return 6 * a *a;
      }
      static int KockaTerfogat(int a) {
         return a * a * a;
      }
      public static void main(String[] args) {
         int b = 2,  terf1, felsz1;
         double c = 1.5, terf2, felsz2;
         System.out.println("Kocka oldala: " + b);
         terf1 = KockaTerfogat(b);
         System.out.println("Terfogat: " + terf1);
         felsz1 = KockaFelszin(b);
         System.out.println("Felszin: " + felsz1);
         terf2 = KockaTerfogat(c);
         System.out.println("\nKocka oldala: " + c);
         System.out.println("Terfogat: " + terf2);
         felsz2 = KockaFelszin(c);
         System.out.println("Felszin: " + felsz2);
      }
}
```

A program futásának eredménye:

```
Kocka oldala: 2
Terfogat: 8
Felszin: 24

Kocka oldala: 1.5
Terfogat: 3.375
Felszin: 13.5
```

Készítsünk alkalmazást, amelyben azonos nevű metódusokkal rendezünk és jelenítünk meg egydimenziós **double** és **int** típusú tömböket! (*OverLoad3*)

```
public class OverLoad3 {
        static void Rendez(double[] t) {
          double e;
          for (int i = 0; i < t.length-1; i++)
             for (int j = i+1; j < t.length; j++)
                if (t[i]>t[j]) {
                     e = t[i];  t[i] = t[j];  t[j] = e;
                }
        }
        static void Rendez(int[] t) {
          int e;
          for (int i = 0; i < t.length-1; i++)
             for (int j = i+1; j < t.length; j++)
                if (t[i]>t[j]) {
                     e = t[i];  t[i] = t[j];  t[j] = e;
                }
        }
        static void Kiir(double[] t) {
          for (int i = 0; i < t.length; i++)
                System.out.print(t[i] + "\t");
          System.out.println();
        }
        static void Kiir(int[] t) {
          for (int i = 0; i < t.length; i++)
                System.out.print(t[i] + "\t");
          System.out.println();
        }
        public static void main(String[] args) {
            double[] dt = { 12.3, 79, 7,2, -3.1, 23 };
            System.out.print("Rendezes elott: ");
            Kiir(dt);  Rendez(dt);
            System.out.print("Rendezes utan : ");
            Kiir(dt);

            int[] et = { 12, 19, 79, 7, -3, 23,35 };
            System.out.print("Rendezes elott: ");
            Kiir(et);  Rendez(et);
            System.out.print("Rendezes utan : ");
            Kiir(et);
        }
}
```

A program futásának eredménye:

```
Rendezes elott: 12.3    79.0    7.0     2.0     -3.1    23.0
Rendezes utan : -3.1    2.0     7.0     12.3    23.0    79.0
Rendezes elott: 12      19      79      7       -3      23      35
Rendezes utan : -3      7       12      19      23      35      79
```

7. Osztályok és objektumok

Az objektum-orientált programozási nyelvek sokkal strukturáltabbak, modulárisabbak és absztraktabbak, mint a hagyományos programozási nyelvek. Ennek jelentősége az alábbiakban rejlik:

- Az adatokat és az adatokat kezelő függvényeket (metódusokat) egyetlen programegységben (objektumban) kezeljük. A metódusok általában az adatokon végeznek műveleteket, ezért a metódusok nagy része nem rendelkezik paraméterrel.
- Az osztályok kialakítása során élhetünk az adatrejtés (*data hiding*) lehetőségével.
- Az öröklés (*inheritance, subclassing*) alkalmazása.
- A származtatott osztály örökli az alaposztály tulajdonságait (adatait) és viselkedését (metódusait) – ezek azonban meg is változtathatók. Lehetőség van új adatok hozzáadására, illetve a metódusok újradefiniálására (*redefine*). A metódusok újradefiniálásával (lecserélésével) megvalósul a polimorfizmus (*polymorphism*), a többalakúság.

A Java nyelv teljesen objektum-orientált, ezért már a legelső példaprogramtól kezdve osztályokat készítünk. Az osztályokat a könyvünk további részeiben objektumpéldányok (*instances*) előállítására fogjuk használni. A példányosítást a referencia objektummodell alapján végzi a Java, ami azt jelenti, hogy az objektumok, a tömbökhöz hasonlóan dinamikusan, a **new** operátor felhasználásával jönnek létre. A lefoglalt területek felszabadításáról a futtató rendszer gondoskodik (*garbage collection*), mi csupán a **null** értékkel jelezhetjük, ha egy dinamikus változóra nincs többé szükségünk.

Az objektum-orientált programozás azonban nem csupán osztályok fejlesztéséből és objektumok létrehozásából áll. Az igazi erejét (és nem utolsó sorban a szépségét) az adja, ahogy az öröklés (bővítés) során lépésről-lépésre (osztályról-osztályra) haladva jutunk el a megfelelő képességű osztályokig. Az öröklési ágak (láncolatok) kialakítása azonban nem egyszerű feladat, általában komoly tervezési munka szükséges hozzá. Könyvünkben eltekintünk az objektum-orientált tervezési módszerek/módszertanok ismertetésétől, hiszen a korlátozott oldalszám csak a Java nyelvi megoldások bemutatását teszi lehetővé.

Az objektum-orientált programozás alapstruktúrája az osztály, melynek definíciójában a **class** kulcsszó után egy azonosító áll. (Az osztályfej további elemeivel később foglalkozunk.) Az osztály törzsében elhelyezkedő adatmezőket, valamint az adatokon műveleteket végző metódusokat kapcsos zárójelek { } fogják közre. Az osztályok leírásának általános formája:

```
// az osztály feje
[módosítók] class Osztálynév [extends Osztálynév]
        [implements Interfésznév1 [, Interfésznév2 ...]]
// az osztály törzse
{
    // az osztálytagok (adatmezők és metódusok)
    // deklarációi
}
```

Felhívjuk a figyelmet arra, hogy a Java-ban minden általunk készített osztály származtatott osztály, még akkor is, ha nem adjuk meg az **extends** részt. Minden osztály közös őse az ***Object*** osztály, melynek metódusait az F1. függelékben foglaltuk össze. Az öröklést a 8. fejezetben ismertetjük részletekbe menően, most csak az explicit származtatás nélküli esetekkel foglalkozunk.

7.1 Adatmezők, metódusok és a this hivatkozás

Az osztály névvel ellátott mezőkben tárolt adatok, és névvel azonosított metódusok gyűjteménye, mely metódusok az adatokat kezelik. Az adatmezőket és a metódusokat együtt az osztály tagjainak (*member*) nevezzük. Osztálytagok esetén az alábbiak szerint szabályozhatjuk a hozzáférést:

- **public** – nyilvános, mindenki hozzáfér.
- **private** – csak az osztály saját metódusai férhetnek hozzá. Öröklés esetén akkor használjuk, ha le akarjuk tiltani a hozzáférést a származtatott osztály számára.
- **protected** – az osztály metódusain kívül csak a származtatott osztály metódusai érik el.
- *csomag* – csak a csomagban (*package*) definiált osztályok férnek hozzá. Ez az alapértelmezés, ha nem adunk meg hozzáférési módot.

Néhány észrevétel az osztálytagok elérésével kapcsolatosan:

- A metódusok általában **public** elérésűek.
- A **private** hozzáféréssel rendelkező metódusokhoz, sem a külvilág, sem pedig az utódosztályok nem férhetnek hozzá.
- Az osztály adatmezői általában **private** vagy **protected** elérésűek.

Megjegyzések az adatmezők használatával kapcsolatosan:

- Az adatmezők egy részével műveletet végzünk, ezért fontos, hogy legyen kezdőértékük.
- Vannak olyan adatmezők, melyeket a megfelelő kezdőértékre, azaz alapállapotba kell állítanunk az algoritmus helyes működéséhez.
- Az eredményeket tároló adatmezőknek nem szükséges kezdőértéket adni.

Néhány gondolat a metódusokkal kapcsolatosan:

A metódusok fontos jellemzője, hogy általában nincsenek paramétereik, mivel az osztály adatmezőin végeznek műveleteket. Kivételt képeznek a kezdőértékadó és az értékmódosító metódusok, melyek a külvilág felől kapják az adatokat, valamint a más osztályok objektumaival kommunikáló metódusok. A metódusok a kezdőértékkel ellátott adatokkal dolgoznak, és az algoritmusuknak megfelelően különféle eredményeket hoznak létre az osztály adatmezőiben. Természetesen a metódusoknak is lehetnek saját helyi (lokális) változóik, melyeket munkarekeszként használunk az algoritmusok programozása során.

A metódusok csoportosítása:

A metódusokat több szempont szerint is csoportba sorolhatjuk. Az alábbiakban szereplő negyedik csoport elemei közönséges metódusok, melyeket nehéz közösen jellemezni. A középső két csoport tagjai az ún. elérési metódusok. Mivel az adatmezők közvetlenül általában nem érhetők el, szükség lehet olyan metódusokra, melyek ellenőrzött módon férnek hozzá az objektum adataihoz. Az első helyen különleges metódusok állnak – a konstruktorok, melyeket közvetlenül az objektumpéldány létrehozásakor aktivizálunk.

- kezdőértékadó,
- új értéket beállító – (gyakran a *set* szóval kezdődik a nevük),
- értéklekérdező – (általában a *get* szócska vezeti be a nevüket),
- az algoritmus megoldásához szükséges, illetve az eredmény megjelenítésére szolgáló metódusok.

Példaként tervezzünk egy egyszerű mintaosztályt, amely kiszámolja két egész szám összegét, és a művelet operandusaival együtt megjeleníti az eredményt! (*MintaOsztaly*)

Első lépésként tervezzük meg az osztály adatmezőit és metódusait! Legyen az osztály azonosítója *MintaOsztaly*!

Adatmezői:

- *x* és *y* egész típusú adatok, melyeknek kezdőértéket kell adnunk.
- *ered* egész típusú változó tárolja a két adat összegét.

Metódusai:

- A *Muvelet()* metódusnak nincsenek paraméterei, mivel az osztály adatmezőivel végez műveletet. Kiszámítja a két adat (az *x* és *y*) összegét, melyet az *ered* változóba tölt.
- A *Kiir()* metódus az elvégzett művelet jelölésével megjeleníti az operandusokat és az eredményt.

```
class MintaOsztaly
{
    // adatmezők (tulajdonságok) deklarációja
    private int x, y;
    private int ered;       // az összeadás eredményét tárolja

    // az osztály metódusai
    public void Muvelet()  // nincs visszatérési érték
    {
      ered = x + y;         // kiszámítja a két adat összegét
    }

    public void Kiir()      // megjeleníti az eredményt
    {
      System.out.println(x + " + " + y + " = " + ered);
    }
}
```

Az osztály minden (nem statikus) metódusának van egy nem látható paramétere – a **this** objektumreferencia. Mivel egy osztállyal több objektumpéldányt is létrehozhatunk, ez azonosítja, hogy melyik példányhoz tartozó adatmezőkkel kell dolgoznia a metódusnak. Az egymásból hívott metódusok esetén a **this** automatikusan továbbadódik. A jobb megértés érdekében alakítsuk át a *MintaOsztaly*-t úgy, ahogy azt a fordítóprogram használja, láthatóvá téve a **this** referenciát! (Felhívjuk a figyelmet arra, hogy az osztály ebben a formában nem fordítható le, hisz például a **this** foglalt szó.)

```
class MintaOsztaly {
    private int x, y;
    private int ered;
    public void Muvelet(MintaOsztaly this)
    {
      this.ered = this.x + this.y;
    }
    public void Kiir(MintaOsztaly this)
    {
       System.out.println(this.x + " + " + this.y + " = " + this.ered);
    }
}
```

7.2 Statikus osztálytagok alkalmazása, osztályok elhelyezése

A könyvünk első hat fejezetében nem készítettünk objektumpéldányokat, azonban mégis futó alkalmazásokat hoztunk létre. Hogyan lehetséges ez? A választ a statikus (**static**) osztálytagok adják.

Ha egy metódus **static** elérésű, akkor anélkül is hívhatjuk, hogy az osztálynak egyetlen példányát létrehoztuk volna. A ***main()*** metódus mindig statikus, mivel a program indulásakor az osztályunknak még nem létezik példánya.

A statikus metódusok (az ún. osztálymetódusok) nem rendelkeznek a **this** paraméterrel, így nem hivatkozhatnak az osztály nem statikus adatmezőire és metódusaira.

A **static** elérésű adatmezők (az osztálymezők) az osztály betöltésekor, egyetlen példányban jönnek létre, így az objektumok megosztva használják azt.

Az összehasonlítás kedvéért alakítsuk át az *MintaOsztalyt* statikus tagokat használó osztállyá, és tegyük futtathatóvá a *MintaPr1* osztály hozzáadásával! (*MintaPr1*)

```
// static osztálytagok használata
class MintaOsztaly {
        // statikus adatmezők
        static int x, y;
        static int ered;

        // statikus metódusok
        public static void Muvelet() {
                ered = x + y;
        }

        public static void Kiir() {
                System.out.println(x + " + " + y + " = " + ered);
        }
}

// a futtató osztály
public class MintaPr1 {
        public static void main(String[] args) {
                MintaOsztaly.x = 12;
                MintaOsztaly.y = 23;
                MintaOsztaly.Muvelet();
                MintaOsztaly.Kiir();
        }
}
```

A program futásának eredménye:

```
12 + 23 = 35
```

Az osztályok elhelyezése a forrásprogramban

A *MintaPr1* alkalmazáshoz néhány megjegyzést kell fűznünk:

- Amennyiben egyetlen forrásfájlban több osztályt is elhelyezünk, az osztályok között csomagkapcsolat alakul ki, ami egy baráti (*friend*) viszony. Ez azt jelenti, hogy az osztályok korlátozás nélkül elérhetik egymás, hozzáférési kulcsszót nem tartalmazó tagjait. (A példában így használjuk az *x* és *y* adatmezőket.)
- Ha egy forrásfájl több osztályt is tárol (a példában a *MintaPr1.java*), akkor az osztályok közül pontosan egy lehet **public** elérésű, és ennek nevével kell az állományt menteni. A fordítás során azonban minden lefordított osztály külön *.class*-fájlban jön létre (a példában *MintaOsztaly.class* és *MintaPr1.class*). Az alkalma-

zás futtatásához a *java* parancs sorában csak a ***main()*** metódust tartalmazó osztály nevét kell megadnunk (*java MintaPr1*).

- Az osztályokat egymásba is skatulyázhatjuk, elzárva például a (tag vagy lokális) belső osztályt a külvilág elől. Ekkor a fordítás során keletkező belső osztály neve a külső osztály nevét is tartalmazza. Az alábbi példa fordításakor a *Kulso.class* és a *Kulso$Belso.class* fájlok jönnek létre.

```
public class Kulso
{
    class Belso
    {
       // ...
    }
    // ...
}
```

- Az osztályok egymásba skatulyázásának speciális esete az anonim osztályok használata. Ekkor a lefordított lokális belső osztály neve a külső osztály nevéből és a dollár jel után egy sorszámból áll. (A következő példában a két állománynév: *Kulso.class* és a *Kulso$1.class*.)

```
public class Kulso
{
    public void letrehoz() {
       int x = new Object() {
                                    public int fv() {
                                           return 12;
                                    }
                                    // ...
                                }.fv();
    }
    // ...
}
```

Az osztályok egymásba skatulyázására használható példákat a grafikus felület programozási feladataiban láthatunk. Anonim osztályként általában más osztály metódusai által hívott függvényeket tároló (adapter) osztályokat használunk. (A példában szereplő ***Object*** osztály nem ilyen!)

7.3 Objektumok létrehozása – konstruktorok

A Java-ban az objektumpéldányok létrehozásának első lépése a megfelelő típusú objektumhivatkozás deklarálása:

```
Osztály objektumreferencia;
```

A következő lépés az *Osztály* típusú objektum létrehozása, és hivatkozásának bemásolása az *objektumreferencia* változóba:

```
objektumreferencia = new Osztály ([argumentumlista]);
```

Ezt a két lépést természetesen össze is vonhatjuk:

```
Osztály objektumreferencia = new Osztály ([arg.lista]);
```

Az értékadás jobb oldalán szereplő kifejezésben az *Osztály* metódusként is viselkedik. Valójában az osztály területének lefoglalása után egy speciális metódus, a konstruktor hívását írjuk elő, melynek jellegzetessége, hogy neve megegyezik az osztály nevével.

Minden osztálynak legalább egy konstruktort kell tartalmaznia, amely az osztály változóinak ad kezdőértéket. Ha nem gondoskodunk konstruktorról, akkor a fordító egy paraméter nélküli, alapértelmezett konstruktort rendel az osztályhoz. Megjegyezzük, hogy az objektum területe nullás bájtokkal töltődik fel a létrehozás után, így az adatmezők automatikusan nulla kezdőértéket kapnak.

Az alábbiakban röviden összefoglaltuk a konstruktorokra vonatkozó szabályokat:

- nevének meg kell egyeznie az osztály nevével,
- nincs visszatérési típusa, a **void** sem használható,
- túlterhelhető, azaz egy osztálynak több, eltérő paraméterezésű konstruktora is lehet,
- nem örökölhető,
- közvetlenül csak egy másik konstruktor első soraként hívható, a **this** hivatkozás felhasználásával: **this** *([argumentumlista]);*
- az alapértelmezett konstruktort csak akkor készíti el a fordító, ha nincs az osztálynak más konstruktora.

Az alábbi példaprogramot követően megvizsgáljuk az alkalmazásban használt megoldásokat.

A *MintaOsztalyt* bővítsük ki konstruktorokkal, és hozzunk létre több objektumpéldányt! (*MintaPr2*)

```
class MintaOsztaly
{
      // az adatmezők deklarációja
      private  int x, y, ered;
      // paraméteres konstruktor
      MintaOsztaly (int x1, int y1)
      {
          x = x1; y = y1; ered = 0;
      }
      // paraméter nélküli konstruktor
      MintaOsztaly ()
      {
          x = y = 1; ered = 0;
      }
```

```
    // Osztály metódusai
    public void Muvelet() {
        ered = x + y;
    }
    public void Kiir() {
        System.out.println(x + " + " + y + " = " + ered);
    }
}
// az alkalmazás indítását biztosító osztály
public class MintaPr2 {
    public static void main(String[] args) {
    int adat1 = 4, adat2 = 5;
    MintaOsztaly peldany1;
    // a paraméteres konstruktor hívása
    peldany1 = new MintaOsztaly(adat1, adat2);
    peldany1.Muvelet();
    peldany1.Kiir();
    // a paraméter nélküli konstruktor hívása
    MintaOsztaly peldany2 = new MintaOsztaly();
    peldany2.Muvelet();
    peldany2.Kiir();
    // a peldany2 objektumpéldány automatikusan felszabadul
    peldany2 = new MintaOsztaly( 2,3);
    peldany2.Muvelet();
    peldany2.Kiir();
    }
}
```

A program futásának eredménye:

```
4 + 5 = 9
1 + 1 = 2
2 + 3 = 5
```

Megjegyzések a példaprogrammal kapcsolatosan:

1. A paraméteres konstruktor a külvilág felől kap értéket, mellyel inicializálja az objektum adatmezőit.

```
MintaOsztaly( int x1, int y1)
{
     x = x1; y = y1;
     ered = 0;          // nem kötelező a 0 kezdőérték adása
}
```

2. A paraméter nélküli konstruktor az objektumpéldányt egy alapértelmezett módon állítja be.

```
MintaOsztaly ( )
{
    x = 1; y = 1; ered = 0;
}
```

3. Ha egy osztálynál nem adunk meg konstruktort, akkor a Java-fordító automatikusan egy paraméter nélküli, alapértelmezett (*default*) konstruktort definiál.
4. Ha a paraméterlistán a paraméterek neve megegyezik a mezőnevekkel, akkor a metóduson belül a **this** objektumreferencia használatával különböztetjük meg az osztály mezőit az azonos nevű paraméterektől.

```
MintaOsztaly ( int x, int y)
{
    this.x = x; this.y = y; ered = 0;
}
```

5. A másoló (*copy*) konstruktor esetén az új osztályt egy létező, azonos típusú objektum adataival inicializáljuk. Természetesen ekkor ennek a két objektumnak azonos értékű adatmezői lesznek.

```
MintaOsztaly (MintaOsztaly obj)
{
    x = obj.x; y = obj.y; ered = 0;
}
```

A feladatmegoldásban az objektumpéldányokat a *MintaPr2* osztály ***main()*** metódusában hozzuk létre az alábbi módszerek valamelyikét alkalmazva.

Első esetben a *MintaOsztaly* típussal deklaráljuk a *peldany1* referencia-változót, amely a *MintaOsztaly* objektumpéldányára hivatkozhat, azonban tartalma a deklarációt követően határozatlan. (Ezt elkerülendő, használhatjuk kezdőértékként a **null** értéket.)

```
MintaOsztaly peldany1;
```

A **new** művelettel létrejön a *MintaOsztaly* típusú objektum, meghívódik a paraméteres konstruktor, majd a kifejezés értékeként megjelenő objektum-hivatkozás bemásolódik a *peldany1* változóba.

```
int adat1 = 4, adat2 = 5;
peldany1 = new MintaOsztaly(adat1, adat2);
```

Az objektumpéldány műveleteit az objektum-referencia nevével minősített metódusnévvel hívjuk, használva a pont (.) operátort.:

```
peldany1.Muvelet();
peldany1.Kiir();
```

A második esetben a objektum-referencia deklarációját, az objektumpéldány létrehozását valamint a paraméter nélküli konstruktor hívását egyetlen utasításban végeztük el.

```
MintaOsztaly peldany2 = new MintaOsztaly();
peldany2.Muvelet();
peldany2.Kiir();
```

A létező objektum referenciáját felülírhatjuk egy újabb **new** művelettel, annak ellenére, hogy a már foglalt memóriaterületet nem szabadítottuk fel.

```
peldany2 = new MintaOsztaly(2, 3);
```

A Java nyelvben nincs külön utasítás az objektumok memóriaterületének felszabadítására. Az objektumok felszámolása a futtatórendszer feladata, azaz a rendszer figyeli, és időnként felszabadítja azokat a területeket, amelyekre már nem hivatkozunk (például a *peldany2* előző objektuma). Az *automatikus szemétgyűjtésnek* nevezett folyamat indítását azonban mi is kérhetjük a ***System.gc()*** metódus hívásával. Megjegyezzük, hogy a **null** érték segítségével mindig „eldobhatjuk" a felesleges objektumokat:

```
peldany2 = null;
```

Mielőbb tovább gombolyítanánk az objektum-orientált programozás gondolatmenetének fonalát, az összehasonlítás kedvéért nézzük meg a feladat megoldásának hagyományos módját!

Valósítsuk meg a *MintaPr2* példa *MintaOsztaly* osztályának működését hagyományos (csak metódusokat használó) módon! Mit tapasztalunk! (*MintaFg*)

```
// A mintafeladat megoldása statikus metódusok segítségével
public class MintaFg {
   static int Muvelet(int w1, int w2){
       return w1 + w2;
   }
   static void Kiir(int w1, int w2, int ered){
       System.out.println(w1 + " + " + w2 + " = " + ered);
   }
   public static void main(String[] args){
       int x = 2, y = 4, ered1;
       ered1 = Muvelet(x, y);
       Kiir(x, y, ered1);
   }
}
```

A program futásának eredménye:

```
4 + 5 = 9
```

A megoldásból világosan kitűnik, hogy a metódusok *értékparaméterekkel* dolgoznak, függvényként vagy eljárásként működnek, és *nincs szoros kapcsolat* az adatok és a metódusok között.

A *MintaOsztalyt* bővítsük ki olyan konstruktorral, melynek paraméternevei megegyeznek az osztály adatmezőinek neveivel, valamint egy másoló konstruktorral, és hozzunk létre 3 objektumpéldányt a konstruktorok aktiválása céljából! Mit tapasztalunk? (*MintaPr3*)

```
// Osztály és objektum, másoló konstruktor használata
class MintaOsztaly
{
    // változók deklarációja
    private int x, y, ered;

    // paraméteres konstruktor
    MintaOsztaly ( int x, int y)
    {
       this.x = x; this.y = y; ered = 0;
    }

    // másoló (copy) konstruktor
    MintaOsztaly (MintaOsztaly obj )
    {
       x = obj.x; y = obj.y; ered = 0;
    }

    // további metódusok
    public void Muvelet() {
       ered = x + y;
    }

    public void Kiir() {
        System.out.println(x + " + " + y + " = " + ered);
    }
}
public class MintaPr3{
    public static void main(String[] args) {
       int adat1 = 4, adat2 = 5;
       MintaOsztaly peldany1;

       // a paraméteres konstruktor hívása
       peldany1 = new MintaOsztaly(adat1, adat2);
       peldany1.Muvelet();
       peldany1.Kiir();

       // copy konstruktor hívása
       MintaOsztaly peldany2 = new MintaOsztaly(peldany1);
       peldany2.Muvelet();
       peldany2.Kiir();

}
```

A program futásának eredménye:

```
4 + 5 = 9
4 + 5 = 9
```

Azt tapasztaljuk, hogy a másoló konstruktor lemásolta a *peldany1* adatmezőit a *példány2* adatmezőibe, így a két eredmény azonos lett.

7.4 Az osztálytagok és az osztályok elérésének szabályozása

A Java programban az osztályok és az osztálytagok deklarációjában egy sor ún. módosító szerepelhet. Ezek szabályozzák az adott elem elérését és felhasználási lehetőségeit. A módosítók teljes listáját névsorba szedve tartalmazza az F1. függelék, itt azonban csoportosítva tekintjük át a használatukat. A módosítók közül egyszerre többet is megadhatunk, azonban nem tetszőleges párosításban:

```
[public | private | protected ] [static] [final]
[public | protected ] abstract
```

Általánosan is elmondhatjuk, hogy a hozzáférési módosítók közül legfeljebb egyet használhatunk.

Hozzáférési módosítók:

- **public** – nyilvános, mindenhonnan elérhető,
- **private** – privát, csak a deklarációt tartalmazó osztályból használható,
- **protected** – védett, csak a saját osztályból és annak utódosztályaiból (*subclass*) érhető el,
- *csomag* – csak a csomagban (*package*) definiált osztályok férnek hozzá. Ez az alapértelmezés, ha nem adunk meg hozzáférési módot.

Egyéb módosítók:

- **static** – osztályhoz tartozó tag, példányok nélkül is használható.
- **final** – végleges, nem változtatható meg.
  - A **final** adatmezők kezdőértéke nem változtatható meg a későbbiekben. A kezdőérték nélküli **final** adatmezők a konstruktorban kaphatnak értéket.
  - Származtatás során megtilthatjuk a **final** metódushoz tartozó műveletek újradefiniálását. Ez azt jelenti, hogy az utódosztályban ezek a metódusok nem cserélhetők le.
  - A teljes osztályra is alkalmazhatjuk a **final** kulcsszót, megakadályozva ezzel, hogy az osztályból új (al)osztályt származtassunk. A **final** osztály előnye, hogy a fordító hatékonyabb kódot állít elő, mivel ebből az osztályból nem származtatunk. Egy másik előnye a biztonság, mivel a *hackerek* gyakran az öröklést használva játsszák ki a Java biztonsági rendszerét.
- **abstract** – Az **abstract** metódusnak nincs törzse – az ilyen metódust az utódosztályban kell kidolgozni. Ha egy osztályban absztrakt metódus szerepel, akkor ezt az **abstract** módosítóval az osztályfejben is jelezni kell. Absztrakt osztállyal nem készíthetünk objektumot (**new**), azonban referencia-változó típusaként szerepelhet.

Tervezzük osztályt, amelynek adatmezői a derékszögű háromszög átfogója és az átfogón lévő szög! Az osztály metódusa számítsa ki a két befogót és jelenítse meg a derékszögű háromszög oldalait! A fokból radiánba váltásra használjuk egy **final** osztály metódusát! (*Atalakito*)

```
public class Atalakito{
    public static void main(String[] args) {
        double c = 5, sz = 30;
        DerekSzH dh= new DerekSzH(c, sz);
        dh.Szamol();
        dh.Kiir();
    }
}
final class Atalakit {
    public static final double pi  =3.141592654;
    public static final double FokRad(double fok) {
        return fok*pi/180;
    }
    public static final double Radfok(double rad) {
        return rad*180/pi;
    }
}
class DerekSzH {
    private double bef1, bef2, atfogo, szog;
    DerekSzH( double atfogo1, double szog1) {
        atfogo = atfogo1; szog = szog1;
    }
    public void Szamol() {
        bef1 = atfogo * Math.sin(Atalakit.FokRad(szog));
        bef2 = atfogo * Math.cos(Atalakit.FokRad(szog));
    }
    public void Kiir() {
        System.out.println("Derekszogu haromszog oldalai");
        System.out.println("a oldal: " + bef1);
        System.out.println("b oldal: " + bef2);
        System.out.println("c oldal: " + atfogo);
    }
}
```

A program futásának eredménye:

```
Derekszogu haromszog oldalai
a oldal: 2.5000000002960414
b oldal: 4.3301270187512735
c oldal: 5.0
```

A *MintaOsztaly* adatmezőit tegyük nyilvánossá (**public**)! Jelenítsük meg a tartalmukat a ***main()*** metódusban! Mit tapasztalunk? (*MintaPr4*)

```
// nyilvános adatmezők használata
class MintaOsztaly {
    public int x, y, ered;

    MintaOsztaly ( int x, int y) {
        this.x = x; this.y = y; ered = 0;
    }

    public void Muvelet() {
        ered = x + y;
    }

    public void Kiir() {
        System.out.println(x + " + " + y + " = " + ered);
    }
}

public class MintaPr4 {
    public static void main(String[] args) {
        int adat1 = 4, adat2 = 5;
        MintaOsztaly peldany1= new MintaOsztaly(adat1, adat2);
        System.out.println("adat1: " + peldany1.x +
                           "  adat2: " + peldany1.y + "\n");
        peldany1.Muvelet();
        peldany1.Kiir();
    }
}
```

A program futásának eredménye:

```
adat1: 4  adat2: 5

4 + 5 = 9
```

Azt tapasztaljuk, hogy minden további nélkül hozzáférhetünk a *peldany1* objektum *x* és *y* adatmezőihez a ***main()*** metódusban.

Amennyiben az adatmezők elérést privátra módosítjuk az előző programban, fordítási hibákat kapunk:

x has private access in MintaOsztaly *System.out.println("adat1: " + peldany1.x +*
y has private access in MintaOsztaly *" adat2: " + peldany1.y + "\n");*

A hibaüzenetek az jelzik, hogy a **private** adatmezőkhöz nem férhetünk hozzá a ***main()*** metódusból.

Legyenek a *MintaOsztaly* adatmezői nem hozzáférhetők (**private**), és írjunk metódusokat a lekérdezésükhöz! Jelenítsük meg a tartalmukat a főprogramban a metódusok hívásával! Mit tapasztalunk? (*MintaPr5*)

```
// private adatmezők lekérdezése metódusokkal
class MintaOsztaly
{
    // adatmezők
    private int x, y, ered;

    MintaOsztaly ( int x, int y) {
        this.x = x; this.y = y; ered = 0;
    }

    public int Getx() { // visszaadja az x adatmező értékét
        return x;
    }

    public int Gety() { // visszaadja az y adatmezőt értékét
        return y;
    }

    public void Muvelet() {
        ered = x + y;
    }

    public void Kiir() {
        System.out.println(x + " + " + y + " = " + ered);
    }
}

public class MintaPr5
{
    public static void main(String[] args) {
        int adat1 = 4, adat2 = 5;
        MintaOsztaly peldany1= new MintaOsztaly(adat1, adat2);;
        System.out.println("adat1: " + peldany1.Getx() +
                           "  adat2: " + peldany1.Gety()+ "\n");
        peldany1.Muvelet();
        peldany1.Kiir();
    }
}
```

A program futásának eredménye:

```
adat1: 4  adat2: 5

4 + 5 = 9
```

A **private** *x* és *y* adatmezők tartalmát sikerült megjeleníteni a *Getx()* és *Gety()* metódusok segítségével.

Tervezzük osztályt, mely megszámlálja egy mondat összes karakterét, valamint statisztikát készít az 'a', 'e', 'i', 'o' és 'u' karakterek előfordulásáról és a szóközök számáról! (*KarPr*)

```
// Szöveg karaktereinek statisztikája
import java.io.*;

class KarakterStat
{
    private String s;      // adatmezők deklarációja
    private int karakterekszama;
    private int maganh_db, szokoz_db;
    private int a_db, e_db, o_db, i_db, u_db;

    KarakterStat () throws IOException {      // kostruktor
        BufferedReader br = new BufferedReader
                            (new InputStreamReader(System.in));
        System.out.print("Szoveg: ");
        s = new String(br.readLine());
        s = s.toLowerCase();          // kisbetűssé alakítjuk
        karakterekszama = s.length(); // karakterek darabszáma
        maganh_db = szokoz_db = 0;
        a_db = e_db = o_db = i_db = u_db = 0;
    }

    public void Vizsgal() {
        for(int i = 0; i< karakterekszama; i++) {
            switch (s.charAt(i)) {
                case 'a' :  maganh_db++; a_db++; break;
                case 'e' :  maganh_db++; e_db++; break;
                case 'i' :  maganh_db++; i_db++; break;
                case 'o' :  maganh_db++; o_db++; break;
                case 'u' :  maganh_db++; u_db++; break;
                case ' ' :  szokoz_db++; break;
            }
        }
    }

    public void Megjelenit() {
        System.out.print("A vizsgalt szoveg: ");
        System.out.println(s);
        System.out.println("\nKarekterek szama  : "
                           + karakterekszama);
        System.out.println("Maganhangzok szama: " + maganh_db);
        System.out.println("      a darabszama: " + a_db);
        System.out.println("      e darabszama: " + e_db);
        System.out.println("      i darabszama: " + i_db);
        System.out.println("      o darabszama: " + o_db);
        System.out.println("      u darabszama: " + u_db);
        System.out.println("Szokozok     szama: " + szokoz_db);
    }
}
```

```
public class KarPr{
    public static void main(String[] args) throws IOException {
        KarakterStat szoveg = new KarakterStat();
        szoveg.Vizsgal();
        szoveg.Megjelenit();
    }
}
```

A program futásának eredménye:

```
Szoveg: Ma szep az ido, hull a ho, es sut a nap.
A vizsgalt szoveg: ma szep az ido, hull a ho, es sut a nap.

Karekterek szama  : 40
Maganhangzok szama: 12
      a darabszama: 5
      e darabszama: 2
      i darabszama: 1
      o darabszama: 2
      u darabszama: 2
Szokozok      szama: 10
```

Tervezzünk osztályt, amely alkalmas általános háromszög kerületének és területének kiszámítására, ha ismerjük az általános háromszög két oldalát és a közbezárt szöget! *(AltHsz)*

```
// Általános háromszög területének és kerületének számítása
class AltalanosHaromszog {
// az adatmezők deklarációja
    private double aold, bold, cold,gamma,ter, ker;

    // paraméteres konstruktor
    AltalanosHaromszog ( double a, double b, double szog) {
        aold = a;
        bold = b;
        gamma = szog;  // fokban
        cold = ter = ker =0;
    }

    // az osztály metódusai
    public void C_oldal() {
        // a harmadik oldal számítása koszinusz tétellel
        cold = Math.sqrt(Math.pow(aold,2) + Math.pow(bold,2)
                - 2 * aold * bold * Math.cos(gamma*Math.PI/180));
    }

    public void Terulet() {
        double s;
        // ha nincs még a c oldal kiszámítva
        if (aold == 0) C_oldal();
        // a félkerület számítása a lokális s változóba
        s = (aold + bold + cold)/2;
        // a terület számítása Heron-képlettel
        ter = Math.sqrt(s * (s-aold)*(s-bold)*(s-cold));
    }
```

```
    public void Kerulet() {
        // ha nincs még  c oldal kiszámítva
        if (aold == 0) C_oldal();
        // a kerület számítása
        ker = aold + bold + cold;
    }
    public void Kiir() {
        // az adatmezők megjelenítése
        System.out.println("Altalanos haromszog adatai\n");
        System.out.println("A oldal: " + aold);
        System.out.println("B oldal: " + bold);
        System.out.println("C oldal: " + cold);
        System.out.println("kozbezart szog: " + gamma);
        // a számítási eredmények megjelenítése
        System.out.println("\nKerulet: " + ker);
        System.out.println("Terulet: " + ter);
    }
}
public class AltHSz{
    public static void main(String[] args) {
        double adat1 = 2, adat2 = 4, szog = 30;
        AltalanosHaromszog ah=
                new AltalanosHaromszog(adat1, adat2, szog);
        ah.C_oldal();
        ah.Terulet();
        ah.Kerulet();
        ah.Kiir();
    }
}
```

A program futásának eredménye:

```
Altalanos haromszog adatai

A oldal: 2.0
B oldal: 4.0
C oldal: 2.4786273498549516
kozbezart szog: 30.0

Kerulet: 8.478627349854952
Terulet: 1.9999999999999996
```

Tervezzük osztályt, amely alkalmas derékszögű háromszög kerületének és területének kiszámítására, ha ismerjük a derékszögű háromszög két befogóját! (*DHaromsz*)

```
// Derékszögű háromszög területének és kerületének számítása
class DerekSzHaromszog
{
    // adatmezők deklarációja
    private double aold, bold,cold,ter, ker;
```

```
        // paraméteres konstruktor
        DerekSzHaromszog ( double a, double b) {
            aold = a;
            bold = b;
            cold = ter = ker =0;
        }
        // az osztály metódusai
        public void C_oldal() {
            cold = Math.sqrt(Math.pow(aold,2) + Math.pow(bold,2));
        }
        public void Terulet() {
            ter = aold * bold/2;
        }
        public void Kerulet() {
            C_oldal();
            ker = aold + bold + cold;
        }
        public void Kiir() {
            System.out.println("Derekszogu haromszog adatai\n");
            System.out.println("A oldal: " + aold);
            System.out.println("B oldal: " + bold);
            System.out.println("C oldal: " + cold);
            System.out.println("\nKerulet: " + ker);
            System.out.println("Terulet: " + ter);
        }
    }
    // az alkalmazást indító main() metódus osztálya
    public class DHaromSz{
        public static void main(String[] args) {
            double adat1 = 3, adat2 = 4;
            DerekSzHaromszog h= new DerekSzHaromszog(adat1, adat2);;
            h.C_oldal();
            h.Terulet();
            h.Kerulet();
            h.Kiir();
        }
    }
```

A program futásának eredménye:

```
Derekszogu haromszog adatai

A oldal: 3.0
B oldal: 4.0
C oldal: 5.0

Kerulet: 12.0
Terulet: 6.0
```

Oldjuk meg az előző *DHaromsz* feladatot statikus metódusok használatával, majd hasonlítsuk össze a két megoldást! (*DHaromszF*)

```
//a derékszögű háromszög feladat megoldása statikus metódusokkal
public class DHaromSzF
{
    static double C_oldal(double aold, double bold) {
        return Math.sqrt(Math.pow(aold,2) + Math.pow(bold,2));
    }

    static double Terulet( double aold, double bold) {
        return aold * bold/2;
    }

    static double Kerulet(double aold, double bold,
                          double cold ) {
        return aold + bold + cold;
    }

    static void Kiir(double aold, double bold, double cold,
                     double ker, double ter ) {
        System.out.println("Derekszogu haromszog adatai\n");
        System.out.println("A oldal: " + aold);
        System.out.println("B oldal: " + bold);
        System.out.println("C oldal: " + cold);
        System.out.println("\nKerulet: " + ker);
        System.out.println("Terulet: " + ter);
    }

    public static void main(String[] args) {
        double a = 3, b = 4, c, ter, ker;
        c = C_oldal(a, b);
        ter = Terulet(a,b);
        ker = Kerulet(a,b,c);
        Kiir(a,b,c,ker,ter);
    }
}
```

A program futásának eredménye:

```
Derekszogu haromszog adatai

A oldal: 3.0
B oldal: 4.0
C oldal: 5.0

Kerulet: 12.0
Terulet: 6.0
```

A *DHaromsz* és a *DHaromszF* feladatok megoldásaiból jól látható, hogy az objektumorientált esetben a metódusok az adatmezőkkel végeznek műveleteket, ezért nincsenek paramétereik. A statikus metódusok használata során az adatokat paraméterként adjuk át az egyes műveletek elvégzéséhez.

Tervezzük olyan osztályt, amely egy adott elemszámú, egész típusú tömböt feltölt 1 és 20 közötti véletlen számokkal! Metódusai kiszámítják a tömb elemeinek összegét, átlagát, és megkeresik a tömb legkisebb és a legnagyobb elemét, valamint megjeleníti az eredményeket! (*TombPr*)

```
// Tömb elemeinek statisztikája
public class TombPr
{
        public static void main(String[] args) {
            int n=15;
            Statisztika st = new Statisztika(n);
            st.AtlagSzamit();
            st.MinKeres();
            st.MaxKeres();
            st.Kiir();
        }
}

class Statisztika
{
    // adatmezők deklarációja
    private int x[], osszeg, min, max;
    private final int db;
    private double atlag;

    // paraméteres konstruktor
    Statisztika (int n) {
        db = n; // csak a konstruktorban lehetséges
        x = new int[n];
        for(int i = 0; i<db; i++)
            x[i] = (int)(20*Math.random()+1);
        min = max = osszeg = 0;
        atlag = 0.;
    }

    // az osztály metódusai
    public void OsszegSzamit() {
        // elemösszeg számítása
        for(int i = 0; i<db; i++)
            osszeg += x[i];
    }

    public void AtlagSzamit() {
        OsszegSzamit();
        atlag = (double)osszeg/db; // az átlag számítása
    }

    public void MinKeres() {
        min = x[0];
        for(int i = 1; i<db; i++)
            if (x[i] < min)
                min = x[i];
    }
```

```
    public void MaxKeres() {
        max = x[0];
        for(int i = 1; i<db; i++)
            if (x[i] > max)
                max = x[i];
    }
    public void Kiir()  {
        // az adatmezők megjelenítése
        System.out.println("A tomb adatai");
        for(int i = 0; i<db; i++) {
            if ( i % 10 == 0)  // egy sorba 10 elem kerül
                System.out.println();
            System.out.print(x[i] + "  ");
        }
        System.out.println("\n\nA tomb elemeinek osszege: "
                           + osszeg);
        System.out.println("A tomb elemeinek atlaga : "
                           + atlag);
        System.out.println("A tomb legnagyobb eleme : " + max);
        System.out.println("A tomb legkisebb eleme  : " + min);
    }
}
```

A program futásának eredménye:

```
A tomb adatai

2  15  12  6  7  18  15  5  7  2
14  3  6  6  5

A tomb elemeinek osszege: 123
A tomb elemeinek atlaga : 8.2
A tomb legnagyobb eleme : 18
A tomb legkisebb eleme  : 2
```

7.5 Objektum és objektumtömb típusú metódusparaméterek

A metódusok primitív típusú paraméterei értékparaméterek, ami azt jelenti, hogy a híváskor megadott argumentum értéke bemásolódik a megfelelő paraméterbe. Ennek következménye, hogy az argumentumban szereplő primitív típusú változó értéke nem módosítható a metóduson belül.

A metódusok tömb és objektum típusú paraméterei azonban referencia-paraméterek, vagyis a híváskor megadott tömb, illetve objektum hivatkozását veszik át. Ezzel a hivatkozással viszont megváltoztathatjuk az argumentumként megadott tömböt vagy objektumot. (Megjegyezzük, hogy paraméterátadás során a tömbökről és az objektumokról nem készül másolat.)

Készítsünk osztály, amely egy másik osztály nyilvános adatmezőit kezeli! (*Param1*)

```
// objektum-referencia típusú metódusparaméterek
import java.io.*;
class SzDatum {
    public int ev, kor, aktualis_ev;
}

class Adatok {
        private String nev, telefon;

        Adatok ( ) { // konstruktor
           nev = "Kiss Attila";
           telefon = "06203337771";
           System.out.print("\nNeve: " + nev);
        }

        // az x objektum-referencia paraméter
        public void KorSzamol(SzDatum x) {
            x.kor  = x.aktualis_ev - x.ev;  // kor számítása
        }

        public void Kiir(SzDatum x) {
            System.out.println("\nSzemelyi adatok" +
                               " a kor szamitasaval\n");
            System.out.println("Neve: " + nev);
            System.out.println("Szuletesi datum: " + x.ev);
            System.out.println("Kora           : " + x.kor);
       }
}

public class Param1 {
    public static void main(String[] args) throws IOException {
        BufferedReader be = new BufferedReader
                            (new InputStreamReader(System.in));
        SzDatum y = new SzDatum();
        Adatok d= new Adatok();
        System.out.print("\nSzuletesi eve: ");
        y.ev =  Integer.valueOf(be.readLine()).intValue();
        System.out.print("Aktualis ev  : ");
        y.aktualis_ev=Integer.valueOf(be.readLine()).intValue();
        d.KorSzamol(y);
        d.Kiir(y);
    }
}
```

A program futásának eredménye:

```
Neve: Kiss Attila
Szuletesi eve: 1974
Aktualis ev  : 2005

Szemely adatok a kor szamitasaval

Neve: Kiss Attila
Szuletesi datum: 1974
Kora           : 31
```

Írjunk programot, amely objektumtömbbe tárolja a háromszög három csúcspontjának koordinátáit! A koordinátapontot a *Koord* osztály kezelje! A *Haromszog* osztály *OldalakSzamitasa* metódusa a paraméterlistán átadott objektumtömb koordinátapontjai alapján határozza meg a háromszög oldalait! Metódusok segítségével számítsuk ki a háromszög kerületét, és írassuk ki az eredményeket! (*Param2*)

```
// Objektumtömb, mint metódusparaméter
import java.io.*;

class Koord
{
    public int x,y;  // koordinátapont
}
class Haromszog
{
    private double aold, bold,cold, ker;
    Haromszog ( ) { // konstruktor
        ker =0;
    }
    public void OldalakSzamitasa(Koord[] lista) {
        aold = Math.hypot((lista[0].x - lista[1].x),
                          (lista[0].y - lista[1].y));
        bold = Math.hypot((lista[1].x - lista[2].x),
                          (lista[1].y - lista[2].y));
        cold = Math.hypot((lista[2].x - lista[0].x),
                          (lista[2].y - lista[0].y));
    }
    public void Kerulet() {  //  háromszög kerületének számítása
        ker = aold + bold + cold;
    }
    public void Kiir() {  // az eredmények megjelenítése
        System.out.println("\nHaromszog oldalai\n");
        System.out.println("A oldal: " + aold);
        System.out.println("B oldal: " + bold);
        System.out.println("C oldal: " + cold);
        System.out.println("\nKerulet: " + ker);
    }
}
public class Param2 {
    public static void main(String[] args) throws IOException {
        BufferedReader be = new BufferedReader
                            (new InputStreamReader(System.in));
        System.out.println("Koordinatak: ");
        Koord[] tav = new Koord[3]; // objektum-ref. tömb
        for(int i = 0; i<3; i++) {
           tav[i] = new Koord(); // az objektum létrehozása
           System.out.print("\nx koordinata: ");
           tav[i].x = Integer.valueOf(be.readLine()).intValue();
           System.out.print("y koordinata: ");
           tav[i].y = Integer.valueOf(be.readLine()).intValue();
        }
```

```
            Haromszog h= new Haromszog();
            h.OldalakSzamitasa(tav); // tav objektumref. tömb
            h.Kerulet();
            h.Kiir();
        }
    }
```

A program futásának eredménye:

```
Koordinatak:

x koordinata: 0
y koordinata: 0

x koordinata: 6
y koordinata: 0

x koordinata: 3
y koordinata: 3
Haromszog oldalai

A oldal: 6.0
B oldal: 4.242640687119285
C oldal: 4.242640687119285

Kerulet: 14.485281374238568
```

7.6 Csomagok (package) készítése, importálása

A Java nyelvben a logikailag összetartozó osztályok csoportokba, ún. *csomagokba* (*packages*) vannak rendezve. A Java API leggyakrabban használt csomagjait az F2. fejezetben foglaltuk össze. A ***java.lang*** csomag kivételével a csomagokban tárolt osztályokat a programmodul elején importálnunk kell, például:

```
// a java.io csomag minden osztályát importáljuk
import java.io.*;

// a java.io csomagból csak a BufferedReader osztályt importáljuk
import java.io.BufferedReader;
```

A ***java.lang*** csomag a Java nyelv alaposztályait tartalmazza, mint például a ***Math***, a ***System***, az ***Integer***, a ***Double*** stb. A csomag nevének minden egyes tagja egy-egy alkönyvtárat jelöl. Például, a ***java.lang.Math*** importálásakor a *java\lang\Math.class* osztályt tesszük hozzáférhetővé a programunk számára. (A csomagok könyvtárszerkezetét gyakran egyetlen JAR-állományba tömörítve terjesztik.)

Java-ban adott a lehetőség, hogy mi is csomagot (*package*) készítsünk a lefordított osztályainkból. Az ehhez szükséges lépéseket példán keresztül mutatjuk be. A példában a *Teglatest* és a *Kocka* osztályokból hozzuk létre a *Testek* csomagot.

A *Testek package* készítésének lépései:

1. Először hozzunk létre egy könyvtárat a csomag nevével (*Testek*), és másoljuk ide az egyes osztályokat önállóan tartalmazó *.java* állományokat (*Teglatest.java* és *Kocka.java*)!

2. Mindkét állomány első sorába helyezzük el a **package** kulcsszót és a csomag nevét tartalmazó utasítást!

```
package Testek;
```

Ahhoz, hogy az osztályok elérhetők legyenek kívülről nyilvánossá (**public**) kell tenni őket. Az osztálytagok hozzáférését azonban igény szerint szabályozhatjuk:

```
// csomag létrehozása
package Testek;
public class Teglatest{
     private double a, b, c;
     private double terf, felsz;
     public Teglatest(double a1, double b1, double c1){
       a = a1; b = b1; c = c1;
}
```

3. Egyenként fordítsuk le a *Testek* könyvtárban található forrásfájlokat *.class* állományokká (*Teglatest. class* és *Kocka. class*)!

A **package** használata nagyon egyszerű. A program **import** utasításában a csomag nevét ponttal elválasztva követi a a beépítendő osztály neve:

```
import Testek.Teglatest;  // csak a Teglatest osztályt
import Testek.*;          // mindkét osztályt
```

Ezután a *Teglatest* osztályt úgy használhatjuk, mintha a *main()* metódus osztályával közös állományban deklaráltuk volna.

A fenti lépések alapján készítsünk *Testek* néven egy csomagot, amely az alábbi *Teglatest* és a *Kocka* osztályokat tartalmazza! (*Testek*)

A *Teglatest.java* forrásfájl tartalma:

```
package Testek;
public class Teglatest
{
    private double a, b, c;
    private double terf, felsz;
    public Teglatest(double a1, double b1, double c1) {
        a = a1; b = b1; c = c1;
    }
    public void Init(double a1, double b1, double c1) {
        a = a1; b = b1; c = c1;
    }
    public void TerfogatSzamol() {
        terf = a * b * c;
    }
    public void FelszinSzamol() {
        felsz = 2*(a * b + b * c + a * c);
    }
```

```
    public void Kiir(){
        System.out.println("Teglatest adatai :");
        System.out.println("a:" + a + " b: " + b + " c: " + c);
        System.out.println("Teglatest felszine : " + felsz);
        System.out.println("Teglatest terfogata: " + terf);
    }
}
```

A *Kocka.java* forrásállomány tartalma:

```
package Testek;
public class Kocka
{
    private double a;
    private double lapatlo, testatlo, terf, felsz;

    public Kocka(double a1) {
        a = a1;
    }
    public void Init(double a1) {
        a = a1;
    }

    public void LapAtlo() {
        lapatlo = a * Math.sqrt(2);
    }
    public void TestAtlo() {
        testatlo = a * Math.sqrt(3);
    }
    public void TerfogatSzamol(){
        terf = a * a * a;
    }
    public void FelszinSzamol(){
        felsz = 6 * a * a;
    }
    public void Kiir(){
        System.out.println("Kocka oldalele  : " + a);
        System.out.println("Kocka lapatloja : " + lapatlo);
        System.out.println("Kocka testatloja: " + testatlo);
        System.out.println("Kocka felszine  : " + felsz);
        System.out.println("Kocka terfogata : " + terf);
    }
}
```

Készítsünk programot, amely a *Testek* nevű csomag *Teglatest* osztályát használja! (*Teglatest_pr2*)

```
import Testek.Teglatest;     // az import használata
class Teglatest_pr2{
   public static void main(String[] args) {
       double a1 = 1, b1 = 2, c1 = 3;
       Teglatest t = new Teglatest(a1, b1, c1);
       t.TerfogatSzamol();
```

```
            t.FelszinSzamol();
            t.Kiir();
            System.out.println();
            t.Init(2,3,4);
            t.TerfogatSzamol();
            t.FelszinSzamol();
            t.Kiir();
        }
    }
```

A program futásának eredménye:

```
Teglatest adatai :
a:1.0 b: 2.0 c: 3.0
Teglatest felszine : 22.0
Teglatest terfogata: 6.0

Teglatest adatai :
a:2.0 b: 3.0 c: 4.0
Teglatest felszine : 52.0
Teglatest terfogata: 24.0
```

7.7 Kivételkezelés

A program futása közben fellépő hibákat a kivételkezelés (*exception-handling*) segítségével tarthatjuk kézben. Mivel Java-ban a kivételek is objektumok, minden egyes kivételhez saját kivételosztály tartozik. Példaként tekintsünk a leggyakrabban előforduló kivételek közül néhányat! (A ***java.lang*** csomag kivételei esetén nem adtuk meg a csomag nevét.)

Kivételosztály	*Leírás*
ArithmeticException	hiba az aritmetikai kifejezésben, például 0-val osztás,
ArrayIndexOutOfBoundsException	tömb indexhatárainak túllépése,
NumberFormatException	szöveg számmá való átalakításakor léphet fel,
NullPointerException	objektum-referencia helyett **null** értéket talál a futtató rendszer,
java.io.IOException	valamilyen input/output hiba lépett fel,
Exception	a kivételek ősosztálya,
RuntimeException	a JVM működése során előforduló események ősosztálya,

A kivételkezelés nem csak lehetőség a Java-nyelvű programokban, de sok esetben kötelesség is, ugyanis e nélkül a program fordítása megszakad. Vannak olyan metódusok, amelyek a fejsorukban jelzik (**throws**), hogy hiba esetén milyen kivételt továbbítanak, ilyen például a ***System.in*** objektum, ***read()*** metódusa:

```
public int read() throws IOException
```

Amennyiben saját metódusban kívánjuk hívni a ***Systen.in.read()*** metódust, választhatunk, hogy egyszerűen továbbítjuk ezt a kivételt, vagy kezeljük azt. Ha az első megol-

dás mellett döntünk, mint ahogy tettük ezt a könyvünk eddigi példaprogramjaiban, a metódus fejsorában jeleznünk kell, hogy mi is továbbadjuk az esetleg fellépő kivételt:

```
public static void main(String []a) throws java.io.IOException
{
   int ch = System.in.read();
}
```

Hasonló a helyzet, ha metóduson belül a **throw** (jelentése „*dob*") utasítással kiváltjuk a fenti kivételt, azonban valahol máshol kívánjuk azt kezelni. Általában feltételes utasítást használunk annak eldöntésére, hogy fennáll-e a kivételes helyzet.

```
public void metodus() throws java.io.IOException
{
    if (feltétel)
        throw new java.io.IOException();
}
```

Gyakran azonban a második megoldást választjuk, vagyis a **try-catch**, illetve a **try-catch-finally** utasításokkal felkészülünk az esetleges hibás állapotok kezelésére. A kivétel feldolgozásához egy **try** (jelentése „*próbáld*") blokkba kell tenni azokat az utasításokat, amelyek kivételt okozhatnak. Ha a **try**-blokk végrehajtása közben kivétel jön létre, a blokk végén álló **catch** segítségével „elkaphatjuk" azt, és megadhatjuk, hogy mely utasításokat kell a kivétel fellépése esetén végrehajtani. Amennyiben semmilyen kivétel sem keletkezik, vagy ha típusa nem egyezik a **catch**-ban szereplővel, akkor a **cath**-blokk utasításai nem futnak le.

```
public static void main(String []a)
{
    try {
        int ch = System.in.read();
    }
    catch (java.io.IOException e) {
        System.out.println("Olvasási hiba lépett fel");
    }
}
```

Egy **try**-blokkhoz több **catch**-blokk is tartozhat, hiszen a **catch** után zárójelben álló, egyetlen paraméter típusa választja ki a kezelendő kivételt. Az aktuális **catch**-blokk végrehajtása után az utolsó **catch**-blokkat követő utasítással folytatódik a program futása. Ha a fellépő kivételt egyik **catch** sem azonosította, az a futtatórendszerhez továbbítódik, és a program futása hibaüzenettel megszakad.

```
public void metodus() {
    try {
       // figyelt utasítások
    }
    catch(ArrayIndexOutOfBoundsException e)     {
        System.out.println("Hibás tömbindex.");
    }
    catch(ArithmeticException e) {
```

```
            System.out.println("Számítási hiba.");
        }
        System.out.println("A program tovább fut");
    }
```

Megjegyezzük, hogy a kivételek ősosztályai (például ***Exception***, ***RuntimeException***) minden belőlük származtatott kivételt azonosítanak.

Ha egy megkezdett tevékenységet mindenképpen be szeretnénk fejezni, akkor a **try** blokkot követő **catch** szerkezet után, a **finally** (jelentése *„végül"*) blokkban megadhatjuk az ehhez szükséges utasításokat. A **finally**-blokk utasításai mindig végrehajtódnak, akár lép fel kivétel, akár nem. Kezelt kivétel esetén a **catch** utasításait követően, kezeletlen kivétel esetén pedig a program futásának hibaüzenettel történő befejezése előtt. A **finally**-blokk utasításai akkor is lefutnak, ha a **return**, a **break** vagy a **continue** utasítással kilépünk a **try**-blokkból.

```
public void metodus() {
    try {
        System.out.println("Figyelés indul...");
        //  utasítások
    }
    catch(ArithmeticException e) {
        System.out.println("Számítási hiba.");
    }
    finally {
        System.out.println("Ez mindeképp lefut!");
    }
    System.out.println("A program tovább fut");
}
```

Ha nem lép fel kivétel a **try**-blokkban, akkor a metódus hívásakor az alábbi szöveg jelenik meg:

Figyelés indul...
Ez mindenképp lefut!
A program tovább fut

A kezelt ***ArithmeticException*** kivétel keletkezése esetén ezt láthatjuk:

Figyelés indul...
Számítási hiba.
Ez mindenképp lefut!
A program tovább fut

A kezeletlen ***NumberFormatException*** kivétel fellépése pedig ezt eredményezi:

Figyelés indul...
Ez mindenképp lefut!
...futás közbeni hibaüzenet ...

Érdemes megjegyezni, hogy a **catch**-blokkok nélküli **try-finally** utasítást is használhatjuk.

Készítsünk programot, amely az általános háromszög két oldalának és a közbezárt szögének beolvasásakor *Formatum* hibát figyel, és hibajelzést ad, ha szám helyett karaktert adtunk meg! (*Kivetel1*)

```
// Kivételkezelés alkalmazása
import java.io.*;
class AltalanosSzHaromszog
{
    private double aold, bold, cold,gamma,ter, ker;

    AltalanosSzHaromszog ( double a, double b, double szog) {
        aold = a; bold = b;
        gamma = szog;  // fokban
        cold = ter = ker =0;
    }

    public void C_oldal() {
        // a harmadik oldal számítása koszinusz tétellel
        cold = Math.sqrt(Math.pow(aold,2) + Math.pow(bold,2)- 2*
               aold * bold * Math.cos(gamma* Math.PI/180 ));
    }

    public void Terulet() {
        double s;  // lokális változó
        // ha nincs még  c oldal kiszámítva
        if (aold == 0) C_oldal();
        // a félkerület számítása a lokális s változóba
        s = (aold + bold + cold)/2;
        // terület számítása Heron-képlettel
        ter = Math.sqrt(s * (s-aold)*(s-bold)*(s-cold));
    }
    public void Kerulet() {
        // ha nincs még  c oldal kiszámítva
        if (aold == 0) C_oldal();
        // kerület számítása
        ker = aold + bold + cold;
    }
    public void Kiir() {
        // az adatmezők megjelenítése
        System.out.println("Altalanos haromszog adatai\n");
        System.out.println("A oldal: " + aold);
        System.out.println("B oldal: " + bold);
        System.out.println("C oldal: " + cold);
        System.out.println("kozbezart szog: " + gamma);
        // az eredmények megjelenítése
        System.out.println("\nKerulet: " + ker);
        System.out.println("Terulet: " + ter);
    }
}
public class Kivetel1 {
    public static void main(String[] args) throws IOException {
        double adat1, adat2, szog;
        BufferedReader be = new BufferedReader
                            (new InputStreamReader(System.in));
```

```
            try {
              System.out.print("A oldal = ");
              adat1 =Double.valueOf(be.readLine()).doubleValue();
              System.out.print("B oldal = ");
              adat2 =Double.valueOf(be.readLine()).doubleValue();
              System.out.print("Ket oldal altal bezart szog (fok) =");
              szog =Double.valueOf(be.readLine()).doubleValue();
              AltalanosSzHaromszog ah=
                      new AltalanosSzHaromszog(adat1, adat2, szog);
              ah.C_oldal();
              ah.Terulet();
              ah.Kerulet();
              ah.Kiir();
            }
            catch(NumberFormatException e) {
              System.out.println("Formatum hiba ");
            }
            catch(IOException e) {
              System.out.println("Olvasasi hiba ");
            }
        }
    }
```

A program futásának eredményei:

```
A oldal = 3
B oldal = 4
Ket oldal altal bezart szog (fok) = 30
Altalanos haromszog adatai

A oldal: 3.0
B oldal: 4.0
C oldal: 2.0531415706603067
kozbezart szog: 30.0

Kerulet: 9.053141570660307
Terulet: 2.9999999999999996
```

Hibás adat megadása a szög esetén:

```
A oldal = 2
B oldal = 3
Ket oldal altal bezart szog (fok) = a
Formatum hiba
```

Egészítsük ki a *TombPr* egész típusú tömböt feldolgozó alkalmazást az 1 és 100 közé eső tömbméret bevitelének lehetőségével, és vértezzük fel a kivételkezelés eszköztárával! (*Kivetel2*)

```
// Tömb elemeinek statisztikája kivételkezeléssel
import java.io.*;
class Statisztika
{
    private int x[], osszeg, min, max;
    private final int db;
```

```
    private double atlag;

    Statisztika (int n) {
        db = n; // csak a konstruktorban lehetséges
        x = new int[db];
        // csak formai megoldás, itt sosem lép fel az
        // indexhatár-túllépési kivétel
        try {
            for(int i = 0; i<db; i++)
                x[i] = (int)(20*Math.random()+1);
        }
        catch (ArrayIndexOutOfBoundsException e) {
              System.out.println("Indexhatar hiba.");
              System.exit(1); // kilépünk a programból
        }
        atlag = min = max = osszeg = 0;
    }

    public void OsszegSzamit() {
        // elemösszeg számítása
        for(int i = 0; i<db; i++)
            osszeg += x[i];
    }

    public void AtlagSzamit() {
        OsszegSzamit();
        atlag = (double)osszeg/db; // az átlag számítása
    }

    public void MinKeres() {
        min = x[0];
        for(int i = 1; i<db; i++)
            if (x[i] < min)
                min = x[i];
    }

    public void MaxKeres() {
        max = x[0];
        for(int i = 1; i<db; i++)
            if (x[i] > max)
                max = x[i];
    }

    public void Kiir()  {
        // az adatmezők megjelenítése
        System.out.println("A tomb elemei:");
        for(int i = 0; i<db; i++) {
            // egy sorba 10 elem kerül
            if ( i % 10 == 0)
                System.out.println();
            System.out.print(x[i] + "  ");
        }
        System.out.println("\n\nA tomb elemeinek osszege: "
                           + osszeg);
        System.out.println("A tomb elemeinek atlaga : "
                           + atlag);
        System.out.println("A tomb legnagyobb eleme : " + max);
```

```
            System.out.println("A tomb legkisebb eleme  : " + min);
        }
    }

    public class Kivetel2 {
            public static void main(String[] args) {
                BufferedReader be = new BufferedReader
                                    (new InputStreamReader(System.in));
                // ha kivétel lép fel, akkor 15 lesz a tömb mérete
                int n=15;
                try {
                    System.out.print("Az elemek szama = ");
                    n =Integer.parseInt(be.readLine());
                }
                catch(NumberFormatException e) {
                    System.out.println("Formatum hiba ");
                }
                catch(IOException e) {
                    System.out.println("Olvasasi hiba ");
                }
                // csak 1 és 100 közé eső értéket fogadunk el
                finally {
                    if (n<1 || n>100) {
                        System.out.println("Rossz tombmeret ");
                        return;  // kilépünk a programból
                    }
                }
                Statisztika st = new Statisztika(n);
                st.AtlagSzamit();
                st.MinKeres();
                st.MaxKeres();
                st.Kiir();
            }
    }
```

Megjegyzések a Kivetel2 példaprogramhoz:

Ahogy a megjegyzésben is szerepel, a *Statisztika (int n)* konstruktorban alkalmazott **try**-blokkban soha sem lép fel az ***ArrayIndexOutOfBoundsException*** kivétel, mivel jól használtuk a **for** ciklus feltételét, és a benne szereplő méret pedig a tömb létrehozási mérete.

```
            try {
                for(int i = 0; i<db; i++)
                    x[i] = (int)(20*Math.random()+1);
            }
            catch (ArrayIndexOutOfBoundsException e) {
                   System.out.println("Indexhatar hiba.");
                   System.exit(1); // kilépünk a programból
            }
```

Azzal a céllal szerepeltettük mégis itt ezt a megoldást, hogy bemutassuk az alkalmazásból való kilépés általános formáját, a ***System.exit()*** metódus hívását. A hívás azonnal megállítja az éppen futó Java Virtuális Gép működését. A hagyományok szerint a

nem nulla argumentum (kilépési kód) valamilyen programhibára utal, míg a 0 azt jelzi, hogy minden rendben volt.

A ***main()*** metódusban szereplő **try-catch-finally** struktúrát úgy alakítottuk ki, hogy kivétel fellépése esetén is legyen értéke a tömbméretet meghatározó *n* változónak.

```
int n=15;
try {
    System.out.print("Az elemek szama = ");
    n =Integer.parseInt(be.readLine());
}
catch(NumberFormatException e) {
    System.out.println("Formatum hiba ");
}
catch(IOException e) {
    System.out.println("Olvasasi hiba ");
}
finally {
    if (n<1 || n>100) {
        System.out.println("Rossz tombmeret ");
        return;
    }
}
```

Amennyiben az adatbevitel sikertelen, 15-elemű tömb létrehozását célozzuk meg. (Nem kezelt kivétel fellépése esetén a program futása megszakad.)

A minden esetben végrehajtódó **finally**-blokkban ellenőrizzük, hogy a tömb mérete 1 és 100 közé esik-e. Amennyiben 1-től kisebb, vagy 100-tól nagyobb a bevitt érték, az üzenet kiírása után kilépünk az alkalmazásból, a ***main()*** metódusból való visszatéréssel (**return**).

Az alábbi futási eredmények jól szemléltetik az elmondottakat:

Kivétel fellépése nélküli futtatás eredménye:

```
Az elemek szama = 23
A tomb elemei:

15  17  1  8  18  20  9  19  16  13
10  2  4  6  18  1  16  6  4  8
4  10  2

A tomb elemeinek osszege: 227
A tomb elemeinek atlaga : 9.869565217391305
A tomb legnagyobb eleme : 20
A tomb legkisebb eleme  : 1
```

Ha rosszul adjuk meg az elemek számát, az alábbi eredményt kapjuk:

```
Az elemek szama = otven
Formatum hiba
A tomb elemei:
```

```
7  16  2  15  12  17  10  6  10  7
13  17  17  5  15

A tomb elemeinek osszege: 169
A tomb elemeinek atlaga : 11.266666666666667
A tomb legnagyobb eleme : 17
A tomb legkisebb eleme  : 2
```

Ha hibás tömbméretet használunk a futás megszakad:

```
Az elemek szama = 123
Rossz tombmeret
```

Tervezzünk csak statikus tagokkal rendelkező osztályt a Fibonacci-sorozat első 92 elemének gyors előállítására! Használjunk statikus inicializálót, a statikus elemtömb feltöltésére! (*Fibonacci*)

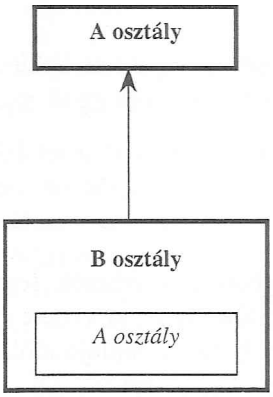
A Fibonacci-sorozat 0. eleme *0*, első eleme *1*, a további elemek pedig a megelőző két elem összegeként keletkeznek. A leggyorsabb megoldás akkor kapjuk, ha az elemeket statikus adattömbben egyszer, az osztály betöltésekor hozzuk létre. Ezek után akárhányszor felhasználhatjuk.

```
class Fibonacci {
    private static final long szamok[];
    public static final int ELEMSZAM = 91; // ennyi fér
                                           // a long típusba!
    public static long getElem(int index) {
        return szamok[index];
    }
    static {  // az elemeket tároló tömb feltöltése
       szamok = new long[ELEMSZAM];
       long e0=0, e1=1, e2;
       szamok[1] = (szamok[0]=0) + 1;
       for (int i=2; i<ELEMSZAM; i++) {
           e2 = e0 + e1;
           szamok[i]=e2;
           e0=e1; e1=e2;
       }
    }
}

public class FiboPelda
{
    public static void main(String[] args) {
        for (int i=0; i<Fibonacci.ELEMSZAM; i++)
            System.out.println(""+i+" : "+Fibonacci.getElem(i));
    }
}
```

8. Öröklés

Az öröklés az objektum-orientált programozás legfőbb tulajdonsága. Az öröklés azt jelenti, hogy egy meglévő osztályból kiindulva, új osztályt hozunk létre (származtatunk). A származtatás során az új osztály örökli a meglévő osztály nyilvános (**public**) és védett (**protected**) tulajdonságai (adatmezőit) és viselkedését (metódusait), melyeket aztán sajátjaként használhat. Azonban az új osztály bővítheti is a meglévő osztályt, új adatmezőket és metódusokat definiálhat, illetve újraértelmezheti (lecserélheti) az öröklött, de működésében elavult metódusokat (*polymorphism*).

A Java nyelv örökléssel kapcsolatos szóhasználata valamelyest eltér a megszokottól, ezért röviden összefoglaljuk az egyes magyar és angol nyelvű kifejezéseket, aláhúzással kiemelve a Java-ban alkalmazottakat.



A osztály, amiből származtatunk:
alaposztály, ősosztály, szülőosztály (*base class, ancestor class, parent class, superclass*),

a művelet:
öröklés, származtatás, bővítés (*inheritance, derivation, extending, subclassing.*)

B osztály, a származtatás eredménye:
utódosztály, származtatott osztály, bővített osztály, gyermekosztály, alosztály (*descendant class, derived class, extended class, child class, subclass*),

A fenti kapcsolatot megvalósító Java programrészlet:

```
class AOsztaly {
    // ...
}

class BOsztaly extends AOsztaly {
    // ...
}
```

A szakirodalomban nem egyértelmű a fogalmak használata, például egy adott osztály alaposztálya vagy ősosztálya tetszőleges elődöt jelölhet, illetve az utódosztály vagy származtatott osztály minősítés bármely osztály esetén alkalmazható. Könyvünkben ezeket a fogalmakat a közvetlen ős (szülő, *super*), illetve közvetlen utód (gyermek, *sub*) értelemben használjuk. Valószínűleg ez a kis fogalomzavar lehetett az oka annak, hogy a Java nyelv fejlesztői új, egyértelmű fogalmakat alkalmaztak: *superclass/ subclass* (szuperosztály / alosztály).

A származtatás általános formájában az utódosztály neve után az **extends** kulcsszó, majd pedig az ősosztály azonosítója következik:

```
class UtódOsztály extends ŐsOsztály {
}
```

8.1 Egyszeres öröklés és a polimorfizmus

A C++ nyelvben egy osztálynak több közvetlen őse is lehet, ezt a többszörös öröklés (*multiple inheritance*) teszi lehetővé. A Java nyelv azonban közvetlenül csak egy osztálytól való származtatást, azaz az egyszeres öröklést (*single inheritance*) támogatja. Ennélfogva az öröklés több lépésben való alkalmazásával igazi fastruktúra alakítható ki (osztály-hierarchia), melynek gyökerében a ***java.lang.Object*** osztály áll.

Amennyiben az osztálydefinícióban nem adjuk meg a származtatás **extends** kulcsszavát, akkor is örökléssel jön létre az osztályunk. A bevezetőben szereplő programrészt valójában az alábbi formában dolgozza fel a fordító:

```
class AOsztaly extends java.lang.Object {
    // ...
}

class BOsztaly extends AOsztaly {
    // ...
}
```

Ahhoz, hogy az ősosztály adatmezői az utódosztályban is elérhetők legyenek, a **private** elérés helyett a **protected** hozzáférést kell alkalmaznunk. A védett módosító, a privát eléréshez hasonlóan elzárja az adatokat az osztály felhasználója elől, azonban az osztály továbbfejlesztője (örököse) számára elérhetővé teszi azokat.

A metódusokat általában **public** vagy **protected** hozzáféréssel adjuk meg, hacsak nem akarjuk valamelyiket **private** hozzáféréssel elrejteni a származtatott osztály elől.

A super hivatkozás

Amikor egy osztályt egy másik osztálytól származtatunk, akkor az utód osztályból a **super** referenciával mindig hivatkozhatunk a közvetlen ős elérhető tagjaira. (Emlékeztetőül, a **this** hivatkozás az aktuális objektumpéldányt jelöli). A **super** hivatkozást ritkán használjuk, hiszen az öröklés során, a védett és a nyilvános tagok az utód osztály saját tagjai lesznek.

Egyik alkalmazási területe a közvetlen ős konstruktorának hívása az utód konstruktorának első utasításából, például:

```
super(a1, b1);
```

Ha a konstruktor első utasításaként nem szerepel a *super(agumentumok);* utasítás, akkor a fordító beszúr ode egy *super();* utasítást, ami meghívja az ősosztály paraméter nélküli konstruktorát.

A származtatott osztályban deklarált adatmezők és metódusok elfedhetik a közvetlen őstől örökölt azonos nevű tagokat. Ekkor a **super** hivatkozás segítségével férhetünk hozzá az elfedett adatmezőkhöz, illetve hívhatjuk az ős nem látható metódusait.

```
super.a = 12;
    super.Kiir();
```

Virtuális öröklés, polimorfizmus

Abban az esetben, amikor utód osztály valamelyik metódusának nemcsak a ***neve***, hanem a ***paraméterlistája*** és ***típusa*** is megegyezik a közvetlen ősosztály egyik metódusáéval, akkor az utód semmissé teszi (*override*), lecseréli, újradefiniálja (*redefine*) az öröklött metódust. Az elmondottak szerint valósul meg Java-ban a polimorfizmus elve.

Más szavakkal ez azt jeleni, hogy mindig az osztály-hierarchia adott szintjétől (osztályától) függ, hogy egy adott metódus milyen tevékenységet végez.

A színfalak mögött a futtatórendszer egy táblázaton (*vtable*) keresztül hívja az osztály metódusait. Minden osztályhoz saját táblázat tartozik, amelyben mindig az aktuális metódusváltozat belépési pontja kerül, felülírva ezzel az esetleg örökölt (azonos fejsorú) metódus belépési pontját. A táblázat az osztály betöltésekor jön létre. Ez a működés a késői kötés (*late binding*) nevet viseli, ellentétben a korai (*early binding*) kötéssel, amikor a fordító közvetlenül hívja a függvényeket.

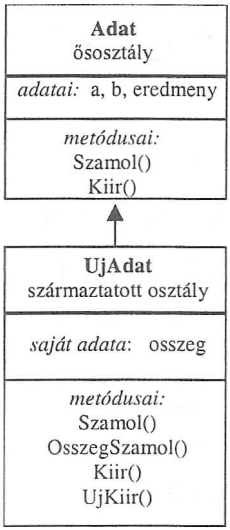
Java-ban a **final** metódusok kivételével minden metódus lecserélhető az öröklés során, vagyis minden metódus virtuális. Szükség esetén azonban, a **super** hivatkozás segítségével a régi (lecserélt) metódusokat is elérhetjük, például:

```
class AOsztaly {
    private int a = 12;
    public int fv() { return a; }
}

class BOsztaly extends AOsztaly {
    private int b = 23;
    public int fv() { return b + super.fv(); }
}
```

Felhívjuk a figyelmet arra, ha a közvetlen ős és az utód valamely metódusainak neve megegyezik, azonban paraméterlistájuk eltérő, akkor egy másik mechanizmus, a túlterhelés (*overloading*) érvényesül, és mindkét metódus elérhető lesz az utódból.

Írjunk programot, amelyben az *Adat* ősosztály *Szamol* metódusa számtani közepet, az *UjAdat* származtatott osztály *Szamol* metódusa pedig mértani közepet számol, valamint saját változójába összegez, és kiírja az új eredményeket! (*Orok1*)

Tervezzük meg az osztályok adatait és metódusait:



Az ősosztály azonosítója: *Adat*

- A **protected** hozzáférésű, valós típusú *a* és *b* változók az adatok.
- A **protected** hozzáférésű, valós típusú *eredmeny* a művelet eredményét tárolja.
- A konstruktora kezdőértéket ad az *a* és a *b* adatoknak.
- A *Szamol()* metódus számtani közepet számol.
- A *Kiir()* metódus megjeleníti az adatokat és az eredményt szövegesen.

```
class Adat  // az ősosztály
{
    protected double a, b;
    protected double eredmeny;

    Adat(double a1, double b1) { // konstruktor
       a = a1;
       b = b1;
    }

    public void Szamol() {
       eredmeny = (a + b)/2;  // számtani közép
    }

    public void Kiir() {
       System.out.println("Adatok: " + a + " " + b);
       System.out.println("Szamtani kozep: " + eredmeny);
    }
}
```

Az származtatott osztály azonosítója: *UjAdat*

- Hozzáfér az örökölt **protected**, valós típusú *a* és *b* változókhoz.
- Saját, **private** hozzáférésű, valós típusú *ujeredmeny* változóval rendelkezik, amely az összeg művelet eredményét tárolja.
- A konstruktora meghívja az ősosztály kostruktorát, hogy kezdőértéket adjon az *a* és a *b* adatoknak.

- A *Szamol()* metódusa mértani közepet számol, felülbírálva az ős osztály *Szamol* metódusát.
- A *Kiir()* metódus megjeleníti az adatokat és az eredményt szövegesen jelzi ki, felülbírálva az ős osztály *Kiir()* metódusát.
- Az *Osszeg()* metódus az adatok összegét számítja ki az *ujeredmeny* saját adatmezejébe.
- Az *UjKiir()* metódusa az összeg eredményt jeleníti meg.

```
class UjAdat extends Adat  // a származtatott osztály
{
    private double osszeg;
    UjAdat(double a1, double b1) { // konstruktor
       super(a1, b1); // az ősosztály konstruktorát hívja
       osszeg = 0;
    }
    public void Szamol() { // felülbírálja az ősosztály Szamol()-t
       eredmeny = Math.sqrt( a * b); // mértani közepet számol
    }
    public void Kiir() { // felülbírálja az ősosztály Kiir()-t
       System.out.println("Adatok: " + a + " " + b);
       System.out.println("Mertani kozep : " + eredmeny);
    }
    public void OsszegSzamol() { // saját összegszámoló metódus
       osszeg = a + b; // összeget számol
    }
    public void UjKiir() { // saját megjelenítő metódusa
       System.out.println("Osszeg : " + osszeg);
    }
}
```

A ***main***() függvény:

- Létrehozza a *ta* nevű *Adat* típusú objektumot, és
- Aktiválja a *ta* objektum *Szamol(), Kiir()* metódusait.
- Létrehozza a *tb* nevű *UjAdat* típusú objektumot.
- Aktiválja a *tb* objektum *Szamol(), Kiir(), OsszegSzamol(), UjKiir()* metódusait.

```
public class Orok1 {
    public static void main(String[] args) {
       Adat ta = new Adat(1,2);
       ta.Szamol();
       ta.Kiir();
       System.out.println();
       UjAdat tb = new UjAdat(1,2);
       tb.Szamol();
       tb.Kiir();
       tb.OsszegSzamol();
       tb.UjKiir();
    }
}
```

A program futásának eredménye:

```
Adatok: 1.0 2.0
Szamtani kozep: 1.5

Adatok: 1.0 2.0
Mertani kozep : 1.4142135623730951
Osszeg : 3.0
```

Írjunk programot, amely a *Teglalap* ősosztálytól származtat *Teglatest* osztályt! Az ősosztály *Szamol1()* metódusa a téglalap területét, a *Szamol2()* metódusa pedig a téglalap kerületét határozza meg. A származtatott osztályban a *Szamol1()* metódus a téglatest térfogatát, a *Szamol2()* metódus a pedig téglatest felszínét számítja. A *Kiir()* metódus pedig megjeleníti az eredményeket. (*Orok2*)

A *Teglalap* ősosztály

adatmezői:

- *a_oldal, b_oldal* a téglalap oldalai,
- *ered1* – a téglalap területe,
- *ered2* – a téglalap kerülete.

metódusai:

- *Szamol1()* – a téglalap területét számítja ki,
- *Szamol2()* – a téglalap kerületét számítja ki,
- *Kiir()* – kiírja a téglalap oldalait, valamint az eredményeket.

```
class Teglalap
{
   protected double a_oldal, b_oldal; // téglalap oldalai
   protected double ered1, ered2;     // eredmények tárolása

   Teglalap(double a1, double b1)  {  // konstruktor
      a_oldal = a1; b_oldal = b1;
   }

   public void Szamol1() {
      ered1 = a_oldal * b_oldal;   // a téglalap területe
   }

   public void Szamol2() {
      ered2 = 2*(a_oldal+b_oldal); // a téglalap kerülete
   }

   public void Kiir() {  // eredmények megjelenítése
      System.out.println("Teglalap oldalai: "
                         + a_oldal + " " + b_oldal);
      System.out.println("Teglalap terulete: " + ered1);
      System.out.println("Teglalap kerulete: " + ered2);
   }
}
```

A *Teglatest* származtatott osztály

adatmezői:

- örökli: *a_oldal*, *b_oldal*, *ered1*, *ered2* adatmezőket,
- saját adatmezője: *c_oldal*.

metódusai:

- konstruktora meghívja az ősosztály konstruktorát, hogy kezdőértéket adjon az ősosztálytól örökölt két oldalnak: *super(a1,b1);* majd a téglatest harmadik oldala kap kezdőértéket: *c_oldal = c1;*
- *Szamol1()* felülbírálja az ősosztály azonos nevű metódusát, és a téglatest térfogatát számítja ki.
- *Szamol2()* felülbírálja az ősosztály azonos nevű metódusát, és a téglatest felszínét számítja ki.
- *Kiir()* felülbírálja az ősosztály azonos nevű metódusát és kiírja az eredményeket.

```
class Teglatest extends Teglalap
{
   private double c_oldal;  // a téglatest harmadik oldala

   Teglatest(double a1, double b1, double c1) { // konstruktor
      super(a1, b1); // az ős osztály konstruktorának hívása
      c_oldal = c1;  // a harmadik oldal kezdőérték adása
   }

   public void Szamol1() {
      ered1 = a_oldal * b_oldal * c_oldal; // téglatest térfogata
   }

   public void Szamol2() {
      ered2 = 2*(a_oldal*b_oldal + a_oldal*c_oldal +
                 b_oldal*c_oldal);   // a téglatest felszíne
   }

   public void Kiir() {  // eredmények megjelenítése
      System.out.println("Teglatest oldalai: "
                         + a_oldal + " " + b_oldal + " " +c_oldal);
      System.out.println("Teglatest terfogata: " + ered1);
      System.out.println("Teglatest felszine : " + ered2);
   }
}
```

A ***main()*** metódus egy *tg* nevű, *Teglalap* típusú objektumot hoz létre, inicializálja és aktiválja a *Szamol1()*, *Szamol2()* és a *Kiir()* metódusait.

Ezt követően *tgt* néven létrehoz egy *Teglatest* típusú objektumot, majd inicializálja és aktiválja a *Szamol1()*, *Szamol2()* és a *Kiir()* metódusait.

```
public class Orok2{
   public static void main(String[] args) {
      Teglalap tg = new Teglalap(1,2);
      tg.Szamol1();
      tg.Szamol2();
      tg.Kiir();
      System.out.println();
      Teglatest tgt = new Teglatest(1,2,3);
      tgt.Szamol1();
      tgt.Szamol2();
      tgt.Kiir();
   }
}
```

A program futásának eredménye:

```
Teglalap oldalai: 1.0 2.0
Teglalap terulete: 2.0
Teglalap kerulete: 6.0

Teglatest oldalai: 1.0 2.0 3.0
Teglatest terfogata: 6.0
Teglatest felszine : 22.0
```

Írjunk programot, amely *Henger* ősosztálytól származtat egy *Tarolo* nevű osztályt! Az ősosztály *Terfogat()* metódusa a henger térfogatát számítja ki. A *Tarolo* osztály *fajsuly* és *suly* adatmezőkkel bővíti az ősosztálytól örökölt tulajdonságokat. A *Sulyszamitas()* metódus kiszámítja a tároló súlyát, az *UjKiir()* metódus pedig kiírja az új eredményeket! (*Orok3*)

Emlékeztetőül megjegyezzük, ha a konstruktor valamelyik paraméterének neve megegyezik az osztály mezőneveinek egyikével, akkor a **this** referenciával egyértelművé tehetjük, hogy melyik név jelöl adatmezőt:

```
Henger(double r, double h)   {
   this.r = r; this.h = h;
}
```

A *Henger* ősosztály

adatmezői:

- *r* – a henger sugara,
- *h* – a henger magassága,
- *terf* – a henger térfogata.

metódusai:

- *Terfogat()* a henger térfogatát számítja ki,
- *Kiir()* kiírja az adatokat és a henger térfogatát.

```
class Henger  // ősosztály
{
   protected double r, h;    // a henger sugara és magassága
   protected double terf;    // a henger térfogata

   Henger(double r, double h) { // konstruktor
      this.r = r; this.h = h;
   }

   public void Terfogat() {
      terf = Math.pow(r,2) * Math.PI * h; // a henger térfogata
   }

   public void Kiir() { // az eredmények megjelenítése
      System.out.println("Henger adatai    : " + r + " " + h);
      System.out.println("Henger terfogata: " + terf);
   }
}
```

A *Tarolo* származtatott osztály

örökölt adatmezői:

- *r* – a henger sugara,
- *h* – a henger magassága,
- *terf* – a henger térfogata.

saját adatmezői:

- *fajsuly* – a hengerben tárolt anyag fajsúlya,
- *suly* – a hengerben tárolt anyag súlya.

metódusai:

- *SulySzamitas()* metódusa aktiválja az ősosztály *Terfogat()* metódusát a henger térfogatának kiszámítására, majd a *terf* és a *fajsuly* szorzatával kiszámítja a hengerben tárolt anyag súlyát.
- *UjKiir()* kiírja tárolt anyag fajsúlyát és súlyát.

```
class Tarolo extends Henger
{
   double fajsuly, suly;      // új tulajdonságok

   Tarolo(double r1, double h1, double fajsuly1) {
      super(r1, h1);          // az ősosztály konstruktorának hívása
      fajsuly = fajsuly1;     // saját tulajdonság értékadása
   }

   public void SulySzamitas() {
      super.Terfogat(); // öröklött metódus hívása
      suly = terf * fajsuly;  // tároló súlyának számítása
   }

   public void UjKiir() {     // új eredmények megjelenítése
      System.out.println("Tarolt anyag fajsulya : " + fajsuly);
      System.out.println("Tarolt anyag sulya    : " + suly);
   }
}
```

A ***main()*** függvény létrehoz egy *Henger* típusú *th* objektumot, inicializálja és meghívja a *Terfogat()* és *Kiir()* metódusait, majd létrehozza a *Tarolo* típusú *t* objektumot és aktivizálja a *SulySzamitas()*, *Kiir()* és *UjKiir()* metódusait.

```
public class Orok3
{
   public static void main(String[] args) {
      Henger th = new Henger(1,1);
      th.Terfogat();
      th.Kiir();
      System.out.println();
      Tarolo t = new Tarolo(1,1,2);
      t.SulySzamitas();
      t.Kiir();
      t.UjKiir();
   }
}
```

A program futásának eredménye:

```
Henger adatai    : 1.0 1.0
Henger terfogata: 3.141592653589793

Henger adatai    : 1.0 1.0
Henger terfogata: 3.141592653589793
Tarolt anyag fajsulya : 2.0
Tarolt anyag sulya    : 6.283185307179586
```

8.2 Absztrakt metódusok és osztályok

Az objektum-orientált fejlesztés során olyan osztályokat is kialakíthatunk, melyeket csak továbbfejlesztésre, származtatásra lehet használni, és vele objektumpéldány nem készíthető, azonban objektum-referencia igen. Ehhez egyszerűen az **abstract** módosítót kell az osztályfejben elhelyezni, és az osztály absztrakttá válik.

Az absztrakt osztályok további jellegzetessége, hogy bizonyos műveletek (metódusok), amelyek szükségesek az osztály működéséhez általában nincsenek kidolgozva – a fejsorukat pontosvessző zárja és hiányzik a törzsük. Az ilyen metódusok fejsorában szintén az **abstract** kulcsszót kell alkalmazni. Az absztrakt metódusok megvalósítása ekkor az utódosztály származtatójának feladata. Amennyiben ezt nem teszi meg, akkor az utódosztály deklarációjában is szerepeltetnie kell az **abstract** módosítót.

Ha egy osztályban definiáltunk absztrakt metódust, akkor ezt az osztályfejben is jelezni kell az **abstract** módosítóval. Az absztrakt osztályban természetesen védett (és nyilvános) elérésű adtamezőket is elhelyezhetünk. Privát hozzáférésű adatmezők is szerelhetnek benne, hiszen nem szükséges, hogy az **abstract** osztály minden metódusa absztrakt legyen.

Az alábbi példában nyomon követhetjük a elmondottakat:

```
abstract class AOsztaly {  // absztrakt osztály
    protected int a = 12;
    public abstract void kiir (); // absztrakt metódus
}

class BOsztaly extends AOsztaly {
    protected int b;
    BOsztaly() { b = 23;}  // konstruktor
    public void kiir() {   // lecseréljük az örökölt kiir()-t
    System.out.println(""+ a + b);  // 1223 jelenik meg
    }
}
```

Az utód és az ősosztályok viszonya

Mielőtt tovább haladnánk, tisztáznunk kell egy fontos kérdést, az ősosztályok és az utódosztályok viszonyával kapcsolatosan. Tekintsük az alábbi hierarchiát!

```
abstract class AOsztaly {  // absztrakt osztály
    protected int a = 12;
    public abstract void kiir ();
    }

class BOsztaly extends AOsztaly {
    protected int b;
    BOsztaly() { b = 23; }
    public void kiir() {
      System.out.println(""+ a + b);  // 1223 jelenik meg
    }
}

class COsztaly extends BOsztaly {
    protected int c;
    COsztaly() { c = 34; }
    public void kiir() {
      System.out.println(""+  a + b + c );  // 122334 jelenik meg
    }
}
```

Készítsünk a *COsztaly*-ra hivatkozó referenciát, és inicializáljuk a *BOsztaly* példányával!

```
COsztaly cx = new BOsztaly();
```

A próbálkozásunk sikertelen, mivel a fordítás „*incompatible types*" hibával megszakad. Ennek oka, hogy a *COsztaly* bővített változata a *BOsztaly*-nak, így általában több tagot (például a *c*) tartalmaz.

Készítsünk most az *AOsztaly*-ra hivatkozó referenciát, inicializáljuk a *COsztaly* példányával, majd pedig hívjuk a *kiir()* metódust!

```
AOsztaly ax = new COsztaly();
ax.kiir();
```

Az *ax* referencia sikeresen hivatkozik a *COsztaly* példányára, sőt a *kiir()* metódus is lefut, megjelenítve az *122334* számsor. Két dologra következtethetünk ebből:

- Valamely ősosztály referenciájával tetszőleges utódobjektumra hivatkozhatunk. Ez azt jelenti, hogy egy utódobjektum mindig helyettesítheti bármely ősének objektumpéldányát – visszafelé ez nem igaz. Például, az ***Object*** típusú referencián keresztül minden objektumot elérhetünk. (A fenti példa módosított változatában típus-átalakítást kellett használnunk.)

```
Object ox = new COsztaly();
((COsztaly)ox).kiir();
```

  Az *AOsztaly*-t tartalmazó példában csak azért nem volt szükség típus-átalakítására, mivel az *AOsztaly* is rendelkezik *kiir()* nevű metódussal, melyet a *COsztaly*-ban lecseréltünk.

- A Java-rendszer futás közbeni típus-információkat (*runtime type information, RTTI*) tárol az objektumok mellett. Ezek alapján az **instanceof** művelettel megvizsgálhatjuk, hogy egy adott referencia objektumának típusa megegyezik-e a kifejezésben megadott osztállyal:

```
Object ox = new COsztaly();
if (ox instanceof COsztaly)
          ((COsztaly)ox).kiir();
```

Az elmondottak alapján objektum-referenciák tömbjét is létrehozhatjuk, a megfelelő ősosztályt választva típusául. Az ilyen tömbök típusaként gyakran absztrakt osztályt használunk.

```
AOsztaly[] at = new AOsztaly [2];  // referencia-tömb jön létre
at[0] = new BOsztaly();            // BOsztaly típusú objektum
at[1] = new COsztaly();            // COsztaly típusú objektum
for (int i = 0; i<at.length; i++)
     at[i].kiir();
```

A programrész futásának eredménye:

```
1223
122334
```

Készítsünk alkalmazást, amelyben a *Teglalap* és a *Kor* osztályokat az absztrakt *Alakzat* osztálytól származtatjuk! Ezt követően tároljuk 4-elemű tömbben a 2 darab *Teglalap* és a 2 darab *Kor* típusú objektumpéldány hivatkozását! (*Abstract*)

Mivel a *Teglalap* és a *Kor* osztályok egyaránt a közös *Alakzat* osztálytól származnak, a közös őssel referencia-tömböt hozhatunk létre. A tömb elemei így a származtatott osztályok objektumaira hivatkozhatnak.

```
static Alakzat[] tk = new Alakzat[4];
tk[0] = new Teglalap(1,2);
tk[1] = new Kor(1);
```

```
// abstract osztály használata
abstract class Alakzat                  // az Alakzat osztály
{
   abstract void Szamol1();
   abstract void Szamol2();
   abstract void Kiir();
}

class Teglalap extends Alakzat          // a Teglalap osztály
{
   private double a,b, ered1, ered2;

   Teglalap( double a1, double b1) {
      a = a1; b = b1;
   }

   public void Szamol1() {
      ered1 = a * b;
   }

   public void Szamol2() {
      ered2 = 2 * (a + b);
   }

   public void Kiir() {
      System.out.println("Teglalap oldalai : " + a + " " + b);
      System.out.println("Teglalap terulete: " + ered1);
      System.out.println("Teglalap kerulete: " + ered2);
   }
}

class Kor extends Alakzat               // a Kor osztály
{
   private double r, ered1, ered2;

   Kor( double r1) {
      r = r1;
   }

   public void Szamol1() {
      ered1 = Math.pow(r,2) * Math.PI;
   }

   public void Szamol2() {
      ered2 = 2 * r * Math.PI;
   }

   public void Kiir() {
      System.out.println("Kor sugara : " + r);
      System.out.println("Kor terulete: " + ered1);
      System.out.println("Kor kerulete: " + ered2);
   }
}
```

```
public class Abstract                    // a futtató osztály
{
   static Alakzat[] tk = new Alakzat[4];

   public static void main(String[] args) {
      tk[0] = new Teglalap(1,2);
      tk[1] = new Kor(1);
      tk[2] = new Teglalap(2,3);
      tk[3] = new Kor(3);
      for( int i = 0; i<tk.length; i++) {
            tk[i].Szamol1();
            tk[i].Szamol2();
            tk[i].Kiir();
            System.out.println();
      }
   }
}
```

A program futásának eredménye:

```
Teglalap oldalai : 1.0 2.0
Teglalap terulete: 2.0
Teglalap kerulete: 6.0

Kor sugara : 1.0
Kor terulete: 3.141592653589793
Kor kerulete: 6.283185307179586

Teglalap oldalai : 2.0 3.0
Teglalap terulete: 6.0
Teglalap kerulete: 10.0

Kor sugara : 3.0
Kor terulete: 28.274333882308138
Kor kerulete: 18.84955592153876
```

8.3 Interfészek (interface)

A objektum-orientált programozás során legtöbbször elegendő az egyszeres öröklés támogatása, vagyis az, hogy minden osztálynak pontosan egy közvetlen őse lehet. A programfejlesztési gyakorlat bizonyította, hogy a többszörös öröklést megengedő nyelveken készített programokban a második, harmadik stb. osztályok általában absztrakt osztályok, melyek a metódusaik megvalósítását írják elő az utód számára.

A Java nyelv fejlesztői a többszörös öröklés bizonyos hiányzó lehetőségeinek pótlására, egy sokkal egyszerűbb és biztonságosabb megoldás, az interfészek (**interface**) használata mellett tették le a voksot. Az interfész sokban emlékeztet az osztályra, azonban csak konstansokat és nyilvános absztrakt metódusokat tartalmazhat. Az interfész fordításakor szintén *.class* állomány keletkezik. Az **abstract** szó használata nem kötelező, hisz maga az interfész, illetve annak minden metódusa hallgatólagosan absztrakt.

Példa az interfészek felépítésére:

```
[public] interface InterfészNév {
            [public] final int konstans = érték;
            [public] int metódus1();
            [public] int metódus2();
}
```

Amennyiben elhagyjuk a **public** módosítókat, csak a csomagban lesz elérhető az interfész.

Az interfészeket az osztálydefiníciókban az **implements** (kivitelez) kulcsszó után adjuk meg:

```
class Osztály [extends ŐsOsztály]
                    [implements Interfész1 [,Interfész2,...]]
{
   // az Osztály tagjai
}
```

A fenti formából jól látszik, hogy az ősosztály nélkül is megvalósíthatunk egy vagy több interfészben kijelölt metódust. Amennyiben valamelyik metódust mégsem készítjük el, akkor az új *Osztályt* **abstract** módosítóval kell ellátni.

A következő kis példa gyakran használt megoldást mutat be, amely látszólag ellentmond az elhangzottaknak.

```
interface Fuggveny {
   public double fv( double x);
}
class Sin implements Fuggveny {
   public double fv(double x) {
        return Math.sin(x);
   }
}
```

A *Fuggveny* interfészből származtatjuk a *Sin* osztályt, amely megvalósítja az *fv()* metódust. Az alábbi osztály első sorában látszólag az interfészt példányosítjuk, azonban ugyanaz zajlik le, mint a *Sin* osztály esetében. Származtatunk egy névtelen osztályt, mellyel azonnal objektumpéldányt hozunk létre, és arra a *cos* névvel hivatkozunk.

```
public class Tablazo
{
   Fuggveny cos = new Fuggveny() {
      public double fv(double x) {
          return Math.cos(x);
      }
   };
```

```
    public void tablaz(Fuggveny f) {
       for (double x=-1; x<1; x+=0.2)
          System.out.println(""+x+"\t"+f.fv(x));
    }
    public void megjelenit() {
       tablaz(cos);
       tablaz(new Sin());
    }
}
```

További érdekessége a megoldásnak, hogy a *tablaz()* metódus egy *Fuggveny* típusú referenciát vár. A metódus hívásakor (a *megjelenit()*-ben) származtatott objektumok referenciáját adjuk át neki argumentumként.

Írjunk programot, amely *SzMK* absztrakt ősosztálytól származtatja *Muvelet* osztályt, és megvalósítja a *HatGyok* interfészt! Az *SzMK* osztály számtani és mértani közép számolására, a *HatGyok* interfész pedig hatványozásra és gyökvonásra használható absztrakt metódust tartalmaz. A *Muvelet* osztály az öröklött műveleteken kívül végezze el az összeadás, a kivonás és a szorzás műveletét! (*Interfesz1*)

Az *SzMK* ősosztály **abstract** osztály, melynek a metódusai szintén absztraktak:

```
// absztrakt osztály: SzMK.java
abstract class SzMK
{
   abstract void SzamtaniKozep();
   abstract void MertaniKozep();
}
```

A nyilvános *HatGyok* interfész **public** hozzáférésű, implicit absztrakt metódusokat tartalmaz:

```
// interface használata: HatGyok.java
public interface HatGyok
{
   public void Hatvanyoz();
   public void Gyokvonas();
}
```

Az interfész használatát bemutató *Muvelet* osztályt, és az alkalmazás futtatásához nélkülözhetetlen ***main()*** metódust tároló, nyilvános *Interfesz1* osztályt az *Interfesz1.java* állomány tartalmazza:

```
// a Muvelet osztály deklarációja
class Muvelet extends SzMK implements HatGyok
{
   double a, b, ered; // a,b műveletek operandusai, ered az eredmény
   char muvjel;       // muvjel a műveleti jelet tartalmazza
```

```
    Muvelet(double a1, double b1, char muvjel1) {
       a = a1; b = b1; muvjel = muvjel1;    // paraméteres konstruktor
    }
    void Szamol() { // műveleti jel alapján elvégzi a műveletet
       switch(muvjel) {
          case '+': ered = a + b; break;
          case '-': ered = a - b; break;
          case '*': ered = a * b; break;
          default: ered = 0;
                   System.out.println("Hibas muveleti jel: " + muvjel);
       }
    }
    public void Hatvanyoz() { // hatványoz
       ered = Math.pow(a,b);
       System.out.println("Hatvanyozas: " + ered);
    }
    public void Gyokvonas() { // gyököt von
       ered = Math.pow(a, 1./b);
       System.out.println("Gyokvonas: " + ered);
    }

    public void SzamtaniKozep() { // számtani közepet számol
       ered = (a + b ) /2;
       System.out.println("Szamtani kozep: " + ered);
    }
    public void MertaniKozep() { // mértani közepet számol
       ered = Math.sqrt(a * b);
       System.out.println("Mertani kozep: " + ered);
    }
    public void Kiir() { // a művelet eredményét jeleníti meg
       System.out.println("Adatok: " + a + " " + b + " " + muvjel);
       System.out.println("Eredmeny: " + ered);
    }
}
```

A *main()* függvény létrehozza a *Muvelet* típusú *x* objektumot, inicializálja azt a konstruktor hívásával, majd pedig aktivizálja az objektum metódusait.

```
public class Interfesz1{
   public static void main(String[] args) {
      Muvelet x = new Muvelet(2,3,'+');
      x.Szamol(); x.Kiir();
      System.out.println();
      x = new Muvelet(4,2,'*');
      x.Szamol(); x.Kiir();
      x.Gyokvonas();
      x.Hatvanyoz();
      x.SzamtaniKozep();
      x.MertaniKozep();
   }
}
```

A program futásának eredménye:

```
Adatok: 2.0 3.0 +
Eredmeny: 5.0

Adatok: 4.0 2.0 *
Eredmeny: 8.0
Gyokvonas: 2.0
Hatvanyozas: 16.0
Szamtani kozep: 3.0
Mertani kozep: 2.8284271247461903
```

Bővítsük az *Interfesz1* programot egy *NegyzetGyoke* interfésszel, amely az adatok négyzetösszegéből von gyököt! (*Interfesz2*)

Az osztálydeklarációban több nyilvános interfészt is megadhatunk, ha nevüket az **implements** kulcsszó után vesszővel elválasztva felsoroljuk:

```
   class Muvelet extends SzMK implements HatGyok, NegyzetekGyoke

// absztrakt osztály: SzMK.java
abstract class SzMK
{
   abstract void SzamtaniKozep();
   abstract void MertaniKozep();
}

// HatGyok interface: HatGyok.java
public interface HatGyok
{
   public void Hatvanyoz();
   public void Gyokvonas();
}

// NegyzetekGyoke interface: NegyzetekGyoke.java
public interface NegyzetekGyoke  // az új osztály az öröklés számára
{
   public void NGyoke();
}

// több interfész használata
class Muvelet extends SzMK implements HatGyok, NegyzetekGyoke
{
   double a, b, ered;
   char muvjel;

   Muvelet(double a1, double b1, char muvjel1) {
      a = a1; b = b1; muvjel = muvjel1;
   }
   . . . // a többi metódus megegyezik az Interfesz1 programbeliekkel
```

```
    public void NGyoke() {      // metódus kidolgozva
        ered = Math.hypot(a,b); // adatok négyzetösszegéből von gyököt
        System.out.println("Negyzetosszegek gyoke: " + ered);
    }
    . . .
}

public class Interfesz2 {
    public static void main(String[] args) {
        Muvelet x = new Muvelet(2,3,'+');
        x.Szamol();
        x.Kiir();
        System.out.println();
        x = new Muvelet(4,2,'*');
        x.Szamol();
        x.Kiir();
        x.Gyokvonas();
        x.Hatvanyoz();
        x.SzamtaniKozep();
        x.MertaniKozep();
        x.NGyoke();
    }
}
```

A program futásának eredménye:

```
Adatok: 2.0 3.0 +
Eredmeny: 5.0

Adatok: 4.0 2.0 *
Eredmeny: 8.0
Gyokvonas: 2.0
Hatvanyozas: 16.0
Szamtani kozep: 3.0
Mertani kozep: 2.8284271247461903
Negyzetosszegek gyoke: 4.47213595499958
```

8.4 Végleges (final) osztályok

A **final** kulcsszót nemcsak konstansok definiálására használjuk, hanem állhat az osztály előtt is:

```
final class Adatok {}
```

Ekkor azt jelöli, hogy az *Adat* osztályból nem származtathatunk újabb osztályokat. A végleges osztályokat ritkán alkalmazzuk, azonban rendelkeznek néhány előnyös tulajdonsággal:

- *Sebesség* – a fordító gyorsabban futó bájtkódot hoz létre, mivel tudja, hogy ebből az osztályból nem származtathatunk másikat.
- *Biztonság*, a *hackerek* nem tudják az öröklést kihasználni, így nem tehetik tönkre a Java programot.

Amennyiben egy osztály valamely metódusának lecserélését szeretnénk megakadályozni a származtatás során, akkor azt a metódust **final**-ként kell deklarálni.

8.5 OOP a gyakorlatban

A könyv eddigi részeiben szereplő kisebb-nagyobb lélegzetvételű példaprogramok mindig csak az éppen bemutatott nyelvi megoldások megértését célozták. A valóságos objektum-orientált fejlesztések során ezeket az eszközöket együtt kell alkalmaznunk, a feladatok sikeres megoldása érdekében. Ezért úgy határoztunk, hogy a könyvünk Java nyelvet ismertető részét, egy nagyobb példaprogrammal zárjuk, melynek elkészítéséhez gyakorlatilag minden eddigi eszközre szükség van.

Fejlesszünk egy ún. „Életjáték" szimulációs programot, ahol a füves mezőn élő nyúl és róka populációk egymásra gyakorolt hatását modellezzük! (*EletJatek*)

Már a legelején megjegyezzük, hogy a megoldásban szereplő konstansok értékét mélyebb biológia ismeretek, tanulmányok nélkül adatuk meg. Itt is inkább a formát, a megvalósítást érdemes tanulmányozni.

A *Jatek* nevű interfészbe gyűjtöttük össze a szükséges konstansokat és a szimulációban szereplő osztályok által kidolgozandó metódusokat.

```
interface Jatek {
   static final int ALLAPOT = 4, ROKAKOR = 2,NYULKOR = 4;
   static final int KIHALT=0, FU=1, NYUL=2, ROKA=3;
   int ki();  // azonosító metódus
   // bizonyos feltételek alapján megadja az adott világpozíció
   // következő állapotát
   Elet kovetkezo(Vilag v);
}
```

Az *Elet* az absztrakt alaposztálya az összes résztvevőnek, a fűnek, a nyúlnak, a rókának és a kihalt állapotnak is. Ugyan implementálja a *Jatek* interfészt, azonban a metódusait nem valósítja meg, így **abstract** módosítóval kell deklarálnunk.

```
abstract class Elet implements Jatek {
   // az világ adott pontjának koordinátái
   protected final int sor, oszlop;

   // az világ objektumban tárolt pontja körüli környezet
   // vizsgálata, melynek alapja, miből hány darab van a pont (0,0)
   // körül (-1,-1)-(1,1)
   protected void ertekel(Vilag v, int ossz[]) {
      ossz[KIHALT] = ossz[FU] = ossz[NYUL] = ossz[ROKA] = 0;
      for (int i=-1;i<=1; ++i)
         for (int j=-1; j<=1; ++j)
            ossz[v.elem(sor+i,oszlop+j).ki()]++;
   }
```

```
    // eltárolja az objektumban az élet végleges helyét a világban
    public Elet(int r, int c) { // konstruktor
      sor=r;
      oszlop=c;
    }
}
```

Az *Elet* osztályból származtatjuk a szimulációban résztvevő elemek osztályait. Minden egyes elem konstruktora az *Elet* konstruktorát hívja, a világ-koordináták átadásával, azonosítja magát a *ki()* metódussal, és *a kovetkezo()* metódussal megvizsgálja, hogy mi lesz a sorsa a világ adott pontján található életnek, mi lesz az új életforma.

```
// Kihalt osztály - amely az élet megszűnését jelöli
class Kihalt extends Elet {
   public Kihalt(int r, int c) {
      super(r,c);
   }

   public int ki() {
      return Elet.KIHALT;
   }

   public Elet kovetkezo(Vilag v) {
      int[] ossz = new int [Elet.ALLAPOT];
      ertekel(v, ossz);
      if (ossz[Elet.ROKA]>1)        // legalább 2 róka szaporodik
         return (new Roka(sor, oszlop));
      else if (ossz[Elet.NYUL]>1)  // legalább 2 nyúl szaporodik
         return (new Nyul(sor, oszlop));
      else if (ossz[Elet.FU]>0)     // legalább 1 fű, az átterjed ide
         return (new Fu(sor, oszlop));
      else
         return (new Kihalt(sor, oszlop));
   }
}
// a róka osztálya
class Roka extends  Elet {
   protected int kor;

   public Roka(int r, int c) {
      this(r,c,0);  // a másik konstruktor hívása
   }

   public Roka(int r, int c, int a) {
      super(r,c);
      kor=a;
   }

   public int ki() {
      return Elet.ROKA;
   }
```

```
    public Elet kovetkezo(Vilag v) {
      int[] ossz = new int [Elet.ALLAPOT];
      ertekel(v, ossz);
      if (ossz[Elet.ROKA]>4)        // ha túl sok a róka, kihalnak
        return (new Kihalt(sor, oszlop));
      else if (kor>Elet.ROKAKOR)    // a túl öreg róka is kihal
        return (new Kihalt(sor, oszlop));
      else                          // a róka öregszik
        return (new Roka(sor, oszlop, kor+1));
    }
}

// a nyúl osztálya
class Nyul extends Elet {
    protected int kor;

    public Nyul(int r, int c) {
      this(r,c,0);  // a másik konstruktor hívása
    }

    public Nyul(int r, int c, int a) {
      super(r,c);
      kor = a;
    }

    public int ki() { return Elet.NYUL; }

    public Elet kovetkezo(Vilag v) {
      int[] ossz = new int [Elet.ALLAPOT];
      ertekel(v,ossz);
      if (ossz[Elet.ROKA]>=ossz[Elet.NYUL]) //a róka megeszi a nyulat
        return (new Kihalt(sor, oszlop));
      else if (kor>Elet.NYULKOR)            // a nyúl túl öreg
        return (new Kihalt(sor, oszlop));
      else                                  // a nyúl öregszik
        return (new Nyul(sor, oszlop, kor+1));
    }
}

// a fű osztálya
class Fu extends Elet {
    public Fu(int r, int c) {
      super(r,c);
    }

    public int ki() { return Elet.FU; }

    public Elet kovetkezo(Vilag v) {
      int[] ossz = new int [Elet.ALLAPOT];
      ertekel(v,ossz);
      if (ossz[Elet.FU]>ossz[Elet.NYUL])    // több a fű, mint a nyúl
        return (new Fu(sor, oszlop));
      else
        return (new Kihalt(sor, oszlop));
    }
}
```

Az élet szimulációját a *Vilag* osztály vezérli, melynek adott méretű kétdimenziós *Elet* típusú *vilag* tömbjének elemei az egyes létformákra hivatkoznak. A *vilag* tömb szélső elemeit nem használjuk a szimulációhoz, hisz ezzel túl bonyolulttá válna a környezet vizsgálata. A *Vilag* osztály metódusainak leírását táblázatban foglaltuk össze:

metódus	*leírás*
konstruktor	létrehozza a kihalt világot,
elem()	biztosítja a világ pontjaiban az élet elérését,
frissites()	lépteti a szimulációt, a régi világból létrehozza az újat,
kezdet()	a világ véletlenszerű feltöltése a létformákkal,
megjelenit()	megjeleníti a világot,

```
// Az életszimulációt vezérlő Vilag osztály
class Vilag
{
   private Elet [][] vilag;
   private final int meret;

   // a konstruktorban létrejön a kihalt világ
   public Vilag(int meret) {
      this.meret = meret;
      vilag = new Elet[meret][meret];
      for (int i=0; i<meret; ++i)
         for (int j=0; j<meret; ++j)
            vilag[i][j]=new Kihalt(i,j);
   }

   // a vilag[][] elemeinek elérési metódusa
   public final Elet elem(int i, int j) {
      return vilag[i][j];
   }

   // az új világ előállítása a régiből
   public void frissites(Vilag v_regi) {
      for (int i=1; i<meret-1; ++i)
         for (int j=1; j<meret-1; ++j)
            vilag[i][j] = v_regi.elem(i,j).kovetkezo(v_regi);
   }

   // a kiindulási világ véletlenszerű létrehozása
   public void kezdet() {
      int allapot;
      for (int i=1; i<meret-1; ++i)
         for (int j=1; j<meret-1; ++j) {
            allapot = (int)(Math.random()*Elet.ALLAPOT);
            switch (allapot) {
               case Elet.KIHALT: vilag[i][j] = new Kihalt(i,j); break;
               case Elet.FU:     vilag[i][j] = new Fu(i,j);     break;
               case Elet.NYUL:   vilag[i][j] = new Nyul(i,j);   break;
               case Elet.ROKA:   vilag[i][j] = new Roka(i,j);   break;
            }
         }
   }
```

```
    // a világ megjelenítése
    public void megjelenit() throws IOException
    {
       System.out.println("\n - . - . - ");
       for (int i=1; i<meret-1; ++i) {
          for (int j=1; j<meret-1; ++j)
             switch (vilag[i][j].ki())
             {
                case Elet.FU:     System.out.print("  fu "); break;
                case Elet.KIHALT: System.out.print(".... "); break;
                case Elet.NYUL:   System.out.print("nyul "); break;
                case Elet.ROKA:   System.out.print("roka "); break;
             }
          System.out.println();
       }
       System.in.read();
    }
}
```

Az *EletJatek* osztály ***main()*** metódusa indítja és futtatja a szimulációt. A futtatáshoz két világot hozunk létre azonos mérettel (*vilag0* és *vilag1*). A *vilag0* világot inicializáljuk és megjelenítjük. Ezt követően előállítjuk belőle a *vilag1* világot, és megjelenítjük azt. Majd ebből állítjuk elő újra a vilag0-ban az új világot, és így tovább, felváltva adott lépésszámban.

```
    // az alkalmazás indítása és futtatása
public class EletJatek
{
    public static void main(String[] args) throws IOException
    {
       final int MERET = 12, LEPES = 5;
       Vilag vilag0 = new Vilag(MERET); // a  régi/új világ
       Vilag vilag1 = new Vilag(MERET); // az új  /régi világ
       vilag0.kezdet();
       vilag0.megjelenit();
       for (int i=0; i<LEPES; i++){
          if (i%2!=0) {
             vilag0.frissites(vilag1);
             vilag0.megjelenit();
          }
          else {
             vilag1.frissites(vilag0);
             vilag1.megjelenit();
          }
       }
    }
}
```

A program futásának eredménye:

A kiindulási világ:

```
roka roka   fu   fu nyul nyul roka roka roka roka
roka roka   fu nyul nyul nyul   fu   fu   fu roka
  fu   fu roka   fu   fu roka   fu   fu   fu   fu
  fu nyul   fu   fu roka roka .... nyul nyul   fu
nyul nyul   fu nyul .... .... roka nyul nyul ....
.... roka .... nyul nyul roka nyul nyul roka roka
roka nyul nyul nyul roka roka nyul nyul .... roka
.... nyul .... roka roka .... nyul nyul .... roka
nyul .... nyul nyul roka   fu nyul roka nyul roka
roka nyul nyul nyul nyul   fu   fu   fu .... nyul
```

A világ egy generáció (lépés) után:

```
roka roka   fu .... nyul nyul roka roka roka roka
roka ....   fu nyul nyul nyul   fu   fu   fu roka
  fu   fu roka   fu .... roka   fu   fu   fu   fu
.... nyul   fu   fu roka roka roka nyul nyul   fu
nyul nyul .... nyul roka roka roka nyul nyul roka
roka roka nyul nyul nyul roka nyul nyul roka roka
roka nyul nyul nyul .... roka nyul nyul roka roka
nyul nyul nyul roka .... roka nyul nyul roka roka
nyul nyul nyul nyul roka .... nyul roka .... roka
roka nyul nyul nyul nyul   fu   fu .... roka nyul
```

9. A Java 2 API általános osztályai

A Java nyelv lehetőségeinek áttekintése után érdemes megismerkedni *J2SE API* néhány, gyakran használt osztályával. Természetesen a példaprogramokban már többel is találkoztunk valamilyen feladat végrehajtása kapcsán. Most azonban maguk az osztályok képezik a tanulmányozásunk tárgyát.

Alapvetően három csomag osztályaiból válogattunk: a *java.lang*, *java.io* és *java.util*. A *java.lang* kivételével a felhasználni kívánt csomag osztályait importálnunk kell (**import**).

A Java nyelvben minden osztálynak közös őse a *java.lang.Object* osztály. Az *Object* osztály metódusait (lásd F1. függelék) minden osztály örökli. Például a *Class* típusú *getClass()* metódussal lekérdezhetjünk az objektum osztályát. A *java.lang* csomag *Boolean*, *Character*, *Number*, *Math*, *System*, *String* és *StringBuffer* osztályainak az *Object* a közvetlen őse, míg a *Byte*, *Short*, *Integer*, *Long*, *Float* és *Double* osztályok a *Number*-tól származnak. (A *Math* és a *System* **final** osztályok, ezért nem örökíthetők.)

A *java.io* csomag osztályai már eddig is segítették a munkánkat (*BufferedReader*, *InputStreamReader*), most azonban a fájlkezelés lehetőségeit tekintjük át: *FileReader*, *FileWriter*, *RandomAccessFile*, *FileOutputStream*, *FileInputStream* stb.

A fejezet végén bemutatjuk a *java.util* csomag néhány osztályát, amelyek többsége a *Java 2 SE 5.0*-tól bevezetett sablonosztály: *Vector*, *Hashtable*, *Stack*, *LinkedList*, *PriorityQueue*, *Properties*, *BitSet* stb. Ezeket a példaprogramokat korábbi Java-fordítókkal csak kis korrekció után lehet lefordítani.

9.1 Primitív típusú adatok objektumokként való kezelése

A primitív típusú változók nem objektumok, de a *java.lang* csomagban minden típushoz egy osztály tartozik:

Character, *Boolean*, *Byte*, *Short*, *Integer*, *Long*, *Float*, *Double*

Nézzünk néhány példát az osztályok alkalmazására!

```
Character kar_obj = new Character('c');
Integer egesz_obj = new Integer(13);
Double valos_obj  = new Double(12.56);
```

A numerikus értékek szöveggé alakítására a *toString()* metódust használhatjuk.

Írjunk programot, amely bemutatja a primitív típusok osztályainak használatát karakter, egész és valós értékek esetén! (*TipusOsztalyok\PrimTipO1*)

```
public class PrimTipO1
{
        public static void main(String[] args) {
            Character kar_obj = new Character('c');
            Integer egesz_obj = new Integer(13);
            Double valos_obj = new Double(12.56);
            String kar_str, egesz_str, valos_str;
            int egesz;
            double valos;
            kar_str = kar_obj.toString();
            egesz_str = egesz_obj.toString();
            valos_str = valos_obj.toString();
            System.out.println("kar_str  : " + kar_str);
            System.out.println("egesz_str: " + egesz_str);
            System.out.println("valos_str: " + valos_str);
            egesz = egesz_obj.intValue();
            valos = valos_obj.doubleValue();
        }
}
```

A program futásának eredménye:

```
kar_str  : c
egesz_str: 13
valos_str: 12.56
```

A ***Character*** osztály **boolean** típusú, statikus metódusaival különböző vizsgálatokat végezhetünk. Minden metódus egy **char** típusú karaktert fogad paraméterként.

metódus	*a vizsgálat tárgya*
`isDigit()`	decimális számjegy-e,
`isLetter()`	betű-e,
`isLowerCase()`	kisbetű-e,
`isUpperCase()`	nagybetű-e,
`isSpaceChar()`	szóköz-e,
`isWhitespace()`	elválasztó karakter-e.

Készítsünk programot, amely bemutatja a ***Character*** osztály metódusainak használatát! (*TipusOsztalyok\PrimTipO2*)

```
public class PrimTipO2 {
        public static void main(String[] args) {
            char ch = 'B';
            if (Character.isLowerCase(ch))
                System.out.println(ch + " - kisbetu");
            if (Character.isUpperCase(ch))
                System.out.println(ch + " - nagybetu");
```

```
            ch = '3';
            if (Character.isLetter(ch))
                System.out.println(ch + " - karakter");
            if (Character.isDigit(ch))
                System.out.println(ch + " - szam");
            ch = ' ';
            if (Character.isSpaceChar(ch))
                System.out.println(ch + " - szokoz");
            if (Character.isWhitespace(ch))
                System.out.println(ch + " - tagolo karakter");
        }
}
```

A program futásának eredménye:

```
B - nagybetu
3 - szam
  - szokoz
  - tagolo karakter
```

Írjunk alkalmazást, amely bemutatja a válaszként leütött karakter kezelését! (*TipusOsztalyok\Valasz*)

```
import java.io.*;
public class Valasz
{
  public static char folytatas() throws IOException  {
    System.out.println("Folytatja ?: i/n");
    char v;
    do {
      v = Character.toUpperCase((char)System.in.read());
      if (!Character.isLetter(v))
         continue;
      else if (v=='N' || v=='I')
         break;
      else
         System.out.println("Az <I> vagy az <N> billentyut nyomja le");
    } while (true);
    return v;
  }

  public static void main (String[] args) {
   try {
      if (folytatas()=='N')
         System.exit(-1);
      else
         System.out.println("Folytatom");
      System.in.read();
   }
   catch(IOException e) {
      System.out.println("IO hiba");
   }
  }
}
```

A program futásának eredménye:

```
Folytatja ?: i/n
i
Folytatom
```

A numerikus típusok osztályai egy sor kellemes megoldást biztosítanak számunkra. Ezek egy része statikus tagokkal érhető el, más részéhez pedig új objektumot kell létrehoznunk. A konstruktorok a megfelelő primitív típusú adatból, vagy sztringből is előállíthatják az objektumpéldányt.

Megtudhatjuk az adott primitív típus értékkészletének határait (*MIN_VALUE*, *MAX_VALUE*), szövegben tárolt számot különböző típusú numerikus adattá alakíthatjuk, de akár – egészek esetén – más számrendszerbe is átválthatjuk a számot.

Készítsünk programot, amely az ***Integer*** osztály példáján bemutatja a numerikus osztályok felhasználási lehetőségeit! (*TipusOsztalyok\PrimTipO3*)

```
public class PrimTipO3
{
    public static void main(String[] args) {
      // Statikus tagok használata:
      System.out.println("A legkisebb int  : " +Integer.MIN_VALUE);
      System.out.println("A legnagyobb int : " +Integer.MAX_VALUE);

      // Decimális -> más számrendszer sztringben
      int a = 1223;
      String s = Integer.toString(a, 16).toUpperCase();
      System.out.println(""+a+" 16-os rendszerben : " + s);
      s = Integer.toString(a, 2).toUpperCase();
      System.out.println(""+a+" 2-es rendszerben  : " + s);
      System.out.println(""+a+" 1-es bitjeinek szama: " +
                         Integer.bitCount(a));
      s = Integer.toString(a, 23).toUpperCase();
      System.out.println(""+a+" 23-as rendszerben : " + s);

      System.out.print(""+a);
      switch (Integer.signum(a)) {
          case -1: System.out.println(" negativ"); break ;
          case  0: System.out.println(" nulla"); break ;
          case +1: System.out.println(" pozitiv"); break ;
       }

       // String -> numerikus adat
       a  = Integer.parseInt("735"+"123");
       System.out.println("String -> int: "+a);

       // Objektumpéldány használatával:
       Integer ao = new Integer("781223");
       System.out.println(""+ao.longValue());
       System.out.println(""+ao.doubleValue());
    }
}
```

A program futásának eredménye:

```
A legkisebb int  : -2147483648
A legnagyobb int : 2147483647
1223 16-os rendszerben : 4C7
1223 2-es rendszerben  : 10011000111
1223 1-es bitjeinek szama: 6
1223 23-as rendszerben : 274
1223 pozitiv
String -> int: 735123
781223
781223.0
```

9.2 Szövegek kezelése

A ***java.lang*** csomag tartalmazza a ***String*** és a ***StringBuffer*** osztályokat, melyek szövegek tárolására alkalmasak. A ***String*** típusú objektumban tárolt szöveg nem változtatható meg, ellentétben a ***StringBuffer*** típusú objektummal, amelyben lévő szöveg módosítható.

9.2.1 A String osztály

A ***String*** osztály *unicode* kódolású szöveg tárolására szolgál. Az objektum szövege azonban nem változtatható meg. Talán érdemes egy pillanatra elidőzni ezen a ponton:

```
String s = "ComputerBooks";
s = "Együtt könnyebb a programozás - Java" ;
```

Első látásra úgy tűnik, hogy megváltoztattuk az *s* ***String*** tartalmát. Valójában azonban eldobtuk a *s* referencia által hivatkozott objektumot, és a második sor után egy új ***String*** objektumra hivatkozik az *s*. A szöveg karakterei sorszámozottak. Az első karakter sorszáma *0*, ha a karakterek száma *N*, akkor az utolsó karakter sorszáma *N-1*.

A ***String*** típusú objektumot többféle módon is létrehozhatunk:

1. A **new** operátorral:
   ```
   String fejlec = new String("A feladat megoldása");
   ```
2. Létrehozhatjuk, mint stringliterál (szövegkonstans), de az előbbi megoldás biztonságosabb:
   ```
   String fejlec = "a feladat megoldása";
   ```
3. Előállíthatjuk karaktertömbből is:
   ```
   char [] ct = {'N','a','t','a',};
   String s = new String(ct);
   ```

A ***String*** osztály sokféle és változatos metódust biztosít a karaktersorozatok kezelésére. Röviden összefoglaltunk néhányat közülük:

- A szövegeket a + műveleti jellel fűzhetjük össze:
  ```
  teljes_szoveg = szoveg1 + szoveg2 + "!";
  ```

- `String concat(String str)` – az objektumban tárolt szöveghez hozzáfűzi az argumentumként megadott szöveget, és ez lesz a visszatérési értéke.

```
tszoveg2 = tszoveg1.concat(" megoldása rendben.");
```

- **`int length()`** – a szöveg hossza

```
hossz = teljes_szoveg.length();
```

- `String valueOf(adattípus)` – az adattípust szöveggé alakítja

```
ValósStr = String.valueOf(ValósAdat);
EgészStr = String.valueOf(EgészAadat);
```

- Sztringet alakít
  - valós számmá:

```
ValósAdat = Double.valueOf(ValósStr).doubleValue();
```

  - egész számmá:

```
EgészAdat = Integer.valueOf(EgészStr).intValue();
```

  - karaktertömbbé:

```
char [] ct = teljes_szoveg.toCharArray();
```

Készítsünk alkalmazást, amely bemutatja a szövegek összefűzését, valamint szövegként megadott adatok számmá, illetve számadatok szöveggé való átalakítását! (*Szovegkezeles\String_pr1*)

```
public class String_pr1
{
      public static void main(String[] args) {
         String szoveg1 = "Minta ";
         String szoveg2 = "feladat";
         String tszoveg1, tszoveg2;
         String Valos1 = "12.5", Valos2;
         String Egesz1 = "46", Egesz2;
         double Vadat1, Vadat2 = 32.56;
         int Eadat1, Eadat2 = 13;
         int hossz;
         tszoveg1 = szoveg1 + szoveg2 + ":";
         hossz = tszoveg1.length();
         System.out.println("tszoveg1: " + tszoveg1);
         System.out.println("tszoveg1 hossza: " + hossz);
         tszoveg2 = tszoveg1.concat(" megoldasa rendben.");
         System.out.println("tszoveg2: " + tszoveg2);

         Vadat1 = Double.valueOf(Valos1).doubleValue();
         Eadat1 = Integer.valueOf(Egesz1).intValue();

         Valos2 = String.valueOf(Vadat2);
         Egesz2 = String.valueOf(Eadat2);

         System.out.println("\nSzoveg szamma alakitva: ");
         System.out.println("Vadat1: " + Vadat1);
         System.out.println("Eadat1: " + Eadat1);
```

```
            System.out.println("\nSzam szovegge alakitva: ");
            System.out.println("Valos2: " + Valos2);
            System.out.println("Egesz2: " + Egesz2);
        }
}
```

A program futásának eredménye:

```
tszoveg1: Minta feladat:
tszoveg1 hossza: 14
tszoveg2: Minta feladat: megoldasa rendben.
Szoveg szamma alakitva:
Vadat1: 12.5
Eadat1: 46
Szam szovegge alakitva:
Valos2: 32.56
Egesz2: 13
```

Sztringek egyezőségének vizsgálata:

- A == operátorral csak az objektumhivatkozásokat hasonlítjuk össze. Csak akkor kapunk igaz értéket, ha mindkét referencia ugyanarra a String típusú objektumra hivatkozik. Felhívjuk a figyelmet, hogy a Java-fordító az azonos szövegkonstansokat egyetlen objektumban tárolja. Ezzel magyarázható az alábbi példa:

```
String a1 = new String("Alma"), a2 = new String("Alma");
System.out.println((a1 == a2));  // false
String b1 = "Alma", b2 = "Alma";
System.out.println((b1 == b2));  // true
System.out.println((a1 == b1));  // false
```

- **boolean** `equals(Object obj)` – ha az aktuális (*szoveg1*) és az argumentumban szereplő (*szoveg2*) sztringobjektumokban tárolt szövegek karakterei megegyeznek, akkor a visszatérési érték **true**, különben pedig **false**.

```
if (szoveg1.equals(szoveg2))
```

- **boolean** `equalsIgnoreCase(String str)` – a tárolt karaktersorozatot összehasonlítja az argumentumként megadott *str* tartalmával, nem megkülönböztetve a kis- és a nagybetűket.

```
if (szoveg1.equalsIgnoreCase(szoveg2))
```

Írjunk programot, amely bemutatja a szövegek összehasonlítására szolgáló ***equals()*** metódus, valamint a == operátor használatát! (*Szovegkezeles\String _pr2*)

```
public class String_pr2 {
      public static void main(String[] args) {
        String szoveg1 = new String("Minta");
        String szoveg2 = "Minta";
        if (szoveg1 == szoveg2)
              System.out.println("az objektumok azonosak");
        else
              System.out.println("az objektumok kulonbozoek");
```

```
        if (szoveg1.equals(szoveg2))
              System.out.println(szoveg1 + " egyezik " + szoveg2);
        else
              System.out.println(szoveg1 + " nem egyezik " + szoveg2);
        if ("Minta".equals(szoveg1))
              System.out.println("Minta" + " egyezik " + szoveg1);
        else
              System.out.println("Minta" + " nem egyezik " + szoveg1);
        if ("MiNTa". equalsIgnoreCase (szoveg1))
              System.out.println("MiNTa" + " egyezik " + szoveg1);
        else
              System.out.println("MiNTa" + " nem egyezik " + szoveg1);
    }
}
```

A program futásának eredménye:

```
az objektumok kulonbozoek
Minta egyezik Minta
Minta egyezik Minta
MiNTa egyezik Minta
```

Szöveg átalakítása, karaktereinek lekérdezése:

- **char** charAt(**int** index) – az index pozícióban lévő karakter olvasása,

```
char c = szoveg.charAt(4);
```

- String toUpperCase() – a szöveget nagybetűssé alakítása,

```
String Nszoveg = szoveg.toUpperCase();
```

- String toLowerCase() – a szöveget kisbetűssé alakítása.

```
String Kszoveg = szoveg.toLowerCase();
```

Készítsünk alkalmazást, amely bemutatja a ***String*** típusú objektumban tárolt szöveg kis- és nagybetűssé alakítását, valamint adott pozíciójában lévő karakter lekérdezését! (*Szovegkezeles\String_pr3*)

```
public class String_pr3 {
      public static void main(String[] args) {
          String szoveg1 = "Feladat";
          String Nszoveg, Kszoveg;
          char c3;
          System.out.println("Mintaszoveg: "+ szoveg1);
          c3 = szoveg1.charAt(4);
          System.out.println("4." + "karakter : " + c3);
          Nszoveg = szoveg1.toUpperCase();
          Kszoveg = szoveg1.toLowerCase();
          System.out.println("Nagybetus  : "+ Nszoveg);
          System.out.println("Kisbetus   : "+ Kszoveg);
      }
}
```

A program futásának eredménye:

```
Mintaszoveg: Feladat
4.karakter : d
Nagybetus  : FELADAT
Kisbetus   : feladat
```

Bizonyos metódusok a ***String*** típusú objektumban tárolt szövegről másolatot készítenek, azon elvégzik a kívánt műveletet, majd pedig visszaadják függvényértékként az eredményt:

- `String trim()` – visszaadja a sztring tartalmát, az elején és a végén lévő szóközök nélkül,

```
String tszoveg = szoveg.trim();
```

- `String replace (char régikarakter, char újkarakter)` – visszaadott sztringben minden *régikarakter* helyett az *újkarakter* szerepel.

```
String csszoveg = szoveg.replace('a','A');
```

- `String substring(int kezdő_index)` – a megadott pozíciótól a karaktersorozat végéig visszaadja a részsztringet,

```
String rszoveg = szoveg.substring(6);
```

- `String substring(int kezdő_index, int vég_index)` – a megadott pozíciók által kijelölt részsztringgel tér vissza.

```
String rszoveg = szoveg.substring(6);
```

Készítsünk alkalmazást, amely bemutatja a szöveg elején és a végén található szóközök törlését, karakterek cseréjét, valamint részsztring lekérdezését!
(*Szovegkezeles\String_pr4*)

```
public class String_pr4 {
        public static void main(String[] args) {
            String szoveg1 = "  almafa  ";
            String szoveg2 = "szalmakalap";
            String szoveg, csszoveg, rszoveg;
            szoveg = szoveg1.trim();
            System.out.println("-" + szoveg1 + "-");
            System.out.println("-" + szoveg + "-");
            csszoveg = szoveg1.replace('a','A');
            System.out.println(szoveg1 + " csere utan " + csszoveg);
            rszoveg = szoveg2.substring(6);
            System.out.println(szoveg2 + " 6. karaktertol: " + rszoveg);
        }
}
```

A program futásának eredménye:

```
-  almafa  -
-almafa-
  almafa   csere utan   AlmAfA
szalmakalap 6. karaktertol: kalap
```

A karaktersorozatokat a más nyelvekben használt lexikografikus (karakterenkénti) módszerrel is összehasonlíthatjuk.

- `int compareTo(String str)` – összehasonlítja az aktuális és a paraméterként kapott sztringek tartalmát, figyelembe véve a kis- és a nagybetűk közötti különbségeket. Negatív számmal tér vissza, ha az aktuális szöveg lexikografikusan kisebb a megadottnál, nullát ad, ha azonosak, és pozitív a metódus értéke, ha az argumentumsztring a nagyobb.

```
if ((k = szoveg0.compareTo(szoveg1)) == 0)
```

- `int compareToIgnoreCase(String str)` – a működése az előzőtől csak annyiban tér el, hogy nem tesz különbséget a kis- és a nagybetűk között.

```
if ((k = szoveg2.compareToIgnoreCase(szoveg3)) == 0)
```

Írjunk programot, amely bemutatja a szövegek összehasonlítására szolgáló metódusok használatát! (*Szovegkezeles\String_pr5*)

```
public class String_pr5 {
    public static void main(String[] args) {
        String szoveg0 = "Alma", szoveg1 = "alma";
        String szoveg2 = "almA", szoveg3 = "korte";
        String szoveg4 = "narancs"; int k;
        System.out.println(szoveg0 + " - " + szoveg1);
        if ((k = szoveg0.compareTo(szoveg1)) == 0)
            System.out.println(szoveg0 + " azonos " + szoveg1);
        else if (k < 0)
            System.out.println(szoveg0 + " kisebb, mint "
                               + szoveg1);
        else
            System.out.println(szoveg0 + " nagyobb, mint "
                               + szoveg1);
        System.out.println(szoveg2 + " - " + szoveg3);
        if ((k = szoveg2.compareToIgnoreCase(szoveg3)) == 0)
            System.out.println(szoveg2 + " azonos " + szoveg3);
        else if (k < 0)
            System.out.println(szoveg2 + " kisebb, mint "
                               + szoveg3);
        else
            System.out.println(szoveg2 + " nagyobb, mint "
                               + szoveg3);
        System.out.println(szoveg4 + " - " + szoveg2);
        if ((k = szoveg4.compareToIgnoreCase(szoveg2)) == 0)
            System.out.println(szoveg4 + " azonos " + szoveg2);
        else if (k < 0)
            System.out.println(szoveg4 + " kisebb, mint "
                               + szoveg2);
        else
            System.out.println(szoveg4 + " nagyobb, mint "
                               + szoveg2);
    }
}
```

A program futásának eredménye:

```
Alma - alma
Alma kisebb, mint alma
almA - korte
almA kisebb, mint korte
narancs - almA
narancs nagyobb, mint almA
```

Karakter és szöveg keresése a tárolt karaktersorozatban:

- `int indexOf(int kar)`
- `int indexOf(int kar, int index_honnan)`
- `int indexOf(String str)`
- `int indexOf(String str, int index_honnan)`

A felsorolt metódusokkal az aktuális szövegben karaktert (*kar*) vagy szöveget (*str*) kereshetünk elölről, vagy a megadott pozíciótól (*index_honnan*). A visszaadott érték a keresett karakter, vagy szövegrészlet első előfordulásának pozíciója, illetve –1 sikertelen esetben.

Készítsünk alkalmazást, amely bemutatja egy karakter első előfordulási helyének megkeresését a sztringben! (*Szovegkezeles\String_pr6*)

```
public class String_pr6 {
     public static void main(String[] args) {
       String szoveg1 = "Erik a Barack.";
       System.out.println(szoveg1 + " a indexe: "
                          + szoveg1.indexOf('a'));
       System.out.println(szoveg1 + " B indexe: "
                          + szoveg1.indexOf('B'));
       }
 }
```

A program futásának eredménye:

```
Erik a Barack. a indexe: 5
Erik a Barack. B indexe: 7
```

9.2.2 A StringBuffer osztály

Mint már korábban említettük, hogy a ***String*** osztály objektumában tárolt szöveg nem változtatható meg, azonban a ***StringBuffer*** osztály objektumainak szövege módosítható: kitörölhetünk belőle, beszúrhatunk szövegrészletet, és a szöveg hossza is változhat. A kapacitása is lekérdezhető, ami meghatározza, hogy milyen hosszúságú szöveg tárolható benne.

A ***StringBuffer*** osztályt biztonságosan lehet használni többszálú programokban is, emiatt azonban nem eléggé hatékony az egyszálú alkalmazásokban. Ezért a *Java API*

biztosít egy másikosztályt is a ***StringBuffer*** feladatainak elvégzésére, ***StringBuilder*** néven. A ***StringBuilder*** egyszálú alkalmazások gyorsabb működését eredményezi.

A ***StringBuffer*** típusú objektumok létrehozása:

- `StringBuffer()` – üres, 16 karakter hosszú sztringpuffert hoz létre,
  ```
  StringBuffer ures1 = new StringBuffer();
  ```
- `StringBuffer(int hossz)` – adott hosszúságú sztringpuffert hoz létre.
  ```
  StringBuffer ures2 = new StringBuffer(30);
  ```
- `StringBuffer(String str)` – az argumentumban megadott sztringet tartalmazó sztringpuffer jön létre:
  ```
  StringBuffer szoveg = new StringBuffer("almafa");
  ```
- `int capacity()` – megadja az sztringpuffer kapacitását.
  ```
  kapacitás = szoveg.capacity();
  ```
- `int length()` – megadja az sztringpufferben tárolt szöveg aktuális hosszát.
  ```
  hossz = szoveg.length();
  ```

Írjunk programot, amely bemutatja a ***StringBuffer*** típusú objektum létrehozását, kapacitásának és a benne tárolt szöveg hosszának lekérdezését! (*Szovegkezeles \StringB_pr1*)

```
public class StringB_pr1
{
      public static void main(String[] args) {
        int hossz1, hossz2, hossz3;
        int kapacitas1, kapacitas2, kapacitas3;
        StringBuffer szoveg = new StringBuffer("almafa");
        hossz1 = szoveg.length();
        kapacitas1 = szoveg.capacity();
        System.out.print(szoveg);
        System.out.println("szoveg hossza: " + hossz1);
        System.out.println("szoveg kapacitasa: " + kapacitas1);

        StringBuffer ures1 = new StringBuffer();
        hossz2 = ures1.length();
        kapacitas2 = ures1.capacity();
        System.out.println("ures1 hossza: " + hossz2);
        System.out.println("ures1 kapacitasa: " + kapacitas2);

        StringBuffer ures2 = new StringBuffer(30);
        hossz3 = ures2.length();
        kapacitas3 = ures2.capacity();
        System.out.println("ures2 hossza: " + hossz3);
        System.out.println("ures2 kapacitasa: " + kapacitas3);

      }
}
```

A program futásának eredménye:

```
szoveg tartalma: almafa
szoveg hossza: 6
szoveg kapacitasa: 22
ures1 hossza: 0
ures1 kapacitasa: 16
ures2 hossza: 0
ures2 kapacitasa: 30
```

Készítsünk alkalmazást, amely bemutatja a szöveg hosszának beállítását, és az adott sorszámú karakter lekérdezését! (*Szovegkezeles\StringB_pr2*)

- **void** `setLength(int ujhossz)` – beállítja a szöveg hosszát, ha kisebb, mint az aktuális, akkor levágja a szöveg végét, ha nagyobb, 0 karakterekkel tölti fel az üres pozíciókat,
- **char** `charAt(int index)` – a szöveg *index* sorszámú karakterét adja vissza.

```
public class StringB_pr2 {
        public static void main(String[] args) {
            StringBuffer szoveg = new StringBuffer("almafa");
            System.out.println(szoveg);
            int hossz = szoveg.length();
            System.out.println(szoveg + " szoveg hossza: " + hossz);
            szoveg.setLength(4);
            int hossz1 = szoveg.length();
            System.out.println(szoveg + " szoveg hossza1: " + hossz1);
            char c = szoveg.charAt(2);
            System.out.println("2. karakter: " + c);
        }
}
```

A program futásának eredménye:

```
almafa
almafa szoveg hossza: 6
alma szoveg hossza1: 4
2. karakter: m
```

Írjunk alkalmazást, amely bemutatja a szövegből való törlés módjait! (*Szovegkezeles\StringB_pr3*)

- `StringBuffer delete(int honnan, int meddig)` – Törlés a szövegből a *honnan* pozíciótól a *meddig*-1 pozícióig - a visszatérési érték a megváltozott objektum referenciája,

```
szoveg.delete(4,6);
```

- `stringBuffer deleteCharAt(int hol)` – Törli a szövegből a *hol* pozícióban található karaktert, és a megváltozott objektummal tér vissza (a szöveg hossza eggyel csökken).

```
szoveg.deleteCharAt(1);
```

```
public class StringB_pr3{
        public static void main(String[] args) {
            StringBuffer szoveg = new StringBuffer("almafa");
            System.out.print(szoveg);
            szoveg.delete(4,6);
            System.out.println(" torles utan: " + szoveg);
            szoveg.deleteCharAt(1);
            System.out.println(" torles utan: " + szoveg);
        }
 }
```

A program futásának eredménye:

```
almafa torles utan: alma
 torles utan: ama
```

Készítsünk programot, amely bemutatja a szöveg végéhez való hozzáfűzést és a szövegbe való beszúrást! (*Szovegkezeles\StringB_pr4*)

- `StringBuffer append(Object obj)`
- `StringBuffer append(típus érték)`

Osztály, vagy tetszőleges primitív típus szöveggé alakítva hozzáadódik a tárolt szöveg végéhez.

```
szoveg.append(" most virágzik.");
```

- `StringBuffer insert(int pozíció, Object obj)`
- `StringBuffer insert(int pozíció, típus érték)`

Osztály, vagy tetszőleges primitív típus szöveggé alakítva beékelődik szövegbe, a megadott pozíciótól kezdve.

```
szoveg.insert(13,ar);
```

```
public class StringB_pr4{
        public static void main(String[] args) {
            StringBuffer szoveg1 = new StringBuffer("Almafa");
            System.out.print(szoveg1);
            szoveg1.append(" most viragzik.");
            System.out.println(" append utan: " + szoveg1);
            StringBuffer szoveg2 = new StringBuffer("Alma kiloja:  Ft");
            int ar = 120;
            System.out.println("szoveg2: " + szoveg2);
            szoveg2.insert(13,ar);
            System.out.println("insert utan: " + szoveg2);
        }
 }
```

A program futásának eredménye:

```
Almafa append utan: Almafa most viragzik.
szoveg2: Alma kiloja:  Ft
insert utan: Alma kiloja: 120 Ft
```

Írjunk alkalmazást, amely bemutatja szövegrész más szövegre való cseréjét, és a szöveg megfordítását! (*Szovegkezeles\StringB_pr5*)

- `StringBuffer replace(int kezdő, int vég, String str)` – a *kezdő* és a *vég*-1 pozíciók közötti szövegrészt kicseréli az *str* sztringben tárolt szövegre,
  `szoveg.replace(0,4,"Körte");`
- `StringBuffer reverse()` – megfordítja a szöveget.
  `szoveg.reverse();`

```
public class StringB_pr5{
       public static void main(String[] args) {
           StringBuffer szoveg = new StringBuffer("Almafa viragzik.");
           System.out.print(szoveg);
           szoveg.replace(0,4,"Korte");
           System.out.println(" replace: " + szoveg);
           szoveg.reverse();
           System.out.println(" reverse: " + szoveg);
       }
 }
```

A program futásának eredménye:

```
Almafa viragzik. replace: Kortefa viragzik.
 reverse: .kizgariv afetroK
```

Készítsünk alkalmazást, amely bemutatja a ***substring()*** metódus használatát! (*Szovegkezeles\StringB_pr6*)

- `String substring(int kezdő, int vég)` - a *kezdő* és *vég*-1 közötti szövegrészt új sztringben adja vissza.
- `String substring(int kezdő)` - a kezdőpozíciótól a szöveg végéig terjedő szövegrészt egy új sztringben adja vissza,
- `String toString()` – sztringben adja vissza az objektum szövegét.

```
public class StringB_pr6{
       public static void main(String[] args) {
             StringBuffer szoveg = new
                                StringBuffer("Almafa viragzik.");
             String resz;
             System.out.print(szoveg);
             resz = szoveg.substring(0,4);
             System.out.println(" subString: " + resz);
             String sz = resz.toString();
             System.out.println("String tipus: " + sz);
       }
 }
```

A program futásának eredménye:

```
Almafa viragzik. subString: Alma
String tipus: Alma
```

Tervezzünk alkalmazást, amely bemutatja a ***StringBuffer*** típusú metódusparaméter használatát! (*Szovegkezeles\StringB_pr7*)

Emlékeztetőül, ha egy metódus paramétere objektumreferencia, akkor a paraméteren keresztül módosíthatjuk a híváskor megadott objektumot.

```
public class StringB_pr7
{
        static void ElejereIr(StringBuffer s){
            s.insert(0,"Az ");
        }

        static void VegereIr(StringBuffer s){
            int hossz = s.length();
            s.insert(hossz,"viragzik.");
        }

        public static void main(String[] args) {
            StringBuffer szoveg1 = new StringBuffer("almafa ");
            System.out.println(szoveg1);
            ElejereIr(szoveg1);
            System.out.println("ElejereIr metodus hivasa utan: "
                               + szoveg1);
            VegereIr(szoveg1);
            System.out.println("VegereIr metodus hivasa utan : "
                               + szoveg1);
        }
}
```

A program futásának eredménye:

```
almafa
ElejereIr metodus hivasa utan: Az almafa
VegereIr metodus hivasa utan : Az almafa viragzik.
```

9.2.3 A StringTokenizer osztály

A ***java.util*** csomagban találjuk meg a ***StringTokenizer*** osztályt, amely lehetővé teszi szöveg részekre bontását, elválasztójelek alapján. Az alapértelmezés szerinti elválasztójelek a szóköz, a tabulátor (*Tab*), a kocsivissza (*CR*), a soremelés (*LF*) és a lapemelés (*FF*). Ezeken kívül mi magunk is definiálhatunk elválasztójelet.

- `StringTokenizer(String str, String delim, boolean returnTokens)` – olyan objektumot hoz létre, amely az *str* sztringben tárolt szöveget fel tudja bontani részekre, a *delim* sztringben megadott elválasztójelek alapján. Ha *returnToknes* paraméter értéke **true,** akkor a szétbontás során a szövegrésszel együtt az elválasztójeleket is megkapjuk, ellenkező esetben csak szövegrészt.
- `StringTokenizer(String str, String delim)` – ebben a konstruktorban a *returnTokens* alapértelmezése **false**.
- `StringTokenizer(String str)` – az alapértelmezett elválasztójelek alapján történik a szöveg részekre bontása, az elválasztójelek visszaadása nélkül.

Az osztály fontosabb metódusai:

- `int Tokens()` – az elválasztójelek alapján visszaadja a szétbontandó szövegrészek darabszámát,
- `boolean hasMoreTokens()` – igaz értéket ad vissza, ha van még felbontandó szövegrész,
- `String nextToken()` – a következő szövegrésszel tér vissza,
- `String nextToken(String delim)` – a *delim* paraméterrel az elválasztójelek lecserélhetők – ezután ez lesz érvényben.

Írjunk programot, amelyben a beadott egész típusú adatokat szóközzel választjuk el! A szétválasztott adatokat tároljuk egész típusú tömbben, majd számítsuk ki az adatok összegét és átlagát! (*Szovegkezeles\Tokenpr1*)

```
import java.io.*;
import java.util.*;
public class Tokenpr1
{
      public static void main(String[] args) throws IOException {
           BufferedReader be = new BufferedReader
                               (new InputStreamReader(System.in));
           System.out.print("adatok = ");
           String adatok =be.readLine(), a;
           StringTokenizer valaszt = new StringTokenizer(adatok);
           int db = valaszt.countTokens();
           int[] x = new int[db];
           int i = 0;
           while (i < db) {
                a = valaszt.nextToken();
                x[i] = Integer.valueOf(a).intValue();
                System.out.println(i + ". adat: " + x[i] );
                i++;
           }
           int osszeg = 0;
           double atlag;
           i = 0;
           while (i < db)
                osszeg += x[i++];
           atlag = (double)osszeg/db;
           System.out.println("osszeg: " + osszeg);
           System.out.println("atlag: " + atlag);
      }
}
```

A program futásának eredménye:

```
adatok = 1 2 3
0. adat: 1
1. adat: 2
2. adat: 3
osszeg: 6
atlag: 2.0
```

Készítsünk alkalmazást, amelyben a valós típusú bemenő adatokat pontosvesszővel tagoljuk! Számítsuk ki az adatok mértani közepét! (*Szovegkezeles\Tokenpr2*)

```
import java.io.*;
import java.util.*;
public class Tokenpr2
{
      public static void main(String[] args) throws IOException {
           BufferedReader be = new BufferedReader
                               (new InputStreamReader(System.in));
           System.out.print("adatok = ");
           String adatok =be.readLine(), a;
           String elvalaszt = new String(";");
           StringTokenizer valaszt =
                          new StringTokenizer(adatok,elvalaszt);
           int db = valaszt.countTokens();
           double y;
           int i = 0;
           double mertani = 1;
           while (i < db) {
                a = valaszt.nextToken();
                y = Double.valueOf(a).doubleValue();
                mertani *= y;
                System.out.println(i + ". adat: " + y);
                i++;
           }
           mertani = Math.pow(mertani,1./db);
           System.out.println("mertani kozep: " + mertani);
      }
}
```

A program futásának eredménye:

```
adatok = 1;2;3;
0. adat: 1.0
1. adat: 2.0
2. adat: 3.0
mertani kozep: 1.8171205928321397
```

Fejlesszünk programot, amelyben a bemenő valós típusú adatokat először pontosvesszővel, majd pedig kettősponttal tagoljuk! Számítsuk ki az adatok számtani közepét! (*Szovegkezeles\Tokenpr3*)

```
import java.io.*;
import java.util.*;
public class Tokenpr3
{
      public static void main(String[] args) throws IOException  {
           BufferedReader be = new BufferedReader
                               (new InputStreamReader(System.in));
           System.out.print("adatok = ");
           String adatok = be.readLine(), a;
```

```
        StringTokenizer valaszt =
                            new StringTokenizer(adatok, ";");
        int db = valaszt.countTokens();
        int i = 0;
        double szamtani = 0, y;
        while (valaszt.hasMoreTokens()) {
            a = valaszt.nextToken(":;");
            y = Double.valueOf(a).doubleValue();
            szamtani += y;
            System.out.println(i + ". adat: " + y);
            i++;
        }
        szamtani /= db ;
        System.out.println("szamtani kozep: " + szamtani);
    }
}
```

A program futásának eredménye:

```
adatok = 1;2:3;
0. adat: 1.0
1. adat: 2.0
2. adat: 3.0
szamtani kozep: 3.0
```

9.3 Állományok kezelése

Amikor egy Java alkalmazás elindul, három (*unicode*) karakteres adatfolyam (*stream*) automatikusan elérhető lesz a program számára. Beolvasásra az ***InputStream*** típusú ***System.in***, kiírásra pedig a ***PrintStream*** típusú ***System.out*** vagy a ***System.err*** objektumokat használhatjuk, a korábbi példákban bemutatott módon. Az alábbiakban – a teljesség igény nélkül – megnézzük, hogyan hozhatunk létre saját adatfolyamokat, amelyek lehetővé teszik a lemezen tárolt állományok feldolgozását.

Alapvetően kétféle adatfolyamot különböztet meg a Java API, a bájtos és a karakteres adatfolyamokat. Külön érdemes megemlíteni azonban az objektumok bájtfolyamba való írását, illetve onnan történő visszaolvasását, mely műveletek a *serialization* nevet viselik. Az adatfolyamok kialakításához és kezeléséhez szükséges osztályokat és interfészeket a ***java.io*** csomagban találjuk.

Az adatfolyamok alapvetően bájtfolyamok, melyeket kezelő osztályok az ***InputStream*** és ***OutputStream*** absztrakt osztályoktól származnak. Nem adatfolyam alapú osztályokat is biztosít számunkra a ***java.io*** csomag:

File	fájlok és könyvtárak osztálya,
RandomAccessFile	tetszőleges elérésű állományok osztálya.

Az ***InputStream*** osztályból származtatott osztályokat adatok bevitelének megszervezéséhez, míg az ***OutputStream*** osztály utódjait az adatok mentésére használhatjuk:

InputStream utódosztályai	*OutputStream* utódosztályai
ByteArrayInputStream	*ByteArrayOutputStream*
FileInputStream	*FileOutputStream*
FilterInputStream	*FilterOutputStream*
ObjectInputStream	*ObjectOutputStream*

A ***FilterInputStream*** osztály legfontosabb alosztályai: ***BufferedInputStream***, ***CheckedInputStream***, ***DataInputStream***, ***PushbackInputStream*** stb.

A ***FilterOutputStream*** osztály legfontosabb utódosztályai: ***BufferedOutputStream***, ***CheckedOutputStream***, ***DataOutputStream***, ***PrintStream*** stb.

A karakteres adatfolyamok írásának, illetve olvasásának absztrakt alaposztályai a ***Writer*** és a ***Reader***. Az ezekből származó, általunk különböző céllal felhasználható osztályokat táblázatban foglaltuk össze:

Reader utódosztályai	*Writer* utódosztályai
BufferedReader	*BufferedWriter*
CharArrayReader	*CharArrayWriter*
FilterReader	*FilterWriter*
InputStreamReader ← *FileReader*	*OutputStreamWriter* ← *FileWriter*
PipedReader	*PipedWriter*
StringReader	*StringWriter*
	PrintWriter

Most már tudjuk értelmezni az adatbevitelhez oly sokszor használt deklarációt:

```
BufferedReader be = new BufferedReader
                    (new InputStreamReader(System.in));
```

A ***System.in*** egy bájtfolyam, melyet az ***InputStreamReader*** karakterfolyammá alakít. Ezt a karakterfolyamot aztán a ***BufferedReader*** osztály ***readLine()*** metódusával soronként érjük el.

A példaprogramok előtt tekintsük át az adatfolyamok kezelésének általános lépéseit!

- Az adatfolyam megnyitása a konstruktor lefutása után automatikusan végbemegy.
- Az adatfolyamba való írást a ***write()*** metódus túlterhelt változatai biztosítják. Az egyes utódosztályok saját metódusokkal bővíthetik ezt a készletet:
- Az adatfolyamból a ***read()*** metódus átdefiniált változataival olvashatunk adatokat. Az egyes utódosztályok saját metódusok is biztosíthatnak az olvasáshoz. A ***skip***() metódussal adott számú adatelemet át is léphetünk, beolvasás nélkül.

- Munkánk végeztével az adatfolyamot le kell zárni a *close()* metódus hívásával. Kivitel esetén a lezárást megelőzi a pufferek adatfolyamba írása, amely műveletet a *flush()* metódussal magunk is elvégezgetjük. A lezárt adatfolyamot nem lehet újra megnyitni.

9.3.1 A File osztály használata

A *File* osztály a számítógépünk fájlrendszeréhez biztosít egyszerű hozzáférést. Az első példában a könyvtárstruktúra bejegyzéseit dolgozzuk fel segítségével, míg a másodikban állománykezelést valósítunk meg.

Írjunk programot, amely rekurzív módon listázza ki a fájlneveket a kijelölt könyvtárágból! (*Fajlkezeles\Rekurziv_DIR*)

```
import java.io.File;
class DirLista
{
    public static void main(String args[]) {
       String dirnev = "./Dir";
       System.out.println("A " + dirnev + " konyvtar tartalma:");
       dirLista(dirnev , ".");
    }
    static void dirLista(String dir, String jel) {
       File f1 = new File(dir);
       if (f1. isDirectory()) {
             String s[] = f1.list();
             for ( int i=0; i < s. length; i++) {
                File f = new File(dir + "/" + s[i]);
                if (f.isDirectory()) {
                    System.out.println("\n|"+jel+s[i]);
                    dirLista(dir + "/" + s[i], jel + jel);
                }
                else
                   System.out.println(jel+jel+s[i]);
             }
       }
       else
          System.out.println(dir + " nem konvvtar!");
    }
}
```

A program futásának eredménye:

```
A ./Dir konyvtar tartalma:

|.AldirA
....Afile1.txt
....Afile2.txt

|..AldirAA
........AAFile1.txt
........AAFile2.txt
```

```
|.AldirB
....Bfile1.txt
....Bfile2.txt
....Bfile3.txt
..File1.txt
..File2.txt
```

Készítsünk programot, amely egész típusú adatokat ír a szöveges *adatok.txt* állományba! (*Fajlkezeles\FileIr_Olvas*)

A példában a szabványos inputot a szokásos módon használjuk adatok beolvasására. Az állomány írása céljából a létrehozott ***File*** típusú objektumot egy ***FileWriter*** típusú objektum létrehozásához használjuk, amely biztosítja a szokásos karakteres kiviteli metódusokat (***append()***, ***write()***). Szöveges adatok esetén azonban sokkal egyszerűbben is elvégezhetjük az adatok kiírását, ha a ***FileWriter*** objektummal egy ***PrintWriter*** objektumot hozunk létre. (Megjegyezzük, hogy ezt közvetlenül a ***File*** objektummal is megtehettük volna.) A ***PrintWriter*** osztály ***print()*** metódusával már egyszerűen kiírhatjuk a vesszővel elválasztott adatainkat.

```
// FileIr.java - adatok szöveges állományba írása
import java.io.*;
public class FileIr
{
   public static void main(String[] args) throws IOException
   {
      BufferedReader be=
           new BufferedReader(new InputStreamReader(System.in));
      // egész adatok beolvasása
      System.out.print("adatok szama = ");
      int db =Integer.valueOf(be.readLine()).intValue();
      int[] x = new int[db];
      for (int i = 0; i <db; i++) {
         System.out.print(i + ".adat = ");
         x[i] = Integer.valueOf(be.readLine()).intValue();
      }
      // adatok kiírása - minden adat után vessző áll a példában
      File f = new File("adatok1.txt");
      System.out.println("\nadatok kiirasa: adatok.txt allomanyba");
      FileWriter ki_stream = new FileWriter(f);
      PrintWriter ki = new PrintWriter(ki_stream);
      ki.print(db);
      ki.print(',');
      for (int i = 0; i <db; i++) {
         ki.print(x[i]);
         ki.print(',');
         System.out.println(x[i]);
      }
      ki.close();
   }
}
```

Az program futásának eredménye:

```
adatok szama = 3
0.adat = 3
1.adat = 5
2.adat = 2

adatok kiirasa: adatok.txt allomanyba
3
5
2
```

Írjunk programot, amelyben egész típusú adatokat olvas a szöveges *adatok.txt* állományból, és kiszámítja az adatok összegét! (*Fajlkezeles\FileIr_Olvas*)

A szövegfájl visszaolvasásához a ***File*** objektummal egy ***FileReader*** olvasó objektumot hozunk létre. Ennek ***read()*** metódusát használva alakítjuk az állomány karaktereit egész számmá, az *intOlvas()* metódusban. A metódusban figyelembe vettük, hogy minden adat után vessző áll.

```
// FileOlvas.java - Adatok olvasása szöveges állományból
import java.io.*;
import java.util.*;
public class FileOlvas
{
   // a soron következő vesszővel határolt egész adat beolvasása
   public static int intOlvas(FileReader bs) throws IOException
   {
      StringBuffer sz = new StringBuffer(12);
      int b;
      do {
           b = bs.read();
           if ((char)b != ',')
               sz.append((char)b);
           else
               break;
           } while (true);
      return Integer.parseInt(sz.toString());
   }

   public static void main(String[] args) throws IOException
   {
      int db;
      File fajl = new File("adatok1.txt");
      FileReader be_stream = new FileReader(fajl);

      // adatok beolvasása állományból
      db = intOlvas(be_stream);
      System.out.println("adatok szama = " + db);

      int[] x = new int[db];
```

```
        for (int i = 0; i < db; i++) {
            x[i] = intOlvas(be_stream);
            System.out.println(i + ".adat = " + x[i]);
        }
        be_stream.close();
        // adatok összegzése
        int osszeg =0;
        for (int i = 0; i <db; i++)
            osszeg += x[i] ;
        System.out.println("osszeg: " + osszeg);
    }
}
```

Az program futásának eredménye:

```
adatok szama = 3
0.adat = 3
1.adat = 5
2.adat = 2
osszeg: 10
```

9.3.2 A RandomAccessFile osztály alkalmazása

A ***RandomAccessFile*** típusú objektum önmagában képes mindenféle típusú adatot állományba írni, illetve onnan visszaolvasni. Az adatok olyan formában tárolódnak, mint ahogy a memóriában.

Az osztály konstruktorának hívásakor meg kell mondanunk, milyen módon kívánjuk a fájlt elérni: `"r"` - csak olvasására, vagy `"rw"` - írásra és olvasásra.

```
new RandomAccessFile(File vagy String, módsztring);
```

A szokásos ***write()*** és ***read()*** metódusok mellett külön metódusokat biztosít a primitív típusú adatok, sztringek és bájttömbök írására/olvasására. Nézzünk ezek közül néhányat!

```
void writeBoolean(boolean v)        boolean readBoolean()
void writeByte(int v)               byte readByte()
void writeChar(int v)               char readChar()
void writeDouble(double v)          double readDouble()
void write(byte[] b)                void readFully(byte[] b)
void writeInt(int v)                int readInt()
void writeChars(String s)           String readLine()
void writeLong(long v)              long readLong()
```

A ***readFully()*** metódus ***read()*** párja is használható, azonban a ***read()*** legfeljebb annyi bájtot olvas be, amennyi még elérhető az állomány végéig. Így előfordulhat, hogy a beérkezett adatbájtok száma kisebb lesz, mint *b.length*.

Az osztály a nevét onnan kapta, hogy az állományon belül pozícionálni is lehet az ***seek()*** és a ***getFilePointer()*** metódusok segítségével.

Fejlesszünk programot, amelyben az egész típusú adatokat a típusos *adatok.dat* állományba írjuk! (*Fajlkezeles\RandomIr_Olvas*)

```
//RandomIr.java - adatok típusos állományba való írása
import java.io.*;
public class RandomIr
{
   public static void main(String[] args) throws IOException
   {
      BufferedReader be =
            new BufferedReader(new InputStreamReader(System.in));
      System.out.print("adatok szama = ");
      int db =Integer.valueOf(be.readLine()).intValue();
      int[] x = new int[db];
      for (int i = 0; i<db; i++) {
          System.out.print(i + ".adat = ");
          x[i] = Integer.valueOf(be.readLine()).intValue();
      }
      // adatok kiírása állományba
      RandomAccessFile f = new RandomAccessFile("Adatok.dat","rw");
      System.out.println("\nadatok kiirasa: adatok.dat allomanyba");
      f.writeInt(db);
      for (int i = 0; i<db; i++) {
          f.writeInt(x[i]);
          System.out.println(x[i]);
      }
      f.close();
    }
}
```

Az program futásának eredménye:

```
adatok szama = 4
0.adat = 3
1.adat = 2
2.adat = 5
3.adat = 7

adatok kiirasa: adatok.dat allomanyba
3
2
5
7
```

Készítsünk alkalmazást, amelyben a típusos *adatok.dat* állományból egész típusú adatokat olvasunk, és kiszámítjuk az adatok összegét! (*Fajlkezeles\RandomIr_Olvas*)

```
//RandomOlvas.java - adatok olvasása típusos állományból
import java.io.*;
import java.util.*;
```

```
public class RandomOlvas
{
   public static void main(String[] args) throws IOException
   {
      int db;
      RandomAccessFile f = new RandomAccessFile("Adatok.dat","r");
      // adatok beolvasása állományból
      db = f.readInt();
      System.out.println("adatok szama = " + db);
      int[] x = new int[db];
      for (int i = 0; i < db; i++) {
          x[i] = f.readInt();
          System.out.println(i + ".adat = " + x[i]);
      }
      f.close();
      // adatok összegzése
      int osszeg =0;
      for (int i = 0; i <db; i++)
         osszeg += x[i] ;
      System.out.println("osszeg: " + osszeg);
   }
}
```

Az program futásának eredménye:

```
adatok szama = 4
0.adat = 3
1.adat = 2
2.adat = 5
3.adat = 7
osszeg: 17
```

9.3.3 A FileOutputStream és a FileInputStream osztályok használata

A bevezetőben elmondottak teljes mértékben igazak a címben szereplő két adatfolyamosztály alkalmazására. A fájlkezelés első lépése a megfelelő objektumpéldány létrehozása, a megnyitandó állomány nevének megadásával.

```
new FileOutputStream (File vagy String);
```

Az állománykezelést segíti az ***InputStream*** osztály ***available()*** metódusa, amellyel megtudhatjuk a fájlban rendelkezésre álló bájtok számát. Nézzük az adatok forgalmazására használható metódusokat, minkét osztály esetén!

Írás	*Olvasás*	*Mennyit?*
`void write(int b)`	`int read()`	egy bájtot,
`void write(byte[] b)`	`int read(byte[] b)`	legfeljebb *b.length* bájtot,
`void write(byte[] b, int off, int len)`	`int read(byte[] b, int off, int len)`	legfeljebb *len* bájtot.

Az egészek kiírását, illetve visszaolvasását magunknak kell megoldanunk, mint ahogy ezt az *intIR()* és *intOlvas()* metódusokban tettük.

Írjunk alkalmazást, amelyben egész típusú adatokat ír bájtfolyamként az *adatok.txt* állományba! (*Fajlkezeles\StreamIr_Olvas*)

```
// StreamIr.java - adatok  szöveges állományba való kiírása
import java.io.*;
public class StreamIr
{
   // a soron következő vesszővel határolt egész adat kiírása
   public static void intIr(FileOutputStream f, int n)
            throws IOException {
      String sz = "" + n + ",";
      f.write(sz.getBytes());
   }

   public static void main(String[] args) throws IOException {
      BufferedReader be =
            new BufferedReader(new InputStreamReader(System.in));
      System.out.print("adatok szama = ");
      int db =Integer.valueOf(be.readLine()).intValue();
      int[] x = new int[db];
      for (int i = 0; i <db; i++) {
          System.out.print(i + ".adat = ");
          x[i] = Integer.valueOf(be.readLine()).intValue();
      }
      // adatok kiírása állományba - minden adat után vessző áll
      FileOutputStream f = new FileOutputStream("adatok1.txt");
      System.out.println("\nadatok kiirasa: adatok.txt allomanyba");
      intIr(f, db);
      for (int i = 0; i <db; i++) {
         intIr(f, x[i]);
         System.out.println(x[i]);
      }
      f.close();
   }
}
```

Az program futásának eredménye:

```
adatok szama = 4
0.adat = 2
1.adat = 1
2.adat = 3
3.adat = 5

adatok kiirasa: adatok.txt allomanyba
2
1
3
5
```

Készítsünk alkalmazást, amelyben egész típusú adatokat olvas bájtfolyamként az *adatok.txt* állományból, és kiszámítja az adatok összegét!
(*Fajlkezeles\StreamIr_Olvas*)

```
//StreamOlvas.java - adatok olvasása szöveges állományból
import java.io.*;
import java.util.*;
public class StreamOlvas
{
   // a soron következő vesszővel határolt egész adat beolvasása
   public static int intOlvas(FileInputStream f) throws IOException
   {
      StringBuffer sz = new StringBuffer(12);
      int b;
      do {
          b = f.read();
          if ((char)b != ',')
             sz.append((char)b);
          else
             break;
      } while (true);
      return Integer.parseInt(sz.toString());
   }

   public static void main(String[] args) throws IOException {
      int db;
      FileInputStream f = new FileInputStream("adatok1.txt");

      // adatok beolvasása állományból
      db = intOlvas(f);
      System.out.println("adatok szama = " + db);

      int[] x = new int[db];
      for (int i = 0; i < db; i++) {
          x[i] = intOlvas(f);
          System.out.println(i + ".adat = " + x[i]);
       }
      f.close();

      // adatok összegzése
      int osszeg =0;
      for (int i = 0; i < db; i++)
          osszeg += x[i] ;
      System.out.println("osszeg: " + osszeg);
   }
}
```

Az program futásának eredménye:

```
adatok szama = 4
0.adat = 2
1.adat = 1
2.adat = 3
3.adat = 5
osszeg: 11
```

9.4 Adattároló osztályok

Gyakran van szükség arra, hogy nagyobb mennyiségű adatot tároljunk és kezeljünk a Java alkalmazásainkból. Erre célra természetesen használhatjuk a tömböket vagy az adatfolyamokat, azonban ekkor a szükséges műveletek (keresés, rendezés stb.) magunknak kell megvalósítani. Azonban sokkal gyorsabban és biztonságosabban célt érünk, nem beszélve arról, hogy a feladatunk lényegére koncentrálhatunk, ha kész megoldásokra építünk.

A *java.util* egyik legnépesebb és legváltozatosabb csomagja a *Java API*-nak. Többek között dátum-, naptár-, időzítő- és tároló (*container*) osztályokat tartalmaz. A tároló osztályok különböző adatstruktúrákat valósítanak meg, melyek adatai objektumok.

A teljes ismertetés helyett, először tisztázzuk a *java.util* csomag osztályainak felhasználásához szükséges ismereteket, majd pedig néhány egyszerű példát mutatunk az alkalmazásukra.

A *java.util* csomag *Java 2 platform 5.0* dokumentációjában *Stack<E>*, *Vector<E>* vagy *Dictionary<K,V>* osztályokat találunk. Eddigi ismereteinkkel sem mi, sem pedig a korábbi Java fordítók nem tudják értelmezni ezeket a neveket. A megértésükhöz át kell tekintenünk a Java nyelvnek a J2SE 5.0-ban bevezetett újdonságát, az általánosított osztályok, osztálysablonok (*generics*, *templates*) alkalmazását.

9.4.1 Általánosított osztályok

A Java lehetővé teszi, hogy saját általánosított osztályokat alakítsunk ki, ahogy azt az alábbi *Template* nevű példa mutatja:

```
class Template<E>
{
      private E d;
      public Template(E n) {
            d = n;
      }

      E getD() {
            return d;
      }
}
```

A példából jól látható, hogy az *E* típus az osztály paraméterként jelenik meg deklarációban. Amikor az osztályt példányosítjuk, akkor egy konkrét osztályt adunk meg az *E* helyén:

```
Template<Integer> a=new Template<Integer>(new Integer(12));
System.out.println(a.getD());
Template<String> b=new Template<String>("december");
System.out.println(b.getD());
Template<Object> c=new Template<Object>(null);
System.out.println(c.getD());
```

Ha nem adunk meg osztályt az objektum előállítása során, a J2SE 5.0 fordítója figyelmeztet bennünket („*warning: [unchecked] unchecked call to Template(E) as a member of the raw type Template*"), és az ***Object*** osztályt használja:

```
Template a=new Template(new Integer(12));
System.out.println(a.getD());
Template b=new Template("december");
System.out.println(b.getD());
Template c=new Template(null);
System.out.println(c.getD());
```

Az alkalmazás ekkor is működik, hisz minden osztály az ***Object*** osztálytól származik, és mindegyik rendelkezik a ***toString()*** metódus saját, felülbírált változatával.

Hogy miért van mégis szüksége az általánosított osztályokra, megértjük, ha például az alábbi *szam* objektumban tárolt egész számot numerikus kifejezésben szeretnénk felhasználni.

```
// az osztálysablont megfelelő módon alkalmazva:
Template<Integer> szam = new Template<Integer>(new Integer(12));
double x = szam.getD().doubleValue() + 23.45;

// az osztálysablont nem korrekt módon alkalmazva,
// a megoldás típus-átalakítást igényel:
Template szam = new Template(new Integer(12));
double x = ((Integer)szam.getD()).doubleValue()+23.45;
```

Az első esetben a szükséges konverziókat a fordítóprogram végzi el helyettünk.

Végezetül tekintsük a *Template* osztályunkat abban a formában, ahogy azt a J2SE korábbi verzióiban definiálhattuk volna.

```
class Template
{
        private Object d;
        public Template(Object n) {
                d = n;
        }
        Object getD() {
                return d;
        }
}
```

(A példaprogramok forrásállományait a fejezet könyvtárában, az „*Adatstrukturak\Sajat template*" alkönyvtárban találjuk.)

9.4.2 A konténerosztályok használata

A dinamikusan növekedni képes ***Vector<E>*** osztály példáján keresztül áttekintjük a tárolók alkalmazásának legfontosabb fogásait. Első lépés a konstruálás, melyet általában többféleképpen is elvégezhetünk:

`Vector()`	kezdeti elemszám 10,
`Vector(Collection<? extends E> c)`	a létrejövő vektort egy másik kollekció elemeivel tölti fel,
`Vector(int capacity)`	adott kezdeti elemszámú vektort hoz létre,
`Vector(int capacity, int increment)`	adott kezdeti elemszámú vektort készít, melynek mérete szükség esetén az *increment* elemszámmal növekedhet.

A sikeres létrehozás után egy sor metódus segíti a felhasználást. Csak a legfontosabbakat gyűjtöttük csokorba:

`boolean add(E o)`	objektum hozzáadása az elemek végéhez,
`void add(int index, E elem)`	elem beszúrása a megadott helyre,
`void addElement(E obj)`	objektum hozzáadása a végéhez, eggyel növelve a vektor méretét,
`int capacity()`	az aktuális kapacitás lekérdezése,
`void clear()`	minden elem törlése,
`boolean contains(Object elem)`	ellenőrzi, hogy a megadott objektum eleme-e a vektornak,
`E elementAt(int index)`	az adott sorszámú elem referenciájával tér vissza,
`E firstElement()`	a 0. elem referenciájával tér vissza,
`E get(int index)`	az adott sorszámú elem referenciájával tér vissza,
`int indexOf(Object elem)`	megadja az adott elem indexét,
`int indexOf(Object elem, int index)`	adott sorszámú elemtől kezdve keresi a megadott elem indexét,
`void insertElementAt(E elem, int index)`	adott pozícióra beszúr egy elemet,
`boolean isEmpty()`	teszteli, hogy üres-e a vektor,
`E lastElement()`	az utolsó elem referenciájával tér vissza,
`int lastIndexOf(E elem)`	az adott elem utolsó előfordulásának indexével tér vissza,
`int lastIndexOf(E elem, int index)`	az adott sorszámú elemtől kezdve visszafelé keresi a megadott elem indexét,
`E remove(int index)`	adott sorszámú elem törlése,
`boolean remove(Object o)`	adott objektum törlése, ha benne van,
`void removeAllElements()`	minden elem eltávolítása,
`boolean removeElement(Object obj)`	az objektum első előfordulását törli,
`void removeElementAt(int index)`	adott indexű elem törlése,
`E set(int index, E elem)`	adott sorszámú elem lecserélése,
`void setElementAt(E obj, int index)`	adott indexű helyre az objektum behelyezése,
`void setSize(int newSize)`	méret beállítása,
`int size()`	az elemszám lekérdezése
`String toString()`	sztringbe másolja a vektor elemeinek szöveges megjelenését,
`void trimToSize()`	a kapacitás az aktuális elemszám lesz.

Tervezzünk alkalmazást, amely bemutatja a dinamikus Vector<E> osztály használatát! (*Adatstrukturak\DinamVektor*)

```
import java.util.*;                                          // Vector<E>
class DinamVektor {
    static void feltoltes(Vector<Integer> v, int n) {
            for (int i=0; i<n; i++)
              v.add(new Random().nextInt(123));
    }

    static void rendezes(Vector<Integer> v) {
       for (int i = 0; i < v.size()-1; i++)
         for (int j = i+1; j < v.size(); j++)
           if (v.elementAt(j).intValue() < v.elementAt(i).intValue())
           {
             int e = v.elementAt(i);
             v.set(i, v.elementAt(j));
             v.set(j,e);
           }
    }

    public static void main(String args[]) {
       // 7 elemmel jön létre, és 12-esével nő az elemek száma
       Vector<Integer> vektor = new Vector<Integer>(7,12);
       System.out.println("Feltoltes:");
       feltoltes(vektor, 10);
       System.out.println(vektor.toString());

       System.out.println("Rendezes:");
       rendezes(vektor);
       System.out.println(vektor.toString());

       // 23 beszúrása a 7. indexű pozícióban
       System.out.println("23 beszurasa:");
       vektor.add(7,23);
       System.out.println(vektor.toString());

       // megkeressük a 23 értékű elem indexét
       System.out.println("a 23 erteku elem indexe:");
       if (vektor.contains(23))
          System.out.println("" + vektor.indexOf(23));
       else
          System.out.println("nincs a vektorban a megadott elem");

       // az 5 indexű elemet tízszeresére növeljük
       System.out.println("az 5 indexu elem tizszerezese:");
       int e = vektor.get(5);
       vektor.set(5, 10*e);
       System.out.println(vektor.toString());

       // a 7. indexű elem törlése
       System.out.println(" a 7. indexu elem torlese:");
       vektor.remove(7);
```

```
        System.out.println(vektor.toString());
        // mimden elem törlése
        System.out.println("minden elem torlese:");
        vektor.clear();
        System.out.println(vektor.toString());
      }
}
```

A program futásának eredménye:

```
Feltoltes:
[84, 71, 38, 45, 41, 120, 4, 39, 49, 85]
Rendezes:
[4, 38, 39, 41, 45, 49, 71, 84, 85, 120]
23 beszurasa:
[4, 38, 39, 41, 45, 49, 71, 23, 84, 85, 120]
a 23 erteku elem indexe:
7
az 5 indexu elem tizszerezese:
[4, 38, 39, 41, 45, 490, 71, 23, 84, 85, 120]
 a 7. indexu elem torlese:
[4, 38, 39, 41, 45, 490, 71, 84, 85, 120]
minden elem torlese:
[]
```

Tervezzünk alkalmazást, amely bemutatja a verem használatát! (*Adatstrukturak\Verem - stack*)

```
import java.util.Stack;                                    // Stack<E>
import java.util.EmptyStackException;

class Verem
{
    static void betesz(Stack<Integer> st, int a) {
       st.push(new Integer(a));
       System.out.println("push ->" + a);
       System.out.println("a verem : " + st + "\n");
    }

    static int kivesz(Stack<Integer> st) {
       System.out.print("pop  <- ");
       int a = st.pop().intValue();
       System.out.println(a);
       System.out.println("a verem : " + st + "\n");
       return a;
    }

    public static void main(String args[]) {
       Stack<Integer> st = new Stack<Integer>();
       int elem;
       System.out.println("a verem : " + st + "\n");
       try {
            betesz(st, 12);
            betesz(st, 23);
            betesz(st, 79);
```

```
            elem = kivesz(st);
            elem = kivesz(st);
            elem = kivesz(st);
            elem = kivesz(st);
        }
        catch (EmptyStackException e)
        {
            System.out.println("a verem ures");
        }
    }
}
```

A program futásának eredménye:

```
a verem : []
push ->12
a verem : [12]
push ->23
a verem : [12, 23]
push ->79
a verem : [12, 23, 79]
pop  <- 79
a verem : [12, 23]
pop  <- 23
a verem : [12]
pop  <- 12
a verem : []
pop  <- a verem ures
```

Tervezzünk alkalmazást, amely sor kezelésére alkalmas, sorba beillesz, illetve kivesz a sorból! (*Adatstrukturak\Sor - queue*)

```
import java.util.*;                                    // PriorityQueue<E>
class Sor
{
    static void betesz(PriorityQueue<Integer> pq, int a) {
       pq.add(new Integer(a));
       System.out.println("add ->" + a);
       System.out.println("a sor : " + pq + "\n");
    }

    static int kivesz(PriorityQueue<Integer> pq) {
       System.out.print("remove <- ");
       int a = pq.remove().intValue();
       System.out.println(a);
       System.out.println("a sor : " + pq + "\n");
       return a;
    }

    public static void main(String args[]) {
       PriorityQueue<Integer> pq = new PriorityQueue<Integer>();
```

```
        int elem;
        System.out.println("a sor : " + pq + "\n");
        try {
             betesz(pq, 12);
             betesz(pq, 23);
             betesz(pq, 79);
             elem = kivesz(pq);
             elem = kivesz(pq);
             elem = kivesz(pq);
             elem = kivesz(pq);
          }
          catch (NoSuchElementException e)
          {
             System.out.println("a sor ures");
          }
      }
}
```

A program futásának eredménye:

```
a sor : []
add ->12
a sor : [12]
add ->23
a sor : [12, 23]
add ->79
a sor : [12, 23, 79]
remove <- 12
a sor : [23, 79]
remove <- 23
a sor : [79]
remove <- 79
a sor : []
remove <- a sor ures
```

Tervezzünk alkalmazást, amely listában tárolt sakklépéseket módosít!
(Adatstrukturak\Lista)

```
import java.util.*;                                    // LinkedList<E>
public class Lista
{
    public static void main(String args[]) {
        final String kodsor = "ABCDEFGH";
        LinkedList<String> lepes = new LinkedList<String>();
        for (int i=0; i <7; i++)
             lepes.add("" + kodsor.charAt((int)(Math.random()*8)));
        System.out.println("" + lepes);
        ListIterator<String> lepesIterator = lepes.listIterator();
```

```
        // Végighaladva a listán, módosítjuk az elemeket
        while(lepesIterator.hasNext()) {
              String elem;
              elem = lepesIterator.next();
              System.out.println(elem);
              lepesIterator.set(elem + (int)(Math.random()*8+1));
        }
        // Fordított sorrendben jelenítjük meg az elemeket
        while(lepesIterator.hasPrevious()) {
              System.out.println(lepesIterator.previous());
        }
        System.out.println("" + lepes);
    }
}
```

A program futásának eredménye:

```
[C, A, F, F, D, E, D]
C
A
F
F
D
E
D
D3
E5
D7
F1
F5
A2
C2
[C2, A2, F5, F1, D7, E5, D3]
```

Fejlesszünk alkalmazást, amely elkészíti az árlap alapján rendelt ételek számláját! (*Adatstrukturak\Penztar*)

```
import java.util.Hashtable;                          // Hashtable<K,V>
class Penztar
{
  public static void main(String args[]) {
     String [] aru  = {"pizza", "kola", "burgonya", "hamburger",
                       "kave", "fagylalt", "salata"};
     int [] ar      = {350, 120, 200, 300, 120, 100, 400};
     // az árlap létrehozása és feltöltése
     Hashtable<String, Integer> arlap = new Hashtable<String,
                                              Integer>();
     for (int i = 0; i < aru.length; i++)
         arlap.put(aru[i], new Integer(ar[i]));
     // számlázás
     szamlazas(arlap, new String[] {"kave", "pizza", "salata", "kola"});
     szamlazas(arlap, new String[]{"hamburger","burgonya","fagylalt"});
  }
```

```
  static void szamlazas(Hashtable<String, Integer> d,
                        String [] rendeles) {
     System.out.println("\n --- Szamla --- ");
     int fizetendo = 0;
     for (int i = 0; i < rendeles.length; i++) {
       int tetel = d.get(rendeles[i]).intValue();
       fizetendo += tetel;
       System.out.println((rendeles[i]+"            ").substring(0,12)
                          + tetel);
     }
     System.out.println("---------------");
     System.out.println("fizetendo:  " + fizetendo);
  }
}
```

A program futásának eredménye:

```
--- Szamla ---
kave        120
pizza       350
salata      400
kola        120
---------------
fizetendo:  990

 --- Szamla ---
hamburger   300
burgonya    200
fagylalt    100
---------------
fizetendo:  600
```

Tervezzünk alkalmazást, amely fájlba menti az angol – magyar szavakat páronként, és visszaolvasva szótárként használható! (*Adatstrukturak\Szotar*)

```
import java.io.*;
import java.util.Properties;                                    // Properties

// A Properties osztály tulajdonságnév/érték párok
// tárolására használható
// a tulajdonságlista fájlba menthető, illetve visszaolvasható
class AMSzotar {
  public static void main(String args[]) throws IOException {
     String[] angol  = {"one", "two", "three", "four", "five",
                        "six", "seven", "eight", "nine", "ten" };
     String[] magyar = {"egy", "ketto", "harom", "negy", "ot",
                        "hat", "het", "nyolc", "kilenc", "tiz" };

     // a szótár létrehozása és feltöltése, amennyiben
     // nem létezik a szótárfájl
     Properties pl = new Properties();
     File szf = new File("Szotar.txt");
     if (!szf.exists()) {
        for (int i = 0; i < angol.length; i++)
            pl.put(angol[i], magyar[i]);
```

```
            // a szótár fájlba mentése
            FileOutputStream fout = new FileOutputStream("Szotar.txt");
            pl.store(fout,"Angol-magyar szotar");
            System.out.println("A szotarfajl sikeresen letrejott.");
        }
        else
        {
            // a szótár betöltése fájlból
            FileInputStream fin = new FileInputStream("Szotar.txt");
            pl.load(fin);
            System.out.println("A szotarfajl sikeresen betoltodott.");
        }
        // fordítás a szótár segítségével
        forditas(pl, new String[] {"two", "three", "one", "two"});
    }

    static void forditas(Properties d, String [] szavak) {
        for (int i = 0; i < szavak.length; i++)
            System.out.print(d.get(szavak[i]) + "\t");
        System.out.println();
    }
}
```

A program futásának eredménye:

```
A szotarfajl sikeresen betoltodott.
ketto   harom   egy     ketto
```

Tervezzünk alkalmazást, amely lottósorsolást szimulál! (*Adatstrukturak\Bitset - Lotto*)

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

public class Lotto
{
    BitSet lotto;
    int hanybol, hanyat;

    public static void main(String [] args) {
        new Lotto(90, 5).sorsolas();
        new Lotto(46, 6).sorsolas();
        new Lotto(35, 7).sorsolas();
    }

    public Lotto(int _hanybol, int _hanyat)
    {
        hanybol = _hanybol;
        hanyat = _hanyat;
        lotto = new BitSet( hanybol+1 );
        // minden bit törlése 1 és hanyat között
        for ( int i = 1; i < lotto.size(); i++ )
            lotto.clear( i );
    }
```

```
    // lottósorsolás
    public void sorsolas() {
        System.out.println("Lottoszamok " + hanyat +"/"
                           + hanybol +" :");
        int kihuzott = 0, szam;
        do {
            szam = (int) (Math.random()*hanybol)+1;
            if (!lotto.get(szam)) {
                lotto.set(szam);
                kihuzott++;
            }
        } while (kihuzott < 5);
        // a lottószámok megjelenítése növekvő sorrendben
        for ( int i = 1; i < lotto.size(); i++ )
            if ( lotto.get( i ) )
                System.out.println("" + i);
        System.out.println();
    }
}
```

A program futásának eredménye:

```
Lottoszamok 5/90 :
5
11
35
43
56
Lottoszamok 6/46 :
8
10
12
15
30
Lottoszamok 7/35 :
5
22
28
32
35
```

Felhívjuk a figyelmet arra, hogy az általánosított osztályokat használó példaprogramok a ***J2SE 1.4,*** illetve a korábbi változatok alatt változtatás nélkül nem fordíthatók le. Ahhoz hogy mégis lefordíthatók legyenek a példák, törölnünk kell a *<típus>* részeket az osztályok nevéből, valamint a megfelelő helyeken el kell végeznünk a szükséges típus-átalakításokat.

9.5 Többszálúság - a Thread osztály

Az operációs rendszerek többségében fontos szerepet játszik a folyamat (*process*) fogalma. Első megközelítésben a folyamat alatt az éppen futó programot érthetjük. A többfeladatos (*multitasking*) rendszerek több folyamat egyidejű futtatását támogatják. Azonban a futó (fő)folyamat több párhuzamosan működő alfolyamat (szál, *thread*) indítását is kezdeményezheti. Java-ban a ***java.lang.Thread*** osztály segítségével több szálon futó alkalmazásokat fejleszthetünk.

A ***Thread*** osztály megvalósítja a ***Runnable*** interfész egyetlen ***run()*** metódusát, amelynek törzse üres, így öröklés során ezt magunknak kell kialakítani, elhelyezve benne a szál által végrehajtani kívánt utastásokat. A ***run()*** metódust a futtatórendszer hívja, amennyiben a szálat a ***Thread*** osztály ***start()*** metódusával elindítjuk. Az alfolyamat működését egy adott ideig felfüggeszthetjük a statikus ***sleep()*** metódus segítségével.

Tervezzünk alkalmazást, amely bemutatja a többszálú programok kialakításának lépéseit! (*Szalak\TobbSzal*)

```
// Az általunk definiált szálak osztálya
class SajatSzal extends Thread
{
    SajatSzal(String név) {
        super(név);
        start();
    }

    public void run() {
        try {
            for(int i = 0; i < 4; i++) {
                System.out.println((Thread.currentThread()).getName()
                    + " szal fut...");
                Thread.sleep(1000);  // várakozás 1 másodpercig
            }
        } catch (InterruptedException e) {}

        System.out.println((Thread.currentThread()).getName()
                        + " szal leall.");
    }
}

// a főszál osztálya
public class TobbSzal
{
    public static void main(String args[])
    {
        SajatSzal szal1 = new SajatSzal("ELSO");
        SajatSzal szal2 = new SajatSzal("MASODIK");
        SajatSzal szal3 = new SajatSzal("HARMADIK");
        SajatSzal szal4 = new SajatSzal("NEGYEDIK");
```

```
        try {
            for(int i = 0; i < 10; i++) {
                System.out.println((Thread.currentThread()).getName()
                                    + " szal fut...");
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {}
    }
}
```

Az alkalmazás futásának eredménye:

```
main szal fut...
ELSO szal fut...
HARMADIK szal fut...
NEGYEDIK szal fut...
MASODIK szal fut...
MASODIK szal fut...
main szal fut...
ELSO szal fut...
NEGYEDIK szal fut...
HARMADIK szal fut...
MASODIK szal fut...
main szal fut...
ELSO szal fut...
HARMADIK szal fut...
NEGYEDIK szal fut...
ELSO szal fut...
main szal fut...
HARMADIK szal fut...
MASODIK szal fut...
NEGYEDIK szal fut...
main szal fut...
ELSO szal leall.
HARMADIK szal leall.
MASODIK szal leall.
NEGYEDIK szal leall.
main szal fut...
main szal fut...
main szal fut...
main szal fut...
main szal fut...
```

Amikor több szál megosztott módon használja (módosítja) ugyanazt az objektumot, ügyelnünk kell az adatok érvényességére. A Java nyelv szinkronizációs megoldásai lehetővé teszik, hogy minden szál a helyes értékkel dolgozzon. A szinkronizált működéshez szükséges metódusokat **synchronized** módosítóval kell ellátnunk, valamint az ***Object*** osztálytól örökölt ***notify()*** metódussal kell „felébresztenünk" az épp „alvó" szálat.

Tervezzünk alkalmazást, amely bemutatja a szinkronizált szálak használatát! (*Szalak\Szinkronizalt_szalak*)

```
// Szinkronizált szálak
public class OsztottErtek
{
   public static void main( String args[]) {
      Tarolo tar = new Tarolo();
      Feltolto fSzal = new Feltolto(tar); // írja az adatokat
      Hasznalo hSzal = new Hasznalo(tar); // kiolvassa az adatokat

      fSzal.start();
      hSzal.start();
   }
}

// Ez a szál létrehozza a tárolt értéket
class Feltolto extends Thread
{
   private Tarolo fTarolo;

   public Feltolto(Tarolo tar) {
      fTarolo = tar;
   }

   public void run() {
      for (int szam = 12; szam < 24; szam++ ) {
         fTarolo.setKozosInt(szam);
         System.out.println( "<-- az osztott taroloban tarolt ertek: "
                             + szam);
         // véletlen ideig várakozik 0 és 2 mp között
         try {
            sleep( (int)( Math.random()*2000));
         }
         catch( InterruptedException e ) {
            System.err.println( "Exception " + e.toString() );
         }
      }
   }
}

// Ez a szál használja a tárolt értéket
class Hasznalo extends Thread
{
   private Tarolo hTarolo;

   public Hasznalo(Tarolo tar) {
      hTarolo = tar;
   }
```

```
   public void run() {
      int ertek;
      do {
         // véletlen ideig várakozik 0 és 2 mp között
         try {
            sleep( (int) ( Math.random() * 2000 ) );
         }
         catch( InterruptedException e ) {
            System.err.println( "Exception " + e.toString() );
         }
         ertek = hTarolo.getKozosInt();
         System.out.println( "--> a tarolobol kivett ertek : "
                             + ertek );
      } while ( ertek != 23); // a leállítás feltétele
   }
}

// Az osztott elérésű tároló szinkronizált metódusokkal
class Tarolo {
   private int kozosInt;
   private boolean irhato = true;

   public synchronized void setKozosInt(int ertek) {
      while (!irhato) {
         try {
            wait();
         }
         catch ( InterruptedException e ) {
            System.err.println( "Exception: " + e.toString() );
         }
      }
      kozosInt = ertek;
      // csak akkor írható felül, ha már kiolvasták a getKozosInt()
      // metódus hívásával
      irhato = false;
      notify();
   }

   public synchronized int getKozosInt() {
      while (irhato) {
         try {
            wait();
         }
         catch ( InterruptedException e ) {
            System.err.println( "Exception: " + e.toString() );
         }
      }
      irhato = true;       // felülírható
      notify();
      return kozosInt;
   }
}
```

Az alkalmazás futásának eredménye:

```
<-- az osztott taroloban tarolt ertek : 12
--> a tarolobol kivett ertek : 12
<-- az osztott taroloban tarolt ertek : 13
--> a tarolobol kivett ertek : 13
<-- az osztott taroloban tarolt ertek : 14
--> a tarolobol kivett ertek : 14
<-- az osztott taroloban tarolt ertek : 15
--> a tarolobol kivett ertek : 15
<-- az osztott taroloban tarolt ertek : 16
--> a tarolobol kivett ertek : 16
<-- az osztott taroloban tarolt ertek : 17
--> a tarolobol kivett ertek : 17
<-- az osztott taroloban tarolt ertek : 18
--> a tarolobol kivett ertek : 18
<-- az osztott taroloban tarolt ertek : 19
--> a tarolobol kivett ertek : 19
<-- az osztott taroloban tarolt ertek : 20
--> a tarolobol kivett ertek : 20
<-- az osztott taroloban tarolt ertek : 21
--> a tarolobol kivett ertek : 21
<-- az osztott taroloban tarolt ertek : 22
--> a tarolobol kivett ertek : 22
<-- az osztott taroloban tarolt ertek : 23
--> a tarolobol kivett ertek : 23
```

10. Grafikus felületű alkalmazások

A grafikus felületű alkalmazások fejlesztésének első lépése a felhasználói felület (*GUI – Graphical User Interface*) megtervezése és megvalósítása. Integrált fejlesztői környezet (IDE) nélkül, ez komoly (munkaigényes) programozási feladatot jelent. A felület kialakítását követően fel kell készíteni a programunkat a különböző események (*events*) kezelésére.

A grafikus felületű alkalmazások kialakításához a ***Java API*** osztályokat és interfészeket bocsát a programozók rendelkezésére, amelyek meggyorsítják a fejlesztési munkát. Alapvetően kétfajta osztálykönyvtár közül választhatunk:

- A korábban kifejlesztett osztálykönyvtár az ***AWT*** (*Abstract Window Toolkit*). Az AWT komponenseit úgy tervezték, hogy az operációs rendszer elemeihez hasonlóak legyenek. Használatuk kissé nehézkes, mivel az *AWT* osztályok közvetítő feladatot látnak el a felhasználó programja és az operációs rendszer között.
- Az újabb osztálykönyvtár a ***Swing***, amely sokkal gazdagabb, mint az *AWT*, és a fent említett korlátozás sem létezik. Természetesen a *Swing* osztály vezérlői sokkal többet tudnak, felépítésük logikusabb. Mivel formatervezésük sajátos, minden platformon azonosan néznek ki a grafikus elemek.

Összehasonlítva a két osztályt tulajdonságaik alapján:

- az *AWT* osztályok használata kevésbé kényelmes, de a futási idejük kisebb,
- a *Swing* összetevőket csak a legújabb böngészők támogatják,
- az áttérés az *AWT* osztályokról a *Swing* osztályokra nem bonyolult.

Ahogy a *Swing* az *AWT* komponenseit fejleszti tovább, úgy bővítik az *AWT* grafikus lehetőségeit a *Java 2D* osztályok. Az ***AWT***, a ***Swing*** és a ***Java 2D*** osztályokat együtt *Java Alaposztályoknak* (***Java Foundation Classes***, ***JFC***) szokták nevezni. A *JFC* osztályait különböző csomagok tárolják:

AWT	***java.awt, java.awt.event***
Swing	***javax.swing***
Java 2D	***java.awt.geom***

A könyvünkben mindhárom osztálykönyvtár használatára mutatunk példákat.

A megvalósítás oldaláról közelítve a grafikus felhasználói felület kialakítását, az építőelemeket két csoportra oszthatjuk. Az egyik csoportba tartoznak a tárolók (*containers*), amelyek a másik csoport elmeit tárolják. Az *AWT* központi osztálya a ***Component***, amely a komponensek által használható közös metódusok gyűjtőhelye. A tárolók ***Container*** ősosztálya is innen származik, amely egyben kiindulópontja a

Swing elemek alaposztályának, a ***JComponent*** osztálynak. (A ***J*** betűvel kezdődő osztálynevek általában *Swing* komponenst jelölnek).

Az alábbiakban táblázatos formában megadtuk a ***Component*** és a ***MenuComponent*** osztályoktól származó komponensek öröklési hierarchiájának bizonyos ágait. Azokat az osztályokat igyekeztünk elsősorban feltüntetni, amelyek előfordulnak a könyvünk példaprogramjaiban. A táblázatokban balról jobbra haladva, egyre későbbi generációk láthatók, például a ***JMenu*** származási láncolata:

Component ← ***Container*** ← ***JComponent*** ← ***AbstractButton*** ← ***JMenuItem*** ← ***JMenu***

*A **Component** osztály utódai*

Button				
Canvas				
Checkbox				
Choice				
Label				
List				
Scrollbar				
TextComponent	***TextArea***			
	TextField			
Container	***ScrollPane***			
	Panel	***Applet***	***JApplet***	
	Window	***JWindow***		
		Frame	***JFrame***	
		Dialog	***FileDialog***	
			JDialog	
	JComponent	***JComboBox***		
		JLabel		
		JMenuBar		
		JPopupMenu		
		JList		
		JPanel		
		JScrollBar		
		JScrollPane		
		JOptionPane		
		JTextComponent	***JTextArea***	
			JTextField	***JPasswordField***
		AbstractButton	***JButton***	
			JToggleButton	***JRadioButton***
				JCheckBox
			JMenuItem	***JMenu***

*A **MenuComponent** osztály utódai*

MenuBar		
MenuItem	***CheckboxMenuItem***	
	Menu	***PopupMenu***

10.1 A grafikus felület felépítésének alapelvei

Tekintsük át, hogy milyen eszközökre van szükségünk a grafikus megjelenítéshez! A grafikus felületű alkalmazások az operációs rendszerben legalább egy ablakkal rendelkeznek. Ebben az ablakban helyezkednek el a különböző komponensek (menü, gombok, listaablak, szöveges vezérlők stb.), és itt program segítségével rajzot is készíthetünk.

A rajzoláshoz szükségünk van egy rajzfelületre. Erre olyan komponensek alkalmasak, mint a panel (*AWT*: ***Panel***, *Swing*: ***JPanel***) vagy az ablak (*AWT*: ***Frame***, *Swing*: ***JFrame***). Ha nem a teljes ablakba, vagy panelre szeretnénk rajzolni, akkor használjuk a festővászon (***Canvas***) komponenst. A ***Canvas*** osztály *AWT* összetevő, nincs *Swing* megfelelője, azonban a ***JPanel*** osztállyal helyettesíthetjük.

Ebben a fejezetben a komponensek használatával ismerkedünk, míg a programozott rajzolás műveleteit a következő fejezetben tárgyaljuk. Ennek ellenére két olyan osztállyal kezdjük az ismerkedést, amelyeknek inkább ott lenne a helyük.

10.1.1 Színek és betűtípusok

A grafikus megjelenítés alapvető sajátossága a színek alkalmazása. A színeket a komponensek, az ablak és a rajzolás színezésére egyaránt használhatjuk. A komponens háttérszínét a ***setBackground()***, az előtér színét a ***setForeground()***, míg a rajzolás színét a ***setColor()*** metódussal állíthatjuk be. Mindegyik metódus egy színt vár paraméterként.

A színek kezelését a ***java.awt*** csomag ***Color*** osztálya segíti. A színeket az RGB-modellnek megfelelően három (piros, zöld és kék) összetevőből állítjuk elő. Mindegyik összetevő értéke 0 és 255 között változhat. A *(0,0,0)* a fekete, míg a *(255,255,255)* a fehér színt jelölik. Ha a három összetevő értéke megegyezik akkor a szürke valamilyen árnyalatát kapjuk A ***Color*** osztályban minden színhez tartozik egy alfa érték *(0.0-1.0)*, amely a szín áttetszőségét határozza meg.

A színeket többféleképpen is megadhatjuk:

- A ***Color*** osztály statikus osztálytagjaival az alábbi színeket érjük el:

black – fekete,
blue – kék,
cyan – türkiz,
darkGray – sötét szürke,
gray – szürke,
green, – zöld,
lightGray – világos szürke,
magenta – lila,
orange – narancs,
pink – rózsaszín,
red – piros,
white– fehér,
yellow – sárga.

- A szín megadásának másik módja az RGB-színösszetevők használata a ***Color*** osztály konstruktorában. A szélsőértékek alkalmazásával az alábbi színekhez jutunk:

szín	*R*	*G*	*B*
fehér	255	255	255
fekete	0	0	0
piros	255	0	0
zöld	0	255	0
kék	0	0	255
türkiz	0	255	255
sárga	255	255	0
lila	255	0	255

Gyakorlásképpen állítsuk be a háttér színét türkizre!

- Létrehozunk egy ***Color*** példányt, inicializálva az R, G és B értékekkel:

```
setBackground(new Color(0,255,255));    // türkiz
```

- Vagy a ***Color*** osztály statikus színkonstansával:

```
setBackground(Color.cyan);              // türkiz
```

A ***Color*** osztály metódusaival hatékonyan kezelhetjük a színeket. Adott a lehetőség a tárolt színnél sötétebb (***darker()***), illetve világosabb (***brighter()***) szín előállítására, az RGB-összetevők lekérdezésére egyenként (***getRed()***, ***getGreen()***, ***getBlue()***), vagy együtt egy ***int*** értékbe ***getRGB()***, illetve egy **float** típusú, háromelemű tömbbe ***getRGBColorComponents()***. Négyelemű **float** tömbbe az RGBA (RGB és alfa) értékeket is lekérhetjük ***getRGBComponents()***.

A grafikus felületen megjelenő szövegek típusát a ***java.awt.Font*** osztály objektumpéldányaival határozhatjuk meg. Egyelőre csak annyit kell a Font osztályról tudnunk, hogy konstruktorának hívásával új betűtípust hozhatunk létre:

```
Font(String fontnév, int stílus, int pontméret)
```

A *stílus* helyén – esetleg vagy (|) kapcsolattal kombinált – konstansok állhatnak: *Font.BOLD*, *Font.ITALIC*, *Font.PLAIN* stb. A *fontnév* paraméterben a betűkészlet nevét kell átadnunk a konstruktornak, például: „*Times New Roman*", „*Courier New*" stb. A pontmérettel pedig a betűk nagyságát állíthatjuk be. (1 pont az *inch* 72. része).

Az alábbi példában létrehozott betűtípus csak programozott rajzolás esetén érvényesül, mivel a vezérlők által használható fontneveket az operációs rendszer erősen korlátozza, bár a stílus és a betűméret megváltoztatható:

```
Font nf = new Font("Times New Roman", Font.BOLD|Font.ITALIC, 23)
```

Természetesen egy sor metódus is segíti a betűtípus osztály alkalmazását.

10.1.2 Az alkalmazás ablaka, a Frame osztály

Alkalmazások készítésekor az ablakosztályunkat *AWT* összetevők esetén a ***Frame*** osztálytól, *Swing* összetevők használata esetén pedig a ***JFrame*** osztálytól kell származtatni.

Az ablakhoz események tartoznak, melyek kezeléséről több lépésben kell gondoskodnunk:

- Implementálnunk kell a szükséges eseményfigyelő interfészt a ***java.awt.event*** csomagból, például a ***WindowListener***.
- Az ablakosztályban (például a konstruktorban) kérnünk kell az ablak eseményeinek figyelését, például az ***addWindowListener(this);*** hívással. (Minden eseményfigyelőhöz saját regisztrációs metódus tartozik.) (Szüksége esetén a figyelés felfüggeszthető a ***removeWindowListener(this);*** hívással.)
- Végül ki kell dolgoznunk az interfészben előírt absztrakt metódusokat. Ha valamelyikre nincs szükségünk, akkor annak törzsét üresen hagyjuk *{ }*.

A különböző komponensek, perifériák eseményfigyeléséhez szükséges ismereteket az F1. függelékben foglaltuk össze. A következőkben csak a ***WindowListener*** interfész metódusait vesszük sorra:

```
public void windowOpened(WindowEvent e) { }     // megnyitották
public void windowClosing(WindowEvent e) {      // lezárása folyamatban
     System.exit(0);
}
public void windowClosed(WindowEvent e) { }     // lezárták
public void windowIconified(WindowEvent e) { } // ikonméretűvé tették
public void windowDeiconified(WindowEvent e){ }// normál méretű lett
public void windowActivated(WindowEvent e) { } // aktív lett
public void windowDeactivated(WindowEvent e){ }// már nem aktív
```

A fenti eljárás általánosan alkalmazható, azonban sok üres metódust kell megvalósítanunk. Az események kezelésének másik módja, amikor nem az ablakosztály implementálja a figyelő interfészeket, hanem ún. adapterosztályt készítünk erre a célra a ***java.awt.event*** csomag megfelelő adapterosztályától származtatva. A fenti események kezelése esetén ez az osztály a ***WindowAdapter***.

Mivel a ***WindowAdapter*** osztály már megvalósította a ***WindowListener*** interfész (valójában az összes ablakhoz kapcsolódó figyelőinterfész) metódusait, nekünk csak a számunkra érdekes metódusokat kell felülbírálni:

```
class SajatWindowAdapter extends WindowAdapter
{
      public void windowClosing(WindowEvent event) {
          System.exit(0);
      }
}
```

Természetesen a figyelés megkezdése is módosul az ablakosztály valamelyik metódusában:

```
      addWindowListener(new SajatWindowAdapter ());
```

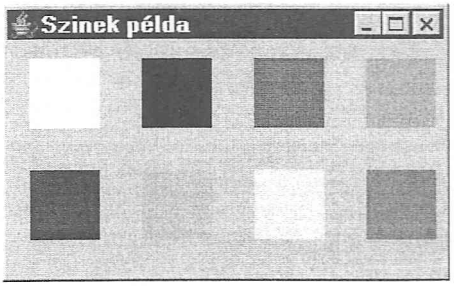
Készítsünk alkalmazást, amely különböző RGB-színű téglalapokat jelenít meg! (*AWT\Szinek*)

A megjelenítéshez grafikus metódusokat használunk a ***paint()*** metódusban, melyeket részleteiben a következő fejezet ismerteti.

```
import java.awt.*;
import java.awt.event.*;
public class Szinek extends Frame implements WindowListener
{
    public Szinek() {
        super("Szinek példa");
        setSize(320,200);
        setBackground(new Color(192,192,192));
        addWindowListener(this);
    }

    public void windowClosing(WindowEvent e)  {
        System.exit(0);
    }
    public void windowOpened(WindowEvent e) { }
    public void windowClosed(WindowEvent e) { }
    public void windowIconified(WindowEvent e) { }
    public void windowDeiconified(WindowEvent e) { }
    public void windowActivated(WindowEvent e) { }
    public void windowDeactivated(WindowEvent e) { }

    public void paint(Graphics g) {
        Rectangle r=getBounds();
        g.setColor(new Color(255,255,255)); // fehér
        g.fillRect(20,40,50,50);
        g.setColor(new Color(0,0,0));       // fekete
        g.fillRect(100,40,50,50);
        g.setColor(new Color(255,0,0));     // piros
        g.fillRect(180,40,50,50);
        g.setColor(new Color(0,255,0));     // zöld
        g.fillRect(260,40,50,50);
        g.setColor(new Color(0,0,255));     // kék
        g.fillRect(20,120,50,50);
        g.setColor(new Color(0,255,255));   // türkiz
        g.fillRect(100,120,50,50);
```

```
        g.setColor(new Color(255,255,0));   // sárga
        g.fillRect(180,120,50,50);
        g.setColor(new Color(255,0,255));   // lila
        g.fillRect(260,120,50,50);
    }

    // a tesztelő főprogram
    public static void main(String[] args) {
        Szinek mw= new Szinek();
        mw.setVisible(true);
    }
}
```

Az alkalmazás futásának eredménye:

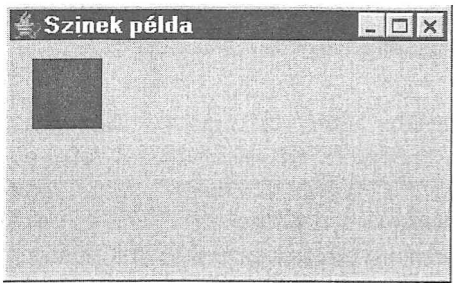


Írjunk programot, amely az ablak háttérszínét állítja! (*AWT\Hatter*)

```
import java.awt.*;
import java.awt.event.*;
public class Hatter extends Frame implements WindowListener
{
    public Hatter() {
        super("Szinek példa");
        setSize(320,200);
        setBackground(Color.cyan);
        addWindowListener(this);
    }

    public void windowClosing(WindowEvent e) {
       System.exit(0);
    }
    public void windowOpened(WindowEvent e) { }
    public void windowClosed(WindowEvent e) { }
    public void windowIconified(WindowEvent e) { }
    public void windowDeiconified(WindowEvent e) { }
    public void windowActivated(WindowEvent e) { }
    public void windowDeactivated(WindowEvent e) { }
    public void paint(Graphics g) {
        Rectangle r=getBounds();
        g.setColor(Color.blue);
        g.fillRect(20,40,50,50);
    }
```

```
// Tesztelő főprogram
    public static void main(String[] args) {
        Hatter mw= new Hatter();
        mw.setVisible(true);
    }
}
```

Az alkalmazás futásának eredménye:



10.2 Az AWT komponensek kezelése

Az *AWT* komponensek kezeléséhez az *AWT* osztályok importálása szükséges:

```
import java.awt.*;
```

Az alkalmazás ablakának osztályát a ***Frame*** osztályból származtatjuk. A ***main()*** metódusban létrehozzuk az ablakosztály példányát, és a ***setVisible()*** hívással megjelenítjük az ablakot. Az ablakosztályban adatmezőként különböző vezérlőelemeket helyezünk el, melyeket általában a konstruktorban véglegesítünk. Az ablaknak felirata is lehet, melyet az ablakosztály konstruktorának továbbítunk, és ez átadja a ***Frame*** osztály konstruktorának.

A következőkben áttekintjük a gyakran előforduló vezérlőkhöz kapcsolódó ismereteket.

10.2.1 Alapvezérlők

A mintafeladatokban használt alapvezérlők fontosabb konstruktorait és metódusait táblázatban foglaltuk össze. Előtte azonban meg kell ismerkednünk néhány segédosztállyal:

A ***Dimension*** osztály objektumai szélesség (***width***) és magasság (***height***) adatokat tárolnak nyilvános adatmezőkben. Ezzel szemben a ***Point*** osztály segítségével pozíciót (***x*** és ***y***) adhatunk meg. Az előző két osztály tudását egyesíti magában a ***Rectangle*** osztály (***width***, ***height***, ***x***, ***y***). Mindhárom osztály különböző konstruktorokkal, és elérési metódusokkal rendelkezik.

A vezérlők többsége a ***Component*** osztálytól származik, így öröklik annak metódusait. Az alapvezérlők közös metódusait az alábbiak szerint csoportosíthatjuk:

Elhelyezkedés és méret

A vezérlők valamilyen tárolóban helyezkednek el, így a pozíciójukat és a méreteiket annak bal felső sarkához rögzített koordináta-rendszerben értelmezzük. (Az x-tengely jobbra, az y-tengely pedig lefelé mutat.) Amennyiben nem alkalmazunk elrendezésmenedzsert *setLayout(**null**);* (lásd a következő alfejezetet), akkor a pozíciók és méretek beállítását magunk végezhetjük el az alábbi metódusokkal. A lekérdezés természetesen mindig használható.

`void setLocation(int x, int y)`
`void setLocation(Point p)`
`Point getLocation(Point rv)`
`Point getLocation()`
`int getY()`
`int getX()`
	pozíció beállítása és lekérdezése

`void setSize(Dimension d)`
`void setSize(int width, int height)`
`Dimension getSize(Dimension rv)`
`Dimension getSize()`
`int getWidth()`
`int getHeight()`
	méretek beállítása és lekérése,

`void setBounds(Rectangle r)`
`void setBounds(int x,int y,`
`int width,int height)`
`Rectangle getBounds(Rectangle rv)`
`Rectangle getBounds()`
	határok (pozíció és méretek együtt) kezelése,

`boolean contains(Point p)` `boolean contains(int x, int y)`	ellenőrzi, hogy a vezérlő bal-felső sarkához mért távolságok belső pontot határoznak-e meg.

Megjelenés

`void setForeground(Color c)` `Color getForeground()`	az előtér (általában a szöveg) színe,
`void setBackground(Color c)` `Color getBackground()`	háttérszín,
`void setFont(Font f)` `Font getFont()`	a szöveg betűkészlete,
`void setVisible(boolean b)` `boolean isVisible()`	legyen látható (*b*=**true**), vagy sem, látható-e?
`void setEnabled(boolean b)` `boolean isEnabled()`	legyen engedélyezett (*b*=**true**), engedélyezett-e?

További hasznos Component metódusok

`Container getParent()`	a vezérlőt tároló ablak objektuma,
`Toolkit getToolkit()`	a vezérlőhöz kapcsolódó eszközök objektuma,
void `add(PopupMenu popup)` **void** `remove(MenuComponent popup)`	felbukkanó menü beállítása és törlése,

`addComponentListener(ComponentListener l)`
`addFocusListener(FocusListener l)`
`addKeyListener(KeyListener l)`
`addMouseListener(MouseListener l)`
`addMouseMotionListener(MouseMotionListener l)`
`addMouseWheelListener(MouseWheelListener l)`
	eseményfigyelők hozzárendelése a vezérlőhöz (*add...*), amelyek lekérdezhetők (*get...*) és le is vehetők (*remove...*),

void `setCursor(Cursor cursor)` `Cursor getCursor()`	az egérmutató beállítása, lekérdezése,
void `repaint()` **void** `repaint(int x, int y, int width, int height)`	a teljes komponens, illetve egy része újrafestésének kérése.

A közös metódusok után sorra vesszük a legfontosabb vezérlőket, feltüntetve a konstruktoraikat, a **final** adatmezőiket és a jellemző metódusaikat. Megjegyezzük, hogy a legtöbb beállító (*set...*) metódusnak létezik lekérdező (*get...*) párja is.

Button *nyomógomb*

`Button()`

`Button(String` *felirat*`)`

void `setLabel(String` *felirat*`)` — a nyomógomb feliratának megváltoztatása.

Checkbox *jelölő- / választógomb*

Az *AWT* nem definiál külön osztályt a jelölőnégyzetek és a választógombok számára. A ***Checkbox*** osztályt alap helyzetben jelölőgombok létrehozására használjuk, melyek közül egyszerre bármennyi bejelölhető. Ha azonban a negyedik konstruktorral az új ***Checkbox*** objektumot egy ***CheckboxGroup*** csoporthoz rendeljük, akkor csak egy gomb lesz kiválasztható, így választógombként működnek. (A *Swing* megkülönbözteti ezt a két vezérlőt a ***JCheckBox*** és a ***JRadioButton*** osztályokkal).

`Checkbox()`

`Checkbox(String` *címke*`)`

`Checkbox(String` *címke*`,` **boolean** *b*`)` — ha *b* állapot értéke **true**, kiválasztott lesz,

`Checkbox(String` *címke*`,` **boolean** *b*`, CheckboxGroup` *csoport*`)` — adott feliratú és állapotú gomb hozzáadása a választógombok csoportjához,

void `setLabel(String` *címke*`)` — a gomb melletti felirat megváltoztatása,

void `setState(boolean` *állapot*`)` — a jelölt állapot módosítása,

void `addItemListener(ItemListener l)` — az állapotváltozás figyelőjének beállítása.

CheckboxGroup *választógombcsoport*

```
CheckboxGroup ()
void setSelectedCheckbox(
                    Checkbox box)
Checkbox getSelectedCheckbox()
```

a kijelölt választógomb beállítása,
a kijelölt választógomb lekérdezése.

Label *statikus szövegmező (címke)*

```
Label()
Label(String szöveg)
Label(String szöveg, int igazítás)
LEFT, RIGHT, CENTER
void setText(String szöveg)
void setAlignment(int igazítás)
```

a szöveg *igazításának* lehetséges értékei,
a címke szövegének beállítása,
szöveg igazítása.

TextField *egysoros szövegmező*

```
TextField()
TextField(String szöveg)
TextField(int oszlopszám)
TextField(String szöveg, int osz)
void setText(String szöveg)
void setColumns(int oszlopszám)
void setEchoChar(char ech)
```

oszlopszám ≈ karakterszám,
(*osz* – oszlopok száma),
szövegmező tartalmának megváltoztatása,
az oszlopszám átállítása,
ech a szöveg beírásakor megjelenő karakter.

TextArea *többsoros, szerkeszthető szövegmező*

```
TextArea()
TextArea(String szöveg)
TextArea(int sorok, int oszlopok)
TextArea(String szöveg, int sorok, int oszlopok)
TextArea(String szöveg, int sorok, int oszlopok, int görgetők)
SCROLLBARS_BOTH, SCROLLBARS_NONE,
SCROLLBARS_VERTICAL_ONLY,
SCROLLBARS_HORIZONTAL_ONLY
void setText(String szöveg)
void setColumns(int oszlopszám)
void setRows(int sorszám)
void append(String szöveg)
void insert(String szöveg, int p)
void replaceRange(String szöveg,
                  int kp, int vp)
```

a *görgetők* lehetséges értékei,
szövegmező tartalmának megváltoztatása,
az oszlopszám átállítása,
a sorok számának beállítása,
szöveg hozzáfűzése,
szöveg beszúrása, adott *p* pozíciótól,
szöveg cseréje adott *kp* és *vp* pozíciók között.

List *görgethető listaablak*

```
List()
List(int sorok_száma)
List(int sorok_száma, boolean b)

void add(String szöveg)
void add(String szöveg, int index)
```

négy látható sort tartalmazó lista jön létre,
adott számú látható sorral jön létre a lista,
ha *b* értéke **true**, több elemet is kiválaszthatunk egyszerre,
elem felvétele a lista végére,
elem beszúrása az adott indexű pozícióba.

```
String getItem(int index)                adott indexű elem lekérdezése,
String[] getItems()                      az összes elem lekérdezése,
int getItemCount()                       a listaelemek számával tér vissza,
void remove(int index)                   adott sorszámú elem törlése,
void remove(String szöveg)               adott tartalmú elem törlése,
void removeAll()                         minden elem törlése,
void replaceItem(String szöveg,          adott sorszámú elem cseréje,
                 int index)
void makeVisible(int index)              láthatóvá teszi az adott indexű elemet,
void setMultipleMode(boolean b)          váltás az egyszeres és a többszörös kiválasztási mód között,

boolean isIndexSelected(int index)       megmondja, ha a megadott elem kiválasztott,
void select(int index)                   adott sorszámú elem kiválasztása,
void deselect(int index)                 adott sorszámú elem jelöltségének törlése,
int getSelectedIndex()                   a kijelölt elem(ek) indexe(i)nek lekérdezés,
int[] getSelectedIndexes()
String getSelectedItem()                 a kijelölt elem(ek) lekérése,
String[] getSelectedItems()
void addItemListener(                    az elemesemények figyelőjének beállítása.
                 ItemListener l)
```

Choice *legördülő lista*

A ***Choice*** vezérlőelem korlátozott képességű, egyszeres kiválasztású listaként programozható, mint ahogy ez a metódusokból is látható.

```
Choice()
void add(String szöveg)                  elem felvétele a lista végére,
String getItem(int index)                adott indexű elem lekérdezése,
void remove(int index)                   adott sorszámú elem törlése,
void remove(String szöveg)               adott tartalmú elem törlése,
void removeAll()                         minden elem törlése,
void select(int index)                   adott sorszámú elem kiválasztása,
void select(String szöveg)               adott tartalmú elem kiválasztása,
int getItemCount()                       a listaelemek számával tér vissza,
int getSelectedIndex()                   a kijelölt elem indexének lekérdezése,
String getSelectedItem()                 a kijelölt elem lekérése,
void addItemListener(                    az elemesemények figyelőjének beállítása.
                 ItemListener l)
```

Scrollbar *görgetősáv, csúszka*

A görgetősávot értékek kiválasztására használjuk, két nyíl között elhelyezett kis gomb mozgatásával. Több komponens rendelkezik beépített görgetősávval, mint például a listaablak, a többsoros szövegmező stb.

```
Scrollbar()
Scrollbar(int irány)
Scrollbar(int irány, int érték, int láthatóMéret,int min int max)
HORIZONTAL, VERTICAL                     a görgető irányának lehetséges értékei,
```

```
void setOrientation(int irány)
void setValue(int újÉrték)
void setVisibleAmount(int újMéret)
void setMaximum(int újMaximum)
void setMinimum(int újMinimum)
void setUnitIncrement(int v)
void setBlockIncrement(int v)
int getValue()
int getMaximum()
int getMinimum()
void addAdjustmentListener(
            AdjustmentListener l)
```

a görgető működési paramétereinek beállítása,

léptetési adatok beállítása,

az aktuális állás lekérdezése,
a beállított értékhatárok lekérése,

a görgetőesemények figyelőjének beállítása.

A menükészítés osztályai:

Az ablakunk menüsorát a ***MenuBar*** osztály objektuma valósítja meg, amelyhez lenyíló menüket csatlakoztatunk (***Menu*** osztály). Mindkét osztály listajellegű műveleteket tartalmaz, hiszen a menüsor objektum menük listája, míg a menük a menüpontok (***MenuItem***) listája. A metódusok ismertetése helyett állítsunk elő egy egyszerű menüt!

```
MenuBar menuSor = new MenuBar();
Menu szinekMenu = new Menu();
MenuItem pirosMenupont = new MenuItem();
MenuItem zoldMenupont = new MenuItem();
MenuItem kekMenupont = new MenuItem();

szinekMenu.setLabel("Színek");
pirosMenupont.setLabel("Piros");
szinekMenu.add(pirosMenupont);
szinekMenu.addSeparator();

zoldMenupont.setLabel("Zöld");
szinekMenu.add(zoldMenupont);
szinekMenu.addSeparator();

kekMenupont.setLabel("Kék");
szinekMenu.add(kekMenupont);
menuSor.add(szinekMenu);
this.setMenuBar(menuSor);
```

10.2.2 A komponensek elrendezése

A felhasználói felület sarkalatos pontja a komponensek megfelelő elhelyezése a tárolókban. Egy megoldásra már utaltunk, amikor magunk állítjuk be a vezérlő pozícióját és méreteit. Ez grafikus fejlesztői környezet nélkül elég fáradtságos tevékenység:

```
setLayout(null);
Button btn = new Button("Küldés");
btn.setBounds(50, 50, 100, 30);
this.add(btn);
```

Hogy a *Java API* megkímélje a programozót a fenti fáradtságos programrészektől, az *AWT* két elrendezés-kezelő interfészt (***LayoutManager, LayoutManager2***), és egy sor

megvalósítást definiál. Ezek közül a legismertebbek a ***BorderLayout***, a ***CardLayout***, a ***FlowLayout***, a ***GridBagLayout*** és a ***GridLayout***. Az elrendezés-kezelők a beépített algoritmusuk alapján határozzák meg a vezérlők pozícióját és méreteit.

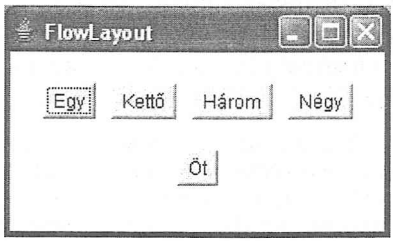
FlowLayout

Az ***add(komponens)*** hívással a tárolóhoz adott vezérlőket sorban, egymás mellé helyezi. Ha az ablak szélessége nem enged meg többet egy sorban, akkor a további komponensek a következő sorba kerülnek.

```
setLayout(new FlowLayout());
setLayout(new FlowLayout(igazítás));
setLayout(new FlowLayout(igazítás, vízszintestáv, függőlegestáv));
```

Az igazítás az alábbi előre definiált konstansokkal adható meg: *FlowLayout.LEFT*, *FlowLayout.CENTER*, *FlowLayout.RIGHT*.

```
setLayout(new FlowLayout(
            FlowLayout.CENTER,10,20));
add(btn1);  // Egy
add(btn2);  // Kettő
add(btn3);  // Három
add(btn4);  // Négy
add(btn5);  // Öt
```



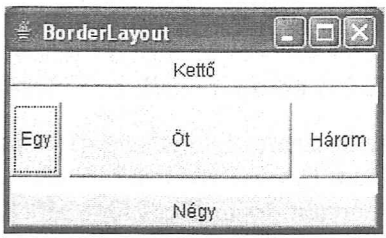
BorderLayout

Segítségével a komponenseket ötféleképpen helyezhetjük el. A tároló bal oldalára (*West*), tetejére (*North*), jobb oldalára (*East*), aljára (*South*), illetve közepére (*Central*).

```
setLayout(new BorderLayout());
setLayout(new BorderLayout(vízszintestáv, függőlegestáv));
```

Ezzel a módszerrel legfeljebb öt komponens helyezhető el az ablakban.

```
setLayout(new BorderLayout(5, 10));
add(btn1, BorderLayout.WEST);
add(btn2, BorderLayout.NORTH);
add(btn3, BorderLayout.EAST);
add(btn4, BorderLayout.SOUTH);
add(btn5, BorderLayout.CENTER);
```



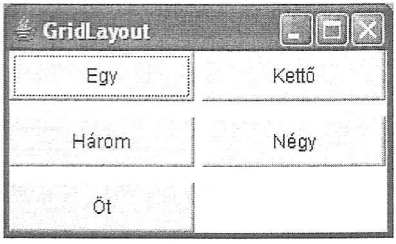
GridLayout

A kezelő a konstruktorban megadott sor- és oszlopszámú táblázatba rendezve helyezi el a komponenseket. Először feltölti az első sort, majd a második és így tovább.

```
setLayout(new GridLayout());
setLayout(new GridLayout(sorok, oszlopok));
setLayout(new GridLayout(sorok, oszlopok, vízsztáv, függtáv));
```

Amennyiben a táblázat elemeinél több vezérlőt helyezünk fel, az elrendezés-kezelő úgy bírálja felül a megadott méreteket, hogy minden komponens megjelenjen.

```
setLayout(new GridLayout(3, 2, 5, 10));
add(btn1);
add(btn2);
add(btn3);
add(btn4);
add(btn5);
```



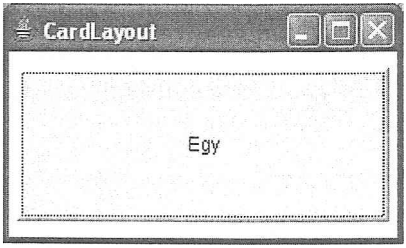
CardLayout

A ***CardLayout*** a külön névvel azonosított komponenseket kártyapakliszerűen egymásra helyezi.

```
CardLayout clay = new CardLayout();
CardLayout clay = new CardLayout(vízszintestáv, függőlegestáv);
setLayout(clay);
```

A vezérlők közül az osztály metódusaival választhatjuk ki azt, amelyik felülre kerül, és így elérhetővé válik a felhasználó számára. Az *AWT\CardLayout* példában az eddig használt öt gomb egymást lapozzák felülre – itt csak a lényegesebb részeket közöljük.

```
CardLayout clay = new CardLayout(5,10);
// ...
setLayout(clay);
add(btn1, "1");
add(btn2, "2");
add(btn3, "3");
add(btn4, "4");
add(btn5, "5");
```



A példaprogram gombnyomás eseményeket kezelő metódusában láthatók a ***BorderLayout*** osztály lapváltó metódusai:

```
public void actionPerformed(ActionEvent event) {
      Object object = event.getSource();
      if (object == btn1)
            clay.show(this, "3");// lapozás a "3" nevű lapra
      if (object == btn3)
            clay.next(this);      // következő lap
      if (object == btn4)
```

```
            clay.show(this, "2");// lapozás a "2" nevű lapra
        if (object == btn2)
            clay.last(this);      // utolsó lap
        if (object == btn5)
            clay.first(this);     // első lap
}
```

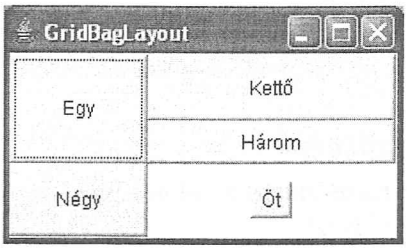
GridBagLayout

Az elrendezés-kezelők egyik legbonyolultabb változata, melyet emiatt elég ritkán használunk. A ***GridBagLayout*** valójában egy olyan táblázat, ahol elég rugalmasan lehet beállítani az egyes elemek pozícióját és méretét.

```
GridBagLayout gblay = new GridBagLayout();
```

Az *AWT\GridBagLayout* példában szemléltetjük a szükséges lépéseket.

```
GridBagLayout gb = new GridBagLayout();
setLayout(gb);
GridBagConstraints gbc =
            new GridBagConstraints();
gbc.fill=GridBagConstraints.BOTH;
gbc.anchor=GridBagConstraints.CENTER;
eloirasok(gbc,0,0, 1,2, 30, 0);
add(btn1,gbc);
eloirasok(gbc,1,0, 1,1, 70,40);
add(btn2,gbc);
eloirasok(gbc,1,1, 1,1, 0, 10);
add(btn3,gbc);
eloirasok(gbc,0,2, 1,1, 0, 50);
add(btn4,gbc);
gbc.fill=GridBagConstraints.NONE;
gbc.anchor=GridBagConstraints.CENTER;
eloirasok(gbc,1,2, 1,1, 0, 0);
add(btn5,gbc)
```



Az *eloirasok()* metódusban beállítjuk a ***GridBagConstraints*** típusú objektum adatmezőit:

```
private void eloirasok(GridBagConstraints gbc, int gx, int gy,
                                int gw, int gh, int wx, int wy) {
        gbc.gridx=gx;          // a cella oszlopának sorszáma
        gbc.gridy=gy;          // a cella sorának sorszáma
        gbc.gridwidth=gw;      // a vezérlő cellája hány sort foglal el
        gbc.gridheight=gh;     // a vezérlő cellája hány oszlopot foglal el
        gbc.weightx=wx;        // a vezérlő szélességének aránya
        gbc.weighty=wy;        // a vezérlő magasságának aránya
}
```

10.2.3 Események kezelése

A grafikus felületű operációs rendszerek alatt a programok futását a különböző perifériáktól (egér, billentyűzet), illetve a grafikus felület elemeitől érkező események vezérlik. A programozás során a számunkra lényeges eseményekre reagálunk, feldolgozzuk azokat.

Java-ban a feldolgozást ún. eseményfigyelő interfészek segítik, melyek által előírt metódusokban végezhetjük el az események kezelését. Az F1. függelékben összefoglaltuk a leggyakrabban előforduló események kezeléséhez szükséges ismereteket, most csak röviden áttekintjük a legfontosabb lépéseket. Az AWT-események az ***AWTEvent*** osztálytól (***java.awt.event*** csomag) származnak, melynek őse a ***java.util.EventObject*** osztály, utódai pedig:

ActionEvent
AdjustmentEvent
ItemEvent
TextEvent
ComponentEvent — ***ContainerEvent***
FocusEvent
PaintEvent
WindowEvent
InputEvent — ***KeyEvent***
MouseEvent

Tekintsük át, hogy a különböző *AWT*-komponensek használata esetén milyen események keletkeznek!

Esemény	*Komponensek*
WindowEvent	***Frame, Dialog, FileDialog, Window***
TextEvent	***TextArea, TextField***
ActionEvent	***Button, List, TextField, MenuItem***
ItemEvent	***CheckBox, Choice, List***
AdjustmentEvent	***Scrollbar***

A 10.1.2. fejezetben az ablakesemények kezelésének lépései mindig alkalmazhatók, azonban komponensek esetén egy adott eseménynek több forrása is lehet. Az eseménykezelők paramétere a megfelelő esemény, melynek ***getSource()*** metódusa azonosítja az esemény forrását:

```
public void actionPerformed(ActionEvent event) {
    Object object = event.getSource();
    if (object == komponens1)
        // tevékenység1
    else if (object == komponens2)
        // tevékenység2
    // ....
}
```

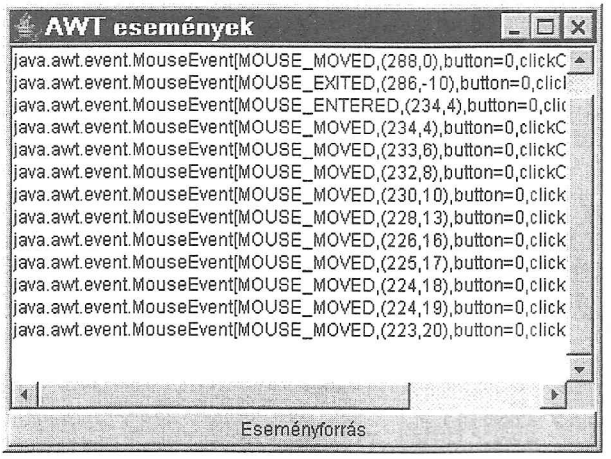
A példaprogramjainkban mindenféle eseményeket kezelünk, így azok áttanulmányozása segítheti az alapvető lépések jobb megértését.

Tervezzünk alkalmazást, amely megjeleníti az egér-, a billentyűzet- és a fókuszeseményeket! (*AWT\Esemenyek*)

```
import java.awt.*;
import java.awt.event.*;
public class HardverEsemenyek extends Frame {
      public HardverEsemenyek () {
            super("AWT események");
            add((out = new TextArea()));
            Button button = new Button("Eseményforrás");
            add(button, "South");
            // saját eseményfigyelő létrehozása
            SajatFigyelo o1 = new SajatFigyelo();
            button.addKeyListener(o1);
            button.addMouseListener(o1);
            button.addMouseMotionListener(o1);
            button.addFocusListener(o1);
            Ablakkezelo aAblakkezelo = new Ablakkezelo();
            this.addWindowListener(aAblakkezelo);
            setSize(400, 300);
            setVisible(true);
      }

      public static void main(String[] args) {
            new HardverEsemenyek();
      }

      // az eseményinformációkat itt jelenítjük meg
      TextArea out;

// a saját eseményfigyelő belső osztály
class SajatFigyelo implements MouseListener, KeyListener,
      MouseMotionListener, MouseWheelListener, FocusListener {
      public void mouseClicked(MouseEvent e)  // egérkattintás
      {     out.append(e.toString() + "\n");
      }
      public void mousePressed(MouseEvent e)  // egérgomb lenyomása
      {     out.append(e.toString() + "\n");
      }
      public void mouseReleased(MouseEvent e) // egérgomb felengedése
      {     out.append(e.toString() + "\n");
      }
      public void mouseEntered(MouseEvent e)  // az egér belépett
      {     out.append(e.toString() + "\n");
      }
      public void mouseExited(MouseEvent e)  // az egér kiment
      {     out.append(e.toString() + "\n");
      }
      public void keyTyped(KeyEvent e)       // karakterbevitel
      {     out.append(e.toString() + "\n");
      }
```

```
        public void keyPressed(KeyEvent e)  // billentyű lenyomása
        {       out.append(e.toString() + "\n");
        }
        public void keyReleased(KeyEvent e) // billentyű felengedése
        {       out.append(e.toString() + "\n");
        }
        // egérmozgatás lenyomott bal gombbal
        public void mouseDragged(MouseEvent e)
        {       out.append(e.toString() + "\n");
        }
        public void mouseMoved(MouseEvent e)  // egérmozgatás
        {       out.append(e.toString() + "\n");
        }
        public void focusGained(FocusEvent e)  // fókuszba került
        {       out.append(e.toString() + "\n");
        }
        public void focusLost(FocusEvent e)  // elveszítette a fókuszt
        {       out.append(e.toString() + "\n");
        }
        public void mouseWheelMoved(MouseWheelEvent e) // egér görgetése
        {       out.append(e.toString() + "\n");
        }
}

        // A kilépés biztosítása
        class Ablakkezelo extends WindowAdapter
        {
                public void windowClosing(WindowEvent event)
                {
                        System.exit(0);
                }
        }
}
```

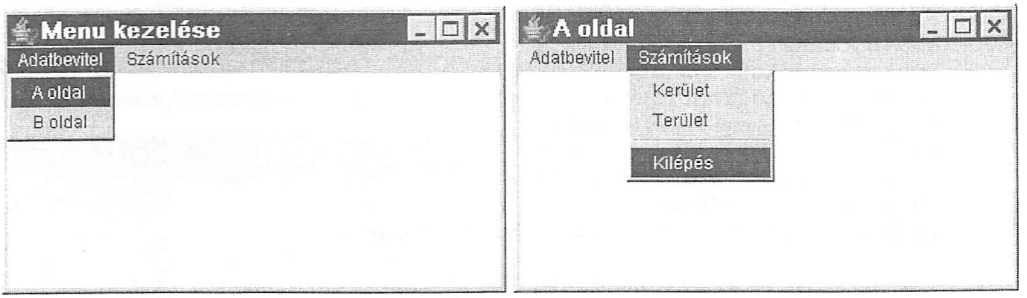
Az alkalmazás futásának eredménye:



Tervezzünk alkalmazást, amely az alkalmazás menüjét alakítja ki, és kezeli a menühasználat során keletkező eseményeket! (*AWT\Menu1*)

```
import java.awt.*;
import java.awt.event.*;

public class Menu1pr extends Frame implements ActionListener
{
    MenuBar menusor = new MenuBar();

    Menu menu1 = new Menu("Adatbevitel");
    MenuItem almenu1 = new MenuItem("A oldal");
    MenuItem almenu2 = new MenuItem("B oldal");
    Menu menu2 = new Menu("Számítások");
    MenuItem almenu3 = new MenuItem("Kerület");
    MenuItem almenu4 = new MenuItem("Terület");
    MenuItem almenu5 = new MenuItem("Kilépés");

    public Menu1pr(String s) {
       super(s);
       menu1.add(almenu1).addActionListener(this);
       menu1.add(almenu2).addActionListener(this);
       menusor.add(menu1);
       menu2.add(almenu3).addActionListener(this);
       menu2.add(almenu4).addActionListener(this);
       menu2.addSeparator();
       menu2.add(almenu5).addActionListener(this);
       menusor.add(menu2);

       this.setSize(350,200);
       this.setVisible(true);
       this.setMenuBar(menusor);

       // a kilépéshez névtelen adapterosztályt definiálunk
       this.addWindowListener( new WindowAdapter()
        {
            public void windowClosing(WindowEvent esemeny) {
                System.exit(0);
            }
        }
    );
  }

  // a menüpontok eseményeinek kezelése, de csak a Kilépés
  // menüpontot vizsgáljuk
  public void actionPerformed(ActionEvent e) {
       this.setTitle(e.getActionCommand());
       if (e.getActionCommand()=="Kilépés")
         this.dispose();
  }

  public static void main(String argv[]) {
      Menu1pr ablak = new Menu1pr("Menu kezelése");
  }
}
```

Az alkalmazás futásának eredménye:



10.2.4 Komponensek használata alkalmazásokban

A következőkben a különböző *AWT*-komponensek alkalmazására mutatunk példákat.

Készítsünk alkalmazást, amely választógombokat használ gömb adatainak számításának vezérlésére (*AWT\Checkbox1*)

```
import java.awt.*;
import java.awt.event.*;
public class Check1pr extends Frame implements ActionListener,
                                                ItemListener
{
   Label label1 = new Label("Gömb sugara");
   Label label2 = new Label("   Felszine");
   Label label3 = new Label("  Térfogata");
   TextField edit1 = new TextField("0",10);
   TextField edit2 = new TextField("0",10);
   TextField edit3 = new TextField("0",10);
   CheckboxGroup muvelet = new CheckboxGroup();
   Checkbox felszin = new Checkbox("Felszin",muvelet,false);
   Checkbox terfogat = new Checkbox("Térfogat",muvelet,false);
   Checkbox terffelsz = new Checkbox("Mindkettő",muvelet,false);
   Button button1 = new Button("SZÁMOL");
   Button button2 = new Button(" TÖRÖL");

   public Check1pr(String s) {
      super(s);
      this.setSize(220,250);
      setLayout(new FlowLayout(FlowLayout.RIGHT));
      setBackground(Color.cyan);
      setForeground(Color.blue);
      add(label1);
      add(edit1);
      add(label2);
      add(edit2);
      add(label3);
      add(edit3);
      add(felszin);
```

```
        add(terfogat);
        add(terffelsz);
        add(button1);
        add(button2);
        button1.addActionListener(this);
        button2.addActionListener(this);
        felszin.addItemListener(this);
        terfogat.addItemListener(this);
        terffelsz.addItemListener(this);
        this.setVisible(true);
        this.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent esemeny) {
                System.exit(0);
            }
        }
    );
}

public void actionPerformed(ActionEvent e) {
    double felsz, terf;
    if (e.getSource()==button1) { // SZÁMOL gomb megnyomása
        double r = new Double(edit1.getText()).doubleValue();
        if(muvelet.getSelectedCheckbox() == felszin) {
            felsz = 4 * Math.pow(r,2) * Math.PI;
            edit2.setText(""+ felsz);
        }
        if(muvelet.getSelectedCheckbox() == terfogat) {
            terf = 4* Math.pow( r, 3) * Math.PI/3;
            edit3.setText(""+ terf);
         }
        if(muvelet.getSelectedCheckbox() == terffelsz) {
            felsz = 4 * Math.pow(r,2) * Math.PI;
            edit2.setText(""+ felsz);
            terf = 4* Math.pow( r, 3) * Math.PI/3;
            edit3.setText(""+ terf);
         }
    }
    if (e.getSource()==button2) { // TÖRÖL gomb lenyomása
        edit1.setText("0");
        edit2.setText("0");
        edit3.setText("0");
    }
}

public void itemStateChanged(ItemEvent e) {
        double ter, ker;  // a választógombok állapotváltozása
        double r = new Double(edit1.getText()).doubleValue();
        if (e.getSource()==  felszin && e.getStateChange()==e.SELECTED)
            edit2.setText("0");edit3.setText("0");
        if (e.getSource()== terfogat && e.getStateChange()==e.SELECTED)
            edit2.setText("0");edit3.setText("0");
        if (e.getSource()==terffelsz && e.getStateChange()==e.SELECTED)
            edit2.setText("0");edit3.setText("0");
}
```

```
    public static void main(String argv[]) {
        Check1pr ablak = new Check1pr("CheckBox kezelése");
    }
}
```

Az alkalmazás futásának eredményei:

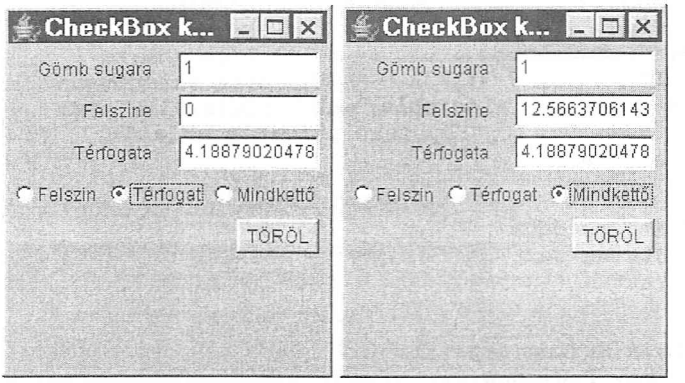
Tervezzünk alkalmazást, amely a gömb felszínének és térfogatának számításához választógombokat használ! (*AWT\Checkbox2*)

```
import java.awt.*;
import java.awt.event.*;
public class Check2pr extends Frame implements ActionListener,
                                                ItemListener
{
   Label label1 = new Label("Gömb sugara");
   Label label2 = new Label("   Felszine");
   Label label3 = new Label("  Térfogata");
   TextField edit1 = new TextField("0",10);
   TextField edit2 = new TextField("0",10);
   TextField edit3 = new TextField("0",10);
   CheckboxGroup muvelet = new CheckboxGroup();
   Checkbox felszin = new Checkbox("Felszin",muvelet,false);
   Checkbox terfogat = new Checkbox("Térfogat",muvelet,false);
   Checkbox terffelsz = new Checkbox("Mindkettő",muvelet,false);
   Button button1 = new Button("TÖRÖL");

   public Check2pr(String s) {
      super(s);
      this.setSize(220,250);
      setLayout(new FlowLayout(FlowLayout.RIGHT));
      setBackground(Color.cyan);
      setForeground(Color.blue);
      add(label1);
      add(edit1);
      add(label2);
      add(edit2);
      add(label3);
```

```
        add(edit3);
        add(felszin);
        add(terfogat);
        add(terffelsz);
        add(button1);
        // eseményfigyelők beállítása
        button1.addActionListener(this);
        felszin.addItemListener(this);
        terfogat.addItemListener(this);
        terffelsz.addItemListener(this);
        this.setVisible(true);
        this.addWindowListener(
        new WindowAdapter()
        {
          public void windowClosing(WindowEvent esemeny) {
             System.exit(0);
          }
        }
    );
  }

  public void actionPerformed(ActionEvent e) {
     if (e.getSource()==button1) { // a TÖRÖL gomb megnyomása
        edit1.setText("0");
        edit2.setText("0");
        edit3.setText("0");
     }
  }

// A választógombok állapotváltozásainak kezelése
  public void itemStateChanged(ItemEvent e) {
     double felsz, terf;
     double r = new Double(edit1.getText()).doubleValue();
     // ha a Felszin gomb kijelölt
     if (e.getSource() == felszin && e.getStateChange()==e.SELECTED) {
        edit3.setText("0");
        felsz = 4 * Math.pow(r,2) * Math.PI;
        edit2.setText(""+ felsz);
      }
     // ha a Térfogat gomb kijelölt
     if (e.getSource() == terfogat && e.getStateChange()==e.SELECTED) {
        edit2.setText("0");
        terf = 4* Math.pow( r, 3) * Math.PI/3;
        edit3.setText(""+ terf);
     }
     // ha a Mindkettő gomb kijelölt
     if(e.getSource() == terffelsz && e.getStateChange()==e.SELECTED) {
        felsz = 4 * Math.pow(r,2) * Math.PI;
        edit2.setText(""+ felsz);
        terf = 4* Math.pow( r, 3) * Math.PI/3;
        edit3.setText(""+ terf);
     }
  }
```

```
    public static void main(String argv[]) {
        Check2pr ablak = new Check2pr("CheckBox kezelése");
    }
}
```

Az alkalmazás futásának eredményei:



10.2.5 Párbeszédablakok kialakítása

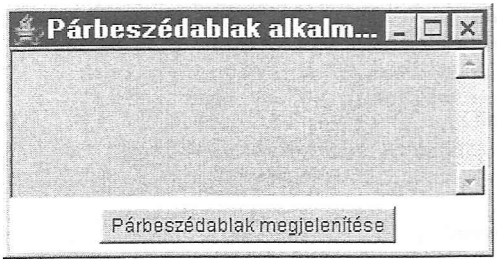
Készítsünk grafikus felületű alkalmazást, amely létrehoz egy egyszerű párbeszédablakot, és megjeleníti azt! (*AWT\Parbeszed*)

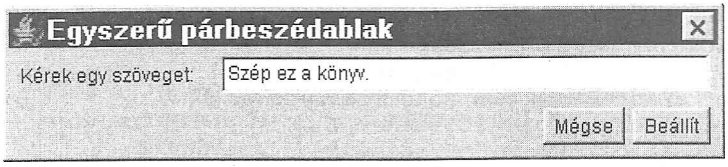
```
import java.awt.*;
import java.awt.event.*;
// A főablak osztálya
public class FoWindow extends Frame implements WindowListener,
                                                ActionListener
{
    private EgyszeruDialog dialog;
    private TextArea textArea;

    public FoWindow() {
        textArea = new TextArea(5, 40);
        textArea.setEditable(false);
        add("Center", textArea);
        Button button = new Button("Párbeszédablak megjelenítése");
        button.addActionListener(this);
        Panel panel = new Panel();
        panel.add(button);
        add("South", panel);
        addWindowListener(this);
    }

    public void windowClosing(WindowEvent e) {
          System.exit(0);
    }
```

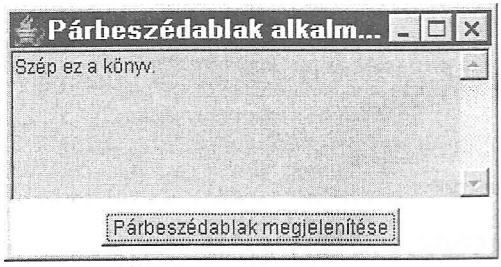
```
    public void windowOpened(WindowEvent e) { }
    public void windowClosed(WindowEvent e) { }
    public void windowIconified(WindowEvent e) { }
    public void windowDeiconified(WindowEvent e) { }
    public void windowActivated(WindowEvent e) { }
    public void windowDeactivated(WindowEvent e) { }

    // gombnyomásra megjelenik a párbeszédablak
    public void actionPerformed(ActionEvent e) {
        if (dialog == null)
            dialog = new EgyszeruDialog(this,
                                        "Egyszerű párbeszédablak");
        dialog.setVisible(true);
    }

    // saját szövegbeállító metódus
    public void setText(String text) {
        textArea.append(text + "\n");
    }

    public static void main(String args[])  {
        FoWindow ablak = new FoWindow();
        ablak.setTitle("Párbeszédablak alkalmazás");
        ablak.pack();
        ablak.setVisible(true);
    }
}

// A párbeszédablak osztálya
class EgyszeruDialog extends Dialog implements  ActionListener
{
   TextField szovegmezo;
   FoWindow parent;
   Button setButton;

   EgyszeruDialog(Frame dw, String title) { // konstruktor
      super(dw, title, false);
      parent = (FoWindow)dw;

      // Az középső rész tartalma
      Panel p1 = new Panel();
      Label label = new Label("Kérek egy szöveget:");
      p1.add(label);
      szovegmezo = new TextField(40);
      szovegmezo.addActionListener(this);
      p1.add(szovegmezo);
      add("Center", p1);

      // Az alsó sor tartalma
      Panel p2 = new Panel();
      p2.setLayout(new FlowLayout(FlowLayout.RIGHT));
      Button b = new Button("Mégse");
      b.addActionListener(this);
      setButton = new Button("Beállít");
      setButton.addActionListener(this);
      p2.add(b);
      p2.add(setButton);
```

```
        add("South", p2);
        // A megfelelő méretű párbeszédablak inicializálása
        pack();
    }
    public void actionPerformed(ActionEvent e) {
      if ((e.getSource() == setButton) ||
                    (e.getSource() == szovegmezo))
        parent.setText(szovegmezo.getText());
      szovegmezo.selectAll();
      this.setVisible(false);
    }
}
```

Az alkalmazás futásának eredményei:







Készítsünk alkalmazást, amely bemutatja a szokásos (névjegy és kiléptető) párbeszédablakok használatát! (*AWT\Parbeszedablakok*)

```
// A főablak osztálya
import java.awt.*;
import java.awt.event.*;
```

```
public class FoAblak extends Frame
{
   public FoAblak() {  // konstruktor
      setLayout(null);
      setSize(400,300);
      setVisible(false);
      setTitle("AWT Alkalmazás");
      // a menürendszer felépítése
      menu1.setLabel("Műveletek");
      menu1.add(letrehozMenuItem);
      letrehozMenuItem.setEnabled(false);
      letrehozMenuItem.setLabel("Létrehozás");
      letrehozMenuItem.setShortcut
                        (new MenuShortcut(KeyEvent.VK_L,false));
      menu1.add(openMenuItem);
      openMenuItem.setLabel("Megnyitás...");
      openMenuItem.setShortcut(new MenuShortcut(KeyEvent.VK_O,false));
      menu1.add(elvalasztoMenuItem);
      menu1.add(kilepMenuItem);
      kilepMenuItem.setLabel("Kilépés");
      mainMenuBar.add(menu1);
      menu2.setLabel("Súgó");
      menu2.add(nevjegyMenuItem);
      nevjegyMenuItem.setLabel("Névjegy...");
      mainMenuBar.add(menu2);
      setMenuBar(mainMenuBar);
      // az eseményfigyelők regisztrálása
      this.addWindowListener(new FoWindowAdapter());
      FoActionListener aListener = new FoActionListener();
      openMenuItem.addActionListener(aListener);
      kilepMenuItem.addActionListener(aListener);
      nevjegyMenuItem.addActionListener(aListener);
   }
   public FoAblak(String felirat) { // paraméteres konstruktor
       this();
       setTitle(felirat);
   }
   // az alkalmazást indító main() metódus
   static public void main(String args[]) {
      (new FoAblak("AWT alkalmazás párbeszédablakokkal"))
                                              .setVisible(true);
   }
   // A kilépést megvalósító adapterosztály
   class FoWindowAdapter extends WindowAdapter {
      public void windowClosing(WindowEvent event) {
         Object object = event.getSource();
         if (object == FoAblak.this)
         // KilepDialog létrehozása és modális megjelenítése
         (new KilepDialog(FoAblak.this, true)).setVisible(true);
      }
   }
```

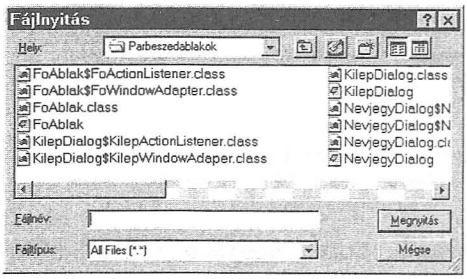
```
   // A menüesemények kezelése
   class FoActionListener implements ActionListener {
       public void actionPerformed(ActionEvent event) {
         Object object = event.getSource();
         if (object == openMenuItem)
              openMenuItem_ActionPerformed(event);
         else if (object == nevjegyMenuItem)
              nevjegyMenuItem_ActionPerformed(event);
         else if (object == kilepMenuItem)
              kilepMenuItem_ActionPerformed(event);
       }
   }

   // a Fájlnyitás párbeszédablak létrehozása és modális megjelenítése
   void openMenuItem_ActionPerformed(ActionEvent event) {
      openFileDialog1 = new FileDialog
           (this, "Fájlnyitás", FileDialog.LOAD);
      openFileDialog1.setVisible(true);
      String konyvtar = openFileDialog1.getDirectory();
      String fajl     = openFileDialog1.getFile();
      if (konyvtar != null)
         setTitle(konyvtar + fajl);
   }

   // a Névjegy párbeszédablak megjelenítése
   void nevjegyMenuItem_ActionPerformed(ActionEvent event) {
       // létrehozás és modális megjelenítés
       (new NevjegyDialog(this, true)).setVisible(true);
   }

   // A kilépés jóváhagyása
   void kilepMenuItem_ActionPerformed(ActionEvent event) {
      // Létrehozás és modális megjelenítés
         (new KilepDialog(this, true)).setVisible(true);
   }

   // deklarációk
   FileDialog openFileDialog1 = new FileDialog(this);
   // a menü deklarációi
   MenuBar mainMenuBar = new MenuBar();
   Menu menu1 = new Menu();
   MenuItem letrehozMenuItem = new MenuItem();
   MenuItem openMenuItem = new MenuItem();
   MenuItem elvalasztoMenuItem = new MenuItem("-");
   MenuItem kilepMenuItem = new MenuItem();
   Menu menu2 = new Menu();
   MenuItem nevjegyMenuItem = new MenuItem();
}

// A kilépést jóváhagyó párbeszédablak
import java.awt.*;
import java.awt.event.*;
```

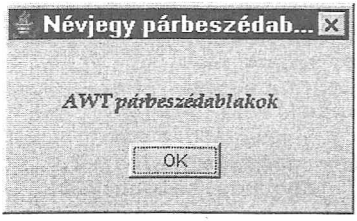
```
public class KilepDialog extends Dialog
{
   public KilepDialog(Frame szuloablak, boolean modal) {
       super(szuloablak, modal);
       // az aktiváló ablak objektumának referenciája
       foAblak = szuloablak;

       setLayout(null);
       setSize(300,180);
       setVisible(false);
       setTitle("Kilépés párbeszédblak");

       igenButton.setLabel(" Igen ");
       add(igenButton);
       igenButton.setBounds(50,120,80,30);

       nemButton.setLabel("  Nem  ");
       add(nemButton);
       nemButton.setBounds(160,120,80,30);

       felirat.setText("Valóban ki akar lépni?");
       felirat.setAlignment(Label.CENTER);
       felirat.setFont(new Font("Dialog", Font.BOLD, 17));
       add(felirat);
       felirat.setBounds(60,60,180,40);

       // Az eseményfigyelés beállítása
       this.addWindowListener(new KilepWindowAdaper());
       KilepActionListener aListener = new KilepActionListener();
       nemButton.addActionListener(aListener);
       igenButton.addActionListener(aListener);
   }

   // konstruktor
   public KilepDialog(Frame szuloablak, String title, boolean modal) {
       this(szuloablak, modal);
       setTitle(title);
   }

   // A szülőablak közepére igazítjuk a párbeszédablakot
   public void setVisible(boolean b) {
       if (b) {
              Rectangle pbounds = getParent().getBounds(); // szülő
              Rectangle dbounds = getBounds();             // dialog
              setLocation(pbounds.x + (pbounds.width
                          - dbounds.width)/ 2,
              pbounds.y + (pbounds.height - dbounds.height)/2);
              Toolkit.getDefaultToolkit().beep();
       }
       super.setVisible(b);
   }

   // A gombnyomásokat figyelő osztály
   class KilepActionListener implements ActionListener
   {
       public void actionPerformed(ActionEvent event) {
           Object object = event.getSource();
           if (object == igenButton) {
```

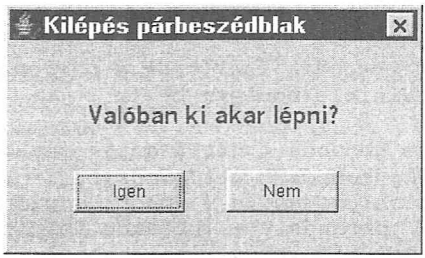
```
            foAblak.setVisible(false);  // a főablak levétele
            foAblak.dispose();          // erőforrások felszabadítása
            KilepDialog.this.dispose(); // erőforrások felszabadítása
            System.exit(0);  // kilépés az alkalmazásból
         }
         else if (object == nemButton)
            KilepDialog.this.dispose();
      }
}

// a névjegy párbeszédablak
import java.awt.*;
import java.awt.event.*;

public class NevjegyDialog extends Dialog
{
   public NevjegyDialog(Frame parent, boolean modal) {
      super(parent, modal);
      setLayout(null);
      setSize(250,150);
      setVisible(false);
      felirat.setText("AWT párbeszédablakok");
      felirat.setBounds(40,60,160,20);
      felirat.setAlignment(Label.CENTER);
      felirat.setFont(new Font("Dialog", Font.BOLD|Font.ITALIC, 14));
      add(felirat);
      okButton.setLabel("OK");
      okButton.setBounds(90,100,65,25);
      setTitle("Névjegy párbeszédablak");
      add(okButton);

      this.addWindowListener(new NevJegyWindowAdapter());
      okButton.addActionListener(new NevJegyActionListener());
   }

   public NevjegyDialog(Frame parent, String title, boolean modal) {
      this(parent, modal);
      setTitle(title);
   }

   // A szülőablak közepére igazítjuk a párbeszédablakot
   public void setVisible(boolean b) {
      if (b){
         Rectangle pbounds = getParent().getBounds(); // szülő
         Rectangle dbounds = getBounds();             // dialog
         setLocation(pbounds.x + (pbounds.width - dbounds.width)/2,
                     pbounds.y + (pbounds.height - dbounds.height)/2);
         Toolkit.getDefaultToolkit().beep();
      }
      super.setVisible(b);
   }
```

```
    class NevJegyActionListener implements ActionListener
    {
        public void actionPerformed(java.awt.event.ActionEvent event) {
            Object object = event.getSource();
            if (object == okButton)
               NevjegyDialog.this.dispose();
        }
    }

    class NevJegyWindowAdapter extends WindowAdapter
    {
        public void windowClosing(WindowEvent event) {
            Object object = event.getSource();
            if (object == NevjegyDialog.this)
               NevjegyDialog.this.dispose();
       }
    }

    // deklarációk
    Label felirat = new Label();
    Button okButton = new Button();
}
```

Az alkalmazás futásának eredményei:







10.3 Swing komponensek alkalmazása

A *Swing*-alkalmazások felépítése alapvetően ugyanazon elveket követi, mint amelyeket az *AWT*-alkalmazásoknál megismertünk. Azonban tapasztalni fogjuk, hogy sok esetben logikusabb, egységesebb és egyszerűbb megoldásokat alkalmazhatunk.

- A ***JFrame*** osztályból származtatjuk az alkalmazás ablakát, melyre a menün (***JMenuBar***) kívül eszköztárat (***JToolBar***) is felhelyezhetünk.
- A vezérlőket általában először egy tárolóhoz rendeljük (mint amilyen a ***JPanel***), majd pedig ezt a tárolót állítjuk be az ablakunk munkaterületének a ***setContentPane***(*tároló*); hívással.
- A *AWT* szokásos elrendezés-kezelőit és – az esetek többségében – az *AWT* eseményfigyelőit használjuk. A *Swing* új komponenseihez azonban új eseményfigyelők tartoznak a ***javax.swing.event*** csomagból.
- A *Swing* osztályok alkalmazásához az alábbi **import** utasításokat ajánlott megadni:

```
import java.awt.*;                  // elrendezés-kezelők, grafika
import java.awt.event.*;            // AWT eseményfigyelők
import javax.swing.*;               // Swing összetevők
import javax.swing.event.*;         // új Swing eseményfigyelők
```

Az alábbi kis példaprogram jól szemlélteti az elmondottakat:

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

public class SwingAlkalmazas extends JFrame implements ActionListener
{
      JButton kilep = new JButton("Kilépés");
      JPanel panel = new JPanel();

      public SwingAlkalmazas() {   // konstruktor
            super("Swing alkalmazás");
            setBounds(120, 230, 250, 150);
            setDefaultCloseOperation(EXIT_ON_CLOSE);

            kilep.addActionListener(this);
            panel.add(kilep);
            setContentPane(panel);
      }

      public static void main(String[] args) {
            (new SwingAlkalmazas ()).setVisible(true);
      }

      // a Kilépés gomb eseménykezelője
      public void actionPerformed(ActionEvent event) {
            if (event.getSource() == kilep)
                  System.exit(0);
      }
}
```

Az alkalmazás ablakában a nyomógomb az alapértelmezés szerinti (***FlowLayout***, középre igazít, és 5 képpont távolságot tart a vezérlők között mindkét irányban) elrendezés szerint jelent meg. A ***setBounds()*** hívással az ablak méretein (*250, 150*) túlmenően a képernyőn való elhelyezkedését (*120, 230*) is meghatároztuk.

Külön felhívjuk a figyelmet a ***JFrame*** osztály ***setDefaultCloseOperation()*** metódusára, amellyel beállíthatjuk, hogy mi a teendő az ablak lezárására tett kísérlet esetén. Az alábbi ***JFrame*** konstans mezőket használhatjuk argumentumként:

- *EXIT_ON_CLOSE* – az ablak lezárása után kilép a programból,
- *DISPOSE_ON_CLOSE* – bezárja az ablakot, eldobja az ablak objektumát, azonban az alkalmazás tovább fut,
- *DO_NOTHING_ON_CLOSE* – nem tesz semmit, csak programból lehet bezárni az ablakot,
- *HIDE_ON_CLOSE* – bezárja az ablakot, és folytatja a futást (alapértelmezés).

Az utolsó három esetben csak a ***WindowListener*** interfész felhasználásával léphetünk ki az alkalmazásból.

10.3.1 Alapvezérlők

A *Swing* komponenseire való áttérés megkönnyítése érdekében az új vezérlők konstruktorainak és metódusainak többsége az *AWT*-beli megfelelőjükkel azonos módon hívható, azonban lényegesen bővítik azok metóduskészletét. Természetesen egy sor új komponensnek nem is létezik párja a ***java.awt*** csomagban.

A 10.2.1 fejezetben ismertetett komponensek elé a *J* betűt helyezve, megkapjuk a vezérlők *Swing* megfelelőjét: ***JButton***, ***JCheckbox***, ***JLabel***, ***JTextField***, ***JTextArea***, ***JList***, ***JScrollBar***. A ***Choice*** helyett a ***JComboBox*** osztályt használhatjuk, míg a ***CheckboxGroup*** funkcionalitását a ***JRadioButton*** és a ***ButtonGroup*** osztályok együtt biztosítják.

Az alábbiakban a különböző komponensekhez csoportosítva fűzünk néhány megjegyzést.

Gombok és címke- JButton, JCheckbox, JRadioButton és ***JLabel***

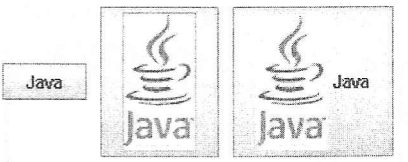
A fenti négy osztály konstruktorai kibővültek, a szokásos paraméterek mellett megjelent bennük az ***Icon***, illetve a gomboknál az ***Action*** típusú paraméter is. Példaként tekintsük a ***JButton*** osztály konstruktorait!

```
JButton()
JButton(String felirat)
JButton(Action a)
JButton(Icon ikon)
JButton(String felirat, Icon ikon)
```

Az *ActionListener* és *EventListener* interfészekből származtatott *Action* interfészt megvalósító objektum segítségével több komponens számára is előírhatjuk ugyanazt a viselkedést. Ekkor a feliratot és/vagy az ikont a ***setText()***, illetve a ***setIcon()*** metódusok segítségével állíthatjuk be.

Nézzünk példákat a szöveget és ikont tartalmazó gombok létrehozására!

```
ImageIcon ikon = new
                 ImageIcon("Java.gif");
JButton jb1 = new JButton("Java");
JButton jb2 = new JButton(ikon);
JButton jb3 = new JButton("Java",ikon);
```



A jelölő- és a választógombok állapotát a ***setSelected()*** metódussal tehetjük kijelöltté (**true** argumentum), illetve törölhetjük a kijelölést (**false** argumentum).

Készítsünk alkalmazást, amely bemutatja a gombok használatával összefüggő legfontosabb metódusokat! (*Swing\JGombok*)

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

public class SwingAlkalmazas extends JFrame implements ActionListener
{
    JButton kilep = new JButton();
    JPanel panel = new JPanel();
    ImageIcon bezar = new ImageIcon("Bezar.gif");

    public SwingAlkalmazas() {
        super("Swing alkalmazás");
        setBounds(120, 230, 250, 150);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        kilep.addActionListener(this);
        kilep.setText("Kilépés");
        kilep.setIcon(bezar);
        // az Alt+K billenyűkkel is lenyomhatjuk
        kilep.setMnemonic(KeyEvent.VK_K);
        // buboréksúgó
        kilep.setToolTipText("Kilépés az alkalmazásból");
        // a szöveg alul és középen
        kilep.setVerticalTextPosition(JButton.BOTTOM);
        kilep.setHorizontalTextPosition(JButton.CENTER);
        kilep.setActionCommand("Lépj ki!");
        panel.setLayout(new FlowLayout(FlowLayout.LEFT));
```

```
        panel.setBackground(Color.white);
        panel.add(kilep);
        setContentPane(panel);
    }

    public static void main(String[] args) {
        (new SwingAlkalmazas ()).setVisible(true);
    }

    public void actionPerformed(ActionEvent event) {
        if ("Lépj ki!".equals(event.getActionCommand()))
            System.exit(0);
    }
}
```

Az alkalmazás futásának eredménye:

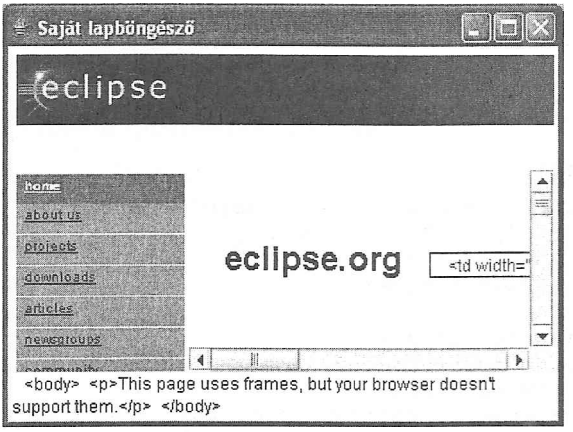
Szöveges komponensek- JTextField, JPasswordField, JFormattedTextField, JTextArea, JEditorPane, JTextPane

Az egysoros szövegmező ***setEchoChar()*** metódusa csak a jelszómező (***JPasswordField***) osztályában található meg. A ***JFormattedTextField*** típusú egysoros szövegmezőben előírt formátum szerint adhatunk meg adatokat. A ***JTextPane*** típusú többsoros szövegmezőben különböző betűtípussal készült szövegrészek és ikonok is szerepelhetnek. A ***JEditorPane*** típusú szövegmezőben különböző formátumú (például statikus *HTML*) dokumentumokat is megjeleníthetünk. A szöveges vezérlők alapvető műveletei megegyeznek az *AWT* fejezetben megismertekkel. Az alábbi kis példaprogram a ***JEditorPane*** használatát szemlélteti:

```
import javax.swing.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
public class SwingAlkalmazas extends JFrame {
    public SwingAlkalmazas() throws MalformedURLException,
                                    IOException {
        super("Saját lapböngésző");
        setSize(400, 300);
        setDefaultCloseOperation(DISPOSE_ON_CLOSE);
        JEditorPane bongeszo =
               new JEditorPane(new URL("http://www.eclipse.org/"));
        getContentPane().add(bongeszo);
    }
```

```
    public static void main(String[] args) {
        try {
            (new SwingAlkalmazas ()).setVisible(true);
        }
        catch (Exception e) {
            System.out.println("Nem sikerült!");
        }
    }
}
```

Az alkalmazás futásának eredménye:



Listaablakok - JList, JComboBox

Talán a legnagyobb változást – előnyükre – „elszenvedett" alapvezérlők; az *AWT* ***List*** és ***Choice*** vezérlői. Emiatt a *Swing* változataikkal kicsit részletesebben kell foglakoznunk. Tekintsük először a gyakrabban használt; lenyíló listát megvalósító ***JComboBox*** osztály konstruktorait!

```
JCombox()
JCombox (ComboBoxModel adatModell)
JCombox (Object[] adatTömb)
JCombox (Vector<?> adatVektor)
```

A kombinált lista elemei bővíthetők, törölhetők és lecserélhetők, ugyanúgy, mint a ***List*** vezérlő esetén. Ezzel szemben a ***JList*** komponens lehetőségei erősen korlátozottak, a konstruálást követően elemei nem módosíthatók, és az elemek görgetéshez is egy másik vezérlőt, a ***JScrollPane***-t kell segítségül hívni.

Készítsünk alkalmazást, amely bemutatja a lista és a kombinált lista használatát! (*Swing\JListak*)

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

```
import javax.swing.event.*;
public class SwingAlkalmazas extends JFrame
                    implements ActionListener, ListSelectionListener
{
    JComboBox cbox;
    JList lista;
    JScrollPane listagorgeto;

    public SwingAlkalmazas()    {
        super("Listakezelés");
        setBounds(120, 230, 250, 210);
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        // a kombinált lista létrehozása és elemenkénti feltöltése
        cbox = new JComboBox();
        cbox.addItem("Piros");
        cbox.addItem("Zöld");
        cbox.addItem("Kék");
        cbox.addActionListener(this);

        String szinek[] = { "Sárga", "Türkiz", "Rózsaszín", "Lila",
                            "Fekete", "Fehér", "Szürke", "Barna",
                            "Narancssárga"};
        // a lista és feltöltése sztingtömbbel
        lista = new JList(szinek);
        lista.addListSelectionListener(this);
        // bal oldali, függőleges görgetősáv hozzáadása a listához
        listagorgeto = new JScrollPane(lista,
                    ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED,
                    ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

        getContentPane().setLayout(new FlowLayout());
        getContentPane().add(cbox);
        getContentPane().add(listagorgeto);
    }

    public static void main(String[] args) {
         (new SwingAlkalmazas ()).setVisible(true);
    }

    // a kombinált lista eseményei
    public void actionPerformed(ActionEvent event) {
        if (event.getSource() == cbox)
            setTitle("ComboBox: " + cbox.getSelectedItem());
    }

    // a lista eseményei
    public void valueChanged(ListSelectionEvent event) {
        if (event.getSource() == lista)
            setTitle("Lista: " + lista.getSelectedValue());
    }
}
```

Az alkalmazás futásának eredménye:

Görgetők - JScrollBar, JScrollPane

Mivel a ***JScrollBar*** osztály metódusai és konstansai teljes mértékben megfelelnek a ***Scrollbar*** komponens tagjainak, így ezzel külön nem foglalkozunk. A ***JScrollPane*** komponens segítségével tetszőleges komponenst elláthatunk vízszintes és/vagy függőleges görgetősávokkal. A előző példaprogramban is alkalmazott konstruktor:

```
JScrollPane(Component nézet, int vízszintes, int függőleges)
```

A görgetősávok meglétéről és működéséről mindkét irányban a ***ScrollPaneConstants*** osztály végleges adattagjaival intézkedhetünk. Nézzünk néhány lehetséges értéket, melyek a függőleges görgetősáv megjelenését szabályozzák!

VERTICAL_SCROLLBAR_ALWAYS - mindig legyen,
VERTICAL_SCROLLBAR_AS_NEEDED - csak ha kell,
VERTICAL_SCROLLBAR_NEVER - soha se legyen.

Keret – a *Border* interfész és a *BorderFactory* osztály

A *Swing* komponensek körül különböző egyszerű, kiemelkedő, besüllyed stb. keretet jeleníthetünk meg. A ***java.swing.border.Border*** interfész megvalósításaként keletkező objektumokat a ***BorderFactory*** osztály statikus metódusai állítják elő. Nézzünk néhány keretet, melyek képét az alábbi lépésekkel állítottuk elő, változtatva a keret típusát:

```
JButton gomb = new JButton("   Java Swing   ");
Border keret = BorderFactory.createLineBorder(Color.black);
gomb.setBorder(keret);
```

BorderFactory metódusának hívása	*Eredmény*
`createLineBorder(Color.black)`	Java Swing
`createRaisedBevelBorder()`	Java Swing
`createLoweredBevelBorder()`	Java Swing
`createEtchedBorder(EtchedBorder.RAISED)`	Java Swing

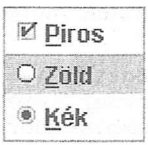
Menü, felbukkanó menü – JMenu, JPopupMenu

A *Swing* alkalmazások egyik legfontosabb vezérlőeleme a menü, mely felépítésének lépései megegyeznek az AWT menüknél alkalmazottakkal. Az ablakunk menüsorát most is a ***JMenuBar*** osztály objektuma valósítja meg, amelyhez lenyíló menüket csatlakoztatunk (***JMenu*** osztály). Az egyes menük a menüpontok (***JMenuItem***) listája. A metódusok ismertetése helyett tekintsük az alábbi egyszerű menüt!

```
ImageIcon ikon = new ImageIcon("szinek.gif");
JMenuBar menuSor = new JMenuBar();
JMenu szinekMenu = new JMenu();
JMenuItem pirosMenupont = new JMenuItem();
JMenuItem zoldMenupont = new JMenuItem();
JMenuItem kekMenupont = new JMenuItem();

szinekMenu.setText("Színek");
szinekMenu.setIcon(ikon);
szinekMenu.setMnemonic((int)'S');
szinekMenu.setToolTipText("Színek beállítása");

pirosMenupont.setText("Piros");
pirosMenupont.setMnemonic((int)'P');
szinekMenu.add(pirosMenupont);
szinekMenu.addSeparator();

zoldMenupont.setText("Zöld");
zoldMenupont.setMnemonic((int)'Z');
szinekMenu.add(zoldMenupont);
szinekMenu.addSeparator();

kekMenupont.setText("Kék");
kekMenupont.setMnemonic((int)'K');
szinekMenu.add(kekMenupont);
menuSor.add(szinekMenu);
this.setJMenuBar(menuSor);
```

Mivel a ***JMenuItem*** osztály az ***AbstractButton*** osztálytól származik, viselkedése sok mindenben egyezik a *Swing* gombjainak viselkedésével. A ***JMenuItem*** alosztályai a ***JMenu***, ***JCheckBoxMenuItem***, és ***JRadioButtonMenuItem***. Végezetül nézzünk egy példát ez utóbbi két osztály alkalmazására felbukkanó menüben (***JPopupMenu***)!

```
JPopupMenu szinekMenu = new JPopupMenu();
JCheckBoxMenuItem pirosMenupont =
                    new JCheckBoxMenuItem();
JRadioButtonMenuItem zoldMenupont =
                    new JRadioButtonMenuItem();
JRadioButtonMenuItem kekMenupont =
                    new JRadioButtonMenuItem();

pirosMenupont.setText("Piros");
pirosMenupont.setMnemonic((int)'P');
szinekMenu.add(pirosMenupont);

zoldMenupont.setText("Zöld");
zoldMenupont.setMnemonic((int)'Z');
szinekMenu.add(zoldMenupont);
kekMenupont.setText("Kék");
```



```
kekMenupont.setMnemonic((int)'K');
szinekMenu.add(kekMenupont);
// a felbukkanó menü megjelenítése, megadva
// az alatta levő komponenst és a pozíciókat
szinekMenu.show(this, 100, 80);
```

10.3.2 Komponensek használata alkalmazásokban

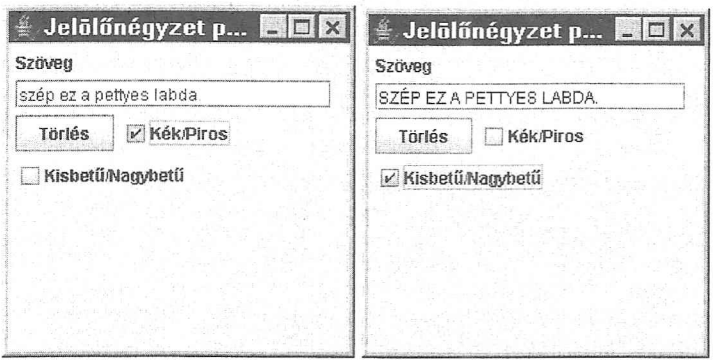
A következőkben a különböző *Swing*-komponensek alkalmazására mutatunk példákat.

Írjunk *Swing*-alkalmazást, amely az alapvezérlők (***JButton***, ***JLabel***, ***JTextField*** és ***JCheckBox***) használatának bemutatására! (*Swing\Check1*)

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

public class Program1 extends JFrame implements ActionListener,
                                                 ItemListener {
    JLabel fejlec = new JLabel("Szöveg");
    JButton button1 = new JButton("Törlés");
    JTextField szoveg = new JTextField(20);
    JCheckBox valtas = new JCheckBox("Kisbetű/Nagybetű");
    JCheckBox szin = new JCheckBox("Kék/Piros");

    public Program1(String nev) {
        super(nev);
        setSize(250,250);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setBackground(Color.gray);

        setLayout(new FlowLayout(FlowLayout.LEFT));
        valtas.setSelected(false);
        add(fejlec); fejlec.setVisible(true);
        add(szoveg);
        add(button1); button1.setVisible(true);
        add(szin);
        add(valtas);

        button1.addActionListener(this);
        valtas.addItemListener(this);
        szin.addItemListener(this);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource()==button1)
            szoveg.setText("");
    }

    public void itemStateChanged(ItemEvent e) {
        if (e.getSource()==szin && e.getStateChange()==e.SELECTED)
            szoveg.setForeground(Color.red);
        else
            szoveg.setForeground(Color.blue);
```

```
        if (e.getSource()==valtas && e.getStateChange()==e.SELECTED)
            szoveg.setText(szoveg.getText().toUpperCase());
        else
            szoveg.setText(szoveg.getText().toLowerCase());
    }

    public static void main(String[] args) {
        Program1 ablak=new Program1("Jelölőnégyzet példa1");
    }
}
```

Az alkalmazás futásának eredményei:

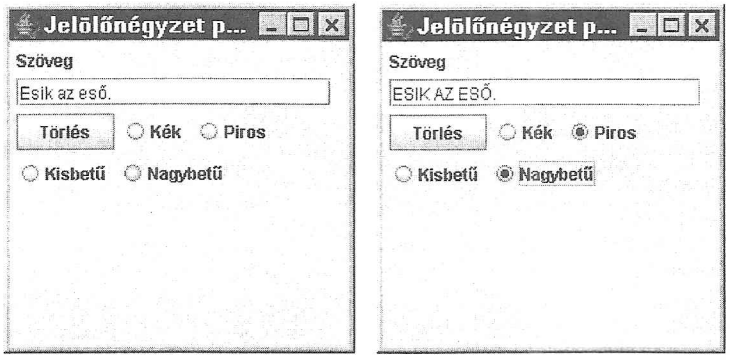


Valósítsuk meg a *Check1* program működését úgy, hogy a jelölőnégyzeteket (***JCheckBox***) választógombokkal (***JRadioButton***) helyettesítjük! (*Swing\Radio1*)

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

public class Program1 extends JFrame implements ActionListener,
                                                 ItemListener {
    JLabel fejlec = new JLabel("Szöveg");
    JButton button1 = new JButton("Törlés");
    JTextField szoveg = new JTextField(20);
    JRadioButton kisbetu = new JRadioButton("Kisbetű");
    JRadioButton nagybetu = new JRadioButton("Nagybetű");
    JRadioButton szin1 = new JRadioButton("Kék");
    JRadioButton szin2 = new JRadioButton("Piros");
    ButtonGroup csoport1 = new ButtonGroup();
    ButtonGroup csoport2 = new ButtonGroup();

    public Program1() {
        super("Jelölőnégyzet példa1");
        setSize(250,250);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setBackground(Color.gray);

        setLayout(new FlowLayout(FlowLayout.LEFT));
        csoport1.add(kisbetu);
        csoport1.add(nagybetu);
```

```
        csoport2.add(szin1);
        csoport2.add(szin2);

        this.add(fejlec); fejlec.setVisible(true);
        this.add(szoveg);
        this.add(button1); button1.setVisible(true);
        this.add(szin1);
        this.add(szin2);
        this.add(kisbetu);
        this.add(nagybetu);

        button1.addActionListener(this);
        szin1.addItemListener(this);
        szin2.addItemListener(this);
        kisbetu.addItemListener(this);
        nagybetu.addItemListener(this);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource()==button1)
            szoveg.setText("");
    }

    public void itemStateChanged(ItemEvent e) {
        if (e.getSource()==szin1 && e.getStateChange()==e.SELECTED)
            szoveg.setForeground(Color.blue);
        if (e.getSource()==szin2 && e.getStateChange()==e.SELECTED)
            szoveg.setForeground(Color.red);

        if (e.getSource()==nagybetu && e.getStateChange()==e.SELECTED)
            szoveg.setText(szoveg.getText().toUpperCase());
        if (e.getSource()==kisbetu && e.getStateChange()==e.SELECTED)
            szoveg.setText(szoveg.getText().toLowerCase());
    }

    public static void main(String[] arguments) {
        Program1 pf = new Program1();
    }
}
```

Az alkalmazás futásának eredményei:

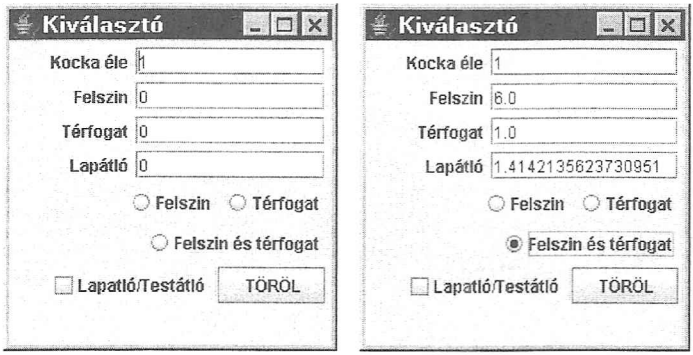


Fejlesszünk alkalmazást, amelyben mind a háromféle gombot (***JButton***, ***JCheckBox*** és ***JRadioButton***) műveletek végrehajtására használjuk! (*Swing\Radio2*)

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

public class Program1 extends JFrame implements ActionListener,
                                                 ItemListener
{
    double a, terf, felsz, testatlo, lapatlo;
    JLabel label1 = new JLabel("         Kocka éle");
    JLabel label2 = new JLabel("            Felszin");
    JLabel label3 = new JLabel("          Térfogat");
    JLabel label4 = new JLabel("           Lapátló");
    JTextField edit1 = new JTextField("1",12);
    JTextField edit2 = new JTextField("0",12);
    JTextField edit3 = new JTextField("0",12);
    JTextField edit4 = new JTextField("0",12);
    JCheckBox atlok = new JCheckBox("Lapatló/Testátló");

    JRadioButton felszin = new JRadioButton("Felszin");
    JRadioButton terfogat = new JRadioButton("Térfogat");
    JRadioButton terf_felsz = new JRadioButton("Felszin és térfogat");
    ButtonGroup csoport = new ButtonGroup();

    JButton button1 = new JButton(" TÖRÖL");

    public Program1() {
       super("Kiválasztó");
       this.setSize(240,250);
       setDefaultCloseOperation(EXIT_ON_CLOSE);
       setLayout(new FlowLayout(FlowLayout.RIGHT));

       felszin.setSelected(false);
       terfogat.setSelected(false);
       terf_felsz.setSelected(false);

       csoport.add(felszin);
       csoport.add(terfogat);
       csoport.add(terf_felsz);

       this.add(label1);
       this.add(edit1);
       this.add(label2);
       this.add(edit2);
       this.add(label3);
       this.add(edit3);
       this.add(label4);
       this.add(edit4);
       this.add(felszin);
       this.add(terfogat);
       this.add(terf_felsz);
       this.add(atlok);
       this.add(button1);
       setVisible(true);
```

```
        button1.addActionListener(this);
        felszin.addItemListener(this);
        terfogat.addItemListener(this);
        atlok.addItemListener(this);
        terf_felsz.addItemListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource()==button1) {
            edit1.setText("0");
            edit2.setText("0");
            edit3.setText("0");
            edit4.setText("0");
        }
    }

    public void itemStateChanged(ItemEvent e) {
        double a = new Double(edit1.getText()).doubleValue();
        if (e.getSource()==felszin && e.getStateChange()==e.SELECTED) {
            edit3.setText("0");
            felsz = 6 * a; edit2.setText(""+ felsz);
        }
        if (e.getSource()==terfogat && e.getStateChange()==e.SELECTED) {
            edit2.setText("0");
            terf = Math.pow(a,3);
            edit3.setText(""+ terf);
        }
        if (e.getSource() == atlok &&
                        e.getStateChange() == e.SELECTED && a > 0) {
            testatlo = a * Math.sqrt(3);
            edit4.setText(""+ testatlo);
            label4.setText("Testátló");
        }
        else {
            lapatlo = a * Math.sqrt(2);
            edit4.setText(""+ lapatlo);
            label4.setText("Lapátló");
        }
        if (e.getSource() == terf_felsz &&
                        e.getStateChange() == e.SELECTED && a > 0) {
            felsz = 6 * a;
            edit2.setText(""+ felsz);
            terf = Math.pow(a,3);
            edit3.setText(""+ terf);
        }
    }

    public static void main(String[] arguments) {
        Program1 pf = new Program1();
    }
}
```

Az alkalmazás futásának eredményei:

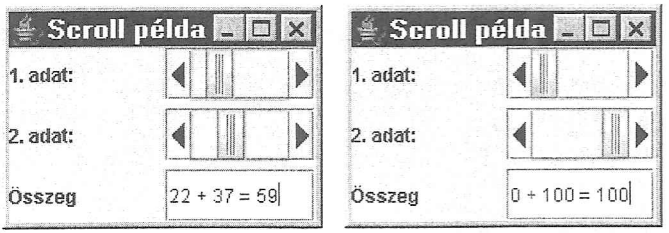


Készítsünk összeadó programot, ahol a tagok értékét két görgetősávval (***JScrollBar***) lehet módosítani 1 és 100 közötti tartományon! (*Swing\Scroll1*)

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class Program1 extends JFrame implements AdjustmentListener
{
    JLabel label1 = new JLabel("1. adat:");
    JLabel label2 = new JLabel("2. adat:");
    JLabel label3 = new JLabel("Összeg");
    JTextField ered = new JTextField(20);

    JScrollBar szam1 = new JScrollBar(Scrollbar.HORIZONTAL, 0, 0, 0,
                       100);
    JScrollBar szam2 = new JScrollBar(Scrollbar.HORIZONTAL, 0, 0, 0,
                       100);
    JButton button2 = new JButton("Összeg");
    JPanel mezo = new JPanel();

    public Program1() {
        super("Scroll példa");
        setSize(200,140);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        mezo.setLayout(new GridLayout(3,2,5,5));
        mezo.add(label1);
        mezo.add(szam1);
        mezo.add(label2);
        mezo.add(szam2);
        mezo.add(label3);
        mezo.add(ered);
        setContentPane(mezo);
        setVisible(true);

        szam1.addAdjustmentListener(this);
        szam2.addAdjustmentListener(this);
    }
```

```
    public void adjustmentValueChanged(AdjustmentEvent e) {
        if(e.getSource() == szam1) {
            ered.setText(String.valueOf(szam1.getValue()) + " + "
                         + String.valueOf(szam2.getValue()) + " = "
                         + String.valueOf(szam1.getValue()
                         + szam2.getValue()));
        }
        if(e.getSource() == szam2) {
            ered.setText(String.valueOf(szam1.getValue()) + " + "
                         + String.valueOf(szam2.getValue()) + " = "
                         + String.valueOf(szam1.getValue()
                         + szam2.getValue()));
        }
    }

    public static void main(String[] args) {
        Program1 pf = new Program1();
    }
}
```

Az alkalmazás futásának eredményei:



Készítsünk színkeverő alkalmazást, ahol az egyes RGB-összetevők értékét görgetősávval, illetve szövegmezőkben megadva lehet módosítani! (*Swing\Szinkevero*)

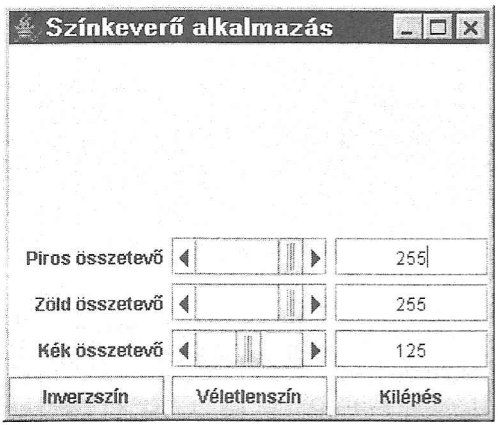
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class SzinKevero extends JFrame
       implements ActionListener, AdjustmentListener, FocusListener
{
    JTextField RErtek, GErtek, BErtek;
    JScrollBar RGorgeto, GGorgeto, BGorgeto;
    JButton    jbInverz, jbVeletlen, jbKilepes;
    JPanel     szinPanel;

    public SzinKevero() {
        super("Színkeverő alkalmazás");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        szinPanel = new JPanel();
        JPanel RGBVezerlok = new JPanel();
        JPanel munkaterulet = new JPanel();
```

```
// a szövegmezők létrehozása
RErtek = new JTextField("0");
RErtek.setHorizontalAlignment(SwingConstants.CENTER);
RErtek.addActionListener(this);
RErtek.addFocusListener(this);
GErtek = new JTextField("0");
GErtek.setHorizontalAlignment(SwingConstants.CENTER);
GErtek.addActionListener(this);
GErtek.addFocusListener(this);
BErtek = new JTextField("0");
BErtek.setHorizontalAlignment(SwingConstants.CENTER);
BErtek.addActionListener(this);
BErtek.addFocusListener(this);
// a görgetősávok létrehozása
RGorgeto = new JScrollBar(SwingConstants.HORIZONTAL,
                          0, 0, 0, 255);
RGorgeto.addAdjustmentListener(this);
GGorgeto = new JScrollBar(SwingConstants.HORIZONTAL,
                          0, 0, 0, 255);
GGorgeto.addAdjustmentListener(this);
BGorgeto = new JScrollBar(SwingConstants.HORIZONTAL,
                          0, 0, 0, 255);
BGorgeto.addAdjustmentListener(this);
// a vezérlőgombok létrehozása
jbInverz = new JButton("Inverzszín");
jbVeletlen = new JButton("Véletlenszín");
jbKilepes = new JButton("Kilépés");
jbInverz.addActionListener(this);
jbVeletlen.addActionListener(this);
jbKilepes.addActionListener(this);
// a vezérlőpanel kialakítása
RGBVezerlok.setLayout(new GridLayout(4, 3, 5, 5));
RGBVezerlok.add(new JLabel("Piros összetevő",
                SwingConstants.RIGHT));
RGBVezerlok.add(RGorgeto);
RGBVezerlok.add(RErtek);
RGBVezerlok.add(new JLabel("Zöld összetevő",
                SwingConstants.RIGHT));
RGBVezerlok.add(GGorgeto);
RGBVezerlok.add(GErtek);
RGBVezerlok.add(new JLabel("Kék összetevő",
                SwingConstants.RIGHT));
RGBVezerlok.add(BGorgeto);
RGBVezerlok.add(BErtek);
RGBVezerlok.add(jbInverz);
RGBVezerlok.add(jbVeletlen);
RGBVezerlok.add(jbKilepes);
// az ablak tartalmának beállítása
munkaterulet.setLayout(new GridLayout(2, 1, 0, 10));
munkaterulet.add(szinPanel);
munkaterulet.add(RGBVezerlok);
```

```
        setContentPane(munkaterulet);
        szinFrissites();
    }

    // a nyomógombesemények kezelése
    public void actionPerformed(ActionEvent evt) {
        Object source = evt.getSource();
        if(source instanceof JTextField)
            gorgetokFrissitese();
        else if(source == jbInverz)
            inverzSzin();
        else if(source == jbVeletlen)
            veletlenSzin();
        else if(source == jbKilepes)
            System.exit(0);
    }

    // görgetéskor frissítjük a szövegmezőket
    public void adjustmentValueChanged(AdjustmentEvent evt) {
        szovegmezokFrissitese();
    }

    // a szövegmezőkbe írt érték alapján módosulnak a görgetősávok
    public void focusGained(FocusEvent evt) { }
    public void focusLost(FocusEvent evt) {
        gorgetokFrissitese();
    }

    void szovegmezokFrissitese() {
        RErtek.setText("" + RGorgeto.getValue());
        GErtek.setText("" + GGorgeto.getValue());
        BErtek.setText("" + BGorgeto.getValue());
        szinFrissites();
    }

    void gorgetokFrissitese() {
        RGorgeto.setValue(Integer.parseInt(RErtek.getText()));
        GGorgeto.setValue(Integer.parseInt(GErtek.getText()));
        BGorgeto.setValue(Integer.parseInt(BErtek.getText()));
        szinFrissites();
    }

    void szinFrissites() {
        int r = RGorgeto.getValue();
        int g = GGorgeto.getValue();
        int b = BGorgeto.getValue();
        szinPanel.setBackground(new Color(r, g, b));
    }

    void inverzSzin() {
        RGorgeto.setValue(255 - RGorgeto.getValue());
        GGorgeto.setValue(255 - GGorgeto.getValue());
        BGorgeto.setValue(255 - BGorgeto.getValue());
        szovegmezokFrissitese();
    }
```

```
    void veletlenSzin() {
        RGorgeto.setValue((int)(Math.random() * 256));
        GGorgeto.setValue((int)(Math.random() * 256));
        BGorgeto.setValue((int)(Math.random() * 256));
        szovegmezokFrissitese();
    }

    // az alkalmazás indítása
    public static void main(String[] args) {
        JFrame frame = new SzinKevero();
        frame.pack();
        frame.setVisible(true);
    }
}
```

Az alkalmazás futásának eredménye:

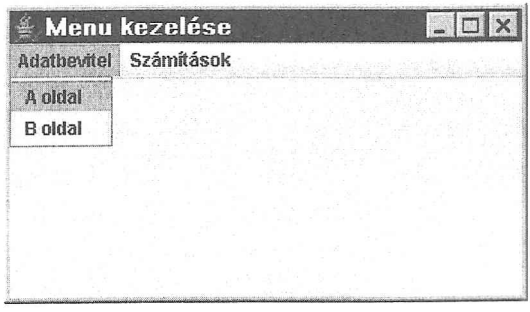


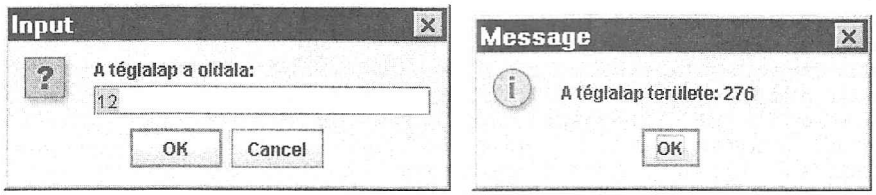
Írjunk menüs Swing-alkalmazást, ahol az adatbekérést és az adatmegjelenítés műveleteit a ***JOptionPane*** párbeszédablakai segítségével végezzük! (*Swing\Menu1*)

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

public class Menu1pr extends JFrame implements ActionListener
{
    JMenuBar menusor = new JMenuBar();
    JMenu menu1 = new JMenu("Adatbevitel");
    JMenuItem almenu1 = new JMenuItem("A oldal");
    JMenuItem almenu2 = new JMenuItem("B oldal");
    JMenu menu2 = new JMenu("Számítások");
    JMenuItem almenu3 = new JMenuItem("Kerület");
    JMenuItem almenu4 = new JMenuItem("Terület");
    JMenuItem almenu5 = new JMenuItem("Kilépés");
    int aoldal=10, boldal=10;
```

```
    public Menu1pr(String s) {
        super(s);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        menu1.add(almenu1).addActionListener(this);
        menu1.add(almenu2).addActionListener(this);
        menusor.add(menu1);
        menu2.add(almenu3).addActionListener(this);
        menu2.add(almenu4).addActionListener(this);
        menu2.addSeparator();
        menu2.add(almenu5).addActionListener(this);
        menusor.add(menu2);
        this.setJMenuBar(menusor);
        this.setSize(350,200);
        this.setVisible(true);
    }
    public void actionPerformed(ActionEvent e) {
        this.setTitle(e.getActionCommand());
        if (e.getActionCommand()=="Kilépés")
            System.exit(0);
        if (e.getActionCommand()=="A oldal")
            aoldal = Integer.parseInt(JOptionPane.showInputDialog(
                                null, "A téglalap a oldala:","12"));
        if (e.getActionCommand()=="B oldal")
            boldal = Integer.parseInt(JOptionPane.showInputDialog(
                                null, "A téglalap a oldala:","23"));
        if (e.getActionCommand()=="Kerület")
            JOptionPane.showMessageDialog(null,
                        "A téglalap kerülete: " + 2*(aoldal+boldal));
        if (e.getActionCommand()=="Terület")
            JOptionPane.showMessageDialog(null,
                        "A téglalap területe: " +(aoldal*boldal));
    }
    public static void main(String argv[]) {
        Menu1pr ablak = new Menu1pr("Menu kezelése");
    }
}
```

Az alkalmazás futásának eredménye:



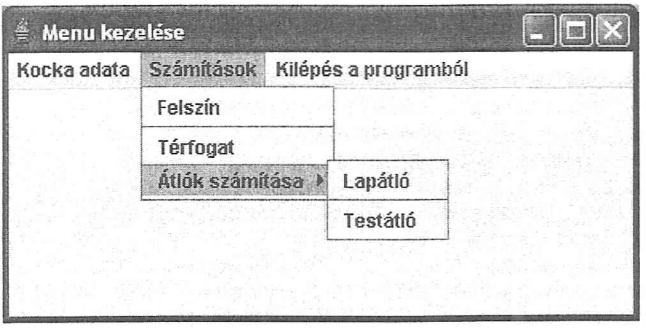


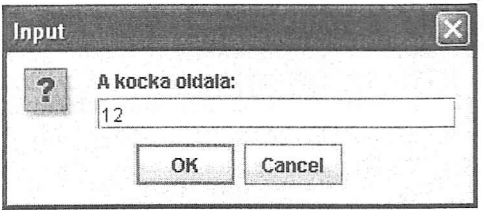
A *Menu1* alkalmazás alapján készítsünk programot, amely bekéri, számolja és megjeleníti egy kocka különböző adatait! (*Swing\Menu2*)

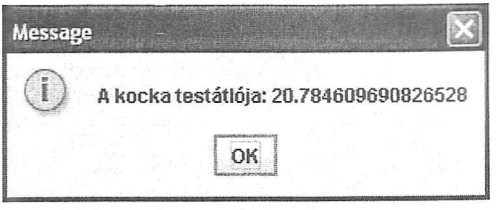
```
import java.awt.event.*;
import javax.swing.*;

public class Menupr extends JFrame implements ActionListener
{
    JMenuBar menusor = new JMenuBar();
    JMenu menu1 = new JMenu("Kocka adata");
    JMenuItem almenu1 = new JMenuItem("Oldaléle");
    JMenu menu2 = new JMenu("Számítások");
    JMenuItem almenu2 = new JMenuItem("Felszín");
    JMenuItem almenu3 = new JMenuItem("Térfogat");
    JMenu menu3 = new JMenu("Átlók számítása");
    JMenuItem almenu4 = new JMenuItem("Lapátló");
    JMenuItem almenu5 = new JMenuItem("Testátló");
    JMenu menu4 = new JMenu("Kilépés a programból");
    JMenuItem almenu6 = new JMenuItem("Kilépés" );
    int oldal=1;

    public Menupr(String s) {
        super(s);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        menu1.add(almenu1).addActionListener(this);
        menusor.add(menu1);

        menu2.add(almenu2).addActionListener(this);
        menu2.addSeparator();
        menu2.add(almenu3).addActionListener(this);
        menu3.add(almenu4).addActionListener(this);
        menu3.addSeparator();
        menu3.add(almenu5).addActionListener(this);
        // almenü beállítása
        menu2.add(menu3);
        menusor.add(menu2);

        menu4.add(almenu6).addActionListener(this);
        menusor.add(menu4);
        this.setJMenuBar(menusor);
        this.setSize(400,200);
        this.setVisible(true);
    }
```

```
    public void actionPerformed(ActionEvent e) {
        this.setTitle(e.getActionCommand());
        if (e.getActionCommand()=="Kilépés")
            System.exit(0);
        if (e.getActionCommand()=="Oldaléle")
         oldal = Integer.parseInt(JOptionPane.showInputDialog(null,
                 "A kocka oldala:","1"));
        if (e.getActionCommand()=="Felszín")
         JOptionPane.showMessageDialog(null, "A kocka felszíne: "
                 +6*(oldal * oldal));
        if (e.getActionCommand()=="Térfogat")
         JOptionPane.showMessageDialog(null, "A kocka térfogata: "
                 + Math.pow(oldal,3));
        if (e.getActionCommand()=="Lapátló")
         JOptionPane.showMessageDialog(null, "A kocka lapátlója: "
                 + oldal * Math.pow(2,1./2));
        if (e.getActionCommand()=="Testátló")
         JOptionPane.showMessageDialog(null, "A kocka testátlója: "
                 + oldal * Math.pow(3,1./2));
    }
    public static void main(String argv[]) {
        Menupr ablak = new Menupr("Menu kezelése");
    }
}
```

Az alkalmazás futásának eredménye:





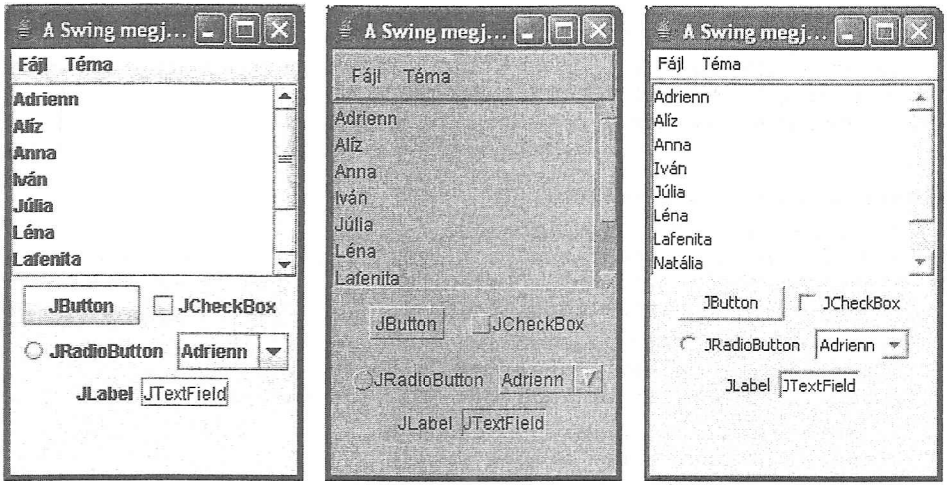


Készítsünk programot, amely bemutatja a *Swing*-vezérlők különböző megjelenítési lehetőségeit! (*Swing\Kinezet*)

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

public class Kinezet extends JFrame
{
    public Kinezet( ) {
    super("A Swing megjelenésének beállítása");
    setSize(200, 350);
    setLocation(230, 120);
    ablakMenu();
    ablakVezerlok();
    setVisible(true);
    }

    protected void ablakMenu() {
        // Fájl menü
        JMenu mnFajl = new JMenu("Fájl", true);
        JMenuItem mpKilpes = new JMenuItem ("Kilépés");
        mnFajl.add(mpKilpes);
        // A kilépésgomb névtelen akcióadapter objektuma
        mpKilpes.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    System.exit(0);
                }
            }
        );

        // Téma menü felépítése
        JMenu mnTema = new JMenu("Téma", true);
        ButtonGroup gombCsoport = new ButtonGroup();
        // A beállítható ablaktémák lekérdezése
        UIManager.LookAndFeelInfo[] tema =
                                UIManager.getInstalledLookAndFeels();
        for (int i = 0; i < tema.length; i++) {
           JRadioButtonMenuItem elem =
                new JRadioButtonMenuItem(tema[i].getName( ), i == 0);
            final String className = tema[i].getClassName();
            elem.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent ae) {
                   try { UIManager.setLookAndFeel(className); }
                   catch (Exception e) { System.out.println(e); }
                   SwingUtilities.updateComponentTreeUI(Kinezet.this);
                }
              }
            );
            gombCsoport.add(elem);
            mnTema.add(elem);
        }
```

```
        // A menüsor felépítése
        JMenuBar menuSor = new JMenuBar( );
        menuSor.add(mnFajl);
        menuSor.add(mnTema);
        setJMenuBar(menuSor);
    }

    protected void ablakVezerlok() {
        // Néhány komponens felhelyezése
        String[] nevek = {"Adrienn", "Alíz", "Anna", "Iván", "Júlia",
                          "Léna", "Lafenita", "Natália",
                          "Petra","Renáta"};
        // nevek görgethető listaablakban
        JPanel foPanel = new JPanel(new GridLayout(2, 1));
        foPanel.add(new JScrollPane(new JList(nevek)));
        // jobb oldalon különböző vezérlők
        JPanel jobbPanel = new JPanel();
        jobbPanel.add(new JButton("JButton"));
        jobbPanel.add(new JCheckBox("JCheckBox"));
        jobbPanel.add(new JRadioButton("JRadioButton"));
        jobbPanel.add(new JComboBox(nevek));
        jobbPanel.add(new JLabel("JLabel"));
        jobbPanel.add(new JTextField("JTextField"));
        // a felület beállítása
        foPanel.add(jobbPanel);
        setContentPane(foPanel);
    }

    public static void main(String[] args)  {
        JFrame ablak = new Kinezet();
        // kilépés az ablak bezárásakor
        ablak.addWindowListener(
            new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                    System.exit(0);
                }
            }
        );
        ablak.setVisible(true);
    }
}
```

Az alkalmazás futásának eredményei a következő oldalon tanulmányozhatók.

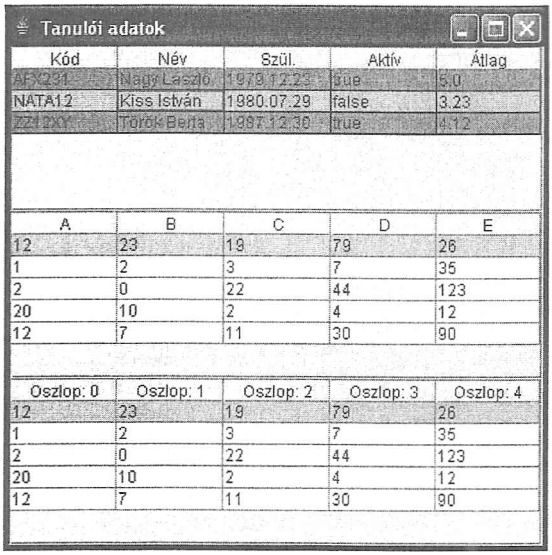


Írjunk programot, amely tanulók adatait, egy mátrixot jeleníti meg táblázatos formában! A táblázatot a *Swing* JTable komponensével hozzuk létre! (*Swing\Tablazat*)

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

public class Tablazat extends JFrame
{
    // a táblázat adatai
    String[] mezok = new String[] {"Kód", "Név", "Szül.", "Aktív",
                                   "Átlag"};
    Object[][] adatok = new Object[][] {
        { "AFX231", "Nagy László", "1979.12.23", Boolean.TRUE,
            new Float(5.00) },
        { "NATA12", "Kiss István", "1980.07.29", Boolean.FALSE,
            new Float(3.23) },
        { "ZZ12XY", "Török Berta", "1987.12.30", Boolean.TRUE,
            new Float(4.12) }
    };

    // a mátrix elemei
    int [][] matrix = new int[][] {
                        { 12, 23, 19, 79,  26 },
                        {  1,  2,  3,  7,  35 },
                        {  2,  0, 22, 44, 123 },
                        { 20, 10,  2,  4,  12 },
                        { 12,  7, 11, 30,  90 } };

    public Tablazat(String cimsor) {
        super(cimsor);
        setSize(400, 400);
        setLocation(230, 120);
```

```
        // A mátrix megjelenítése
        TableModel tablaModell = new AbstractTableModel() {
             public int getColumnCount() {
                 return matrix[0].length;
             }
             public int getRowCount() {
                 return matrix.length;
             }
             public Object getValueAt(int sor, int oszlop) {
                 return new Integer(matrix[sor][oszlop]);
             }
        };
        JTable uresTabla = new JTable(tablaModell);

        // Feltöltött színes tábla létrehozása
        JTable adatTabla = new JTable(adatok, mezok);
        adatTabla.setGridColor(Color.blue);
        adatTabla.setBackground(Color.green);
        adatTabla.setForeground(Color.red);

        // Tábla dinamikus felépítése
        DefaultTableModel defTablaModell = new DefaultTableModel();
        JTable dinTabla = new JTable(defTablaModell);
        for(int oszlop = 0; oszlop < matrix[0].length; oszlop++)
            defTablaModell.addColumn("Oszlop: " + oszlop);
        Object [] adatSor = new Object[matrix[0].length];
        for(int sor = 0; sor < matrix.length; sor++) {
            for(int oszlop = 0; oszlop < matrix[0].length; oszlop++)
                adatSor[oszlop] = new Integer(matrix[sor][oszlop]);
            defTablaModell.addRow(adatSor);
        }

        // A táblák elhelyezése az ablakban
        getContentPane().setLayout(new GridLayout(3,1));
        getContentPane().add(new JScrollPane(adatTabla));
        getContentPane().add(new JScrollPane(uresTabla));
        getContentPane().add(new JScrollPane(dinTabla));
        setVisible(true);
    }

    public static void main(String[] args)  {
        // A táblázatot tároló ablak létrehozása
        Tablazat tabla = new Tablazat("Tanulói adatok");
        tabla.addWindowListener(
            new WindowAdapter() {
                 public void windowClosing(WindowEvent e) {
                     System.exit(0);
                 }
            }
        );
    }
}
```

Az alkalmazás futásának eredménye:



Készítsünk alkalmazást, amely bemutatja a szokásos (névjegy és kiléptető) párbeszédablakok használatát! (*Swing\Parbeszedablakok*)

```
// A főablak osztálya: -- FoAblak.java --
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class FoAblak extends JFrame
{
   public FoAblak() {  // konstruktor
      getContentPane().setLayout(null);
      setSize(400,300);
      setVisible(false);
      setTitle("Swing Alkalmazás");
      // csak mi léptetjük ki
      setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
      // a menürendszer felépítése
      menu1.setText("Műveletek");
      menu1.setActionCommand("Műveletek");
      menu1.setMnemonic((int)'M');
      menu1.add(letrehozMenuItem);
      letrehozMenuItem.setEnabled(false);
      letrehozMenuItem.setText("Létrehozás");
      letrehozMenuItem.setActionCommand("Létrehozás");
      letrehozMenuItem.setMnemonic((int)'L');
      letrehozMenuItem.setAccelerator
            (KeyStroke.getKeyStroke(KeyEvent.VK_L, Event.CTRL_MASK));
      menu1.add(openMenuItem);
```

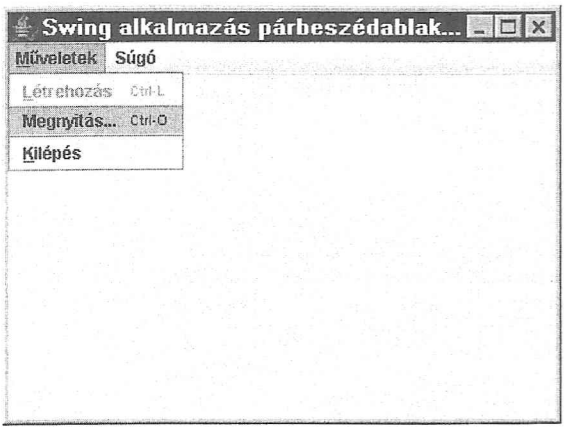
```
    openMenuItem.setText("Megnyitás...");
    openMenuItem.setActionCommand("Megnyitás...");
    openMenuItem.setMnemonic((int)'O');
    openMenuItem.setAccelerator
            (KeyStroke.getKeyStroke(KeyEvent.VK_O, Event.CTRL_MASK));
    menu1.add(elvalasztoMenuItem);
    menu1.add(kilepMenuItem);
    kilepMenuItem.setText("Kilépés");
    kilepMenuItem.setActionCommand("Kilépés");
    kilepMenuItem.setMnemonic((int)'K');
    mainMenuBar.add(menu1);

    menu2.setText("Súgó");
    menu2.setActionCommand("Súgó");
    menu2.setMnemonic((int)'O');
    menu2.add(nevjegyMenuItem);
    nevjegyMenuItem.setText("Névjegy...");
    nevjegyMenuItem.setActionCommand("Névjegy...");
    nevjegyMenuItem.setMnemonic((int)'N');
    mainMenuBar.add(menu2);
    setJMenuBar(mainMenuBar);

    // az eseményfigyelők regisztrálása
    this.addWindowListener(new FoWindowAdapter());
    FoActionListener aListener = new FoActionListener();
    openMenuItem.addActionListener(aListener);
    kilepMenuItem.addActionListener(aListener);
    nevjegyMenuItem.addActionListener(aListener);
}

public FoAblak(String felirat) { // paraméteres konstruktor
    this(); // a paraméternélküli konstruktor hívása
    setTitle(felirat);
}

// az alkalmazást indító main() metódus
static public void main(String args[]) {
    // az alkalmazás ablakának létrehozása és megjelenítése
    (new FoAblak("Swing alkalmazás párbeszédablakokkal"))
                .setVisible(true);
}

// A kilépést megvalósító adapter
class FoWindowAdapter extends WindowAdapter
{
     // az ablak bezárása
     public void windowClosing(WindowEvent event) {
        Object object = event.getSource();
        if (object == FoAblak.this)
             kilépesAlkalmazasbol();
     }
}
```

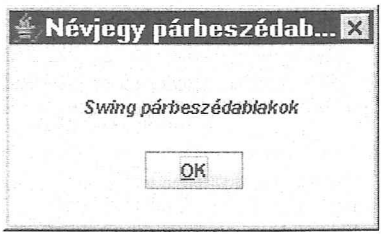
```
// A menüesemények kezelése
class FoActionListener implements ActionListener
{
    public void actionPerformed(ActionEvent event) {
        Object object = event.getSource();
        if (object == openMenuItem)
            openMenuItem_ActionPerformed(event);
        else if (object == nevjegyMenuItem)
            nevjegyMenuItem_ActionPerformed(event);
        else if (object == kilepMenuItem)
            kilépesAlkalmazasbol();
    }
}

// a Fájlnyitás párbeszédablak létrehozása és modális megjelenítése
void openMenuItem_ActionPerformed(ActionEvent event) {
    openFileDialog1 =
        new FileDialog(this, "Fájlnyitás", FileDialog.LOAD);
        openFileDialog1.setVisible(true);
        String konyvtar = openFileDialog1.getDirectory();
        String fajl     = openFileDialog1.getFile();
        if (konyvtar != null)
            setTitle(konyvtar + fajl);
}

// A Névjegy párbeszédablak létrehozása és modális megjelenítése
void nevjegyMenuItem_ActionPerformed(ActionEvent event) {
    NevjegyDialog nevjegy = new NevjegyDialog(this);
    nevjegy.setModal(true);
    nevjegy.setVisible(true);
}

// A kilépési folyamat kezelése visszakérdezéssel
void kilépesAlkalmazasbol() {
        // hangjelzés
        Toolkit.getDefaultToolkit().beep();
        // a beépített jóváhagyó párbeszédablak
        int valasz = JOptionPane.showConfirmDialog(this,
                        "Valóban ki akar lépni?",
                        "Kilépés párbeszédablak" ,
                        JOptionPane.YES_NO_OPTION,
                        JOptionPane.QUESTION_MESSAGE);
        // ha a válasz igen
        if (valasz == JOptionPane.YES_OPTION) {
            this.setVisible(false);     // az ablak eltűntetése
            this.dispose();             // erőforrások felszabadítása
            System.exit(0);             // kilépés az alkalmazásból
        }
}

// deklarációk
FileDialog openFileDialog1 = new FileDialog(this);
// a menüelemek deklarációi
JMenuBar mainMenuBar = new JMenuBar();
JMenu menu1 = new JMenu();
JMenuItem letrehozMenuItem = new JMenuItem();
```

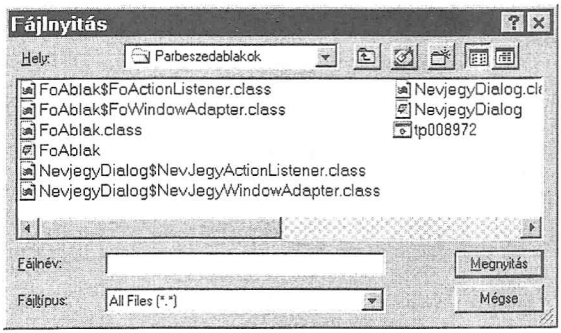
```
   JMenuItem openMenuItem = new JMenuItem();
   JSeparator elvalasztoMenuItem = new JSeparator();
   JMenuItem kilepMenuItem = new JMenuItem();
   JMenu menu2 = new JMenu();
   JMenuItem nevjegyMenuItem = new JMenuItem();
}

// Névjegy párbeszédablak: -- NevjegyDialog.java --
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class NevjegyDialog extends JDialog
{
   public NevjegyDialog(Frame parent) {
       super(parent);
       getContentPane().setLayout(null);

       setSize(250,150);
       setVisible(false);
       setModal(true);

       felirat.setText("Swing párbeszédablakok");
       felirat.setBounds(40,30,160,20);
       felirat.setHorizontalAlignment(SwingConstants.CENTER);
       felirat.setFont(new Font("Dialog",
                          Font.BOLD | Font.ITALIC, 12));
       getContentPane().add(felirat);

       okButton.setText("OK");
       okButton.setActionCommand("OK");
       okButton.setMnemonic((int)'O');
       okButton.setBounds(90,70,65,25);
       getContentPane().add(okButton);
       setTitle("Névjegy párbeszédablak");

       this.addWindowListener(new NevJegyWindowAdapter());
       okButton.addActionListener(new NevJegyActionListener());
   }

   public NevjegyDialog(Frame parent, String title) {
      this(parent);
      setTitle(title);
   }

   // A szülőablak közepére igazítjuk a párbeszédablakot
   public void setVisible(boolean b) {
       if(b) {
           Rectangle bounds = (getParent()).getBounds();
           Dimension size = getSize();
           setLocation(bounds.x + (bounds.width - size.width)/2,
                       bounds.y + (bounds.height -
                       size.height)/2);
            Toolkit.getDefaultToolkit().beep();
       }
       super.setVisible(b);
   }
```

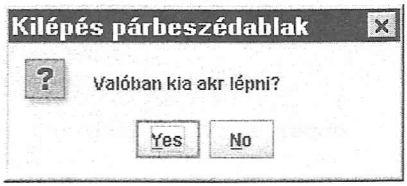
```
    class NevJegyActionListener implements ActionListener
    {
        public void actionPerformed(java.awt.event.ActionEvent event) {
            Object object = event.getSource();
            if (object == okButton)
                NevjegyDialog.this.setVisible(false);
        }
    }

    class NevJegyWindowAdapter extends WindowAdapter
    {
        public void windowClosing(WindowEvent event) {
            Object object = event.getSource();
            if (object == NevjegyDialog.this)
                NevjegyDialog.this.setVisible(false);
        }
    }

    // deklarációk
    JLabel felirat = new JLabel();
    JButton okButton = new JButton();
}
```

Az alkalmazás futásának eredménye:









10.3.3 Összetett fájlkezelő Swing-alkalmazások

Két példán keresztül szemléltetjük, miként alkalmazhatjuk a fejezetben elmondottakat állománykezelő alkalmazásokban.

Tervezzünk személyes adatok szöveges állományban való tárolására használható alkalmazást, *Swing* grafikus felülettel! (*Swing\Cimtar*)

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.Vector;
import javax.swing.*;
import javax.swing.border.LineBorder;

public class Cimtar extends JFrame
{
    public Cimtar() {  // konstruktor
        final String[] sFelirat= { "Név:", "Ir.szám:", "Város:",
                        "Utca:", "Házszám:", "Telefon:", "eMail:"};
        mszam = sFelirat.length;
        // A vezérlőelemek létrehozása
        borderLayout1 = new BorderLayout();
        vKeret = new LineBorder(Color.yellow); // sárga keret
        jScrollPane1 = new JScrollPane(); // mező a lista görgetéséhez
        jPanel1 = new JPanel();
        jList1 = new JList();
        jbHozzaad = new JButton("Hozzáad");
        jbTorol = new JButton("Töröl");
        jbKilep = new JButton("Kilép");
        jbMegjelenit = new JButton("Megjelenít");

        jLabel1 = new JLabel();
        jCimkek = new JLabel[mszam];
        for (int i=0; i<mszam; i++)
          jCimkek[i] = new JLabel(sFelirat[i]);

        jSzovegMezok = new JEditorPane[mszam];
        for (int i=0; i<mszam; i++)
          jSzovegMezok[i] = new JEditorPane();

        adatok = new MemFajl();

        try {
            jAlkalmazasInit();
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
```

```
// a memóriából az adatsorokat állományba írjuk
// az adatelemeket a % jel tagolja a sorban
void adatokFajlbaIrasa() {
    byte abyte0[] = new byte[0]; // ezzel a referenciával
                                 // hivatkozunk a sztringből
    String s = "";               // készített bájtsorra
    try {
        FileOutputStream foutput =
               new FileOutputStream("cimlista.txt");
        for(int index = 0; index < adatok.nev.size(); index++) {
            for (int i=0; i<mszam; i++)
              s += adatok.getMezo(i,index) + "%";
            s += "\n";
        }
         byte abyte1[] = s.getBytes();
        foutput.write(abyte1);
        foutput.close();
    }
    catch(Exception _ex) { }
}

// az adatsorokat az állományból felolvassuk a memóriába
// az adatokat a % karakter választja el egymástól a sorban
void adatokFajlbolListaba() {
    int i = 23;
    int j = 0;
    StringBuffer spuffer = new StringBuffer();
    try {
        FileInputStream finput =
           new FileInputStream("cimlista.txt");
        while(i != -1)
            try {
                i = finput.read();
                if(i >= 32)     // a vezérlő karaktereket kihagyjuk
                    if(i != 37)  // % elválasztó karakter-e
                        spuffer.append((char)i);
                    else {
                        adatok.setMezo(j, spuffer.toString());
                        j = j==mszam-1 ? 0 : j+1;
                        spuffer = spuffer.delete
                                          (0, spuffer.length());
                    }
            }
            catch(Exception e) {
                System.out.println("Fajlolvasasi hiba \n" + e);
            }
    }
    catch(FileNotFoundException e) {
        System.out.println("A fajl nem talalhato \n" + e);
    }
     // A nevek felvitele a listába
    jList1.setListData(adatok.nev);
}
```

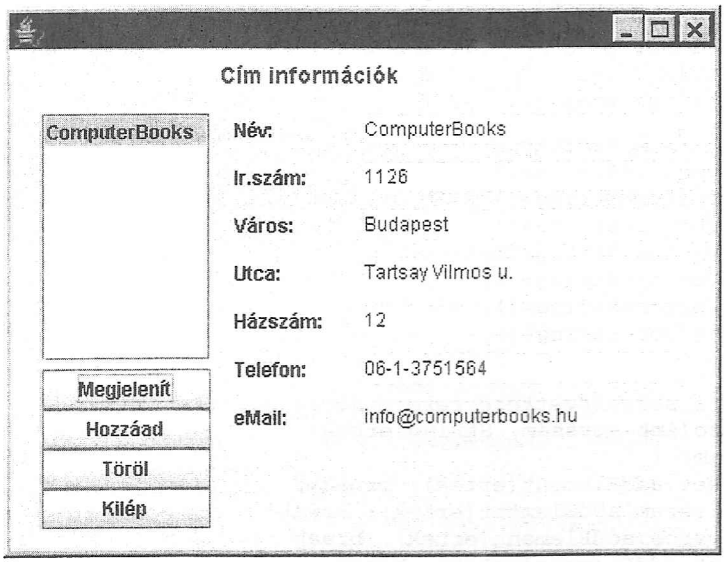
```
// A Hozzáad gomb eseménykezelője
void jbHozzaad_actionPerformed(ActionEvent actionevent) {
   jList1.removeAll();
   for (int i=0; i<mszam; i++)
         adatok.setMezo(i,jSzovegMezok[i].getText());
   jList1.setListData(adatok.nev); // A nevek felvitele a listába
   adatokFajlbaIrasa();
}

// A Töröl gomb eseménykezelője
void jbTorol_actionPerformed(ActionEvent actionevent) {
    jList1.removeAll();
    int index = jList1.getSelectedIndex();
    if(index >= 0) {
        adatok.delRekord(index);
        jList1.setListData(adatok.nev);
        szovegMezokTorlese();
    }
    adatokFajlbaIrasa();
}

// A Kilép gomb eseménykezelője
void jbKilep_actionPerformed(ActionEvent actionevent) {
    System.exit(0);
}

// A Megjelenít gomb eseménykezelője
void jbMegjelenit_actionPerformed(ActionEvent actionevent) {
    int index = jList1.getSelectedIndex();
    if(index >= 0)
        for (int i=0; i<mszam; i++)
           jSzovegMezok[i].setText(adatok.getMezo(i,index));
}

// Az alkalmazás GUI felépítése
private void jAlkalmazasInit() throws Exception {
    getContentPane().setLayout(borderLayout1);
    setSize(new Dimension(450, 345));
    jPanel1.setLayout(null);
    // A listát görgetősávval látjuk el
    jScrollPane1.getViewport().setView(jList1);
    jScrollPane1.setBounds(new Rectangle(20, 40, 105, 155));
    jList1.setBounds(new Rectangle(20, 40, 105, 155));
    // A nyomógombok elhelyezése
    jbMegjelenit.setBounds(new Rectangle(20, 200, 105, 25));
    jbHozzaad.setBounds(new Rectangle(20, 225, 105, 25));
    jbTorol.setBounds(new Rectangle(20, 250, 105, 25));
    jbKilep.setBounds(new Rectangle(20, 275, 105, 25));
    // Minden nyomógombhoz külön eseménykezelőt hozunk létre
    jbHozzaad.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent actionevent) {
             jbHozzaad_actionPerformed(actionevent);
         }
    });
```

```
        jbTorol.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent actionevent) {
                jbTorol_actionPerformed(actionevent);
            }
        });
        jbKilep.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent actionevent) {
                jbKilep_actionPerformed(actionevent);
            }
        });
        jbMegjelenit.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent actionevent) {
                jbMegjelenit_actionPerformed(actionevent);
            }
        });
        // A címet tároló címke beállítása
        jLabel1.setText("Cím információk");
        jLabel1.setHorizontalAlignment(0);
        jLabel1.setForeground(Color.red);
        jLabel1.setFont(new Font("Dialog", 1, 14));
        jLabel1.setBounds(new Rectangle(75, 5, 225, 20));
        // Az adatbeviteli címke/szövegmező párok beállítása
        for (int i=0; i<mszam; i++) {
            jCimkek[i].setHorizontalAlignment(2);
            jCimkek[i].setForeground(Color.blue);
            jCimkek[i].setBounds(new Rectangle(140, 40+i*30, 65, 20));
            jSzovegMezok[i].setBounds(new Rectangle(220, 40+i*30,
                                                    200, 20));
            jSzovegMezok[i].setBorder(vKeret);
        }
        szovegMezokTorlese();
        jList1.setBorder(vKeret);
        getContentPane().add(jPanel1, "Center");
        adatokFajlbolListaba();
        // A vezérlők felhelyezése a panelra
        jPanel1.add(jScrollPane1);
        jPanel1.add(jbHozzaad, null);
        jPanel1.add(jbTorol, null);
        jPanel1.add(jbKilep, null);
        jPanel1.add(jbMegjelenit, null);
        jPanel1.add(jLabel1, null);
        for (int i=0; i<mszam; i++) {
            jPanel1.add(jCimkek[i], null);
            jPanel1.add(jSzovegMezok[i], null);
        }
        setVisible(true);
        addWindowListener(new WindowAdapter() { // az ablak zárása
            public void windowClosing(WindowEvent windowevent) {
                System.exit(0);
            }
        });
    }
```

```
    void szovegMezokTorlese() { // a szövegmezők tartalmának törlése
        for (int i=0; i<mszam; i++)
            jSzovegMezok[i].setText("");
    }

    // Az alkalmazás indítása
    public static void main(String args[]) {
        Cimtar cimtar = new Cimtar();
    }

    // Adatmező-deklarációk
    BorderLayout borderLayout1;
    LineBorder vKeret;
    JScrollPane jScrollPane1;
    JPanel jPanel1;
    JList jList1;
    JButton jbHozzaad, jbTorol;
    JButton jbKilep, jbMegjelenit;
    JLabel jLabel1, jCimkek[];
    JEditorPane[] jSzovegMezok;
    MemFajl adatok;
    int mszam;
}

// A memóriában tárolt fájltartalom osztálya, amelyben
class MemFajl {        // minden adatmezőt sorszám azonosít
            Vector<String> nev;       // 0
            Vector<String> irszam;    // 1
            Vector<String> varos;     // 2
            Vector<String> ut;        // 3
            Vector<String> hsz;       // 4
            Vector<String> tel;       // 5
            Vector<String> email;     // 6

    public MemFajl() { // konstruktor
        nev    = new Vector<String>();
        irszam = new Vector<String>();
        varos  = new Vector<String>();
        ut     = new Vector<String>();
        hsz    = new Vector<String>();
        tel    = new Vector<String>();
        email  = new Vector<String>();
    }

    // Adat felvitele a soronkövetkező rekord adott sorszámú mezőjébe
    public void setMezo(int sorszam, String ertek) {
        switch (sorszam) {
            case 0: nev.addElement(ertek); break;
            case 1: irszam.addElement(ertek); break;
            case 2: varos.addElement(ertek); break;
            case 3: ut.addElement(ertek); break;
            case 4: hsz.addElement(ertek); break;
            case 5: tel.addElement(ertek); break;
            case 6: email.addElement(ertek); break;
        }
```

```
    }
    // Adott indexű rekord, adott sorszámú mezőjének lekérdezése
    public String getMezo(int sorszam, int index) {
        String ertek = "";
        switch (sorszam) {
            case 0: ertek = nev.elementAt(index); break;
            case 1: ertek = irszam.elementAt(index); break;
            case 2: ertek = varos.elementAt(index); break;
            case 3: ertek = ut.elementAt(index); break;
            case 4: ertek = hsz.elementAt(index); break;
            case 5: ertek = tel.elementAt(index); break;
            case 6: ertek = email.elementAt(index); break;
        }
        return ertek;
    }

    // Adott indexű rekord eltávolítása
    public void delRekord(int index) {
        nev.removeElementAt(index);
        irszam.removeElementAt(index);
        varos.removeElementAt(index);
        ut.removeElementAt(index);
        hsz.removeElementAt(index);
        tel.removeElementAt(index);
        email.removeElementAt(index);
    }
}
```

Az alkalmazás futásának eredménye:



Fejlesszünk típusos (*random access*) fájlkezelést végző alkalmazást *Swing* grafikus felülettel! (*Swing\RandomfajlGUI*)

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;

class Application extends JFrame implements ActionListener
{
    // Az alkalmazás indítása
    public static void main(String[] args) {
       Application azApplication = new Application("Egyszerű adatbázis");
       azApplication.setSize(380,240);
       azApplication.setVisible(true);
    }

    // Az adatmezők deklarációi
    long recsum, recno;
    JButton btnFrissit, btnHozzaad, btnKovetkezo, btnElozo,
            btnElso, btnUtolso;
    JLabel labNev, labKor, labFizetes;
    JTextField txtNev, txtKor, txtFizetes;
    JPanel [] panels = new JPanel[2];

    // paraméteres konstruktor
    public Application(String nev) {
       super(nev);
       // Kilép az ablak bezárásakor
       setDefaultCloseOperation(EXIT_ON_CLOSE);
       initGUI(); // A grafikus felület kialakítása
    }

    // GridBagLayout segédfüggvény
    private void  gbcParameterzes(GridBagConstraints gbc, int gx,
                          int gy, int gw, int gh, int wx, int wy) {
       gbc.gridx=gx;
       gbc.gridy=gy;
       gbc.gridwidth=gw;
       gbc.gridheight=gh;
       gbc.weightx=wx;
       gbc.weighty=wy;
    }

    // A grafikus felhasználói felület kialakítása
    public void initGUI() {
       Container munkaterulet = getContentPane();
       // Két panelre osztjuk  a felületet
       for (int i=0; i<panels.length; i++)
           panels[i] = new JPanel();
       munkaterulet.setLayout(new GridLayout(2,1,5,5));
       for (int i=0; i<panels.length; i++)
             munkaterulet.add(panels[i]);
```

```
// Adatmezők GridBagLayout elrendezésben
labNev = new JLabel("  Név");
labKor = new JLabel("  Életkor");
labFizetes = new JLabel("  Fizetés");
txtNev = new JTextField("");
txtKor = new JTextField("") ;
txtFizetes = new JTextField("") ;

panels[0].setLayout(new GridBagLayout());
GridBagConstraints gbc = new GridBagConstraints();
gbc.fill=GridBagConstraints.BOTH;
gbc.anchor=GridBagConstraints.CENTER;

gbcParameterzes(gbc,0,0,1,1,10,20);
panels[0].add(labNev,gbc);              // 0,0
gbcParameterzes(gbc,1,0,1,1,90,0);
panels[0].add(txtNev,gbc);              // 1,0

gbcParameterzes(gbc,0,1,1,1,10,20);
panels[0].add(labKor,gbc);              // 0,1
gbcParameterzes(gbc,1,1,1,1,90,0);
panels[0].add(txtKor,gbc);              // 1,1

gbcParameterzes(gbc,0,2,1,1,10,20);
panels[0].add(labFizetes,gbc);          // 0,2
gbcParameterzes(gbc,1,2,1,1,90,0);
panels[0].add(txtFizetes,gbc);          // 1,2
panels[0].doLayout();

// A vezérlőgombok rácsos elrendezésben
panels[1].setLayout(new GridLayout(4,1,0,2));

btnHozzaad = new JButton("Hozzáadás új rekordként");
panels[1].add(btnHozzaad);
btnHozzaad.addActionListener(this);

btnFrissit = new JButton("Rekord frissítése");
panels[1].add(btnFrissit);
btnFrissit.addActionListener(this);

btnElso = new JButton("Első rekord");
panels[1].add(btnElso);
btnElso.addActionListener(this);

btnUtolso = new JButton("Utolsó rekord");
panels[1].add(btnUtolso);
btnUtolso.addActionListener(this);

btnElozo = new JButton("Előző rekord");
panels[1].add(btnElozo);
btnElozo.addActionListener(this);

btnKovetkezo = new JButton("Következő rekord");
panels[1].add(btnKovetkezo);
btnKovetkezo.addActionListener(this);
```

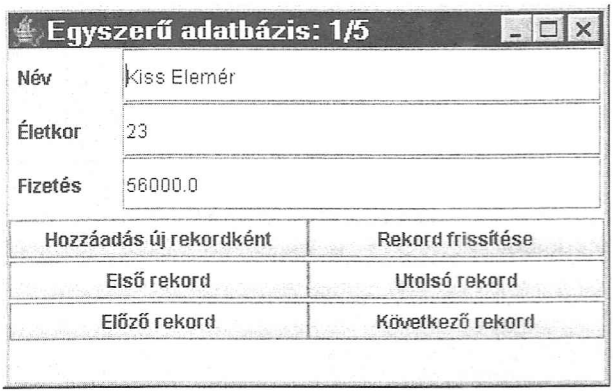
```
    // Ha az állomány nem létezik, létrehozzuk
    recsum = fileSize();
    if (recsum <= 0) {
       try {
            RandomAccessFile file =
                      new RandomAccessFile("Database.dat","rw");
            file.close();
            recno  = -1; recsum = 0;
            }
       catch (IOException ioe){ }
    }
    else
        recno = showRecord(0);
    kijelzes();
}

// A rekordszám meghatározása
public long fileSize() {
    long sr = -1;
    Record rc = new Record();
    try {
       RandomAccessFile file=
                new RandomAccessFile("Database.dat","r");
       try {
          while (true) {
              sr++;
              file.seek(sr*rc.getSize());
              rc.read(file);
          }
       }
       finally {
           file.close();
       }
    }
    catch (IOException ioe) { }
    return sr;
}

// A kért rekord beolvasása és megjelenítése
public long showRecord(long rno) {
    try {
       txtNev.setText(""); txtKor.setText("");
       txtFizetes.setText("");

       Record rc = new Record();
       RandomAccessFile file=
                new RandomAccessFile("Database.dat","r");
       file.seek(rno * rc.getSize());
       rc.read(file);
       file.close();

       txtNev.setText(rc.getNev());
       txtKor.setText(""+rc.getKor());
       txtFizetes.setText(""+rc.getFizetes());
    } catch (IOException ioe) { }
    return rno;
}
```

```
    // A rekordsorszám kijelzése az ablak címsorában
    private void kijelzes() {
       setTitle("Egyszerű adatbázis: " + (recno+1) + "/" + (recsum));
    }

    // A gombnyomásesemények kezelése
    public void actionPerformed(ActionEvent e) {
       // az aktuális rekord frissítése
       if (e.getSource() == btnFrissit) {
          try {
               Record rc = new Record(txtNev.getText(),
               Integer.parseInt(txtKor.getText()), Double.valueOf
                      (txtFizetes.getText()).doubleValue());
               RandomAccessFile file=
                      new RandomAccessFile("Database.dat","rw");
               file.seek(recno*rc.getSize());
               rc.write(file);
               file.close();
          }
          catch (IOException ioe) { }
       }
       // Új rekord hozzáadása a fájl végéhez
       else if (e.getSource() == btnHozzaad) {
          try {
               Record rc = new Record(txtNev.getText(),
               Integer.parseInt(txtKor.getText()), Double.valueOf
                       (txtFizetes.getText()).doubleValue());
               RandomAccessFile file=
                       new RandomAccessFile("Database.dat","rw");
               file.seek(recsum*rc.getSize());
               rc.write(file);
               recsum++; recno++;
               file.close();
          }
          catch (IOException ioe) { }
       }
       // Mozgás a fájlon belül
       else if (e.getSource() == btnElso) {
          recno = showRecord(0);
       }
       else if (e.getSource() == btnElozo) {
          if (recno>0)
             recno = showRecord(recno-1);
       }
       else if (e.getSource() == btnKovetkezo) {
          if (recno<recsum-1)
               recno = showRecord(recno+1);
       }
       else if (e.getSource() == btnUtolso) {
          recno = showRecord(recsum-1);
       }
       kijelzes();
    }
}
```

```
// Az adatrekord osztálya
class Record
{
    // Adatmezők
    boolean _ervenyes;  // érvényes-e a rekord (törléshez) (1 bájt)
    int _kor;           // életkor (4 bájt)
    double _fizetes;    // fizetés (8 bájt)
    String _nev;        // név

    // Konstruktorok
    Record() {
       this("",0,0,false);
    }

    Record(String nev, int kor, double fizetes){
       this(nev, kor, fizetes, true);
    }

    Record(String nev, int kor, double fizetes, boolean ervenyes){
       setNev(nev);
       setKor(kor);
       setFizetes(fizetes);
       setErvenyes(ervenyes);
    }

    // Rögzített rekordmérettel dolgozunk
    // érvényes: 1, kor: 4, fizetés: 8, név: max 109, rekordvége:1
    int getSize() {
       return 123;
    }

    // A _nev mező elérési metódusai
    String getNev() {
       return _nev;
}
    void setNev(String nev) {
       _nev = nev;
}

    // A _kor mező elérési metódusai
    public int getKor() {
       return _kor;
    }
    public void setKor(int kor) {
       _kor = kor;
    }

    // A _fizetes mező elérési metódusai
    public double getFizetes() {
       return _fizetes;
    }
    public void setFizetes(double fizetes) {
       _fizetes = fizetes;
    }
```

```
    // Az _ervenyes mező elérési metódusai
    public boolean getErvenyes() {
       return _ervenyes;
}
    public void setErvenyes(boolean ervenyes) {
       _ervenyes = ervenyes;
    }

    // Adatok írása állományba
    public void write(RandomAccessFile file) throws IOException {
       long fp = file.getFilePointer();
       file.writeBoolean(_ervenyes);
       file.writeInt(_kor);
       file.writeDouble(_fizetes);
       file.writeUTF(_nev);
       file.seek(fp + getSize()-1);
       file.writeByte(255);  // a rekord végének jelzése
    }

    // Adatok olvasás állományba
    public void read(RandomAccessFile file) throws IOException {
       _ervenyes = file.readBoolean();
       _kor = file.readInt();
       _fizetes = file.readDouble();
       _nev = file.readUTF();
    }
}
```

Az alkalmazás futásának eredménye:



11. Grafika programozása

Minden grafikus felhasználói felületet kezelő rendszerben megtalálható egy adatstruktúra, amelyben a grafikus környezet (*context*) adatait tárolja. A grafikus környezet lehet a teljes képernyő, a képernyő egy kisebb része (mint amilyen egy komponens), vagy akár a nyomtató. A megoldás előnye, hogy minden grafikus környezet (kontextus) különböző grafikus jellemzőkkel (háttérszín, rajzszín, betűtípus stb.) rendelkezhet.

11.1 Grafikus megjelenítés AWT és Swing környezetben

Java-ban hagyományosan a ***java.awt.Graphics*** osztály képviseli a grafikus környezetet, melynek példányai akkor jönnek létre, amikor a komponensek megjelennek a képernyőn (*setVisible(**true**);*). A grafikus környezet objektumát ***Component***, illetve a belőle származtatott osztályok ***paint()*** és ***update()*** metódusai paraméterként kapják, azonban a ***getGraphics()*** metódussal magunk is lekérdezhetjük.

`void paint(Graphics g)`	a komponens megfestése,
`void paintAll(Graphics g)`	az adott komponens és a rajta található összes építőelem megfestése,
`void update(Graphics g)`	*AWT*-komponensek esetén törli a hátteret és hívja a ***paint()*** metódust,
`Graphics getGraphics()`	a nem megjelenített építőelem esetén **null** értékkel tér vissza.

A komponensek újrafestését a futtató rendszer automatikusan kezdeményezi, ha

- a komponens először jelenik meg a képernyőn,
- átméretezték,
- az építőelem, vagy annak egy része kikerült a takarásból.

A Java-alkalmazásból a komponens ***repaint()*** metódusának hívásával kérhetjük felületének újrafestését: A ***repaint()*** az AWT (ún. nehézsúlyú) építőelemek esetén az ***update()*** metódust aktivizálja, míg a (könnyűsúlyú) *Swing*-építőelemek esetén közvetlenül a ***paint()*** metódust hívja.

`void repaint()`	a teljes komponens újrafestése,
`void repaint(int x, int y, int szélesség, int magasság)`	komponens adott téglalapalakú területének újrafestése.

AWT-építőelemek esetén az eredeti ***paint()*** metódus semmit sem tesz, azonban saját építőelemet származtatva belőle, kiegészíthetjük a látványát. *Swing* környezetben az

ősosztály ***paint()*** metódusa gondoskodik a megjelenítésről, így azt mindig ajánlott meghívni az utódban felülbírált metódusból, például: ***super.paint(g);***

*A **JComponent*** osztálytól származó *Swing*-komponensek megjelenítése valamelyest eltér az *AWT*-elemekétől, azonban ennek részletezésétől eltekintünk. Annyit azonban érdemes megemlíteni, hogy külön metódusok gondoskodnak az építőelem, a keret és a gyermekkomponensek festéséről. Mivel a ***repaint()*** hívással az újrafestést nem azonnal, hanem „amilyen gyorsan csak lehet" elvet követve végzi a rendszer, hasznosak lehetnek a ***JComponent*** azonnali újrafestést kérő ***paintImmediately()*** metódusai.

```
void paintImmediately(int x, int y, int w, int h)
void paintImmediately(Rectangle r)
```

A teljes *Swing*-építőelem újrafestését az alábbi hívással kérhetjük:

```
komponens.paintImmediately(komponens.getBounds());
```

11.2 A grafikus környezet (context) állapotváltozói

A grafikus kontextust megvalósító ***Graphics*** osztály objektumai egy sor belső tárolóval (állapotváltozóval) rendelkeznek, amelyek meghatározzák a megjelenítést az adott környezetben.

Belső tároló	*Leírás*
felület	A ***Graphics*** objektumhoz rendelt és általa kezelt képernyőfelület. (Egy adott felület több objektumhoz is tartozhat.)
aktuális szín	A felületen történő (vonalas vagy festett) rajzoláshoz használt szín. A szín nincs hatással a képek megjelenítésére.
aktuális betűtípus	A felületen a szöveg aktuális színnel és betűtípussal jelenik meg.
vágó téglalap	A grafikus felület téglalap alakú területe, amit a ***Graphics*** objektum megrajzol/fest. A téglalapon kívüli felületrészek változatlanok maradnak.
festési mód	A *Paint* (festés) módban egyszerűen felülírja a grafikus objektum a felület képpontjait. *XOR* módban a felület aktuális képpontja és az aktuális szín között logikai xor műveletet végez. Kivételt képeznek a felület aktuális (alternatív) színű képpontjai, melyek színét az alternatív (aktuális) színnel cseréli fel. A *XOR* mód lényege, hogy ugyanazt a műveletet kétszer elvégezve, visszakapjuk az eredeti állapotot.
origó	Az origó, a koordináta-rendszer kezdőpontja általában a bal felső sarokban található, azonban áthelyezhető. Alaphelyzetben a rajzterületet a *(0,0)*, *(getSize().width-1, getSize().height-1)* pontok határolják.

A belső tárolók értékét a ***Graphics*** osztály metódusainak segítségével lekérdezhetjük, illetve módosíthatjuk.

Az aktuális szín

```
void setColor(Color c)
Color getColor()
```

Az aktuális színt a 10. fejezetben bemutatott ***Color*** osztály példányaival beállíthatjuk, illetve megtudhatjuk. *AWT*-építőelemek esetén az alapértelmezés szerinti szín a fekete, míg ***Swing*** esetén függ a megjelenítés stílusától.

Az aktuális betűtípus

```
void setFont(Font font)
Font getFont()
FontMetrics getFontMetrics()
FontMetrics getFontMetrics(Font f)
```

A 10. fejezetben megemlített ***Font*** osztály példányai képviselik a betűtípusokat. A ***getFontMetrics()*** hívásával az aktuális, illetve a megadott betűtípusról szerezhetünk információkat. Például, a ***FontMetrics*** osztály ***getHeight()*** metódusával megtudhatjuk az adott font betűinek befoglaló magasságát, a ***stringWidth(sztring)*** hívással pedig megkapjuk az argumentumban szereplő szöveg megjelenítési szélességét. Az alapértelmezés szerinti betűtípusok *AWT* és *Swing* környezetben különböznek:

```
new Font("Dialog", Font.PLAIN, 12)    // AWT
new Font("Dialog", Font.BOLD, 12)     // Swing
```

A vágó téglalap

`void clipRect(int x, int y, int szélesség, int magasság )`	Az aktuális és a megadott téglalap metszete lesz az új vágó téglalap,
`void setClip(int x, int y, int szélesség, int magasság )`	A megadott téglalap lesz az új vágó téglalap,
`void setClip(Shape clip)`	A *clip* alakzat határozza meg az új vágó téglalapot,
`Shape getClip()`	Az aktuális vágóterület lekérdezése,
`Rectangle getClipBounds()`	Új téglalapban adja vissza az aktuális vágó téglalapot,
`Rectangle getClipBounds( Rectangle r)`	A megadott téglalapban az aktuális vágó téglalappal tér vissza.

A metódusok segítségével különböző módon korlátozhatjuk a rajzterületet, az aktuális grafikus felületen belül.

A festési mód

`void setPaintMode()`	A felülfestési (*paint*) mód kijelölése.
`void setXORMode(Color c1)`	A *XOR* mód beállítása, ahol *c1* az alternatív szín.

Az origó

```
void translate (int x, int y)
```

A ***translate()*** metódus hívásával koordináta-rendszer origója a megadott (*x*,*y*) pontba kerül. Amennyiben kerettel ellátott komponensre rajzolunk (***Frame***, ***JFrame***), a munkaterület méretének meghatározásánál figyelembe kell venni a keret és az ablakfej által lefoglalt részeket is. Ebben segítségünkre lehet a ***Container*** és a ***JComponent*** osztályok ***getInsets()*** metódusa által visszaadott ***java.awt.Insets*** osztály:

```
// Az origó áthelyezése az ablak munkaterületének bal felső sarkába
g.translate(getInsets().left, getInsets().top);
// Az ablak befoglaló méreteinek lekérdezése
Rectangle r = getBounds();
// A munkaterület méretének meghatározása
r.height -= getInsets().bottom + getInsets().top;
r.width  -= getInsets().left + getInsets().right;
```

11.3 Grafikus primitívek

A ***Graphics*** osztály metódusainak segítségével grafikus primitíveket (vonalakat, téglalapokat, lekerekített téglalapokat, köröket stb.) rajzolhatunk, szöveget és képet jeleníthetünk meg. Csoportosítsuk a ***Graphics*** osztály fontosabb rajzoló metódusait!

Vonal rajzolása

```
drawLine(int x1, int y1, int x2, int y2)
```

Az egyenes szakasz a grafikus környezet koordináta-rendszerében megadott (*x1*,*y1*) és (*x2*,*y2*) pontokat köti össze.

Téglalapok rajzolása

`drawRect(int x, int y, int w, int h)`	téglalap,
`drawRoundRect(int x, int y, int w, int h, int aw, int ah)`	lekerekített sarkú téglalap, az ív szélessége (*aw*), magassága (*ah*),
`clearRect(int x, int y, int w, int h)`	téglalap törlése a háttérszínnel, a festési mód használata nélkül,
`fillRect(int x, int y, int w, int h)`	rajzszínnel kitöltött téglalap,
`fillRoundRect(int x, int y, int w, int h, int aw, int ah)`	kitöltött lekerekített téglalap,
`void draw3DRect(int x, int y, int w, int h, boolean kiemelt)`	3D-s téglalap,
`void fill3DRect(int x, int y, int w, int h, boolean kiemelt)`	rajzszínnel kitöltött 3D téglalap.

A téglalapot a bal felső (*x*, *y*) sarokpont koordinátáival és a méreteivel (*w* - szélesség, *h* - magasság) jelöljük ki. A 3D-téglalapok *kiemelt* paraméterének **true** értéket adva, a téglalap kiemelkedik, ellenkező esetben pedig süllyesztett lesz.

Sokszög (polygon) rajzolása

`drawPolygon(int x[], int y[], int n)`	zárt sokszög,
`drawPolygon(Polygon p)`	zárt sokszög,
`fillPolygon(int x[], int y[], int n)`	zárt, kitöltött sokszög,
`fillPolygon(Polygon p)`	zárt, kitöltött sokszög,
`drawPolyline(int x[], int y[], int n)`	nyitott vonallánc.

A sokszög pontjait az (*x[i]*, *y[i]*) koordináta-párok jelölik ki, ahol $0 \leq i < n$. Az első és az utolsó pontok (amennyiben eltérőek) összekötésével zárt sokszög jön létre. A ***Polygon*** osztály az **int** típusú adattagokon (*npoints*, *xpoints[]*, *ypoints[]*) túlmenően, hasznos műveletekkel is segíti a sokszögek kezelését.

Ellipszis rajzolása

`drawOval(int x, int y, int w, int h)`	kör vagy ellipszis,
`fillOval(int x, int y, int w, int h)`	kitöltött kör vagy ellipszis.

Az ellipszist a befoglaló téglalapjával adjuk meg, melynek csúcspontja az (*x*, *y*), méretei pedig (*w*, *h*). Ha a téglalap szélessége (*w*) megegyezik a magasságával (*h*), kört kapunk.

Ellipszisív rajzolása

`void drawArc(int x, int y, int w, int h, int kezdőSzög, int ívszög)`	ellipszisív,
`void fillArc(int x, int y, int w, int h, int kezdőSzög, int ívszög)`	kitöltött ellipsziscikk.

A befoglaló téglalapjával (*x*, *y*, *w*, *h*) kijelölt ellipszis ívét a fokban megadott kezdőszög és a szögtartomány jelöli ki. A kezdőszöget a három órának megfelelő *0* pozícióhoz viszonyítva adjuk meg. A pozitív értékek az óramutató járásával ellentétes, a negatív értékek pedig az óramutató járásával egyező irányt jelölnek.

Szöveg megjelenítése

`void drawString(String szöveg, int x, int y)`	szöveg,
`void drawChars(char[] adat, int kezdőindex, int darab, int x, int y)`	szöveg karaktertömbből,
`void drawBytes (byte[] adat, int kezdőindex, int darab, int x, int y)`	szöveg bájttömbből.

A szöveg a megadott pozíciótól, az aktuális szín és betűtípus felhasználásával jelenik meg. (A pozícióba a balszélső karakter alapvonalának kezdőpontja kerül.)

Képek megjelenítése

A ***drawImage()*** metódusok segítségével ***Image*** típusú képet jeleníthetünk meg a grafikus felületen. A lehetőségekről bővebben a 11.5. alfejezetben szólunk.

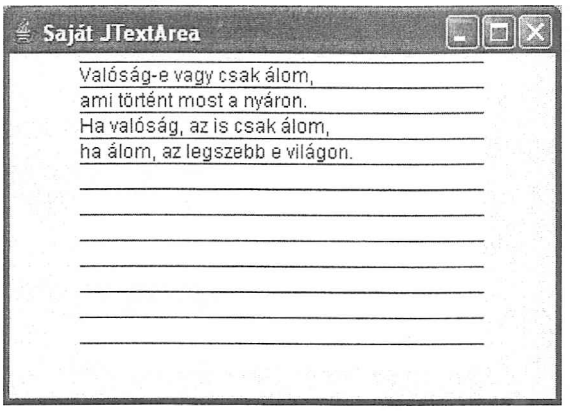
11.4 Grafika programozott megjelenítése

A következőkben néhány példaprogram segítségével szemléltetjük a bevezető, elméleti rész ismereteit.

Tervezzünk alkalmazást, amelyben a JTextArea komponens felületét bevonalazzuk a jobb olvashatóság érdekében! (*Alapok\Vonalazas*)

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

class SajatTextArea extends JTextArea
{
    public SajatTextArea(int sorok, int oszlopok) {
        super(sorok, oszlopok);
    }

    public void paint(Graphics g) {
        super.paint(g);              // a JTextArea megrajzolása
        Color c = g.getColor();      // elmentjük az aktuális színt
        Dimension size = getSize(); // a JTextArea méretei
        int ybetu=g.getFontMetrics().getHeight(); // a betűk magassága
        g.setColor(Color.darkGray);
        for (int y=0; y < size.height; y += ybetu)
            g.drawLine(0, y,size.width-1,y);
        g.setColor(c);               // a rajzszín visszaállítása
    }
}

public class JAblak extends JFrame
{
    SajatTextArea szovegmezo;

    public JAblak() {
        super("Saját JTextArea");
        setSize(350, 250);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        JPanel panel = new JPanel();
        szovegmezo = new SajatTextArea(12, 23);
        szovegmezo.setText("Valóság-e vagy csak álom,\n"+
                           "ami történt most a nyáron.\n"+
                           "Ha valóság, az is csak álom,\n"+
                           "ha álom, az legszebb e világon." );
        szovegmezo.setForeground(Color.blue);
        panel.add(szovegmezo);
        setContentPane(panel);
        setVisible(true);
    }

    public static void main(String[] arguments) {
        new JAblak();
    }
}
```

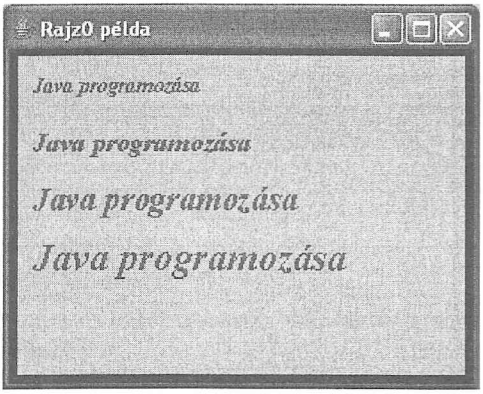
Az alkalmazás futásának eredménye:



Írjunk programot, amely bemutatja szöveg különböző méretben való megjelenítését! (*Alapok\Rajz0*)

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

public class Rajz0 extends JFrame
{
    public Rajz0()
    {
       super("Rajz0 példa");
       setDefaultCloseOperation(EXIT_ON_CLOSE);
       setSize(320,260);
       setBackground(Color.blue);
    }

    public void paint(Graphics g)
    {
       Rectangle r=getBounds();
       g.setColor(Color.cyan);
       g.fillRect(10,10,r.width-20,r.height-20);
       for (int i = 1; i<5; i++) {
          Font f = new Font("Times New Roman",
                            Font.BOLD|Font.ITALIC,10+i*4);
          g.setFont(f);
          g.setColor(Color.red);
          g.drawString("Java programozása", 20, 20+i*40);
       }
    }

    public static void main(String[] args) {
       Rajz0 mw = new Rajz0();
       mw.setVisible(true);
    }
}
```

Az alkalmazás futásának eredménye:

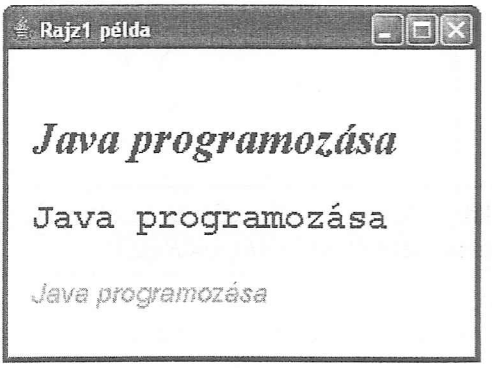


Készítsünk alkalmazást, amely bemutatja szöveg különböző típusú és méretű megjelenítését! (*Alapok\Rajz1*)

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

public class Rajz1 extends JFrame
{
    public Rajz1() {
       super("Rajz1 példa");
       setDefaultCloseOperation(EXIT_ON_CLOSE);
       setSize(320,240);
       setBackground(Color.white);
    }

    public void paint(Graphics g) {
       Rectangle r=getBounds();
       g.setColor(Color.white);
       g.fillRect(0,0,r.width-1,r.height-1);

       Font f = new Font("Times New Roman",Font.BOLD+Font.ITALIC, 30);
       g.setFont(f);
       g.setColor(Color.blue);
       g.drawString("Java programozása", 20, 100);

       Font f1 = new Font("Courier New",Font.BOLD, 24);
       g.setFont(f1);
       g.setColor(Color.darkGray);
       g.drawString("Java programozása", 20, 150);

       Font f2 = new Font("Arial",Font.ITALIC, 18);
       g.setFont(f2);
       g.setColor(Color.green);
       g.drawString("Java programozása", 20, 200);
    }
```

```
    public static void main(String[] args) {
        Rajz1 mw = new Rajz1();
        mw.setVisible(true);
    }
}
```

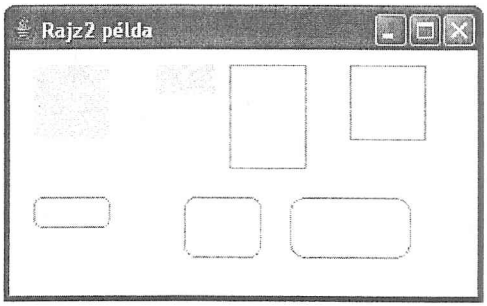
Az alkalmazás futásának eredménye:



Tervezzünk alkalmazást, amely bemutatja a téglalap, a lekerekített téglalap és a befestett téglalap megjelenítését! (*Alapok\Rajz2*)

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

public class Rajz2 extends JFrame
{
    public Rajz2() {
        super("Rajz2 példa");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(320,200);
        setBackground(Color.white);
    }

    public void paint(Graphics g) {
        Rectangle r=getBounds();
        g.setColor(Color.yellow);
        g.fillRect(20,40,50,50);
        g.fillRect(100,40,40,20);
        g.setColor(Color.green);
        g.drawRect(150,40,50,70);
        g.drawRect(230,40,50,50);
        g.drawRoundRect(20,130,50,20,10,10);
        g.drawRoundRect(120,130,50,40,15,15);
        g.drawRoundRect(190,130,80,40,20,20);
    }

    public static void main(String[] args) {
        Rajz2 mw = new Rajz2();
        mw.setVisible(true);
    }
}
```

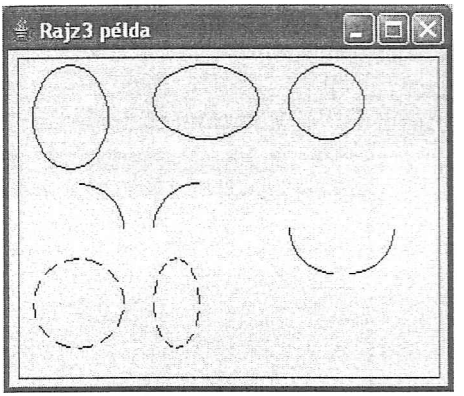
Az alkalmazás futásának eredménye:



Készítsünk alkalmazást, amely bemutatja a kör, a körív, az ellipszis és az ellipszisív folytonos valamint szaggatott vonallal történő megjelenítését! (*Alapok\Rajz3*)

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

public class Rajz3 extends JFrame
{
    public Rajz3() {
        super("Rajz3 példa");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300,260);
        setBackground(Color.yellow);
    }

    public void paint(Graphics g) {
        Rectangle r=getBounds();
        g.setColor(Color.red);
        g.drawRect(10,35,r.width-20,r.height-45);

        g.setColor(Color.blue);
        g.drawOval(20,40,50,70);
        g.drawOval(100,40,70,50);
        g.drawOval(190,40,50,50);

        g.drawArc(20,120,60,60,0,90);
        g.drawArc(100,120,60,60,90,90);
        g.drawArc(190,120,60,60,180,90);
        g.drawArc(200,120,60,60,270,90);

        for( int i = 0; i<360; i+=30)
            g.drawArc(20,170,60,60,i,20);

        for( int i = 0; i<360; i+=30)
            g.drawArc(100,170,30,60,i,20);
    }

    public static void main(String[] args) {
        Rajz3 mw = new Rajz3();
        mw.setVisible(true);
    }
}
```

Az alkalmazás futásának eredménye:

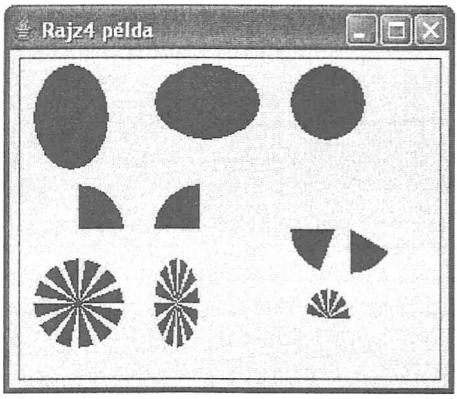


Tervezzünk alkalmazást, amely befestett kör, körív, ellipszis és ellipszisív folytonos valamint szaggatott határvonallal történő megjelenítését szemlélteti! (*Alapok\Rajz4*)

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

public class Rajz4 extends JFrame
{
    public Rajz4() {
        super("Rajz4 példa");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300,260);
        setBackground(Color.yellow);
    }

    public void paint(Graphics g) {
        Rectangle r=getBounds();
        g.setColor(Color.red);
        g.drawRect(10,35,r.width-20,r.height-45);

        g.setColor(Color.blue);
        g.fillOval(20,40,50,70);
        g.fillOval(100,40,70,50);
        g.fillOval(190,40,50,50);

        g.fillArc(20,120,60,60,0,90);
        g.fillArc(100,120,60,60,90,90);
        g.fillArc(190,120,60,60,180,70);
        g.fillArc(200,120,60,60,270,60);

        for( int i = 0; i<360; i+=30)
            g.fillArc(20,170,60,60,i,20);

        for( int i = 0; i<360; i+=30)
            g.fillArc(100,170,30,60,i,20);

        for( int i = 0; i<180; i+=30)
            g.fillArc(200,190,30,40,i,20);
    }
```

```
    public static void main(String[] args) {
        Rajz4 mw = new Rajz4();
        mw.setVisible(true);
    }
}
```

Az alkalmazás futásának eredménye:

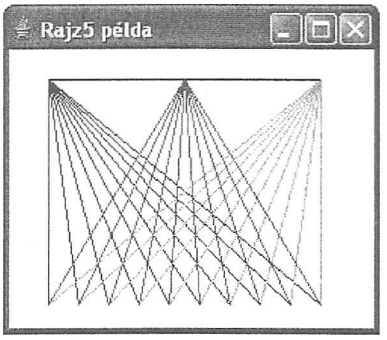


Írjunk különböző színű vonalakkal rajzoló programot! (*Alapok\Rajz5*)

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

public class Rajz5 extends JFrame
{
    public Rajz5() {
        super("Rajz5 példa");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(250,220);
        setBackground(Color.white);
    }

    public void paint(Graphics g) {
        Rectangle r=getBounds();
        g.drawLine(30,50, 210,50);

        g.setColor(Color.blue);
        for( int i = 0; i<10; i++)
            g.drawLine(30,50,30+i*20,200);

        g.setColor(Color.green);
        for( int i = 0; i<10; i++)
            g.drawLine(210,50,30+i*20,200);

        g.setColor(Color.red);
        for( int i = 0; i<10; i++)
            g.drawLine(120,50,30+i*20,200);
    }
```

```
    public static void main(String[] args) {
        Rajz5 mw = new Rajz5();
        mw.setVisible(true);
    }
}
```

Az alkalmazás futásának eredménye:

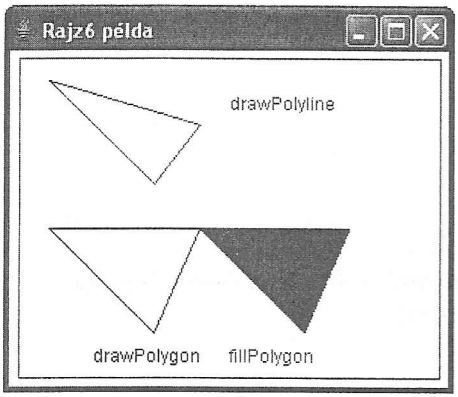


Készítsünk alkalmazást, amely bemutatja a ***drawPolyline()***, ***drawPolygon()*** és a ***fillPolygon()*** metódusok használatát! (*Alapok\Rajz6*)

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

public class Rajz6 extends JFrame
{
    int[] x1 = {30,100,130,30};
    int[] y1 = {50,120,80, 50};
    int[] x2 = {130,200,230,130};
    int[] y2 = {150,220,150,150};
    int[] x3 = {30,100,130,30};
    int[] y3 = {150,220,150,150};

    public Rajz6() {
       super("Rajz6 példa");
       setDefaultCloseOperation(EXIT_ON_CLOSE);
       setSize(300,260);
       setBackground(Color.white);
    }

    public void paint(Graphics g) {
       Rectangle r=getBounds();
       g.setColor(Color.red);
       g.drawRect(10,35,r.width-20,r.height-45);
       g.drawPolyline(x1,y1,4);
       g.drawString("drawPolyline", 150, 70);
       g.setColor(Color.darkGray);
       g.fillPolygon(x2,y2,4);
       g.drawString("fillPolygon", 150, 240);
```

```
        g.setColor(Color.blue);
        g.drawPolygon(x3,y3,4);
        g.drawString("drawPolygon", 60, 240);
    }

    public static void main(String[] args) {
        Rajz6 mw = new Rajz6();
        mw.setVisible(true);
    }
}
```

Az alkalmazás futásának eredménye:

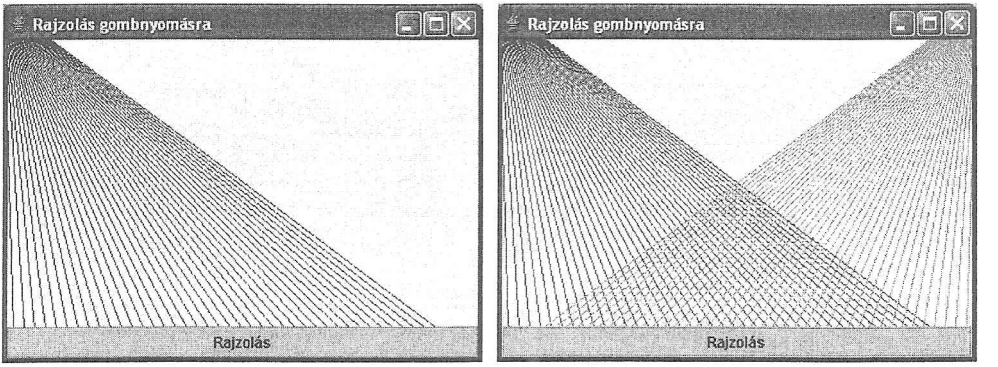


Tervezzünk alkalmazást, amelyben nyomógomb megnyomásakor rajzolunk! (*Alapok\GrafGombra*)

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

public class GrafGombra extends JFrame
{
    private JButton button1 = new JButton();

    public GrafGombra() {
        setTitle("Rajzolás gombnyomásra");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(400,300);

        button1.setText("Rajzolás");
        button1.setBackground(Color.lightGray);

        getContentPane().setLayout(new BorderLayout());
        getContentPane().add(button1, BorderLayout.SOUTH);

        Esemenykezelo eEsemenykezelo = new Esemenykezelo();
        button1.addActionListener(eEsemenykezelo);
        setVisible(true);
}
```

```
    // az alkalmazás indítása kivételfigyeléssel
    static public void main(String args[]) {
        try {
            (new GrafGombra()).setVisible(true);
        }
        catch (Throwable t) {
            System.err.println(t);
            t.printStackTrace();
            System.exit(1);
        }
    }

class Esemenykezelo implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        Object object = event.getSource();
        if (object == button1)
            button1_ActionPerformed(event);
    }
}

    public void paint(Graphics g) {
        g.setColor(Color.red);
        for (int x=0; x <= getSize().width; x+=10)
            g.drawLine(0, 0, x, getSize().height);
        button1.repaint();
    }

     void button1_ActionPerformed(ActionEvent event) {
         Graphics g=getGraphics();
         g.setColor(Color.green);
         for (int x= getSize().width; x>=0; x-=10)
            g.drawLine(getSize().width, 0, x, getSize().height);
     }
}
```

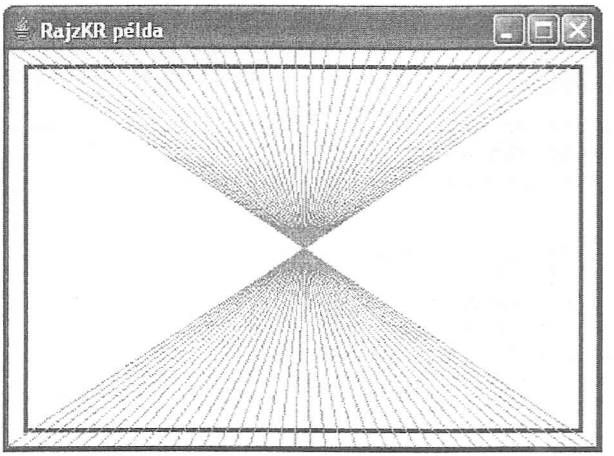
Az alkalmazás futásának eredménye:



Készítsünk alkalmazást, amely bemutatja az origó áthelyezését! (*Alapok\RajzKR*)

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

// Rajzolás az ablak munkaterületének koordináta-rendszerében
public class RajzKR extends JFrame
{
     public RajzKR() {
       super("RajzKR példa");
       setDefaultCloseOperation(EXIT_ON_CLOSE);
       setSize(400,300);
       setBackground(Color.white);
    }

    public void paint(Graphics g)
{
       // -- Az origó áthelyezése az ablak munkaterületének
       // -- bal felső sarkába a keret és az ablakfej
       // -- méretével korrigálva
       g.translate(getInsets().left, getInsets().top);

       // -- Az ablak befoglaló méreteinek lekérdezése
       Rectangle r=getBounds();

       // -- A munkaterület méretének meghatározása
       r.height -= getInsets().bottom + getInsets().top;
       r.width  -= getInsets().left + getInsets().right;

       // -- Keretezés vastag piros vonallal
       g.setColor(Color.red);
       g.drawRect(10,10 ,r.width-20,r.height-20);
       g.drawRect(11,11 ,r.width-22,r.height-22);
       g.drawRect(12,12 ,r.width-24,r.height-24);

       // -- Sugársor rajzolása
       g.setColor(Color.green);
       for (int x=0; x<r.width; x+=10) {
           g.drawLine(r.width/2, r.height/2, x, r.height-1);
           g.drawLine(r.width/2, r.height/2, x, 0);
       }
    }

     // az alkalmazás indítása kivételfigyeléssel
     static public void main(String args[]) {
         try {
             (new RajzKR ()).setVisible(true);
         }
         catch (Throwable t) {
             System.err.println(t);
             t.printStackTrace();
             System.exit(1);
         }
     }
}
```

Az alkalmazás futásának eredménye:

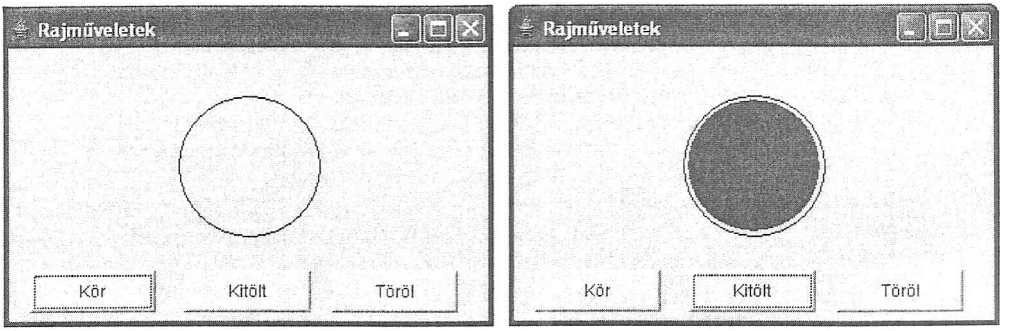


Tervezzünk alkalmazást, amely bemutatja egy kör rajzolását, befestését, valamint a rajz törlését nyomógombok használatával! (*Alapok\Rajzmuveletek*)

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

public class rajzmuveletek extends JFrame
                           implements ActionListener
{
    Button circleButton = new Button("Kör");
    Button fillButton   = new Button("Kitölt");
    Button resetButton  = new Button("Töröl");
    boolean kör=false, kitölt=false, töröl=false;
    final int sugar=50, dr=3;

    // konstruktorok
    public rajzmuveletek() {
        this("Java alkalmazás példa");
    }

    public rajzmuveletek(String fejléc) {
        super(fejléc);
        inicializalas();
        kozepreHelyez ();
    }

    public void inicializalas(){
      setSize(350,230);
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      setBackground(Color.yellow);
      setLayout(null);
```

```
        circleButton.setBounds(15,160,90,30);
        fillButton.setBounds(125,160,90,30);
        resetButton.setBounds(230,160,90,30);

        getContentPane().add(circleButton);
        getContentPane().add(fillButton);
        getContentPane().add(resetButton);

        circleButton.addActionListener(this);
        fillButton.addActionListener(this);
        resetButton.addActionListener(this);
    }

    // Az ablakot a képernyő közepére helyezi
    public void kozepreHelyez() {
        Toolkit toolkit  = Toolkit.getDefaultToolkit();
        Dimension kMeret = toolkit.getScreenSize();
        Dimension aMeret = getSize();
        setLocation((kMeret.width - aMeret.width) / 2,
                    (kMeret.height - aMeret.height) / 2);
    }

    public void paint(Graphics g) {
        if(töröl) {
            g.setColor(getBackground());
            g.fillRect(0,0,350,230);
            töröl = kör = kitölt = false;
        }

        if(kör) {
            g.setColor(Color.blue);
            g.drawOval(175-sugar,115-sugar, 2*sugar, 2*sugar);
        }

        if(kitölt) {
            g.setColor(Color.red);
            g.fillOval(175-(sugar-dr),115-(sugar-dr),
                       2*(sugar-dr), 2*(sugar-dr));
        }
    }

    public void actionPerformed(ActionEvent e) {
        if(e.getSource()==circleButton)
            kör = true;
        else if(e.getSource()==fillButton)
            kitölt=true;
        else
            töröl=true;
        repaint();
    }

    public static void main(String[] args) {
         rajzmuveletek mw = new rajzmuveletek("Rajzműveletek");
         mw.setVisible(true);
    }
}
```

Az alkalmazás futásának eredményei:



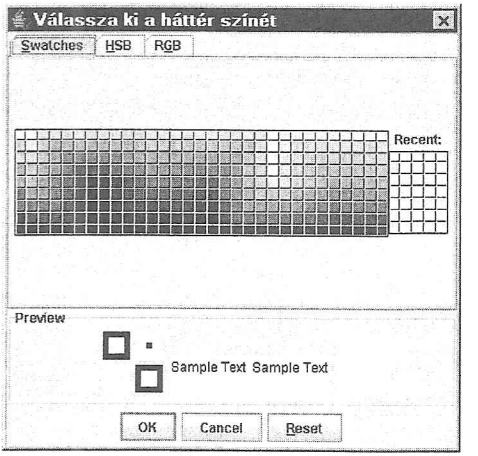
Készítsünk egyszerű rajzolótáblát, ahol a rajz- és a háttér színe párbeszédablakból választva beállítható, illetve az egérrel készült rajz törölhető! (*Alapok\Irkafirka*)

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

public class IrkaFirka extends JFrame implements ActionListener
{
    RajzTabla rajzTabla;

    public IrkaFirka(  ) {
        super("Szabadkézi rajzolás");
        setSize(300, 330);
        setLocation(230, 120);

        Container content = getContentPane(  );
        content.setLayout(new BorderLayout(  ));

        rajzTabla = new RajzTabla();
        content.add(rajzTabla, BorderLayout.CENTER);

        JPanel alsoPanel = new JPanel(  );
        JButton torlesGomb = new JButton("Törlés");
        torlesGomb.addActionListener(this);

        JButton rajzSzinGomb = new JButton("Rajzszín");
        rajzSzinGomb.addActionListener(this);

        JButton hatterSzinGomb = new JButton("Háttérszín");
        hatterSzinGomb.addActionListener(this);

        alsoPanel.add(torlesGomb);
        alsoPanel.add(rajzSzinGomb);
        alsoPanel.add(hatterSzinGomb);
        content.add(alsoPanel, BorderLayout.SOUTH);
        setVisible(true);
    }
```

```
    public void actionPerformed(ActionEvent e) {
       if (e.getActionCommand()=="Törlés")
           rajzTabla.torles();
       else if (e.getActionCommand()=="Rajzszín")
           rajzTabla.rajzSzin = JColorChooser.showDialog( this,
                                "Válassza ki a rajzszínt",
                                rajzTabla.rajzSzin);
       else if (e.getActionCommand()=="Háttérszín") {
           rajzTabla.hatterSzin = JColorChooser.showDialog( this,
                                "Válassza ki a háttér színét",
                                rajzTabla.hatterSzin);
           rajzTabla.torles();
       }
    }

    public static void main(String[] args) {
        new IrkaFirka();
    }
}

// A RajzTabla osztály deklarációja
class RajzTabla extends JPanel
{
    public Color rajzSzin;
    public Color hatterSzin;
    Image image;
    Graphics graphics;
    int hovaX, hovaY, honnanX, honnanY;

    public RajzTabla() {
        rajzSzin = Color.black;
        hatterSzin = Color.white;
        setDoubleBuffered(false);

        //az egérgombok lenyomásakor tároljuk a kezdőpont koordinátáit
        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                honnanX = e.getX();
                honnanY = e.getY();
            }
        });

        // az egér mozgatásakor rajzolunk
        addMouseMotionListener(new MouseMotionAdapter() {
            public void mouseDragged(MouseEvent e) {
                hovaX = e.getX( );
                hovaY = e.getY( );
                if (graphics != null) {
                    graphics.setColor(rajzSzin);
                    graphics.drawLine(honnanX, honnanY, hovaX, hovaY);
                }
                repaint();
                honnanX = hovaX;
                honnanY = hovaY;
            }
        });
    }
```

```
    public void paintComponent(Graphics g) {
        // első alkalommal létrehozzuk a háttérképet,
        // amelyre rajzolunk
        if (image == null) {
            image = createImage(getSize().width, getSize().height);
            graphics = image.getGraphics();
            torles();
        }
        // megjelenítjük a háttérképet
        g.drawImage(image, 0, 0, null);
    }
    public void torles() {
        graphics.setColor(hatterSzin);
        graphics.fillRect(0, 0, getSize().width, getSize().height);
        repaint();
    }
}
```

Az alkalmazás futásának eredményei:



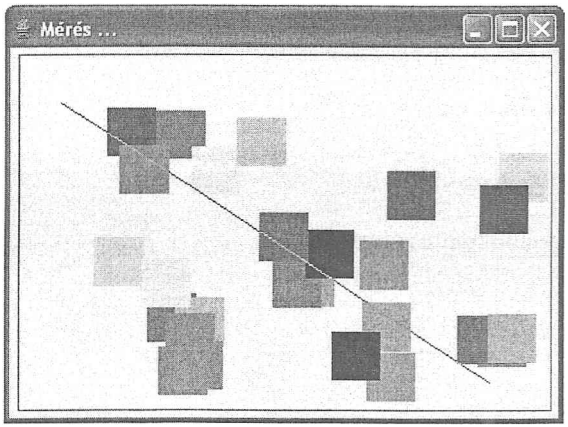
Tervezzünk alkalmazást, melynek ablakában az egér bal gombjának lenyomásával, mozgatásával és felengedésével távolságot mérhetünk! A vonalzó megjelenítéséhez használjuk a *XOR*-módot! (*Alapok\ TavolsagMeres*)

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

public class TavolsagMeres extends JFrame
{
       boolean vonalzoBe = false;
       Point kezdo = new Point(), veg = new Point();
```

```
public TavolsagMeres() {
   super("Távolságmérés");
   setSize(400, 300);
   setDefaultCloseOperation(EXIT_ON_CLOSE);
   // szálkereszt egérmutató beállítása
   setCursor(Cursor.getPredefinedCursor(
                              Cursor.CROSSHAIR_CURSOR));
   setVisible(true);
   addMouseListener(new SajatMouseAdapter());
   addMouseMotionListener(new SajatMouseMotionAdapter());
}

// Az ablak tartalmának véletlenszerű előállítása
public void paint(Graphics g) {
   super.paint(g);
   final int L=35; // a kis négyzetek oldalhossza
   Rectangle r = getBounds();

   // sárga háttér
   g.setColor(Color.yellow);
   g.fillRect(r.x, r.y, r.width, r.height);

   // szürke keret
   g.setColor(Color.darkGray);
   g.drawRect(r.x+10, r.y+35, r.width-20, r.height-45);

   // kifestett négyzettel jelöljük a pontokat a kereten belül
   // véletlen pozíciókban, véletlen színnel
   int x, y;
   for (int i=0; i<23+(int)(Math.random()*12); i++) {
      x = r.x+10+L+(int)(Math.random()*(r.width-20-2*L));
      y = r.y+35+L+(int)(Math.random()*(r.height-45-2*L));
      g.setColor(new Color((int)(Math.random()*256),
                           (int)(Math.random()*256),
                           (int)(Math.random()*256)));
      g.fillRect(x, y, L,L);
   }
}

// A kezdő- és a végpontokat XOR-módban köti össze
private void vonalRajz() {
   // ha a két pont egybeesik
   if (veg.equals(kezdo))
         return;

   Graphics g = getGraphics();
   Color c = g.getColor(); // elmentjük

   // rajzolás XOR-módban
   g.setColor(Color.blue);
   g.setXORMode(Color.yellow);
   g.drawLine(kezdo.x, kezdo.y, veg.x, veg.y);

   // visszaállítjuk az eredeti állapotokat
   g.setPaintMode();
   g.setColor(c);
}
```

```
    public static void main(String[] arguments) {
      new TavolsagMeres();
    }

    // Az egér bal oldali gombjának lenyomását és
    // felengedését kezeljük
    class SajatMouseAdapter extends MouseAdapter
    {
      // egérgomb lenyomása
      public void mousePressed(MouseEvent e) {
        // a bal oldali gomb le
        if (e.getButton() == MouseEvent.BUTTON1) {
            veg.x = kezdo.x = e.getX();
            veg.y = kezdo.y = e.getY();
            setTitle("Mérés ...");
            vonalzoBe = true;
        }
      }

      // egérgomb felengedése
      public void mouseReleased(MouseEvent e) {
        // a bal oldali gomb fel
        if (e.getButton() == MouseEvent.BUTTON1) {
            vonalzoBe = false;
            vonalRajz();
            veg.x = e.getX();
            veg.y = e.getY();

            // távolságszámítás
            double tavolsag = Math.hypot(kezdo.x-veg.x,
                                         kezdo.y-veg.y);
            setTitle("Távolság: "+tavolsag);
        }
      }
    }

    // Az egér lenyomott bal gombbal való mozgatását kezeljük -
    // dragged
    class SajatMouseMotionAdapter extends MouseMotionAdapter
    {
      public void mouseDragged(MouseEvent e) {
        if (vonalzoBe) {  // mérés folyamatban
            vonalRajz();
            veg.x = e.getX();
            veg.y = e.getY();
            vonalRajz();
        }
      }
    }
}
```

Az alkalmazás futásának eredménye:



11.5 Multimédiás elemek

A multimédia egy időben nagyon felkapott téma volt az informatikában, mára azonban hétköznapi dologgá vált, amellyel lépten-nyomon találkozhatunk. A nevében benne van, hogy sokféle közeget kezel, mi azonban csupán a képek és a hangállományok kezelésébe vezetjük be az Olvasót.

11.5.1 Képek betöltése és megjelenítése

Java alkalmazásokban a képeket több osztály felhasználásával is betölthetjük, kezelhetjük és módosíthatjuk. Képeket futás közben is előállíthatunk, megadva a képpontok (*pixelek*) színét.

A képek kezelésének egyik alaposztálya a ***java.awt.Image***, melynek példányait az esetek többségében nem mi hozzuk létre, csak lekérjük a hivatkozásukat a ***java.awt.Toolkit*** osztály ***getImage()*** metódusával:

```
Image kép = getToolkit().getImage("elérési_út");
Image kép = Toolkit.getDefaultToolkit().getImage("elérési_út");
```

A Java API csak a ***JPEG***, a ***PNG*** és a ***GIF*** képformátumot támogatja. Más formátumú képeket a betöltés előtt ilyen formátumúvá kell alakítanunk. Az alkalmazás futtatásához szükséges képeket általában az ablak konstruktorában olvassuk be.

Az ***Image*** típusú objektumban tárolt képet a ***Graphics*** osztály ***drawImage()*** metódusaival jeleníthetjük meg, adott (x, y) pozícióban a saját méreteit használva, vagy a megadott méretekkel (w, h). A metódusok **true** visszatérési értékkel jelzik, hogy a teljes képet sikerült rámásolni a grafikus felületre. Némely metódus hívásában szereplő ***Color*** típusú háttérszínt a kép áttetsző részeinek kifestésére használja a rendszer.

```
drawImage(kép, x, y, képfigyelő)
drawImage(kép, x, y, háttérszín, képfigyelő)
drawImage(kép, x, y, w, h, képfigyelő)
drawImage(kép, x, y, w, h, háttérszín, képfigyelő)
```

A ***drawImage()*** metódus utolsó paramétere az ***ImageObserver*** interfész típusú képfigyelő, amelyből megtudható, hogy a teljes kép betöltése és átalakítása belső formátumúvá befejeződött-e. (Gyakorlatilag minden komponens megvalósítja ezt az interfészt, így utolsó argumentumként a **this** hivatkozást szoktuk megadni.) Amennyiben az **Image** típusú kép méretei (***getWidth()*** és ***getHeight()***) már rendelkezésre állnak, a metódus figyelmen kívül hagyja ezt a paramétert.

A *Swing* objektumtár egy új képosztályt definiált ***javax.swing.ImageIcon*** (ikonkép) néven. Az ***ImageIcon*** osztály példányát létrehozó konstruktorok paraméterként ***Image*** típusú objektumot, képállománynevet vagy URL-hivatkozást várnak. Metódusokkal lekérdezhetjük az ikonkép méreteit, illetve a benne tárolt ***Image*** típusú képet.

```
int getIconHeight()
int getIconWidth()
Image getImage()
void setImage(Image kép)
ImageObserver getImageObserver()
void setImageObserver(ImageObserver képfigyelő)
void paintIcon(Component c, Graphics g, int x, int y)
```

Külön metódus gondoskodik az ikonkép grafikus felületen történő megjelenítéséről. (Amennyiben a képhez nem tartozik képfigyelő, a *c* komponenst használja erre a célra.)

A képek betöltése elég időigényes folyamat, főleg, ha nagyméretű képekből, sokat kell beolvasnunk az alkalmazás futtatásához. A ***java.awt.MediaTracker*** osztály segítségével ellenőrizhetjük, hogy a megadott képek betöltődtek-e már, vagy sem. Első lépésként a „médiakövető" objektumához azonosítóval ellátva hozzá kell rendelnünk a képeket. (Egy azonosító alatt akár több kép is szerepelhet.)

```
addImage(Image kép, int azonosító)
```

Ezt követően talán a legbiztosabb megoldás, ha felfüggesztjük (***waitForAll()***) a program futását, amíg az utolsó kép be nem töltődik. Az alkalmazás sikertelen esetben is tovább fut, ezért a ***paint()*** metódusban ellenőriznünk kell, hogy történt-e hiba.

```
boolean isErrorAny()
boolean isErrorID(int azonosító)
```

Az alábbi programrészt használva csak akkor jelenik meg kép, ha mindkettő betöltése sikeres volt:

```
MediaTracker koveto;
```

```
Image kep0, kep1;

private void kepekBetoltese() {
    koveto = new MediaTracker(this);
    kep0 = getToolkit().getImage("Kep1.jpg");
    koveto.addImage(kep0, 0);
    kep1 = Toolkit.getDefaultToolkit().getImage("Kep2.jpg");
    koveto.addImage(kep1, 1);
    try {
        koveto.waitForAll();
    } catch (InterruptedException e) {}
}

public void paint(Graphics g) {
    if (!koveto.isErrorAny()) {
        g.drawImage(kep0, 30, 30, this);
        g.drawImage(kep1, 150, 150, this);
    }
}
```

A sikeresen beolvasott képeket külön-külön is láthatóvá tehetjük:

```
public void paint(Graphics g) {
    if (!koveto.isErrorID(0))
        g.drawImage(kep0, 30, 30, this);
    if (!koveto.isErrorID(1))
        g.drawImage(kep1, 150, 150, this);
}
```

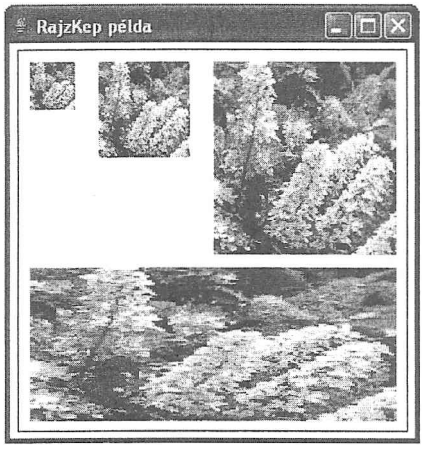
A Java *API* további interfészeket és osztályokat is tartalmaz a képkezelő alkalmazások hatékony kialakításához. A ***java.awt.image*** csomag elemeivel a képeket szűrhetjük, vághatjuk, képpontokká alakíthatjuk, illetve pixeltömbből felépíthetjük stb. Néhány megoldásra az alábbi példákból kapunk ízelítőt, azonban a lehetőséget teljes ismertetése külön kötetet igényelne.

Írjunk alkalmazást, amely bemutatja a kép betöltését, nagyítását, kicsinyítését, illetve torzítását! (*Multimedia\RajzKep*)

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

public class RajzKep extends JFrame
{
    ImageIcon ikonKep;
    Image kep;
```

```
    public RajzKep() {
        super("RajzKep példa");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(320,340);
        setBackground(Color.white);
        ikonKep = new ImageIcon("kepek/tavasz.jpg");
        kep=ikonKep.getImage();
    }
    public void paint(Graphics g) {
        Rectangle r = getBounds();
        g.setColor(Color.blue);
        g.drawRect(10,35,r.width-20,r.height-45);
        int wkep = kep.getWidth(this);
        int hkep = kep.getHeight(this);
        int xpoz = 20, ypoz = 45;
        // 25 %
        g.drawImage(kep, xpoz, ypoz, wkep / 4, hkep / 4, this);
        // 50 %
        xpoz += (wkep / 4) + 18;
        g.drawImage(kep, xpoz , ypoz, wkep / 2, hkep / 2, this);
        // 100%
        xpoz += (wkep / 2) + 18;
        g.drawImage(kep, xpoz, ypoz, this);
        // 200% x, 80% y
        g.drawImage(kep, 20, hkep + 55, wkep * 2,
                    (int)(hkep * 0.8), this);
    }
    public static void main(String[] args)  {
         (new RajzKep()).setVisible(true);
    }
}
```

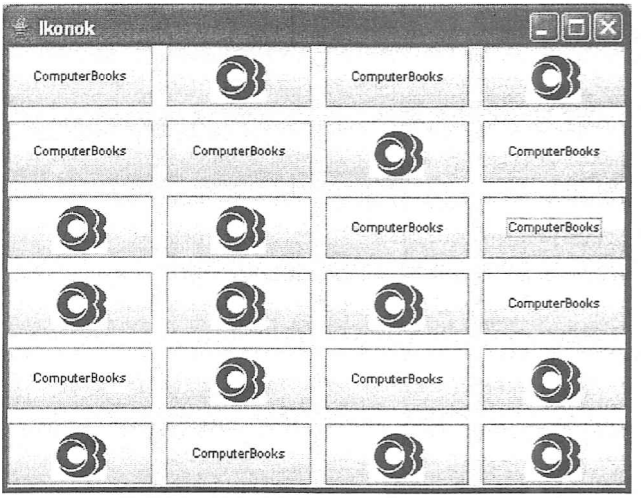
Az alkalmazás futásának eredménye:



Készítsünk alkalmazást, amelyben a 6x4-es rácsban elhelyezett nyomógombok ikonja- és felirata gombnyomáskor felcserélődik! (*Multimedia\Ikonok*)

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

public class Ikonok extends JFrame implements ActionListener
{
       JButton[] gombok = new JButton[24];
       ImageIcon ikon = new ImageIcon("cb.gif");
       Font betuk = new Font("InputDialog",Font.PLAIN,9);

       public Ikonok() {
              super("Ikonok");
              setSize(450, 350);
              setDefaultCloseOperation(EXIT_ON_CLOSE);
              JPanel mezo = new JPanel();
              mezo.setLayout(new GridLayout(6,4,10,10));

              // a 24 gomb létrehozása és elhelyezése a rácsban
              for (int i=0; i<24; i++) {
                     gombok[i]=new JButton(ikon);
                     gombok[i].setFont(betuk);
                     gombok[i].setForeground(Color.blue);
                     gombok[i].addActionListener(this);
                     mezo.add(gombok[i]);
              }
              setContentPane(mezo);
              setVisible(true);
       }

       // a gombnyomás esemény kezelése
       public void actionPerformed(ActionEvent event) {
              JButton btn = (JButton)event.getSource();
              // ha nincs ikon a gombon
              if (btn.getIcon() == null) {
                  btn.setIcon(ikon);  // legyen kép
                  btn.setText(null);  // felirat törlése
              }
              // ha van kép a gombon
              else {
                 btn.setIcon(null);  // levesszük a képet
                 btn.setText("ComputerBooks"); // felirat
              }
       }

       public static void main(String[] arguments) {
              Ikonok ikb = new Ikonok();
       }
}
```

Az alkalmazás futásának eredménye:



Tervezzünk alkalmazást, amely bemutatja kép forgatását, valamint fekete színre való váltását! (A képeket pixelenként dolgozzuk fel.) (*Multimedia\Kepmuveletek*)

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

public class KepMuveletek extends JFrame
{
    Image kep=null, kep90=null, kepbw=null;

    public KepMuveletek() {
        super("Képműveletek példa");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(280,280);
        setBackground(Color.white);

        MediaTracker mt = new MediaTracker (this);
        kep = Toolkit.getDefaultToolkit(  ).getImage("CBooks.jpg");
        mt.addImage (kep, 0);
        try {
           mt.waitForAll(); // várakozás a kép betöltéséig
           int wmeret  = kep.getWidth(this);
           int hmeret  = kep. getHeight(this);
           int keppontok[]  = new int [wmeret * hmeret];

           // Az eredeti kép képpontjainak lekérdezése
           PixelGrabber pg  = new PixelGrabber (kep, 0, 0, wmeret,
                                    hmeret, keppontok, 0, wmeret);
```

```
            // Ha a képpontok előállítása sikeres volt
            if (pg.grabPixels() &&
                    ((pg.status()& ImageObserver.ALLBITS) !=0)) {
              // 90-fokkal elforgatott kép létrehozása
              kep90 = createImage(new MemoryImageSource(hmeret, wmeret,
                       forgatas90(keppontok,wmeret,hmeret),0, hmeret));
              // fekete-fehér kép elkészítése
              kepbw = createImage(new MemoryImageSource(wmeret, hmeret,
                      feketefeher(keppontok,wmeret, hmeret),0,wmeret));
            }
        }
        catch (InterruptedException e) {
           e.printStackTrace();
        }
    }

    public void paint (Graphics g) {
        g.drawImage (kep, 20, 50, this);          // eredeti kép
        if (kep90 != null)
            g.drawImage (kep90, 170, 65, this); // 90-kal elforgatott
        if (kepbw != null)
            g.drawImage (kepbw, 20, 150, this); // fekete-fehér
    }

    // A kép elforgatása 90 fokkal
    private int[] forgatas90 (int keppontok[], int w, int h) {
        int forgKeppontok[] = null;
        if ((w * h) == keppontok.length) {
            forgKeppontok = new int [w * h];
            int forgIndex=0;
            for (int x = w-1; x >= 0; x--)
                for (int y = 0; y < h; y++)
                    forgKeppontok[forgIndex++] = keppontok[y*w + x];
        }
        return forgKeppontok;
    }

    // A kép fekete-fehér változatának előállítása
    private int[] feketefeher (int keppontok[], int w, int h) {
        int ujKeppontok[] = null;
        if ((w * h) == keppontok.length) {
            ujKeppontok = new int [w * h];
            int ujIndex=0;
            for (int y = 0; y < h; y++)
                for (int x = 0; x < w; x++) {
                    Color sz = new Color(keppontok[y*w + x]);
                    int bw = (sz.getRed() + sz.getBlue()
                              + sz.getGreen())/3;
                    sz = new Color(bw, bw, bw);
                    ujKeppontok[ujIndex++] = sz.getRGB();
                }
        }
        return ujKeppontok;
    }
```

```
    public static void main(String[] args)  {
        KepMuveletek mw= new KepMuveletek();
        mw.setVisible(true);
    }
}
```

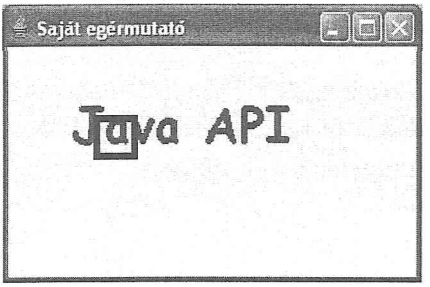
Az alkalmazás futásának eredménye:

Készítsünk alkalmazást, amelyben a jobb egérgombbal az ablak egérmutatóját váltogatjuk, a saját tervezésű és a beépített kéz kurzor felváltva történő megjelenítésével. (*Multimedia\Egermutato*)

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.MemoryImageSource;

public class Egermutato extends JFrame {
       Cursor mutato;
       boolean sajat=true;
       Font betutipus = new Font("Comic Sans MS", Font.BOLD, 35);

       public Egermutato() {
           super("Saját egérmutató");
           setSize(300, 200);
           setDefaultCloseOperation(EXIT_ON_CLOSE);
           sajatCursor();
           setVisible(true);
           addMouseListener(new SajatMouseAdapter());
       }

       protected void sajatCursor() {
           Toolkit tk = Toolkit.getDefaultToolkit();
           int maxSzinek = tk.getMaximumCursorColors();
           // 50x50 képpontméretű kurzort kívánunk létrehozni
           Dimension d = tk.getBestCursorSize(50, 50);
           int w = d.width, h = d.height;
```

```
        // az egérmutató érzékeny pontja
        Point p = new Point(0,0);

        // az egérmutató képét itt állítjuk elő
        // 5 pixel széles kék négyzet, melynek
        // belső része átlátszó
        int [] keppontok = new int[w*h];
        for (int x=0, k=0; x<w; x++)
           for (int y=0; y<h; y++)
             if ((x<5 || x>w-6) || (y<5 || y>h-6))
                 keppontok[k++] = 0xff0000ff;
             else
                 keppontok[k++] = 0;

        // a kép létrehozása a pixeltömbből
        Image kurzorKep = createImage(new
                          MemoryImageSource(w, h, keppontok, 0, w));

        // az egérmutató előállítása a képből és megjelenítése
        mutato = tk.createCustomCursor(kurzorKep, p, null);
        this.setCursor(mutato);
    }

    // a teszteléshez egy szöveget jelenítünk meg az ablakban
    public void paint(Graphics g) {
        super.paint(g);
        g.setColor(Color.darkGray);
        g.setFont(betutipus);
        g.drawString("Java API", 50,100);
    }

    // Saját egérkezelő osztály, amely figyeli az egér
    // jobb gombjával való kattintást
    class SajatMouseAdapter extends MouseAdapter
    {
        public void mouseClicked(MouseEvent e) {
            // a jobb oldali egérgomb
            if (e.getButton()==MouseEvent.BUTTON3)
               if (sajat) {
                  sajat = false;
                  // a kéz egérmutató beállítása
                  setCursor(Cursor.getPredefinedCursor(
                                          Cursor.HAND_CURSOR));
               }
               else {
                  sajat = true;
                  setCursor(mutato);
               }
        }
    }

    public static void main(String[] arguments) {
        new Egermutato();
    }
}
```

Az alkalmazás futásának eredménye:



11.5.2 Hang megszólaltatása

A Java 2 – az első változathoz képest – komoly hangkezelési lehetőségekkel rendelkezik, melynek összetevőit a ***Java Sound API*** tartalmazza. Az *API* a hang lejátszásán túlmenően zenék előállítását és állományba való mentését is lehetővé teszi.

A JVM hanglejátszója a mintavételezett (*sampled*) *digital audio* és MIDI (*Musical Intstrument Digital Interface*) hangadatok megszólaltatására képes. A hangot az első esetben *AU*, *WAVE* vagy *AIFF* típusú állományok, míg a második esetben *MIDI*-fájlok tárolják. E kettősség a Java-csomagok kialakításában is nyomon követhető:

```
javax.sound.sampled,
javax.sound.sampled.spi,
javax.sound.midi,
javax.sound.midi.spi.
```

Az AudioSystem osztály alkalmazásával

A ***digitális audio*** állományok lejátszásához a ***javax.sound.sampled*** csomag elemeit használjuk. A hang megszólaltatásához szükséges alaplépések a következők:

1. Létrehozunk egy ***Clip*** típusú változót, valamint a hangállományt megnyitjuk egy ***File*** típusú objektumban:

```
Clip lejátszó = null;
File f = new File(hangállomány)
```

2. Több lépésben ellenőrizzük, hogy a hangfájlban tárolt adatok formátumát támogatja-e az audiorendszer:

```
AudioFileFormat aff = AudioSystem.getAudioFileFormat(f);
AudioFormat af = aff.getFormat();
DataLine.Info info = new DataLine.Info(Clip.class, af);
if (!AudioSystem.isLineSupported(info)) // nem játszható le!
```

3. Megszerezzük a megszólaltatáshoz szükséges információkat, valamint a hangállomány ***File*** típusú objektumát ***AudioInputStream*** típusú adatfolyammá alakítjuk:

```
lejátszó = (Clip)AudioSystem.getLine(info);
AudioInputStream hangadatok =
                    AudioSystem.getAudioInputStream(f)
```

4. Megszólaltatjuk a zenét, esetleg kérve annak ismétlését:

```
lejátszó.open(hangadatok);
lejátszó.loop(12);
lejátszó.start();
```

5. Megállítjuk a lejátszást:

```
lejátszó.stop();
lejátszó.close();
```

A MidiSystem osztály segítségével

A ***MIDI***-állományok lejátszásához a ***javax.sound.midi*** csomag elemeit használjuk. A hang megszólaltatásához szükséges alaplépések sokban hasonlítanak a fentiekben bemutatott lépésekre:

1. Létrehozunk egy ***Sequencer*** típusú változót, valamint a MIDI-állományt megnyitjuk egy ***File*** típusú objektumban:

```
Sequencer lejátszó = null;
File f = new File(hangállomány)
```

2. Megpróbáljuk elérni a Midi-rendszer lejátszóját (*Sequencer*), csak sikeres esetben lehet folytatni a lépéseket:

```
lejátszó = MidiSystem.getSequencer();
if (lejátszó == null) // nincs mivel lejátszani!
```

3. A ***File*** típusú objektumból a hangadatokat (MIDI-szekvenciákat) a ***Sequence*** típusú objektumba másoljuk, amit aztán a lejátszóhoz rendelünk:

```
Sequence hangadatok = MidiSystem.getSequence(f);
lejátszó.setSequence(hangadatok);
```

4. Elindítjuk a lejátszást, esetleg kérve a zene ismétlését:

```
lejátszó.open();
lejátszó.setLoopCount(12);
lejátszó.start();
```

5. Leállítjuk a zenélést:

```
lejátszó.stop();
lejátszó.close();
```

Az AudioClip interfész felhasználásával

A fentieknél sokkal egyszerűbb hanglejátszást biztosít az ***AudioClip*** típusú objektum használata. Ebben az esetben a lejátszás első lépéseként az ***Applet*** osztály statikus

```
AudioClip newAudioClip(URL webcím)
```

metódusával be kell töltenünk a hangfájlt, létrehozva az ***AudioClip*** interfész példányát. Ezt követően az ***AudioClip play()*** metódusával egyszer, a ***loop()*** metódusával pedig ismétlődve (a ***stop()*** hívásáig) zenélhetünk. Egyetlen gondot a webcím előállítása jelenti, amihez először le kell kérnünk (***getClass()***) az ablakosztály információit a ***Class*** típusú objektumba, melynek

```
URL getResource(String név)
```

metódus visszaadja az argumentumban szereplő hangfájl elérési adatait az ***URL*** típusú objektumban.

Fejlesszünk alkalmazást, amely gombokkal vezérelhető módon szólaltatja meg egy *WAVE* típusú állományban tárolt zenét! (*Multimedia\DigitalAudio*)

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.sound.sampled.*;
import java.io.*;

public class JAblak extends JFrame implements ActionListener
{
   JButton beButton = new JButton("Start");
   JButton kiButton = new JButton("Stop");
   Clip lejatszo = null;

   public JAblak() {
       super("Digital Audio-lejászó");
       getContentPane().setLayout(null);
       beButton.setBounds(10,80,90,30);
       kiButton.setBounds(115,80,90,30);
       kiButton.setEnabled(false);
       setSize(230,150);
       getContentPane().add(beButton);
       getContentPane().add(kiButton);
       beButton.addActionListener(this);
       kiButton.addActionListener(this);
       setVisible(true);
    }

   // Digital Audio állományok lejátszása
   public void playDA(String allomany) {
      try {
         File f = new File(allomany);
         // Ellenőrizzük, hogy a fájlban tárolt adatok formátumát
         // támogatja-e az AudioSystem
         AudioFileFormat aff = AudioSystem.getAudioFileFormat(f);
         AudioFormat af = aff.getFormat();
         DataLine.Info info = new DataLine.Info(Clip.class, af);
```

```
            if (!AudioSystem.isLineSupported(info)) {
                JOptionPane.showMessageDialog(null,
                                        "A Line nem támogatott!");
                kiButton.setEnabled(false);
                beButton.setEnabled(true);
                return;
            }
            lejatszo = (Clip)AudioSystem.getLine(info);
            // a hangadatok folyama
            AudioInputStream hangadatok =
                                AudioSystem.getAudioInputStream(f);
            lejatszo.open(hangadatok);
            lejatszo.start();
        } catch (UnsupportedAudioFileException e){
            System.out.println("UnsupportedAudioFileException " + e);
        } catch (LineUnavailableException e) {
            System.out.println("LineUnavailableException " + e);
        } catch (IOException e) {
            System.out.println("IOException " + e);
        }
    }

    public void actionPerformed(ActionEvent e) {
        if(e.getSource()==beButton) {
            kiButton.setEnabled(true);
            beButton.setEnabled(false);
            playDA("Zene.wav");
        } else
        if(e.getSource()==kiButton) {
            kiButton.setEnabled(false);
            beButton.setEnabled(true);
            lejatszo.stop();
            lejatszo.close();
        }
    }

    public static void main(String[] arguments) {
        new JAblak();
    }
}
```

Tervezzünk alkalmazást, amely gombokkal vezérelve szólaltatja meg egy *midi*-állomány tartalmát! (*Multimedia\Midi*)

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.sound.midi.*;
import java.io.*;
```

```
public class JAblak extends JFrame implements ActionListener
{
   JButton beButton = new JButton("Start");
   JButton kiButton   = new JButton("Stop");
   Sequencer lejatszo = null;

   public JAblak() {
      super("MIDI-lejászó");
      getContentPane().setLayout(null);
      beButton.setBounds(10,80,90,30);
      kiButton.setBounds(115,80,90,30);
      kiButton.setEnabled(false);
      setSize(230,150);
      getContentPane().add(beButton);
      getContentPane().add(kiButton);
      beButton.addActionListener(this);
      kiButton.addActionListener(this);
   }

   public void playMIDI(String allomany) {
      try {
         File f = new File(allomany);
         lejatszo = MidiSystem.getSequencer();
         if (lejatszo == null) {
                JOptionPane.showMessageDialog(null,
                                   "A MIDI Sequencer nem érhető el");
                kiButton.setEnabled(false);
                beButton.setEnabled(true);
                return;
         }
         Sequence hangadatok = MidiSystem.getSequence(f);
         lejatszo.setSequence(hangadatok);
         lejatszo.open();
         lejatszo.start();
      } catch (InvalidMidiDataException e){
         System.out.println("InvalidMidiDataException " + e);
      } catch (MidiUnavailableException e) {
         System.out.println("MidiUnavailableException " + e);
      } catch (IOException e) {
         System.out.println("IOException " + e);
      }
   }

   public void actionPerformed(ActionEvent e) {
      if(e.getSource()==beButton) {
         kiButton.setEnabled(true);
         beButton.setEnabled(false);
         playMIDI("Zene.mid");
      } else
      if(e.getSource()==kiButton) {
         kiButton.setEnabled(false);
         beButton.setEnabled(true);
         lejatszo.stop();
         lejatszo.close();
      }
}
```

```
    public static void main(String[] arguments) {
        (new JAblak()).setVisible(true);
    }
}
```

Készítsünk alkalmazást, amely bemutatja *AU* típusú állományok gombokkal vezérelt lejátszását! Használjuk az AudioClip interfészt! (*Multimedia\AudioClip*)

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.applet.*;
import java.net.URL;

public class Hangok extends JFrame implements ActionListener
{
    JButton beButton = new JButton("Madárdal be");
    JButton kiButton   = new JButton("Madárdal ki");
    JButton sipButton  = new JButton("Sípoló hang");
    AudioClip hatterzene, mpjel;

    public Hangok() throws Exception {
        super("Hangkezelő Java alkalmazás példa");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(null);
        hatterzene = Applet.newAudioClip
                        (getClass().getResource("madar.au"));
        mpjel = Applet.newAudioClip(getClass().getResource("sip.au"));

        getContentPane().setLayout(null);
        beButton.setBounds(10,80,90,30);
        kiButton.setBounds(115,80,90,30);
        sipButton.setBounds(220,80,90,30);
        setSize(330,150);
        getContentPane().add(beButton);
        getContentPane().add(kiButton);
        getContentPane().add(sipButton);
        beButton.addActionListener(this);
        kiButton.addActionListener(this);
        sipButton.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) {
       if(e.getSource()==beButton)
           hatterzene.loop();
       else if(e.getSource()==kiButton)
           hatterzene.stop();
        else
           mpjel.play();
   }

     public static void main(String[] args) throws Exception {
       (new Hangok()).setVisible(true);
     }
}
```

11.6 A Java2D grafika rövid áttekintése

A *Sun* cég a Java JDK 1.2 változatában egy új alapokra helyezett, hatékony és jó minőségű grafika előállítására alkalmas, kétdimenziós grafikus osztálykönyvtárat jelentetett meg. Az osztálykönyvtár alapját képező, a már megismert ***Graphics*** osztályból származtatott ***Graphics2D*** absztrakt osztály a ***java.awt*** csomagban található, azonban a ***Java 2D API*** további elemeit egy külön csomag, a ***java.awt.geom*** tárolja.

Mivel a ***Graphics2D*** a ***Graphics*** osztályhoz hasonlóan absztrakt osztály, csaknem kizárólag a futtatórendszertől szerezzük meg az objektumát. Ez persze feltételezi, hogy a JVM a grafikus felületet mindig a ***Graphics2D***-vel modellezi, így az mindkét említett osztállyal elérhető, például:

```
public void paint(Graphics g) {
    Graphics2D g2 = (Graphics2D) g;
     // ...
}
```

Az alábbiakban röviden áttekintjük a ***Java 2D*** lehetőségeit, majd pedig egy példaprogramon keresztül szemléltetjük az elmondottak alkalmazását.

Koordináta-rendszerek és transzformációk

A ***Graphics*** által használt felhasználói koordináta-rendszeren (*User Space*) túlmenően létezik egy ún. eszköz (képernyő, ablak, nyomtató stb.) koordináta-rendszer (*Device Space*) is. A ***Graphics2D*** osztály metódusai automatikusan az eszköz koordináta-rendszerbe transzformálják a felhasználói rendszerben elvégzett műveletek eredményét.

Az adatok koordináta-rendszerek közötti átalakítása programozható, melynek során tetszőleges affin síktranszformáció (forgatás, kicsinyítés/nagyítás, eltolás stb.) előírható. A transzformációkat az ***AffineTransform*** osztály objektumai képviselik. A konstruktorban beállított átalakításhoz metódusokkal újabbakat adhatunk (például, forgatás - ***rotate()***, eltolás - ***translate()***), de akár agy másik ***AffineTransform*** objektum transzformációja is hozzákapcsolható (***concatenate()***). Az elkészített transzformációt a ***Graphics2D*** osztály ***transform()*** metódusának hívásával érvényesíthetjük.

A transzformációk pontos elvégzése érdekében a *Java 2D*-ben valós koordinátákat használunk.

Grafikus primitívek objektumai

Java 2D-ben a grafikus primitíveket, amelyeket a ***Graphics*** osztály metódusaiként értünk el, a ***java.awt.geom*** csomag osztályai modellezik. A ***Graphics2D*** csupán két metódust használ ezek megjelenítésére a ***draw***() és a ***fill***(). Az első a grafikus primitív körvonalát rajzolja meg, míg a második a kifesti a belterületét. Minkét metódus csak

olyan objektumot fogad el argumentumként, melyek osztályai megvalósítják a *java.awt.Shape* interfészt.

Szöveg megjelenítéshez a *Graphics2D* osztály *drawString()*, kép láthatóvá tételére pedig a *drawImage()* metódusait használhatjuk.

Néhány grafikus primitívekhez több osztály is tartozik. Példaként tekintsük az egyenes vonal osztályait: *Line2D*, *Line2D.Double*, *Line2D.Float*! A *Line2D* absztrakt osztály, melynek a *Line2D.Double* alosztálya **double** típusú, a *Line2D.Float* alosztálya pedig **float** típusú koordinátákat használ. További osztályok:

Osztály	*Leírás*
Arc2D	ellipszisív,
CubicCurve2D	harmadfokú (Beziér) görbe,
Ellipse2D	ellipszis,
Line2D	egyenes vonal,
Polygon	sokszög,
QuadCurve2D	másodfokú görbe,
Rectangle2D	téglalap,
RoundRectangle2D	lekerekített sarkú téglalap.

A *Polygon* kivételével minden osztály rendelkezik *.Double* és *.Float* alosztályokkal. A táblázatban nem szereplő *GeneralPath* osztály egyenes szakaszokból, másodfokú és harmadfokú görbékből összeállított görbét definiál. Az *Area* osztály pedig lehetővé teszi, hogy a *Shape* objektumok között a *CAG* (*Constructive Area Geometry*) műveleteket (unió, metszet, különbség) végezzünk.

A felsorolásban nem szerepelnek, mivel nem jeleníthetők meg, a pontok *Point2D*, *Point2D.Double* és *Point2D.Float* osztályai. (Megjegyezzük, hogy a már korábban használt *Point* osztály is a *Pont2D* leszármazottja).

Megjelenítés

A *Graphics2D* kibővítette a grafikus felület hagyományos megjelenítését meghatározó *Graphics* attribútumokat, melyeket szintén metódusokkal módosíthatunk:

Beállító metódus	*Attribútum*
`setStroke(Stroke toll)`	az alakzat körvonalát rajzoló toll,
`setPaint(Paint ecset)`	az alakzat belterületét festő ecset,
`setComposite(Composite kombinálás)`	színek kombinálása,
`setTransform(AffineTransform aTrafo)`	transzformáció,
`setClip(Shape alakzat)`	vágó alakzat,
`setFont(Font betűtípus)`	betűkészlet,
`setRenderingHints(RenderingHints.Key hintKey, Object hintValue)`	a megjelenítési algoritmus paraméterezése.

A *java.awt.Stroke* interfészt megvalósító *java.awt.BasicStroke* osztályt használhatjuk a tollak előállításához, és a konstruálás során beállíthatjuk a toll vastagságát, szagga-

tottságát, intézkedhetünk a vonalvégek (*end cap*) kialakításáról, valamint a vonalak csatlakozásairól (*line joins*).

A kitöltés paraméterezéséhez használt ***java.awt.Paint*** interfészt több osztály is megvalósítja, többek között a már jól ismert ***Color*** osztály, illetve annak leszármazottja a ***SysColor*** osztály. Számunkra most a ***GradientPaint*** és a ***TexturePaint*** osztályok az érdekesek. A ***GradientPaint*** osztály konstruktorában két pontot és két színt kell megadnunk, melynek hatására a kifestés a két pont által meghatározott irányban, a beállított két szín közötti finom átmenettel megy végbe. A művelet ismétlődővé is tehető. (A pontok bárhol lehetnek a felhasználói koordináta-rendszerben.) A ***TexturePaint*** osztály használata esetén, a kitöltést előre megadott minta (kép) alapján végzi a rendszer.

A *Java 2D* a ***java.awt.font*** csomag bevezetésével különböző tipográfiai lehetőségekkel egészítette ki az eddigi szövegkezelési műveleteket. Amennyiben a szöveget a ***java.awt.font.TextLayout*** osztály objektumának felhasználásával, a ***Graphics2D draw()*** metódusával jelenítjük meg, akkor annak minden betűjére érvényesek lesznek a fenti beállítások (toll, ecset stb.) A ***TextLayout*** objektum létrehozásához a szöveg és a betűtípus mellett, egy ***FontRenderContext*** típusú objektumra is szükség van, melyet a ***Graphics2D*** osztály ***getFontRenderContext()*** metódusával kaphatunk meg.

Tervezzünk programot, amely bemutatja a ***Java 2D API*** alapvető funkcióinak alkalmazását! (*Grafika2D*)

```
//Grafika2D
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
public class Grafika2D extends JComponent
{
  private Image kep;

  public Grafika2D()
  {
    kep = Toolkit.getDefaultToolkit().getImage("cica.jpg");
  }
  public void paint(Graphics g)
  {
    Graphics2D g2 = (Graphics2D)g;

    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
    RenderingHints.VALUE_ANTIALIAS_ON);

    // az ablak középpontjának koordinátái
    int cx = getSize( ).width / 2;
    int cy = getSize( ).height / 2;

    // a koordináta-rendszer elhelyezése a középpontban
```

```
    g2.translate(cx, cy);
    // és elforgatása 30 fokkal az óramutató járásával ellentétesen
    g2.rotate(-30 * Math.PI / 180);

    // *** Vágás használata
    Shape regiVago = g2.getClip( );
    // ezzel az ellipszissel vágjuk a rajzolt elemeket
    Shape e = new Ellipse2D.Double(-cx, -cy/2, cx * 2, cy);
    g2.clip(e);

    // téglalap létrehozása
    Shape r = new Rectangle2D.Float(-cx, -cy, cx * 3 / 4, cy * 2);
    // vonalas minta 45 fokos szögben kék és
    // sárga színek közötti átmenettel
    g2.setPaint(new GradientPaint(40, 40, Color.blue,50, 50,
                Color.yellow, true));
    g2.fill(r);

    // zöld színnel kitöltött ellipszis (most kör) rajzolása
    Shape k = new Ellipse2D.Float(0, 0, cx, cy);
    g2.setPaint(Color.green);
    g2.fill(k);
    // a lila határvonal beállítása
    g2.setPaint(Color.magenta);
    g2.setStroke(new BasicStroke(5));
    g2.draw(k);
    g2.setClip(regiVago);

    // *** szöveg megjelenítése
    g2.setFont(new Font("Times New Roman", Font.BOLD, 80));
    // vízszintes vonalas minta fekete-piros átmenettel
    g2.setPaint(new GradientPaint(-cx, 0, Color.black, cx, 0,
                Color.red, false));
    g2.drawString("Java 2D", -cx * 2 / 3,  cy / 5 );

    // *** az alábbi képet 60 %-ban áttetsző módon jelenítjük meg
    Composite regiC = g2.getComposite();
    AlphaComposite ac = AlphaComposite.getInstance
                                    (AlphaComposite.SRC_OVER,0.60F);
    g2.setComposite(ac);
    // kép megjelenítése
    g2.drawImage(kep, -cx / 3, -5 * cy / 6, this);
    g2.setComposite(regiC);
}


public static void main(String[] args)
{
    JFrame keret = new JFrame("2D java grafika");
    Container tarolo = keret.getContentPane( );
    tarolo.setLayout(new BorderLayout( ));
    tarolo.add(new Grafika2D(), BorderLayout.CENTER);
    keret.setSize(400, 400);
    keret.setLocation(100, 100);
    keret.addWindowListener(
    new WindowAdapter()
    {
```

```
        public void windowClosing(WindowEvent e)
        {
            System.exit(0);
        }
    }
  );
  keret.setVisible(true);
 }
}
```

Az alkalmazás futásának eredménye:

12. Kisalkalmazások (appletek)

Az eddig alkalmazások önállóan futottak a JVM felügyelete alatt, kihasználva az operációs rendszer adta szöveges és grafikus lehetőségeket. A grafikus felületű alkalmazások esetén egy ***Frame*** osztálytól származtatott ablakban jelentek meg a vezérlők és a grafika.

A Java azonban lehetővé teszi olyan kisalkalmazások (*appletek*) létrehozását is, amelyek egy másik alkalmazás felügyelete alatt működnek. Ez a másik alkalmazás lehet valamilyen Internet-böngésző, vagy például az ***appletviewer*** segédprogram.

A JVM és böngésző a kisalkalmazások biztonságos alkalmazása érdekében egy sor korlátozást valósít meg:

- nem érhető el belőle annak a számítógépnek a fájlrendszere, amelyen fut, még olvasásra sem,
- csak azzal a kiszolgálóval alakíthat ki hálózati kapcsolatot (*sockets*), amelyről letöltötték,
- nem indíthat más folyamatokat (***exec()***),
- nem állíthatja le a JVM futását az ***exit()*** metódussal,
- nem érheti el a rendszertulajdonságokat, például az alábbi utasítás sem futtatható: `System.getProperties().list(System.out);`
- nem nyomtathat a kisalkalmazást futtató számítógéphez kapcsolt nyomtatókon,
- nem használhatja a vágólapot (*clipboard*),
- nem hívhat natív metódusokat stb.

12.1 Az appletek felépítése

A Java szemszögéből az applet egy tároló, amelynek ***java.applet.Applet*** osztálya a ***Panel*** osztályból származik, és speciális vezérlőmetódusokkal rendelkezik: ***init()***, ***start()***, ***stop()***, ***destroy()***. Az applet konstruktorát ritkán használjuk, hisz a konstruktor futtatásakor még nem alakult ki a teljes futási környezet, így bizonyos inicializálási feladatok nem végezhetők el benne, például az átméretezés. Ezért a konstruktor helyett általában az ***init()*** metódust alkalmazzuk.

További fontos eltérés az alkalmazásoktól, hogy az appletek futtatásához nélkülözhetetlen egy HTML-fájl, amely *<applet>* leíróeleme kijelöli, és paraméterezi a futtatandó applet osztályállományát.

A Java appletek a HTML-fájllal együtt általában egy webkiszolgálón tárolódnak, ahonnan az ügyfél böngészője letölti azokat. A sikeres letöltés után a böngésző elin-

dítja az appletet, feltéve, hogy a gépünkre telepítettük a java virtuális gépet. (JVM). A futó applet a HTML-dokumentum részeként jelenik meg,

Pontokba szedve összefoglaltuk kisalkalmazásokról szóló tudnivalókat:

- Az appletek nem rendelkeznek ***main()*** metódussal, helyette az alábbi metódusokat definiálhatjuk, melyeket a böngésző hív:

  init(), ***start()***, ***stop()***, ***destroy()***

- Az appleteket származtathatjuk a ***java.applet.Applet*** *AWT* konténerosztályból vagy a ***javax.swing.JApplet*** -től, ha a *Swing* osztályait használjuk.
- A HTML-dokumentumban az alábbi formában kell elhelyeznünk az *Applet1.java* forrásfájl fordításával előállított *Applet1.class* állomány nevét:

  ```
  <APPLET CODE="Applet1.class"> </APPLET>
  ```

  Az utasításban a kisalkalmazás munkaterületének méretét is beállíthatjuk a szélesség (*WIDTH*) és a magasság (*HEIGHT*) képpontokban mért értékeinek megadásával:

  ```
  <APPLET CODE="Applet1.class" WIDTH=350 HEIGHT=200> </APPLET>
  ```

  Megadhatjuk az applet forrásállományának hivatkozását is, amely így letölthető a kiszolgálóról:

  ```
  <A HREF = "Applet1.java"> Az applet forráskódja</A>
  ```

12.1.1 A kisalkalmazás osztályának deklarálása

Az appletet egy **public** elérésű főosztállyal valósítjuk meg, melyet az ***Applet*** osztálytól származtatunk:

```
import java.awt.*;
import java.applet.*;
public class Applet1 extends Applet
{
  // . . .
}
```

Ugyancsak készíthetünk kisalkalmazást a *swing* ***JApplet*** osztályának felhasználásával:

```
import java.awt.*;
import javax.swing.*;
public class Applet1 extends JApplet
{
  // . . .
}
```

Nézzük meg részletesen a következő példaprogramot!

Türkiz háttérben, véletlenszám-generátorral előállított koordinátájú pontban, véletlenszerű színnel írjuk ki a „Java 5" szöveget! (*Alapok\Minta_AWT*)

(Megjegyezések: a *„Java 5"* felirat a *Java 2 SE 5.0* verziójára utal; a példaprogram *swinget* használó változata az *Alapok\Minta_Swing* könyvtárban található.)

Az *Applet1* főosztályon belül deklarált *font* adatmező a megjelenített szöveg stílusát határozza meg, míg a *pozíció* adatmező a kiírás pozícióját definiálja.

```
public class Applet1 extends Applet
{
        Font font;
        Point pozicio;
}
```

12.1.2 A főosztály konstruktora

A kisalkalmazás osztályának konstruktorát ritkán használjuk, inkább az ***init()*** metódusban adjuk meg az kiindulási beállításokat, hisz azok legtöbbször a böngészővel, illetve a megjelenítéssel kapcsolatosak. Amennyiben mégis definiálunk paraméter nélküli konstruktort, annak nyilvános elérésűnek kell lennie, mivel JVM csak így tudja előállítani az osztály objektumát.

```
public Applet1()
{
        pozicio = new Point();
        font =new Font("Times New Roman", Font.BOLD, 30);
}
```

A példa kedvéért két objektum létrehozását az ***init()*** metódus helyett a konstruktorban helyeztük el. A ***Point*** típusú objektum *x* és *y* tagjait a koordináták tárolására használjuk, míg a *font* adatmezővel a *Font* osztály egy példányát azonosítjuk (*Times New Roman* betűtípus, *BOLD* stílus és *30* pontos betűméretet).

12.1.3 Az init() és a destroy() metódusok

Az ***init()*** metódust, amelynek feladata a kisalkalmazás futtatásához szüksége előkészítő lépések elvégzése, a böngésző (vagy az ***appletviewer***) hívja egyetlen-egyszer, a kisalkalmazás osztályának betöltését követően. A példánk ***init()*** metódusában csak az applet háttérszínét állítottuk be.

```
public void init()
{
        setBackground(Color.cyan);
}
```

Az *init()* metódus párja a *destroy()*, melyet a böngésző az applet befejezésekor hív. A *destroy()* metódusban intézkedhetünk az *init()*-ben lefoglalt erőforrások (például kommunikációs csatornák) felszabadításáról. (Ritkán használjuk.)

12.1.4 A start() és a stop() metódusok – a kisalkalmazás indítása, megállítása

A böngésző (*appletviewer*) az *init()* lefutása után aktivizálja a *start()* metódust. Ezt követően a *stop()* és a *start()* metódusokat felváltva hívja, ha kisalkalmazást tartalmazó oldal nem látható, illetve ismét látható lesz.

Amennyiben az applet oldaláról elnavigálunk, majd pedig visszalépünk rá, illetve használjuk a böngésző frissítés gombját, az *init()* és a *start()* metódusok egyaránt lefutnak.

Az *Applet* osztálytól örökölt *start()* metódus üres, és csak speciális esetekben bíráljuk felül, párban a *stop()* metódussal, például animációk készítésénél.

A mintafeladat *start()* metódusában véletlenszám-generátor segítségével állítjuk be az előtér (rajzolás) színét, valamint a szöveg megjelenítési pozícióját.

```
public void start()
{
      pozicio.x = (int) (90 * Math.random()+50);
      pozicio.y = (int) (100* Math.random()+50);
      setForeground(new Color(  (int)(Math.random()*256),
                                (int)(Math.random()*256),
                                (int)(Math.random()*256) ) );
}
```

12.1.5 A paint() – az applet grafikus megjelenítése

Az alkalmazásokhoz hasonlóan, a *paint()* metódus felelős a kisalkalmazás felületének újrarajzolásáért. Ha nem bíráljuk felül, akkor csak egy üres téglalap jelenik meg a böngészőben. Amennyiben az applet csak vezérlőket tartalmaz, nincs is szükség a *paint()* metódusra.

Ha az applet tartalmaz *paint()* metódust, akkor az először *start()* hívása után aktivizálódik, majd ezt követően mindig, ha a kisalkalmazás területét újra kell rajzolni. (Ezeket hívásokat a böngésző vezérli.)

A példában az újrarajzolás során beállítjuk a betűtípust, és megjelenítjük a szöveget:

```
public void paint(Graphics g)
{
      g.setFont(font);
      g.drawString("Java 5", pozicio.x, pozicio.y);
}
```

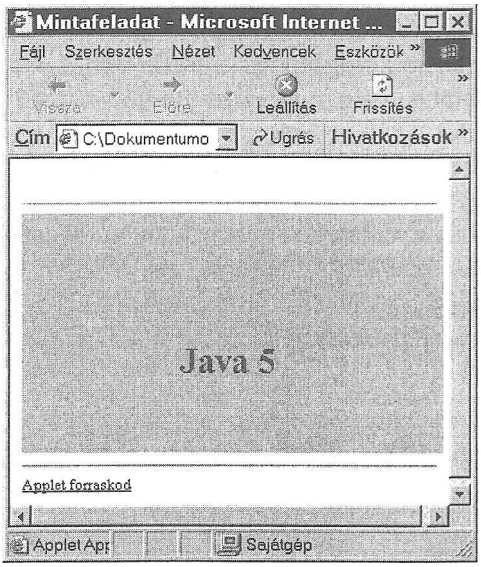
12.1.7 Az applet a HTML-dokumentumban

Bemutatjuk azt az egyszerű HTML-kódot (*Applet1.html*), amely megjeleníti az *Applet1.class* állományban tárolt kisalkalmazást, és két vízszintes vonallal elválasztja az oldal többi részétől:

```
<HTML>
<HEAD>
<TITLE> Mintafeladat </TITLE>
<BODY>
<BR>

<HR>
<APPLET CODE="Applet1.class" WIDTH=350 HEIGHT=200> </APPLET>
<HR>
<a href= "Applet1.java">Applet forraskod</a>
</BODY>
</HTML>
```

12.1.8 A példaapplet tesztelése a böngészőben

Minden kisalkalmazás feladatot külön könyvtárban tárolunk, ahol megtalálható az applet *.java* forráskódja, a lefordított *.class* valamint a *.html* állomány, amely lehetővé teszi az applet tesztelését a böngészőben. Jelenítsük meg az *Alapok\Minta_AWT* könyvtárban található *Applet1.html* dokumentumot a böngészőben!



12.2 Az appletek paraméterezése

A kisalkalmazás segítségével a HTML-oldalainkat dinamikussá, látványossá tehetjük. Sajnos a különböző böngészőkben az appletek megjelenése és futtatása eltérő lehet. A legtöbb böngésző ismeri az *APPLET* HTML-leíróelem alábbi paraméterezését:

```
<APPLET NAME="Applet1" CODE="Applet1.class"
        CODEBASE ="http://www.javahol.hu/appletek/"
        WIDTH="350" HEIGHT="200" ALIGN="TOP"
        HSPACE="10" VSPACE="10"
        ALT="Ha böngésző támogatja, egy applet jelenik meg itt"
        <STRONG> Az applet betöltése folyamatban...</STRONG>
        <PARAM NAME  = "Név1" VALUE = "Érték1">
        <PARAM NAME  = "Név2" VALUE = "Érték2">
        <PARAM NAME  = "Név3" VALUE = "Érték3">
</APPLET>
```

A *NAME*, a *CODE* és a *CODEBASE* paraméterek együtt határozzák meg, hogy melyik kisalkalmazást kell letölteni és elindítani. Ha *CODEBASE* hiányzik, akkor a böngésző a HTML-állomány könyvárában keresi a *CODE* állományt.

A következő öt paraméter a kisalkalmazás területének elhelyezkedését szabályozza a weblapon. A méretmegadással már foglalkoztunk, az *ALIGN* szerint (például *MIDDLE*, *CENTER*, *RIGHT*) igazítja a böngésző az applet téglalapját, és körülötte *VSPACE* és *HSPACE* távolságban találhatók a weblap többi összetevői.

Az *ALT* paraméterben megadott szöveg akkor jelenik meg, ha az appletet nem tudta megjeleníteni a böngésző. A *STRONG* leíróelemben megadott szöveg addig látható, amíg a JVM betöltődik, és a kisalkalmazás elindul.

A PARAM leíróelem használata

Az applet és a HTML-dokumentum között fontos az adatcsere megvalósítása. Ennek egyik formája, a kisalkalmazása paraméterezése a *PARAM* leíróelemmel, melynek *NAME* tagja a paraméter nevét, míg a *VALUE* tagja a paraméter értékét tárolja, szöveges formában. A futó appletből a ***String*** típusú ***getParameter()*** metódussal kérdezhetjük le az adott nevű paraméterhez tartozó értéket.

Tervezzünk appletet, amely a html-kódból paraméterként átvesz egy nevet és egy életkort! (*Alapok\Parameter1*)

A Param.java állomány tartalma:

```
import java.applet.*;
import java.awt.*;
```

```
public class Param extends Applet
{
   String nev;
   int kor;
   public void init() {
     nev = getParameter("Nev");
     String parameter = getParameter("Kor");
     if (parameter != null)
       kor = Integer.parseInt(parameter);
   }
   public void paint(Graphics g) {
     g.drawString(nev + "  " + kor, 35, 45);
   }
}
```

A start.html fájl tartalma:

```
<applet code = Param.class width=300 height=300>
  <param name = Nev value="Kiss Arpad">
  <param name = Kor value=12>
</applet>
```

Az applet futásának eredménye:

Kiss Arpad 12

Java applet és JavaScript kommunikáció

A futó Java kisalkalmazás és a weboldal között kétirányú adatcsere is kialakítható, ha a HTML-kódot JavaScript-tel bővítjük. A kommunikáció alapja, hogy a JavaScript utasításokból elérhetjük a Java applet nyilvános adatmezőit, illetve hívhatjuk a **public** metódusait.

Készítsünk alkalmazást, amelyben az appletben és a weblapon elhelyezett szövegmezők kölcsönösen felveszik egymás értékét. (*Alapok\Paramter2*)

A ParamJS.java állomány tartalma:

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

public class ParamJS extends JApplet implements ActionListener
{
      public void init() {
            setBackground(Color.cyan);
            setLayout(new FlowLayout(FlowLayout.LEFT));
            add(cimke);
            add(adatmezo);
            adatmezo.setText(""+kod);
            adatmezo.addActionListener(this);
            resize(200,50);
      }
```

```
        public void actionPerformed(ActionEvent e){
                kod = Integer.parseInt(adatmezo.getText());
                megvaltozott = true;
        }
        public void frissit(String s) {        // kívülről érjük el
                kod=Integer.parseInt(s);
                adatmezo.setText(s);
        }
        JLabel cimke = new JLabel(" Kód: ");
        JTextField adatmezo = new JTextField(10);
        public int kod=1223;                   // kívülről érjük el
        public boolean megvaltozott = true;    // kívülről érjük el
}
```

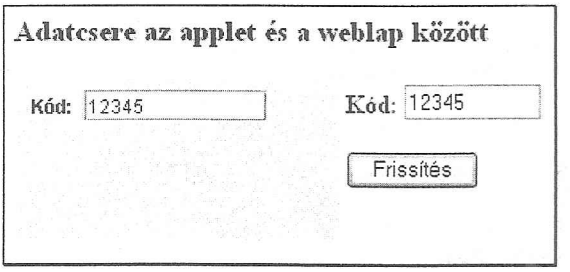
A index.html fájl tartalma:

```
<HTML>
<HEAD>
<SCRIPT type="text/javascript">
<!--
function kialvaso() {
 if (document.ParamJS.megvaltozott) {
   values.kod.value=document.ParamJS.kod;
   document.ParamJS.megvaltozott=false;
 }
 setTimeout('kialvaso()', 100)
}
//-->
</SCRIPT>
</HEAD>
<BODY onload="setTimeout('kialvaso()', 100)">
<H3>Adatcsere az applet és a weblap között</H3>
<table> <tr>
   <td width=200>
        <APPLET  name=ParamJS code=ParamJS.class
                 height=100 width=200></APPLET>
   </td>
   <td valign=top>
     <form name=values>
        <B>  Kód:</B>
        <input name=kod value="0" size=10
               onkeyup="this.value=this.value % 50000">
       <br>
       <input value=" Frissítés "
         type=button onclick="document.ParamJS.frissit(kod.value);">
     </form>
  </td>
</tr> </table>
</BODY>
</HTML>
```

A megoldásban problémát az okozott, hogyan, mikor vegyük át a kisalkalmazásból a megváltozott kódértéket. Ehhez definiáltunk egy JavaScript függvényt, amely 0,1 má-

sodpercenként ellenőrzi az applet *megvaltozott* változójának értékét, és ha kell, frissíti a weblap szövegmezejének tartalmát.

Az applet futásának eredménye:



12.3 Az applet és alkalmazás egyben

A hajsampon reklámnak tűnő cím nagyon is jogos igényt rejt. Hogyan lehet úgy fejleszteni Java nyelven, hogy az alkalmazásként és appletként is ugyanúgy legyen futtatható.

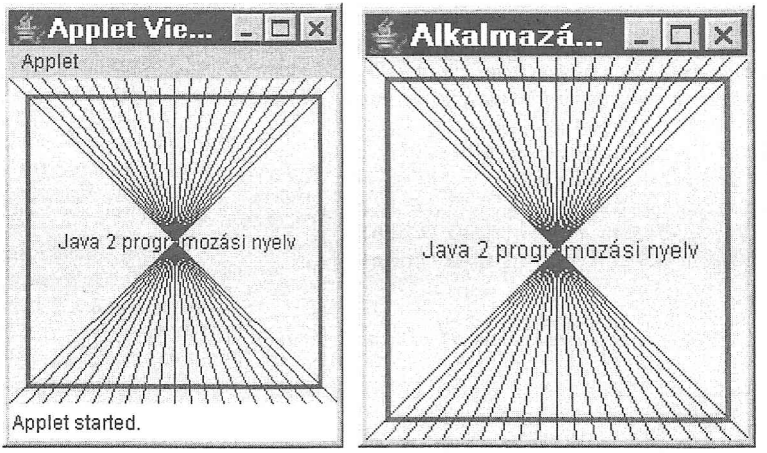
A megoldás lényege, hogy a kisalkalmazás nyilvános osztályában definiáljuk a szokásos ***main()*** metódust, ami senkit sem zavar. A ***main()*** metódusban létrehozzuk az alkalmazás ablakát, amelyben aztán megjeleníti az applet példányát.

Tervezzünk appletet, amelyet egy alkalmazás jelenít meg a böngészőben látható applettel azonos méretben! (*Alapok\Kisalkalmazás*)

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Kisalkalmazas extends Applet
{
   // Az applet méretei
   public final int appletW = 200, appletH = 200;

   // Az alkalmazás indításához szükséges metódus
   public static void main(String[] args)
   {
      Alkalmazas alkalmazas = new Alkalmazas
                                 ("Alkalmazás és applet egyben");
      alkalmazas.setVisible(true);
   }

   public void init() {
      setSize(appletW,appletH);
      setBackground(Color.yellow);
      setForeground(Color.blue);
   }
```

```
   public void paint(Graphics g)
   {
      // Az ablak befoglaló méreteinek lekérdezése
      Rectangle r=getBounds();
      // Ha nem appletként, vagyis alkalmazásként fut
      if (!isActive()) {
         g.translate(getInsets().left, getInsets().top);
         // A munkaterület méretének meghatározása
         r.height -= getInsets().bottom + getInsets().top;
         r.width  -= getInsets().left + getInsets().right;
      }
      // Keretezés vastag piros vonallal
      g.setColor(Color.red);
      g.drawRect(10,10 ,r.width-20,r.height-20);
      g.drawRect(11,11 ,r.width-22,r.height-22);
      g.drawRect(12,12 ,r.width-24,r.height-24);
      // Sugársor rajzolása
      g.setColor(Color.blue);
      for (int x=0; x<=r.width; x+=10) {
         g.drawLine(r.width/2, r.height/2, x, r.height-1);
         g.drawLine(r.width/2, r.height/2, x, 0);
      }
      g.drawString("Java 2 programozási nyelv",30,105);
   }
}

   // Az alkalmazás futtatásakor az appletet magunk aktiváljuk
   class Alkalmazas extends Frame
   {
      private Kisalkalmazas applet;

      public Alkalmazas(String cimsor) {
         super(cimsor);
         addWindowListener(new AlkalmazasAblak());
         applet=new Kisalkalmazas();
         applet.init();
         // Az alkalmazás ablakméreteit úgy módosítjuk, hogy
         // a munkaterület mérete megegyezzen az appletével
         this.setVisible(true);
         this.setSize(applet.appletW+getInsets().left+
              getInsets().right,
         applet.appletH+getInsets().top+getInsets().bottom);
         applet.start();
         add(applet);
      }

      // Az ablak lezárásának kezelése
      class AlkalmazasAblak extends WindowAdapter
      {
          public void windowClosing(WindowEvent e) {
            applet.stop();
            applet.destroy();
            System.exit(0);
          }
      }
   }
```

Az applet és az alkalmazás futásának eredménye:



12.4 Komponensek használata appletekben

Az *AWT* és a *Swing* típusú komponensek használatára különféle feladatokat készítettünk, és ezeket külön alfejezetekben mutatjuk be. A kisalkalmazásokban ugyanúgy kell a különböző építőelemeket kezelni, mint a Java alkalmazásokban. Ezért az elkövetkezőben csak azokat a lépéseket magyarázzuk, ahol a megértést segítheti egy-két gondolat hozzáfűzése a látottakhoz.

12.4.1 AWT komponensek használata

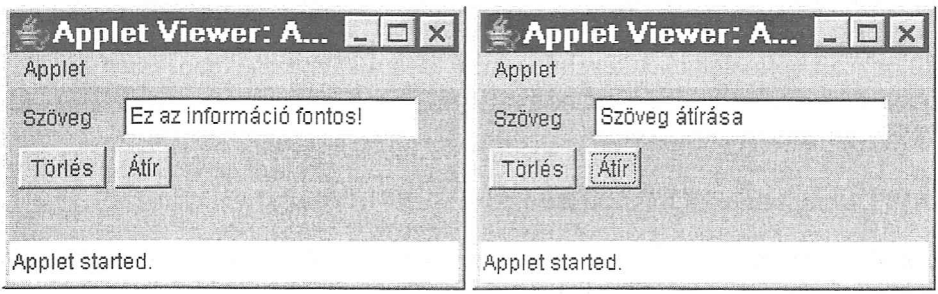
Ebben az alfejezetben az *AWT* (*Abstract Window Toolkit*) osztálykönyvtár komponenseinek használatára látunk példákat.

Tervezzünk appletet, amely a szövegmezőbe beírt szöveget a *Törlés* gomb megnyomásával törli, az *Átír* gomb megnyomásával pedig a „*Szöveg átírása*" szöveggel helyettesíti! (*AWT\Text1*)

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class Applet1 extends Applet implements ActionListener
{
   Label fejlec = new Label("Szöveg");
   Button button1 = new Button("Törlés");
   Button button2 = new Button("Átír");
   TextField szoveg = new TextField(20);
```

```
    public void init()
    {
       setSize(250,250);
       setLayout(new FlowLayout(FlowLayout.LEFT));
       this.setBackground(new Color(0,250,255));
       this.setForeground(new Color(0,0,0));
       this.add(fejlec);
       this.add(szoveg);
       this.add(button1);
       this.add(button2);
       button1.addActionListener(this);
       button2.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e)
    {
       if (e.getSource()==button1)
            szoveg.setText("");
       if (e.getSource()==button2)
            szoveg.setText("Szöveg átírása");
    }
}
```

Az applet futásának eredménye:

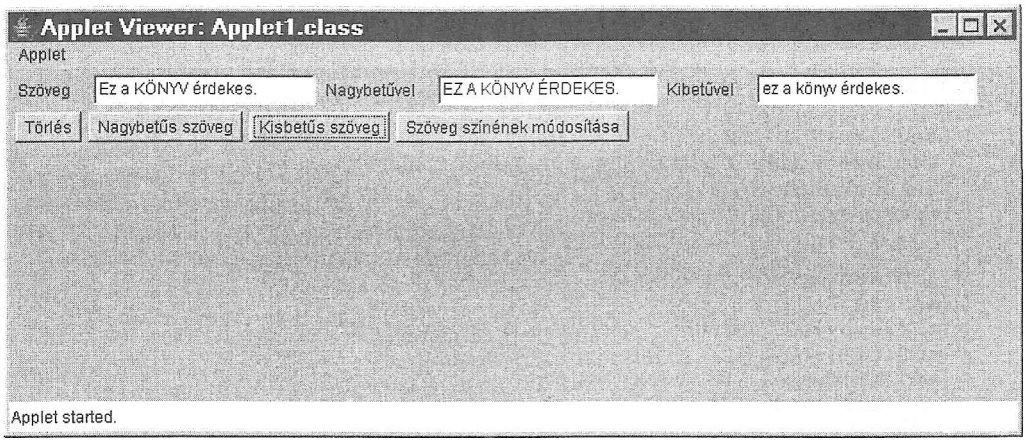


Készítsünk appletet, amely egy szövegmezőbe írt szöveget nagybetűssé, kisbetűssé alakít két másik szövegmezőbe, valamint a szöveg színét is meg tudja változtatni! (*AWT\Text2*)

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class Applet1 extends Applet implements ActionListener
{
   Label fejlec1 = new Label("Szöveg");
   Label fejlec2 = new Label("Nagybetűvel");
   Label fejlec3 = new Label("Kibetűvel");
   Button button1 = new Button("Törlés");
   Button button2 = new Button("Nagybetűs szöveg");
   Button button3 = new Button("Kisbetűs szöveg");
```

```
    Button button4 = new Button("Szöveg színének módosítása");
    TextField szoveg1 = new TextField(20);
    TextField szoveg2 = new TextField(20);
    TextField szoveg3 = new TextField(20);

    public void init()
    {
       setSize(750,250);
       setLayout(new FlowLayout(FlowLayout.LEFT));
       setBackground(Color.cyan);
       setForeground(Color.black);

       add(fejlec1);
       add(szoveg1);
       add(fejlec2);
       add(szoveg2);
       add(fejlec3);
       add(szoveg3);
       add(button1);
       add(button2);
       add(button3);
       add(button4);
       button1.addActionListener(this);
       button2.addActionListener(this);
       button3.addActionListener(this);
       button4.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e)
    {
       String sz = new String("                                    ");
       String sz2 = new String("                                    ");

       if (e.getSource()==button1)
       {
          szoveg1.setText("");
          szoveg2.setText("");
          szoveg3.setText("");
       }

       if (e.getSource()==button2)
          szoveg2.setText( szoveg1.getText().toUpperCase());

       if (e.getSource()==button3)
          szoveg3.setText(szoveg1.getText().toLowerCase());

       if (e.getSource()==button4)
       {
          szoveg1.setForeground(Color.red);
          szoveg2.setForeground(Color.blue);
          szoveg3.setForeground(Color.green);
       }
    }
}
```

Az applet futásának eredménye:



Tervezzünk appletet, amely egy nyomógomb megnyomásakor megváltoztatja a háttér és a szöveg színét! (*AWT\Button*)

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class Applet1 extends Applet implements ActionListener
{
     Label label1 = new Label();
     Button button1 = new Button("Szinek változtatása");
     Font font = new Font("Times New Roman",Font.BOLD,30);

     public void init()
     {
        setSize(320,200);
        setBackground(Color.cyan);
        setForeground(Color.blue);
        label1.setText("Háttér és előtér színe változik");
        setLayout(new BorderLayout());
        add("North",label1);
        add("South",button1);
        button1.addActionListener(this);
     }

     public void paint(Graphics g)
     {
        g.setFont(font);
        g.drawString("Java 5",90,100);
     }
```

```
    public void actionPerformed(ActionEvent e)
    {
       if (e.getSource()==button1)
       {
            setBackground(new Color((int)(Math.random()*256),
             (int)(Math.random()*256),(int)(Math.random()*256)));
            setForeground(new Color((int)(Math.random()*256),
             (int)(Math.random()*256),(int)(Math.random()*256)));
       }
    }
}
```

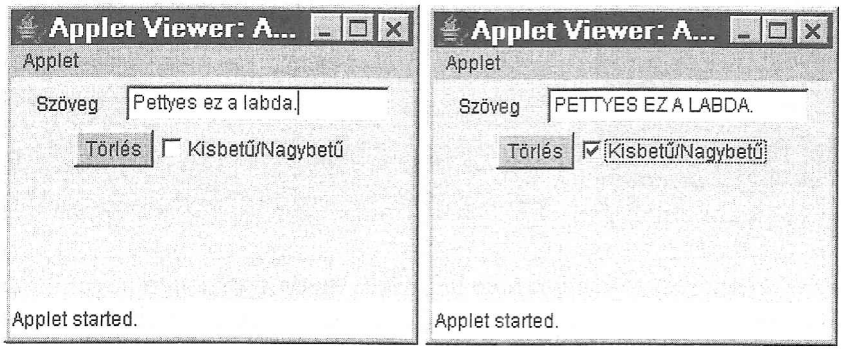
Az applet futásának eredménye:

Tervezzünk appletet, amely egy szövegmezőbe beírt szöveget a *Törlés* nyomógombbal törli, és egy jelölőnégyzet (*Checkbox*) a szöveget kisbetűssé, illetve nagybetűssé változtatja! (*AWT\Check*)

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class Applet1 extends Applet
                     implements ActionListener, ItemListener
{
   Label fejlec = new Label("Szöveg");
   Button button1 = new Button("Törlés");
   TextField szoveg = new TextField(20);
   Checkbox valtas = new Checkbox("Kisbetű/Nagybetű",false);

   public void init() {
      setSize(250,250);
      setBackground(Color.yellow);
      setForeground(Color.black);

      add(fejlec);
      add(szoveg);
      szoveg.setForeground(Color.blue);
```

```
        add(button1);
        add(valtas);
        button1.addActionListener(this);
        valtas.addItemListener(this);
    }

        public void actionPerformed(ActionEvent e) {
            if (e.getSource()==button1) szoveg.setText("");
        }

        public void itemStateChanged(ItemEvent e) {
            if (e.getSource()==valtas && e.getStateChange()==e.SELECTED)
                szoveg.setText(szoveg.getText().toUpperCase());
            else
                szoveg.setText(szoveg.getText().toLowerCase());
        }
}
```

Az applet futásának eredményei:

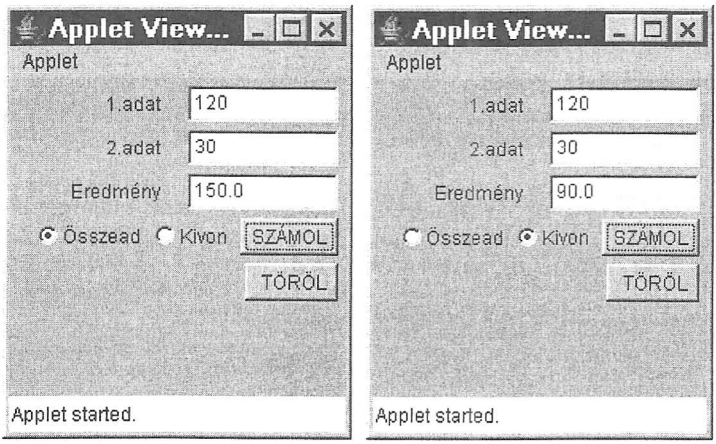


Készítsünk appletet, amely két adaton kétféle műveletet tud végrehajtani, attól függően, hogy melyik választógomb (*Összead* illetve *Kivon*) van bejelölve! A műveletet a *Számol* nyomógomb hajtja végre, a *Töröl* nyomógomb pedig törli az adatmezőket. (*AWT\CheckboxGroup*)

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class Applet1 extends Applet
                    implements ActionListener, ItemListener
{
   Label label1 = new Label("            1.adat");
   Label label2 = new Label("            2.adat");
   Label label3 = new Label("          Eredmény");

   TextField edit1 = new TextField("0",10);
   TextField edit2 = new TextField("0",10);
   TextField edit3 = new TextField("0",10);
```

```
    CheckboxGroup muvelet = new CheckboxGroup();
    Checkbox osszead = new Checkbox("Összead",muvelet,false);
    Checkbox kivon = new Checkbox("Kivon",muvelet,false);

    Button button1 = new Button("SZÁMOL");
    Button button2 = new Button(" TÖRÖL");
    Panel p1=new Panel();

    public void init() {
       setSize(210,200);
       setLayout(new FlowLayout(FlowLayout.RIGHT));
       setBackground(Color.cyan);
       setForeground(Color.blue);
       add(label1);
       add(edit1);
       add(label2);
       add(edit2);
       add(label3);
       add(edit3);
       add(osszead);
       add(kivon);
       add(button1);
       add(button2);
       button1.addActionListener(this);
       button2.addActionListener(this);
       osszead.addItemListener(this);
       kivon.addItemListener(this);
    }

    public void actionPerformed(ActionEvent e) {
       double ered;
       if (e.getSource()==button1) {
          double adat1 = new Double(edit1.getText()).doubleValue();
          double adat2 = new Double(edit2.getText()).doubleValue();
          if(muvelet.getSelectedCheckbox() == osszead) {
              ered = adat1 + adat2;
              edit3.setText(""+ ered);
          }
          if(muvelet.getSelectedCheckbox() == kivon) {
              ered = adat1-adat2;
              edit3.setText(""+ ered);
          }
       }
       if (e.getSource()==button2) {
          edit1.setText("0");
          edit2.setText("0");
          edit3.setText("0");
       }
    }

    public void itemStateChanged(ItemEvent e) {
       if (e.getSource() == osszead && e.getStateChange()==e.SELECTED)
              edit3.setText("");
       if (e.getSource() == kivon && e.getStateChange()==e.SELECTED)
              edit3.setText("");
    }
}
```

Az applet futásának eredményei:

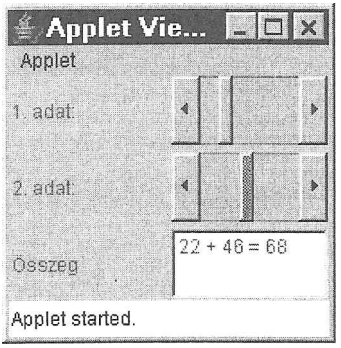


Tervezzünk appletet, amelyben két, 1 és 100 határokkal beállított görgetősávot (*Scrollbar*) tartalmaz! A görgetősávok csúszkáit tologatva, automatikusan összegezi a csúszkák pozícióinak megfelelő értékeket egy szövegmezőben. (*AWT\Gorgeto*)

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class Applet1 extends Applet implements AdjustmentListener
{
   Label label1 = new Label("1. adat:");
   Label label2 = new Label("2. adat:");
   Label label3 = new Label("Összeg");
   TextField ered = new TextField(20);

   Scrollbar szam1 = new Scrollbar(Scrollbar.HORIZONTAL, 0, 0, 0, 101);
   Scrollbar szam2 = new Scrollbar(Scrollbar.HORIZONTAL, 0, 0, 0, 101);
   Button button2 = new Button("Összeg");

   public void init() {
        setSize(200,140);
        setBackground(new Color(200,227,72));
        setForeground(new Color(255,0,255));
        setLayout(new BorderLayout());
        setLayout(new GridLayout(3,2,5,5));

        add(label1);    add(szam1);
        szam1.setBlockIncrement(1);
        add(label2);    add(szam2);
        szam2.setBlockIncrement(1);
        add(label3);    add(ered);
        szam1.addAdjustmentListener(this);
        szam2.addAdjustmentListener(this);
   }
```

```
    public void adjustmentValueChanged(AdjustmentEvent e)
    {
        if (e.getSource() == szam1) {
            ered.setText(String.valueOf(szam1.getValue()) +
                " + " + String.valueOf(szam2.getValue()) + " = " +
                String.valueOf(szam1.getValue()+szam2.getValue()));
        }
        if (e.getSource() == szam2) {
            ered.setText(String.valueOf(szam1.getValue()) + " + " +
                String.valueOf(szam2.getValue()) + " = " +
                String.valueOf(szam1.getValue()+szam2.getValue()));
        }
    }
}
```

Az applet futásának eredménye:

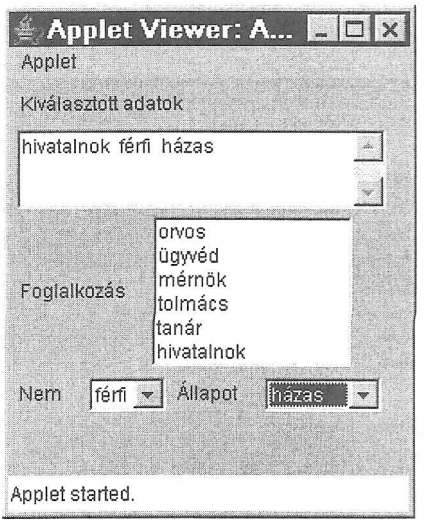


Készítsünk appletet, amely egy listából és két kombinált listából kiválasztott tételeket szövegablakban jeleníti meg! (*AWT\Lista*)

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class Applet1 extends Applet implements ItemListener
{
    Label felirat1 = new Label("Kiválasztott adatok");
    Label felirat2 = new Label("Foglalkozás");
    Label felirat3 = new Label("Nem");
    Label felirat4 = new Label("Állapot");
    TextArea text = new TextArea("",2,30);
    List foglalkozas = new List(6,true);
    Choice nem = new Choice();
    Choice allapot = new Choice();
    Panel mezo = new Panel();
```

```
    public void init() {
        setSize(250,250);
        setBackground(Color.cyan);
        setLayout(new FlowLayout(FlowLayout.LEFT));
        add(felirat1);
        add(text);
        foglalkozas.add("orvos");
        foglalkozas.add("ügyvéd");
        foglalkozas.add("mérnök");
        foglalkozas.add("tolmács");
        foglalkozas.add("tanár");
        foglalkozas.add("hivatalnok");
        add(felirat2);
        add(foglalkozas);
        nem.addItem(" - ");
        nem.addItem("férfi");
        nem.addItem("nő");
        add(felirat3);
        add(nem);
        allapot.addItem(" - ");
        allapot.addItem("nőtlen");
        allapot.addItem("hajadon");
        allapot.addItem("házas");
        allapot.addItem("özvegy");
        add(felirat4);
        add(allapot);
        foglalkozas.addItemListener(this);
        nem.addItemListener(this);
        allapot.addItemListener(this);
        add(mezo);
    }

    public void itemStateChanged(ItemEvent e) {
        if (e.getSource() == nem &&
            e.getStateChange() == ItemEvent.SELECTED) {
              text.append("  ");
              text.append( (String) e.getItem());
        }
        if (e.getSource() == allapot &&
            e.getStateChange() == ItemEvent.SELECTED) {
              text.append("  ");
              text.append( (String) e.getItem());
        }
        if (e.getSource() == foglalkozas &&
            (e.getStateChange() == ItemEvent.SELECTED ||
             e.getStateChange() == ItemEvent.DESELECTED)) {
              text.setText(foglalkozas.getItem(
                   Integer.parseInt(e.getItem().toString()) ));
              int index = foglalkozas.getSelectedIndex();
              foglalkozas.deselect(index);
        }
    }
}
```

Az applet futásának eredménye:



12.4.2 Swing komponensek használata

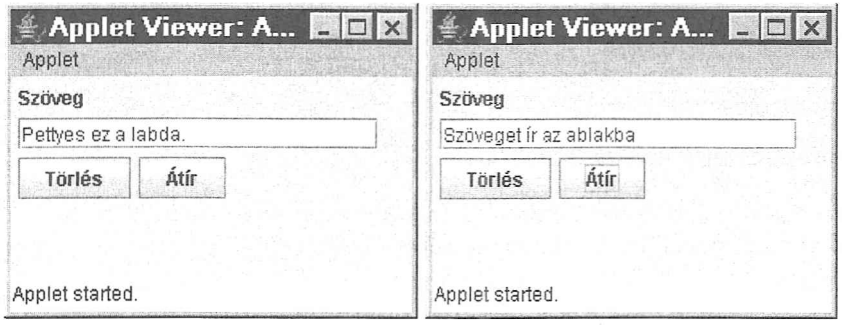
Ebben az alfejezetben a *Swing* osztálykönyvtár komponenseinek appletekben való alkalmazására mutatunk példákat. Napjainkban a Java fejlesztések során egyre inkább a látványosabb, nagyobb tudással felvértezett swing-komponenseket használják.

Készítsünk appletet, amely egy szövegmezőbe írt szöveget töröl, valamint tárolt szöveggel felülír! (*Swing\Text1*)

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

public class Applet1 extends JApplet implements ActionListener
{
   JLabel fejlec = new JLabel("Szöveg");
   JButton button1 = new JButton("Törlés");
   JButton button2 = new JButton("Átír");
   JTextField szoveg = new JTextField(20);
   JPanel mezo = new JPanel();

   public void init()
{
      setSize(250,250);
      mezo.setLayout(new FlowLayout(FlowLayout.LEFT));
      setBackground(new Color(0,250,255));
      setForeground(new Color(0,0,0));
```

```
        mezo.add(fejlec);
        mezo.add(szoveg);
        mezo.add(button1);
        mezo.add(button2);
        setContentPane(mezo);
        button1.addActionListener(this);
        button2.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource()==button1)
            szoveg.setText("");
        if (e.getSource()==button2)
            szoveg.setText("Szöveget ír az ablakba");
    }
}
```

Az applet futásának eredményei:

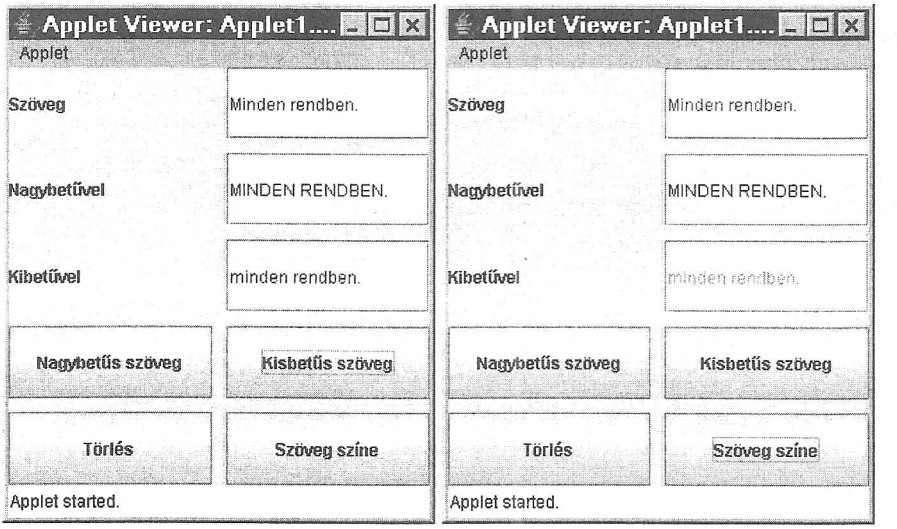


Tervezzünk appletet, amely nyomógombok segítségével a szövegmezőbe beírt szöveget nagybetűssé, illetve kisbetűssé alakítva a megfelelő szövegablakba ír! Egy gombbal a szövegek színe is változtatható legyen! (*Swing\Text2*)

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

public class Applet1 extends JApplet implements ActionListener
{
  JLabel fejlec1 = new JLabel("Szöveg");
  JLabel fejlec2 = new JLabel("Nagybetűvel");
  JLabel fejlec3 = new JLabel("Kibetűvel");
  JButton button1 = new JButton("Törlés");
  JButton button2 = new JButton("Nagybetűs szöveg");
  JButton button3 = new JButton("Kisbetűs szöveg");
  JButton button4 = new JButton("Szöveg színe");
  JTextField szoveg1 = new JTextField(20);
  JTextField szoveg2 = new JTextField(20);
  JTextField szoveg3 = new JTextField(20);
  JPanel mezo = new JPanel();
```

```
    public void init() {
        setSize(300,300);
        mezo.setLayout(new GridLayout(5,2,10,10));
        mezo.add(fejlec1);  mezo.add(szoveg1);
        mezo.add(fejlec2);  mezo.add(szoveg2);
        mezo.add(fejlec3);  mezo.add(szoveg3);
        mezo.add(button2);  mezo.add(button3);
        mezo.add(button1);  mezo.add(button4);
        setContentPane(mezo);
        button1.addActionListener(this);
        button2.addActionListener(this);
        button3.addActionListener(this);
        button4.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource()==button1) {
            szoveg1.setText("");
            szoveg2.setText("");
            szoveg3.setText("");
        }
        if (e.getSource()==button2)
            szoveg2.setText(szoveg1.getText().toUpperCase());
        if (e.getSource()==button3)
            szoveg3.setText(szoveg1.getText().toLowerCase());
        if (e.getSource()==button4) {
            szoveg1.setForeground(Color.red);
            szoveg2.setForeground(Color.blue);
            szoveg3.setForeground(Color.green);
        }
    }
}
```

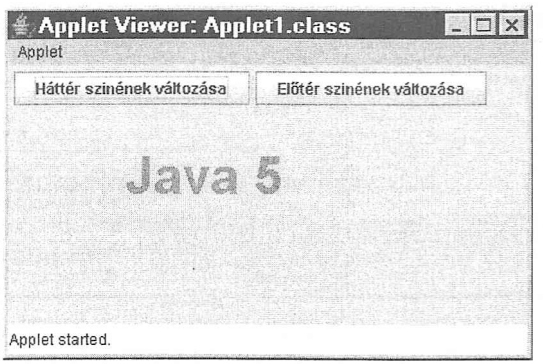
Az applet futásának eredményei:



Készítsünk appletet, amely nyomógombok segítségével változtatja a háttér és a szöveg írásának a színét! (*Swing\Button*)

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

public class Applet1 extends JApplet implements ActionListener
{
   JButton button1 = new JButton("Háttér szinének változása");
   JButton button2 = new JButton("Előtér szinének változása");
   Font font = new Font("Courier",Font.BOLD,40);
   JPanel mezo = new JPanel();

   public void init() {
     mezo.setSize(250,250);
     mezo.setBackground(Color.cyan);
     mezo.setForeground(Color.blue);

     mezo.setLayout(new FlowLayout(FlowLayout.LEFT));
     mezo.add(button1);
     mezo.add(button2);
     setContentPane(mezo);
     setVisible(true);

     button1.addActionListener(this);
     button2.addActionListener(this);
   }

   public void paint(Graphics g) {
     g.setFont(font);
     g.drawString("Java 5",90,100);
   }

   public void actionPerformed(ActionEvent e)
   {
      if (e.getSource()==button1){
         setBackground(new Color((int)(Math.random()*256),
                                 (int)(Math.random()*256),
                                 (int)(Math.random()*256)));
      }

      if (e.getSource()==button2) {
         setForeground(new Color((int)(Math.random()*256),
                                 (int)(Math.random()*256),
                                 (int)(Math.random()*256)));
      }
   }
}
```

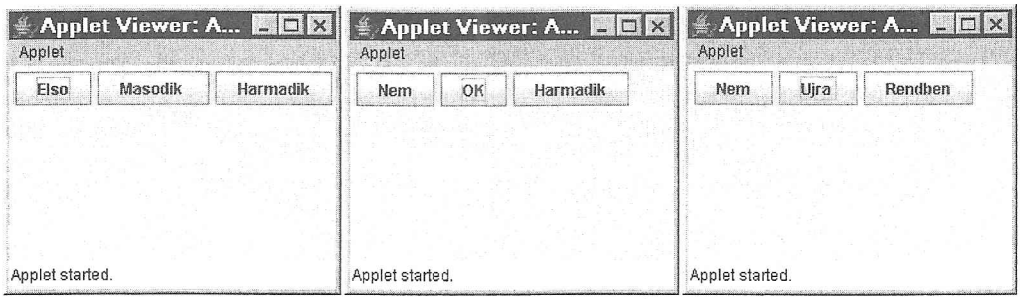
Az applet futásának eredménye:



Írjunk appletet, amely három nyomógomb megnyomásakor megváltoztatja a nyomógombok feliratát! (*Swing\Button3*)

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

public class Applet1 extends JApplet implements ActionListener
{
    JButton button1 = new JButton("Elso");
    JButton button2 = new JButton("Masodik ");
    JButton button3 = new JButton("Harmadik");
    JPanel mezo = new JPanel();

    public void init(){
       setSize(250,150);
       mezo.setLayout(new FlowLayout(FlowLayout.LEFT));
       mezo.add(button1);
       mezo.add(button2);
       mezo.add(button3);
       setContentPane(mezo);
       button1.addActionListener(this);
       button2.addActionListener(this);
       button3.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e)
    {
       if (e.getSource()==button2 && button2.getText()=="OK")
             button2.setText("Ujra ");
       if (e.getSource()==button1)
             button1.setText("Nem");
       if (e.getSource()==button2 && button2.getText()=="Masodik ")
             button2.setText("OK");
       if (e.getSource()==button3)
             button3.setText("Rendben");
    }
}
```

Az applet futásának eredményei:

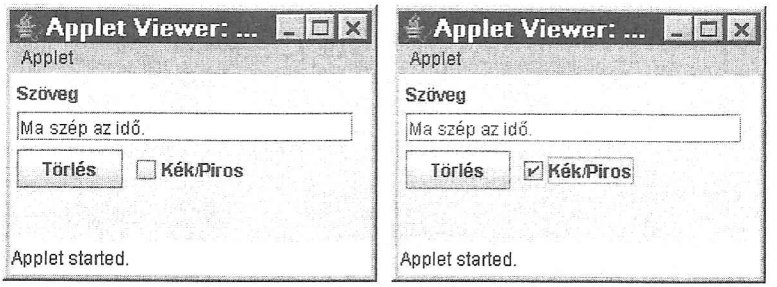


Tervezzünk appletet, amelyben egy szövegmezőbe írt szöveget a *Törlés* nyomógombbal (*JButton*) törli, és jelölőnégyzet (*JCheckBox*) segítségével a szöveg színét kékre, illetve pirosra változtatja! (*Swing\Check*)

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

public class Applet1 extends JApplet
                     implements ActionListener,ItemListener
{
   JLabel fejlec = new JLabel("Szöveg");
   JButton button1 = new JButton("Törles");
   JTextField szoveg = new JTextField(20);
   JCheckBox szin = new JCheckBox("Kék/Piros",false);
   JPanel mezo = new JPanel();

   public void init() {
     setSize(250,250);
     mezo.setLayout(new FlowLayout(FlowLayout.LEFT));
     szin.setSelected(false);
     szoveg.setForeground(Color.blue);

     mezo.add(fejlec);
     mezo.add(szoveg);
     mezo.add(button1);
     mezo.add(szin);
     setContentPane(mezo);
     button1.addActionListener(this);
     szin.addItemListener(this);
   }

   public void actionPerformed(ActionEvent e) {
     if (e.getSource()==button1)
        szoveg.setText("");
   }
```

```
    public void itemStateChanged(ItemEvent e)  {
      if (e.getSource() == szin && e.getStateChange() == e.SELECTED)
            szoveg.setForeground(Color.red);
      else
            szoveg.setForeground(Color.blue);
    }
}
```

Az applet futásának eredményei:



Készítsünk appletet, amely két adaton elvégzi a választógombokkal (*Összead, Kivon*) kijelölt műveletet! A *Töröl* nyomógomb segítségével a szövegmezők tartalma törölhető. (*Swing\RadioButton1*)

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

public class Applet1 extends JApplet
                    implements ActionListener, ItemListener
{
   JLabel label1 = new JLabel("            1.adat");
   JLabel label2 = new JLabel("            2.adat");
   JLabel label3 = new JLabel("Eredmény");
   JTextField edit1 = new JTextField("0",10);
   JTextField edit2 = new JTextField("0",10);
   JTextField edit3 = new JTextField("0",10);
   ButtonGroup muvelet = new ButtonGroup();
   JRadioButton osszead = new JRadioButton("Összead",false);
   JRadioButton kivon = new JRadioButton("Kivon",false);
   JButton button2 = new JButton(" TÖRÖL");
   JPanel mezo = new JPanel();

   public void init()
   {
      setSize(210,200);
      mezo.setLayout(new FlowLayout(FlowLayout.RIGHT));
      muvelet.add(osszead);
      muvelet.add(kivon);
      mezo.add(label1);
      mezo.add(edit1);
      mezo.add(label2);
```

```
        mezo.add(edit2);
        mezo.add(label3);
        mezo.add(edit3);
        mezo.add(osszead);
        mezo.add(kivon);
        mezo.add(button2);
        setContentPane(mezo);
        setVisible(true);
        button2.addActionListener(this);
        osszead.addItemListener(this);
        kivon.addItemListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource()==button2) {
            edit1.setText("0");
            edit2.setText("0");
            edit3.setText("0");
            osszead.setSelected(false);
            kivon.setSelected(false);
        }
    }

    public void itemStateChanged(ItemEvent e) {
        double ered;
        double adat1 = new Double(edit1.getText()).doubleValue();
        double adat2 = new Double(edit2.getText()).doubleValue();
        if (e.getSource() == osszead && e.getStateChange()==e.SELECTED){
          ered = adat1 + adat2;
          edit3.setText(""+ ered);
        }
        if (e.getSource() == kivon && e.getStateChange()==e.SELECTED) {
          ered = adat1-adat2;
          edit3.setText(""+ ered);
        }
    }
}
```

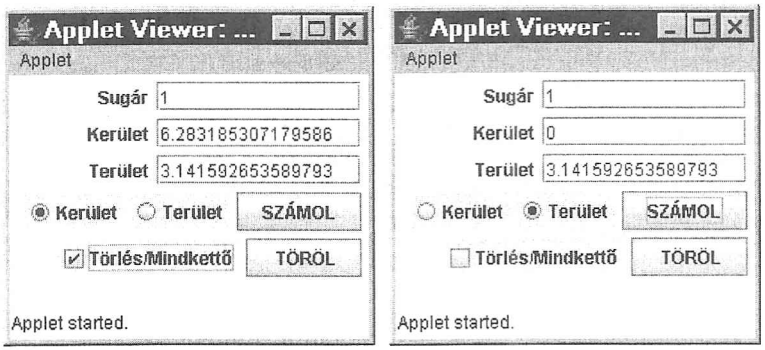
Az applet futásának eredményei:

Fejlesszünk kisalkalmazást, amelyben a kör területének és kerületének számításához választógombot, egyszerre való számításához pedig jelölőnégyzetet használhatunk! (*Swing\RadioButton2*)

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

public class Applet1 extends Applet
                    implements ActionListener, ItemListener
{
   JLabel label1 = new JLabel("              Sugár");
   JLabel label2 = new JLabel("            Kerület");
   JLabel label3 = new JLabel("            Terület");
   JTextField edit1 = new JTextField("1",12);
   JTextField edit2 = new JTextField("0",12);
   JTextField edit3 = new JTextField("0",12);
   JCheckBox mind = new JCheckBox("Törlés/Mindkettő");
   JRadioButton kerulet = new JRadioButton("Kerület");
   JRadioButton terulet = new JRadioButton("Terület");
   ButtonGroup csoport = new ButtonGroup();
   JButton button1 = new JButton("SZÁMOL");
   JButton button2 = new JButton(" TÖRÖL");
   JPanel p1 = new JPanel();

   public void init(){
      setSize(240,200);
      setLayout(new FlowLayout(FlowLayout.RIGHT));
      kerulet.setSelected(true);
      csoport.add(kerulet);
      csoport.add(terulet);
      this.add(label1);
      this.add(edit1);
      this.add(label2);
      this.add(edit2);
      this.add(label3);
      this.add(edit3);
      this.add(kerulet);
      this.add(terulet);
      this.add(button1);
      this.add(mind);
      this.add(button2);

      button1.addActionListener(this);
      button2.addActionListener(this);
      kerulet.addItemListener(this);
      terulet.addItemListener(this);
      mind.addItemListener(this);
   }
```

```
   public void actionPerformed(ActionEvent e)
   {
      double ter, ker;

      // a SZÁMOL gomb megnyomásakor
      if (e.getSource()==button1) {
         double r =new Double(edit1.getText()).doubleValue();
         if (kerulet.isSelected()) {        // Kerület
             ker = 2 * r * Math.PI;
             edit2.setText(""+ ker);
         }

         if(terulet.isSelected()) {         // Terület
             ter = Math.pow( r, 2) * Math.PI;
             edit3.setText(""+ ter);
         }
      }

      // a TÖRÖL gomb megnyomásakor
      if (e.getSource()==button2) {
         edit1.setText("0");
         edit2.setText("0");
         edit3.setText("0");
      }
   }

   public void itemStateChanged(ItemEvent e)
   {
      double ter, ker;
      double r = new Double(edit1.getText()).doubleValue();

      // a választógombok váltásakor töröljük az eredmény mezőket
      if (e.getSource() == kerulet && e.getStateChange()==e.SELECTED) {
        edit2.setText("0");
        edit3.setText("0");
      }

      if (e.getSource() == terulet && e.getStateChange()==e.SELECTED) {
        edit2.setText("0");
        edit3.setText("0");
      }

      // a jelölőnégyzet bejelölésekor
      if (e.getSource() == mind && e.getStateChange()==e.SELECTED
           && r > 0) {
        ker = 2 * r * Math.PI;
        edit2.setText(""+ ker);
        ter = Math.pow(r, 2) * Math.PI;
        edit3.setText(""+ ter);
      }
      else {
        edit2.setText("0");
        edit3.setText("0");
      }
   }
}
```

Az applet futásának eredményei:

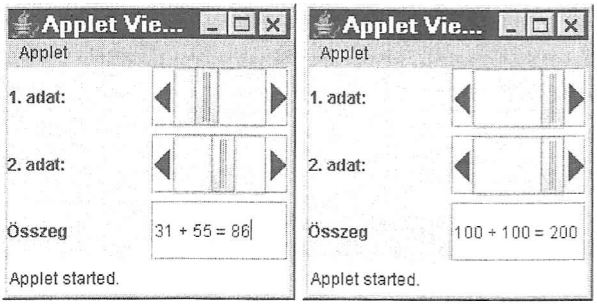


Készítsünk appletet, amelyben két, 1-100 közötti határral beállított görgetősávon (*JscrollBar*) tologatva a csúszkát, a csúszkák pozícióinak megfelelő értékek összege egy szövegmezőbe íródik! (*Swing\Gorgeto*)

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

public class Applet1 extends JApplet implements AdjustmentListener
{
   JLabel label1 = new JLabel("1. adat:");
   JLabel label2 = new JLabel("2. adat:");
   JLabel label3 = new JLabel("Összeg");
   JTextField ered = new JTextField(20);
   JScrollBar szam1 = new JScrollBar(Scrollbar.HORIZONTAL, 0, 0, 0, 100);
   JScrollBar szam2 = new JScrollBar(Scrollbar.HORIZONTAL, 0, 0, 0, 100);
   JButton button2 = new JButton("Összeg");
   JPanel mezo = new JPanel();

   public void init() {
      setSize(200,140);
      mezo.setLayout(new GridLayout(3,2,5,5));
      mezo.add(label1);
      mezo.add(szam1);
      mezo.add(label2);
      mezo.add(szam2);
      mezo.add(label3);
      mezo.add(ered);
      setContentPane(mezo);
      setVisible(true);
      szam1.addAdjustmentListener(this);
      szam2.addAdjustmentListener(this);
   }
```

```
    public void adjustmentValueChanged(AdjustmentEvent e) {
        if(e.getSource() == szam1) { // egyik görgető
            ered.setText(String.valueOf(szam1.getValue()) + " + "
                    + String.valueOf(szam2.getValue()) + " = "
                    + String.valueOf(szam1.getValue()
                    + szam2.getValue()));
        }
        if(e.getSource() == szam2) { // másik görgető
            ered.setText(String.valueOf(szam1.getValue()) + " + "
                    + String.valueOf(szam2.getValue()) + " = "
                    + String.valueOf(szam1.getValue()
                    + szam2.getValue()));
        }
    }
}
```

Az applet futásának eredményei:

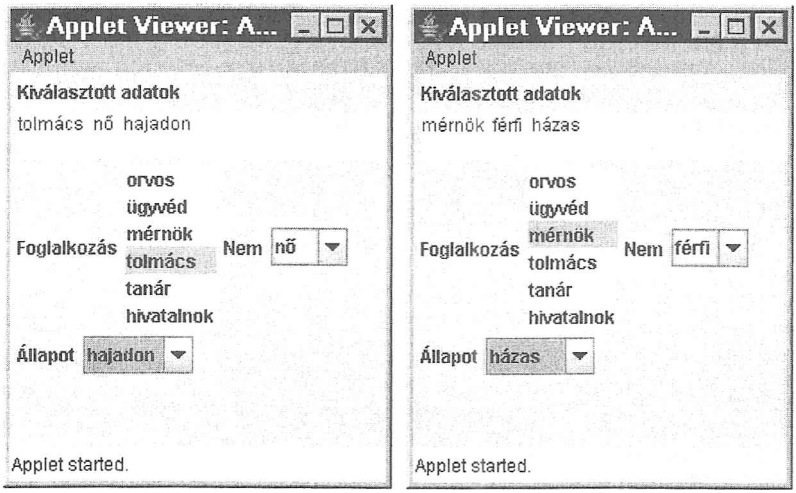


Készítsünk kisalkalmazást, amely egy listából és két kombinált listából kiválasztott tételeket szövegablakban szóközzel elválasztva jeleníti meg! (*Swing\Lista*)

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

public class Applet1 extends JApplet
                    implements ItemListener, ListSelectionListener
{
    JLabel felirat1 = new JLabel("Kiválasztott adatok");
    JLabel felirat2 = new JLabel("Foglalkozás");
    JLabel felirat3 = new JLabel("Nem");
    JLabel felirat4 = new JLabel("Állapot");
    JTextArea text = new JTextArea(2,30);
    JList foglalkozas = new JList();
    JComboBox nem = new JComboBox();
    JComboBox allapot = new JComboBox();
    JPanel mezo = new JPanel();
```

```
    public void init() {
        String [] ltetel =
             {"orvos","ügyvéd","mérnök","tolmács","tanár","hivatalnok"};
        this.setSize(250,250);
        this.setBackground(Color.cyan);
        mezo.setLayout(new BorderLayout());
        mezo.setLayout(new FlowLayout(FlowLayout.LEFT));
        foglalkozas.setListData(ltetel);
        mezo.add(felirat1);
        mezo.add(text);
        mezo.add(felirat2);
        mezo.add(foglalkozas);
        nem.addItem(" - ");
        nem.addItem("férfi");
        nem.addItem("nő");
        mezo.add(felirat3);
        mezo.add(nem);
        allapot.addItem(" - ");
        allapot.addItem("nőtlen");
        allapot.addItem("hajadon");
        allapot.addItem("házas");
        allapot.addItem("özvegy");
        mezo.add(felirat4);
        mezo.add(allapot);
        setContentPane(mezo);
        setVisible(true);
        foglalkozas.addListSelectionListener(this);
        nem.addItemListener(this);
        allapot.addItemListener(this);
    }

    public void itemStateChanged(ItemEvent e) {
        if (e.getSource() == nem  &&
            e.getStateChange() == ItemEvent.SELECTED) {
                text.append("  ");
                text.append( (String) e.getItem());
        }
        if (e.getSource() == allapot  &&
            e.getStateChange() == ItemEvent.SELECTED) {
                text.append("  ");
                 text.append( (String) e.getItem());
        }
    }

    public void valueChanged(ListSelectionEvent e) {
        if (e.getSource() == foglalkozas) {
                text.setText(foglalkozas.getSelectedValue().toString());
                allapot.setSelectedIndex(0);
                nem.setSelectedIndex(0);
        }
    }
}
```

Az applet futásának eredményei:

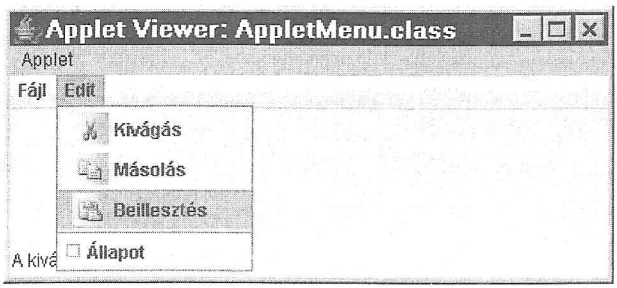


Tervezzünk appletet, amely két főmenüt és almenüket tartalmaz! (*Swing\Menu1*)

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

public class AppletMenu extends JApplet implements ActionListener
{
    public void init() {
       final String msg = "A kiválasztott menüpont: ";

       // A menürendszer felépítése
       JMenuBar jmenubar = new JMenuBar();

       // A Fájl menü
       JMenu jmenu1 = new JMenu("Fájl");
       JMenuItem jmenuitem1 = new JMenuItem("Új..."),
                 jmenuitem2 = new JMenuItem("Megnyitás..."),
                 jmenuitem3 = new JMenuItem("Mentés...");

       // ALT+X gyorsítóbillentyű a Kilépés menüponthoz
       jmenuitem3.setMnemonic(KeyEvent.VK_S);
       KeyStroke keystroke = KeyStroke.getKeyStroke
                 (KeyEvent.VK_S, Event.ALT_MASK);
       jmenuitem3.setAccelerator(keystroke);

       jmenu1.add(jmenuitem1);
       jmenu1.add(jmenuitem2);
       jmenu1.addSeparator();
       jmenu1.add(jmenuitem3);
```

```
        jmenuitem1.setActionCommand(msg + jmenuitem1.getText());
        jmenuitem2.setActionCommand(msg + jmenuitem2.getText());
        jmenuitem3.setActionCommand(msg + jmenuitem2.getText());

        jmenuitem1.addActionListener(this);
        jmenuitem2.addActionListener(this);
        jmenuitem3.addActionListener(this);

        // Az Edit menü képes menüpontokkal
        ImageIcon iconkivagas =
              new ImageIcon(getClass().getResource("Kivagas.jpg"));
        ImageIcon iconmasolas =
              new ImageIcon(getClass().getResource("Masolas.jpg"));
        ImageIcon iconbeillesztes =
              new ImageIcon(getClass().getResource("Beillesztes.jpg"));

        JMenu jmenu2 = new JMenu("Edit");
        JMenuItem jmenuitem4 = new JMenuItem("Kivágás",iconkivagas),
                  jmenuitem5 = new JMenuItem("Másolás", iconmasolas),
                  jmenuitem6 = new JMenuItem("Beillesztés",
                                                  iconbeillesztes);

        jmenu2.add(jmenuitem4);
        jmenu2.add(jmenuitem5);
        jmenu2.add(jmenuitem6);
        jmenu2.addSeparator();  // menüpontok elválasztása

        // jelölőnégyzet a menüpontban
        JCheckBoxMenuItem jmenuitem7 =
              new JCheckBoxMenuItem("Állapot", null);
        jmenu2.add(jmenuitem7);

        jmenuitem4.setActionCommand(msg + jmenuitem4.getText());
        jmenuitem5.setActionCommand(msg + jmenuitem5.getText());
        jmenuitem6.setActionCommand(msg + jmenuitem6.getText());

        jmenuitem4.addActionListener(this);
        jmenuitem5.addActionListener(this);
        jmenuitem6.addActionListener(this);
        jmenuitem7.addActionListener(this);

        jmenubar.add(jmenu1);
        jmenubar.add(jmenu2);
        setJMenuBar(jmenubar);
    }

    public void actionPerformed(ActionEvent e)
    {
      JMenuItem jmenuitem = (JMenuItem)e.getSource();
      // A menüválasztás megjelenítése az állapotsorban
      if (jmenuitem.getText().equals("Állapot"))
            showStatus("Állapot: " +
              ((JCheckBoxMenuItem)jmenuitem).getState());
      else
            showStatus(jmenuitem.getActionCommand());
    }
}
```

Az applet futásának eredménye:

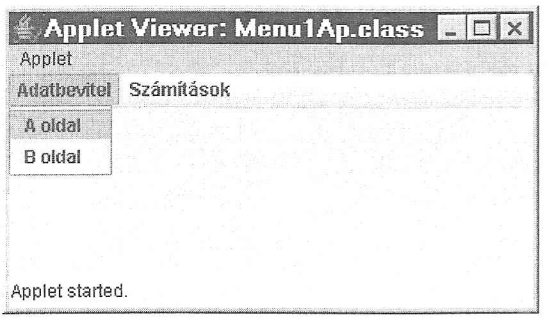


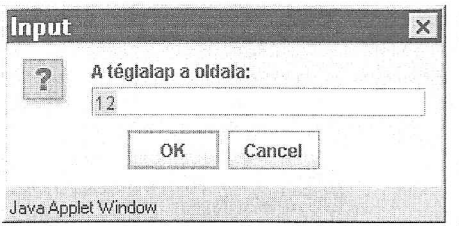
Fejlesszünk appletet, amely menüvezérelve számítja ki a téglalap területét és kerületét! A téglalap oldalhosszát *Input* ablakból olvassa be, az eredményt pedig *Message* ablakban jeleníti meg. (*Swing\Menu2*)

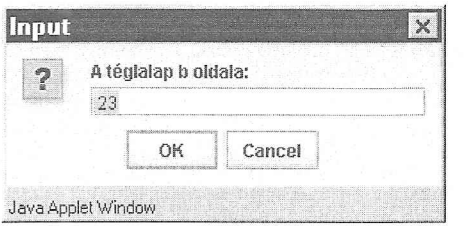
```
import java.awt.event.*;
import javax.swing.*;

public class Menu1Ap extends JApplet implements ActionListener
{
   // a menüelemek létrehozása
   JMenuBar menusor = new JMenuBar();
   JMenu menu1 = new JMenu("Adatbevitel");
   JMenuItem almenu1 = new JMenuItem("A oldal");
   JMenuItem almenu2 = new JMenuItem("B oldal");
   JMenu menu2 = new JMenu("Számítások");
   JMenuItem almenu3 = new JMenuItem("Kerület");
   JMenuItem almenu4 = new JMenuItem("Terület");
   int aoldal=10, boldal=10;

   public void init() {
       // a menü felépítése
       menu1.add(almenu1).addActionListener(this);
       menu1.add(almenu2).addActionListener(this);
       menusor.add(menu1);
       menu2.add(almenu3).addActionListener(this);
       menu2.add(almenu4).addActionListener(this);
       menusor.add(menu2);
       setJMenuBar(menusor);
       setSize(350,200);
   }

   public void actionPerformed(ActionEvent e) {
       showStatus(e.getActionCommand());
       if (e.getActionCommand()=="A oldal")
         aoldal = Integer.parseInt(JOptionPane.showInputDialog(null,
                  "A téglalap a oldala:","12"));
       if (e.getActionCommand()=="B oldal")
         boldal = Integer.parseInt(JOptionPane.showInputDialog(null,
                  "A téglalap b oldala:","23"));
```

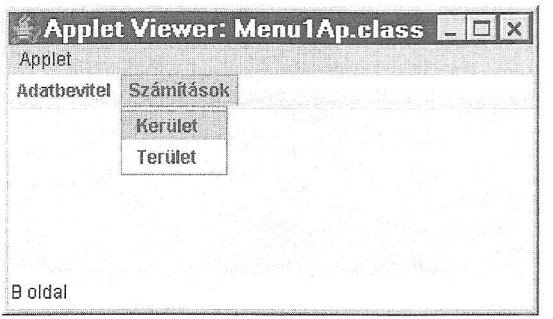
```
        if (e.getActionCommand()=="Kerület")
             JOptionPane.showMessageDialog(null,
                "A téglalap kerülete: "+2*(aoldal+boldal));
        if (e.getActionCommand()=="Terület")
             JOptionPane.showMessageDialog(null,
                "A téglalap területe: "+(aoldal*boldal));
    }
}
```

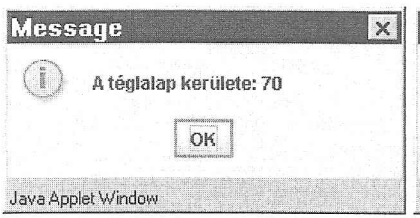
Az applet futásának eredményei:

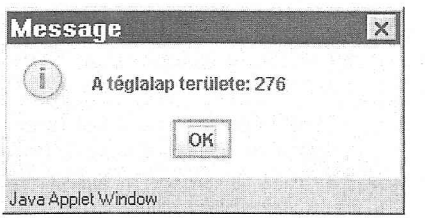








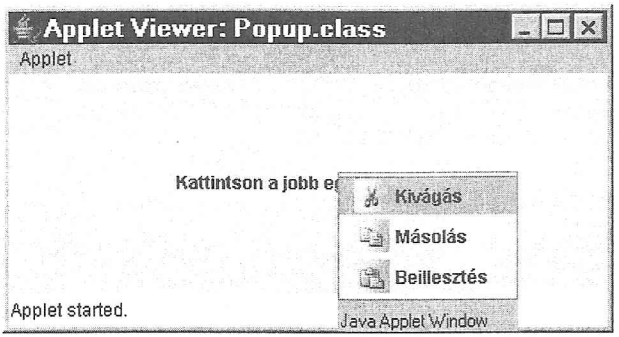




Készítsünk appletet, amely az egér jobb gombjával kattintva felbukkanó menüt jelenít meg! (*Swing\PopupMenu*)

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class Popup extends JApplet
                   implements MouseListener, ActionListener
{
    JLabel jlabel = new JLabel("Kattintson a jobb egérgombbal!",
                    JLabel.CENTER);
    JPopupMenu jpopupmenu = new JPopupMenu();

    public void init()
    {
       Container contentPane = getContentPane();
      // a menű felépítése, képes menüpontokkal
      ImageIcon iconkivagas =
            new ImageIcon(getClass().getResource("Kivagas.jpg"));
      ImageIcon iconmasolas =
            new ImageIcon(getClass().getResource("Masolas.jpg"));
      ImageIcon iconbeillesztes =
            new ImageIcon(getClass().getResource("Beillesztes.jpg"));

      JMenuItem jmenuitem1 = new JMenuItem("Kivágás",iconkivagas),
           jmenuitem2 = new JMenuItem("Másolás", iconmasolas),
           jmenuitem3 = new JMenuItem("Beillesztés", iconbeillesztes);

       jpopupmenu.add(jmenuitem1);
       jpopupmenu.add(jmenuitem2);
       jpopupmenu.add(jmenuitem3);

       jmenuitem1.addActionListener(this);
       jmenuitem2.addActionListener(this);
       jmenuitem3.addActionListener(this);

       jlabel.addMouseListener(this);
       contentPane.add(jlabel);
    }

    // a jobb oldali egérgombbal jeleníthető meg
    public void mousePressed (MouseEvent e) {
        if((e.getModifiers() & InputEvent.BUTTON3_MASK) ==
            InputEvent.BUTTON3_MASK)
        jpopupmenu.show(jlabel, e.getX(), e.getY());  // megjelenik
    }

    public void mouseClicked(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}

    public void actionPerformed(ActionEvent e) {
       JMenuItem jmenuitem = (JMenuItem)e.getSource();
       showStatus(jmenuitem.getActionCommand());      // kijelzés
    }
}
```

Az applet futásának eredménye:

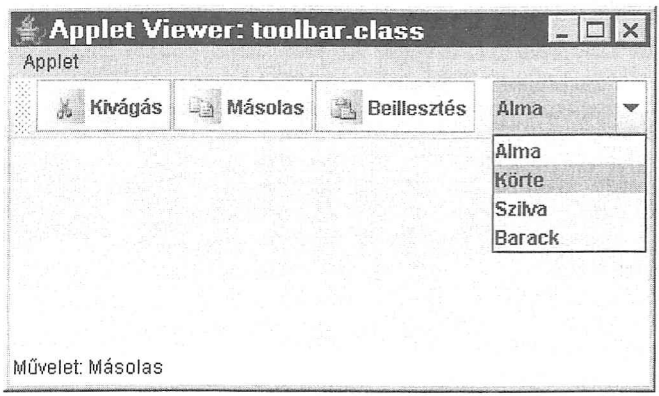


Tervezzünk kisalkalmazást, amely eszközsort helyez az applet ablakának felső sorába! (*Swing\Toolbar*)

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class toolbar extends JApplet
             implements ActionListener, ItemListener
{
    // az eszközsor 3 képes gombból és egy kombinált listából áll
    JButton jbutton1 = new JButton("Kivágás",
            new ImageIcon(getClass().getResource("Kivagas.jpg")));
    JButton jbutton2 = new JButton("Másolas",
            new ImageIcon(getClass().getResource("Masolas.jpg")));
    JButton jbutton3 = new JButton("Beillesztés",
            new ImageIcon(getClass().getResource("Beillesztes.jpg")));
    JComboBox jcombobox = new JComboBox();

    public void init() {
        Container contentPane = getContentPane();
        JToolBar jtoolbar = new JToolBar();

        jbutton1.addActionListener(this);
        jbutton2.addActionListener(this);
        jbutton3.addActionListener(this);

        jcombobox.addItem("Alma");
        jcombobox.addItem("Körte");
        jcombobox.addItem("Szilva");
        jcombobox.addItem("Barack");
        jcombobox.addItemListener(this);

        jtoolbar.add(jbutton1);
        jtoolbar.add(jbutton2);
        jtoolbar.add(jbutton3);
        jtoolbar.addSeparator();
        jtoolbar.add(jcombobox);
        contentPane.add(jtoolbar, BorderLayout.NORTH);
    }
```

```
    public void actionPerformed(ActionEvent e) { // gombnyomás
        JButton btn = (JButton)e.getSource();
        showStatus("Művelet: " + btn.getText());
    }

    public void itemStateChanged(ItemEvent e) { // listaválasztás
        String outString = "";
        if(e.getStateChange() == ItemEvent.SELECTED)
        outString += "Kiválasztott elem: " + (String)e.getItem();
        showStatus(outString);
    }
}
```

Az applet futásának eredménye:



12.5 Grafika alkalmazása az appletekben

A kisalkalmazásokban a grafikus megjelenítés a ***paint()*** metódus feladata. Ha valami változás következtében újra szeretnénk rajzoltatni az applet felületét, akkor a ***repaint()*** metódus hívásával aktivizálhatjuk a ***paint()*** metódust, melynek működést megelőzi az ***update()*** metódus lefutása:

```
public void update(Graphics g)
{
      // . . .
}
```

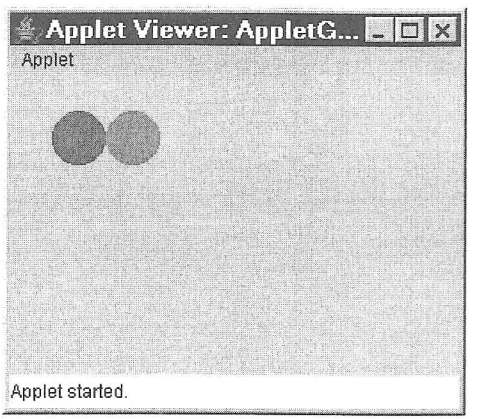
A grafika alkalmazását szintén példák sorával szemléltetjük.

Készítsünk appletet, amely a színes háttérre, véletlenszerű középpontban és sugárral egymás mellé két különböző színnel befestett kört rajzol! (*Grafika\AppletGrafika*)

```
import java.awt.*;
import java.applet.*;
```

```
public class AppletGrafikaextends Applet
{
  class Korok {
      int kx,ky,sugar;
  }

  Color szin1, szin2;
  Korok kr;

  public AppletGrafika () {
      kr = new Korok();
  }

  public void init() {
     this.setBackground(new Color((int)(Math.random()*256),
                                  (int)(Math.random()*256),
                                  (int)(Math.random()*256)));
     szin1 = new Color((int)(Math.random()*256),
                       (int)(Math.random()*256),
                       (int)(Math.random()*256));
     szin2 = new Color((int)(Math.random()*256),
                       (int)(Math.random()*256),
                       (int)(Math.random()*256));
  }

  public void start() {
    kr.kx = (int)(100*Math.random()+20 );
    kr.ky = (int)(100*Math.random()+20);
    kr.sugar = (int)(50*Math.random()+20);
  }

  public void paint(Graphics g) {
    g.setColor(szin1);
    g.fillOval(kr.kx,kr.ky,kr.sugar,kr.sugar);
    g.setColor(szin2);
    g.fillOval(kr.kx+kr.sugar,kr.ky,kr.sugar,kr.sugar);
  }
}
```

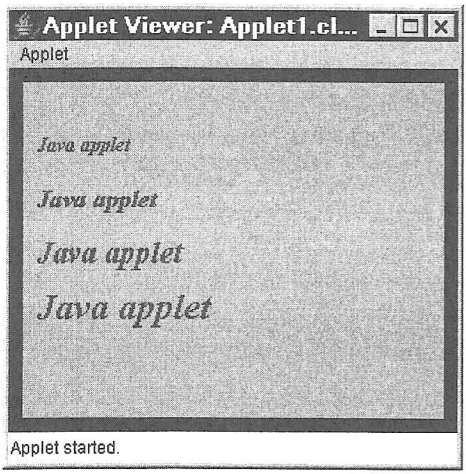
Az applet futásának eredménye:



Készítsünk kisalkalmazást, amely szöveget jelenít meg négyféle méretben, színes háttérrel, és színes téglalap alakú kerettel! (*Grafika\Rajz0*)

```
import java.awt.*;
import java.applet.*;

public class Applet1 extends Applet
{
   public void init(){
       setBackground(Color.blue);
       resize(320,260);
   }

   public void paint(Graphics g) {
       Rectangle r=getBounds();
       g.setColor(Color.cyan);
       g.fillRect(10,10,r.width-20,r.height-20);
       g.setColor(Color.red);
       for (int i = 1; i<5; i++) {
            Font f=new Font("Times New Roman",
                            Font.BOLD+Font.ITALIC, 10+i*4);
            g.setFont(f);
            g.drawString("Java applet", 20, 20+i*40);
       }
   }
}
```

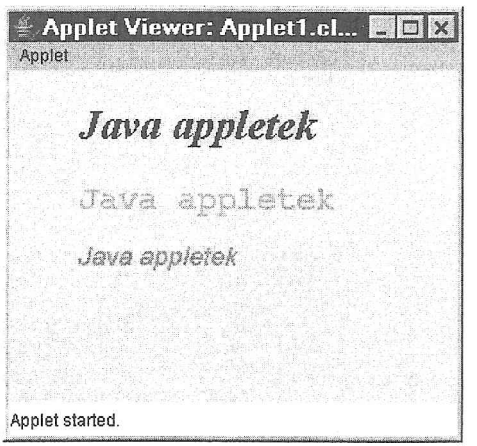
Az applet futásának eredménye:



Készítsünk appletet, amely a ***drawString()*** metódussal különböző típusú és méretű szöveget ír egy sárgára festett téglalapba! (*Grafika\Rajz1*)

```
import java.awt.*;
import java.applet.*;

public class Applet1 extends Applet
{
   public void init() {
      resize(320,240);
      setBackground(Color.white);
   }

   public void paint(Graphics g)
   {
      Rectangle r=getBounds();
      g.setColor(Color.yellow);
      g.fillRect(0,0,r.width-1,r.height-1);
      Font f=new Font("Times New Roman",Font.BOLD+Font.ITALIC, 30);
      g.setFont(f);
      g.setColor(Color.blue);
      g.drawString("Java appletek", 50, 50);

      Font f1=new Font("Courier New",Font.BOLD, 24);
      g.setFont(f1);
      g.setColor(Color.cyan);
      g.drawString("Java appletek ", 50, 100);

      Font f2=new Font("Arial",Font.ITALIC, 18);
      g.setFont(f2);
      g.setColor(Color.green);
      g.drawString("Java appletek ", 50, 140);
   }
}
```

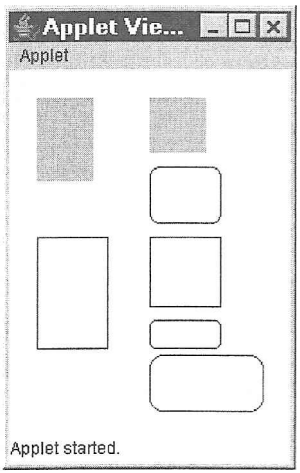
Az applet futásának eredménye:



Tervezzünk appletet, amely befestett téglalapot, téglalap alakú keretet és lekerekített téglalapot rajzol! (*Grafika\Rajz2*)

```
import java.awt.*;
import java.applet.*;

public class Applet1 extends Applet
{
   public void init()
   {
      resize(200,260);
      setBackground(Color.white);
   }

   public void paint(Graphics g)
   {
      Rectangle r=getBounds();
      g.setColor(Color.cyan);
      g.fillRect(20,20,40,60);
      g.fillRect(100,20,40,40);
      g.setColor(Color.blue);
      g.drawRect(20,120,50,80);
      g.drawRect(100,120,50,50);
      g.drawRoundRect(100,180,50,20,10,10);
      g.drawRoundRect(100,70,50,40,15,15);
      g.drawRoundRect(100,205,80,40,20,20);
   }
}
```

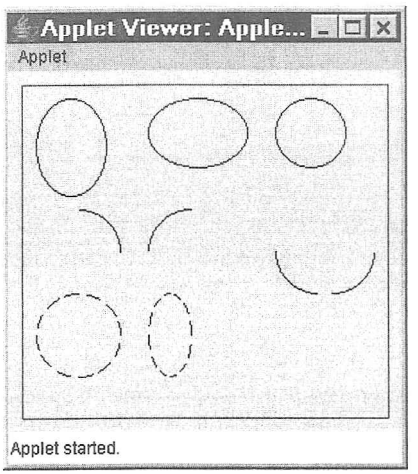
Az applet futásának eredménye:



Írjunk appletet, amely ellipszist, kört, ellipszisívet, körívet valamint szaggatott kört és ellipszist rajzol! (*Rajz3*)

```
import java.awt.*;
import java.applet.*;

public class Applet1 extends Applet
{
   public void init() {
       resize(280,260);
       setBackground(Color.yellow);
   }

   public void paint(Graphics g) {
      Rectangle r=getBounds();
      g.setColor(Color.red);
      g.drawRect(10,10,r.width-20,r.height-20);
      g.setColor(Color.blue);

      g.drawOval(20,20,50,70);
      g.drawOval(100,20,70,50);
      g.drawOval(190,20,50,50);

      g.drawArc(20,100,60,60,0,90);
      g.drawArc(100,100,60,60,90,90);
      g.drawArc(190,100,60,60,180,90);
      g.drawArc(200,100,60,60,270,90);

      for( int i = 0; i<360; i+=30)
         g.drawArc(20,160,60,60,i,20);
      for( int i = 0; i<360; i+=30)
         g.drawArc(100,160,30,60,i,20);
   }
}
```

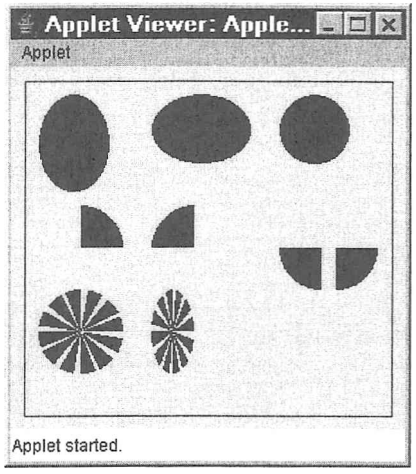
Az applet futásának eredménye:



Készítsünk appletet, amely befestett ellipszist, kört, ellipszisívet, körívet valamint szaggatott kört és ellipszist rajzol! (*Grafika\Rajz4*)

```
import java.awt.*;
import java.applet.*;

public class Applet1 extends Applet
{
   public void init() {
      resize(280,260);
      setBackground(Color.yellow);
   }

   public void paint(Graphics g) {
      Rectangle r=getBounds();
      g.setColor(Color.red);
      g.drawRect(10,10,r.width-20,r.height-20);
      g.setColor(Color.blue);

      g.fillOval(20,20,50,70);
      g.fillOval(100,20,70,50);
      g.fillOval(190,20,50,50);

      g.fillArc(20,100,60,60,0,90);
      g.fillArc(100,100,60,60,90,90);
      g.fillArc(190,100,60,60,180,90);
      g.fillArc(200,100,60,60,270,90);

      for( int i = 0; i<360; i+=30)
         g.fillArc(20,160,60,60,i,20);

      for( int i = 0; i<360; i+=30)
         g.fillArc(100,160,30,60,i,20);
   }
}
```

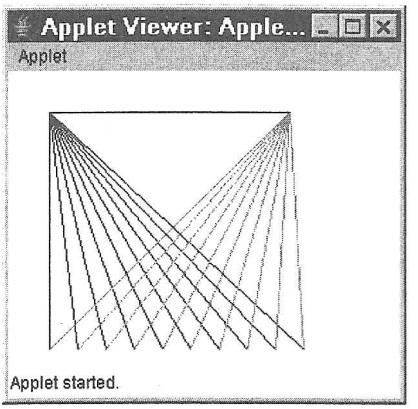
Az applet futásának eredménye:



Írjunk kisalkalmazást, amely színes vonalakat rajzol! (*Grafika\Rajz5*)

```
import java.awt.*;
import java.applet.*;

public class Applet1 extends Applet
{
   public void init() {
       resize(280,220);
       this.setBackground(Color.white);
   }

   public void paint(Graphics g) {
      Rectangle r=getBounds();
      g.drawLine(30,30, 200,30);
      g.setColor(Color.blue);
      for( int i = 0; i<10; i++)
          g.drawLine(30,30,30+i*20,200);
      g.setColor(Color.green);
      for( int i = 0; i<10; i++)
          g.drawLine(200,30,30+i*20,200);
   }
}
```

Az applet futásának eredménye:

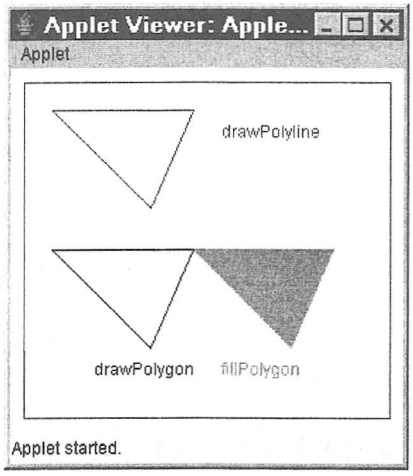


Készítsünk appletet, amely háromszöget rajzol egyenesekből, poligonnal és befestett poligonnal! (*Grafika\Rajz6*)

```
import java.awt.*;
import java.applet.*;

public class Applet1 extends Applet
{
    int[] x1 = {30,100,130,30};
    int[] y1 = {30,100,30,30};
```

```
    int[] x2 = {130,200,230,130};
    int[] y2 = {130,200,130,130};
    int[] x3 = {30,100,130,30};
    int[] y3 = {130,200,130,130};
    public void init() {
       resize(280,260);
       setBackground(Color.white);
    }
    public void paint(Graphics g) {
      Rectangle r=getBounds();
      g.setColor(Color.red);
      g.drawRect(10,10,r.width-20,r.height-20);
      g.drawPolyline(x1,y1,4);
      g.drawString("drawPolyline", 150, 50);
      g.setColor(Color.green);
      g.fillPolygon(x2,y2,4);
      g.drawString("fillPolygon", 150, 220);
      g.setColor(Color.blue);
      g.drawPolygon(x3,y3,4);
      g.drawString("drawPolygon", 60, 220);
    }
}
```

Az applet futásának eredménye:

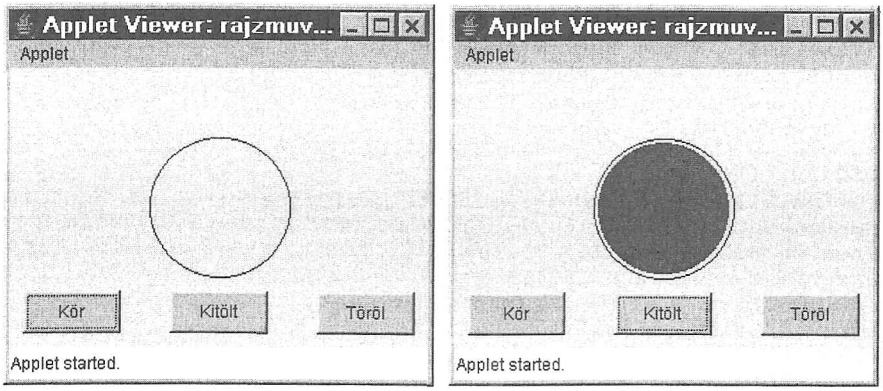


Tervezzünk appletet, amely rajzol egy kék színű kört, kitölti piros színnel, valamint letörli a rajzot! Oldjuk meg a feladatot három nyomógombhoz kapcsolva a műveleteket! (*Grafika\Rajzmuveletek*)

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
```

```
public class rajzmuveletek extends Applet implements ActionListener
{
  Button circleButton = new Button("Kör");
  Button fillButton   = new Button("Kitölt");
  Button resetButton  = new Button("Töröl");
  boolean kör=false, kitölt=false, töröl=false;
  final int sugar=50, dr=3;

  public void init(){
    setLayout(null);  // a gombok rögzített pozícióba kerülnek
    circleButton.setBounds(10,160,70,30);
    fillButton.setBounds(115,160,70,30);
    resetButton.setBounds(220,160,70,30);
    resize(300,200);
    setBackground(Color.yellow);
    add(circleButton);
    add(fillButton);
    add(resetButton);
    circleButton.addActionListener(this);
    fillButton.addActionListener(this);
    resetButton.addActionListener(this);
  }

  public void paint(Graphics g){
    if(töröl){
      g.setColor(getBackground());
      g.fillRect(0,0,300,200);
      töröl = kör = kitölt = false;
    }
    if(kör){
      g.setColor(Color.blue);
      g.drawOval(150-sugar,100-sugar, 2*sugar, 2*sugar);
    }
    if(kitölt){
      g.setColor(Color.red);
      g.fillOval(150-(sugar-dr),100-(sugar-dr),
                 2*(sugar-dr), 2*(sugar-dr));
    }
  }

  public void actionPerformed(ActionEvent e) {
    if (e.getSource()==circleButton)     // Kör gomb
      kör=true;
    else if (e.getSource()==fillButton) // Kitölt gomb
      kitölt=true;
    else
      töröl=true;                        // Töröl gomb
    repaint();
  }
}
```

Az applet futásának eredményei:



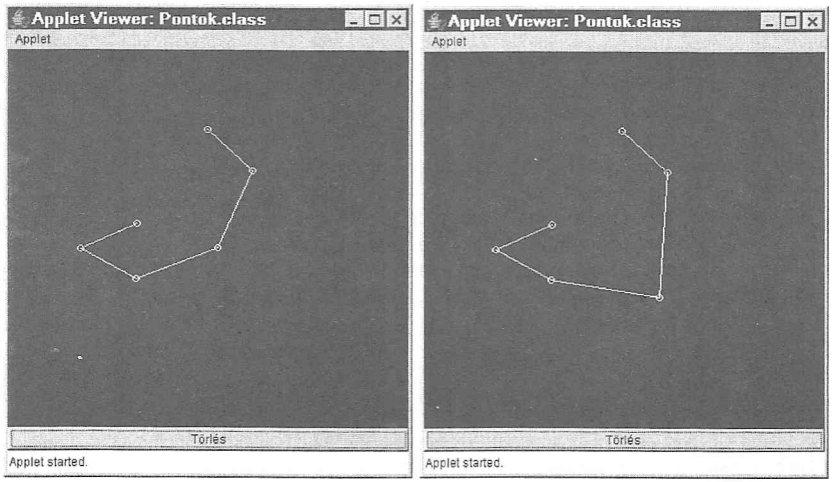
Fejlesszünk kisalkalmazást, amely az egét bal gombjának kattintásával egy pontláncot épít fel! A szakaszvégpontokra való bal kattintással a pontok áthelyezhetők, jobb egérgombbal kattintva pedig törölhetők. A teljes rajz pedig nyomógomb megnyomásával törölhető. (*Grafika\Pontok*)

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Pontok extends Applet implements MouseListener,
                    MouseMotionListener, ActionListener
{
  Vector<Point> pontok = new Vector<Point>();  // J2SE 5.0 kell
  Button torol = new Button("Törlés");
  int aktindex = -1;

  public void init() {
     addMouseListener(this);
     addMouseMotionListener(this);
     setBackground(Color.black);
     setLayout(new BorderLayout());
     add("South", torol);
     torol.addActionListener(this);
  }

  public void actionPerformed(ActionEvent e) {
     if (e.getSource()==torol) { // teljes törlés
         pontok.clear();
         repaint();
     }
  }
```

```
// a b pont bele esik-e az a pont 3 sugarú környezetébe
boolean beleTalalt(Point a, Point b) {
    if (a.x-3<=b.x && b.x<=a.x+3 && a.y-3<=b.y && b.y<=a.y+3)
        return true;
    else
        return false;
}

public void mouseClicked(MouseEvent e) {  // egérkattintás
    // a bal egérgombbal való kattintáskor új pontot veszünk fel
    if ((e.getModifiers() & e.BUTTON1_MASK) != 0) {
        pontok.add(e.getPoint());
        repaint();
     }
     else {
         // a másik egérgombbal való kattintáskor levesszük a
         // mutatott pontot, ha van ilyen pont
         Enumeration ve = pontok.elements();
         Point pe = e.getPoint();
         while (ve.hasMoreElements()) {
             Point pv = (Point) ve.nextElement();
             if (beleTalalt(pv, pe)) {
                 pontok.removeElement(pv);
                 repaint();
                 break; // egyszerre egy pontot
             }
         }
     }
}

// a bal gomb lenyomásakor megkeressük a közeli pont indexét
public void mousePressed(MouseEvent e) {
     if ((e.getModifiers() & e.BUTTON1_MASK) != 0) {
         Enumeration ve = pontok.elements();
         Point pe = e.getPoint();
         while (ve.hasMoreElements()) {
             Point pv = (Point)ve.nextElement();
             if (beleTalalt(pv, pe)) {
                 aktindex=pontok.indexOf(pv);
                 break;
             }
         }
     }
}

// a bal gomb felengedésekor töröljük az indexet
public void mouseReleased(MouseEvent e) {
     if ((e.getModifiers() & e.BUTTON1_MASK) != 0)
         aktindex = -1;
}

public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
public void mouseMoved(MouseEvent e) {}
```

```
    // ha a bal gombot lenyomva mozgatjuk az egeret, módosítjuk a
    // pont pozícióját
    public void mouseDragged(MouseEvent e) {
        if ((e.getModifiers() & e.BUTTON1_MASK) != 0 && aktindex>-1) {
            pontok.set(aktindex, e.getPoint());
            repaint();
        }
    }

    // a vonallánc kirajzolása
    public void paint(Graphics g) {
        if (pontok.size()==0) return; // még nincsenek pontok
        g.setColor(Color.yellow);
        Enumeration e = pontok.elements();
        Point p1 = (Point) e.nextElement();
        g.drawOval(p1.x-3, p1.y-3, 6, 6);;
        while (e.hasMoreElements()) {
            Point p2 = (Point) e.nextElement();
            g.drawOval(p2.x-3, p2.y-3, 6, 6);
            g.drawLine(p1.x, p1.y,p2.x,p2.y);
            p1 = p2;
        }
    }
}
```

Az applet futásának eredményei:

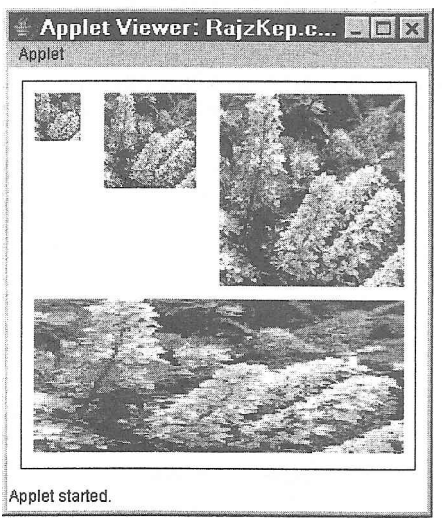


Tervezzünk appletet, amely betölt egy képet, majd különböző nagyításban megjeleníti azt! (*Grafika\RajzKep*)

```
import java.awt.*;
import java.applet.*;
```

```
public class RajzKep extends Applet
{
    Image kep;

    public void init() {
       resize(320,320);
       this.setBackground(Color.white);
       kep = getImage(getCodeBase(),"kepek/tavasz.jpg");
    }

    public void paint(Graphics g) {
       Rectangle r=getBounds();
       g.setColor(Color.blue);
       g.drawRect(10,10,r.width-20,r.height-20);

       int wkep = kep.getWidth(this);
       int hkep = kep.getHeight(this);
       int xpoz = 20, ypoz = 20;

       // 25 %
       g.drawImage(kep, xpoz, ypoz, wkep / 4, hkep / 4, this);

       // 50 %
       xpoz += (wkep / 4) + 18;
       g.drawImage(kep, xpoz , ypoz, wkep / 2, hkep / 2, this);

       // 100%
       xpoz += (wkep / 2) + 18;
       g.drawImage(kep, xpoz, ypoz, this);

       // 200% x, 80% y
       g.drawImage(kep, 20, hkep+30, wkep*2, (int)(hkep*0.8), this);
    }
}
```

Az applet futásának eredménye:

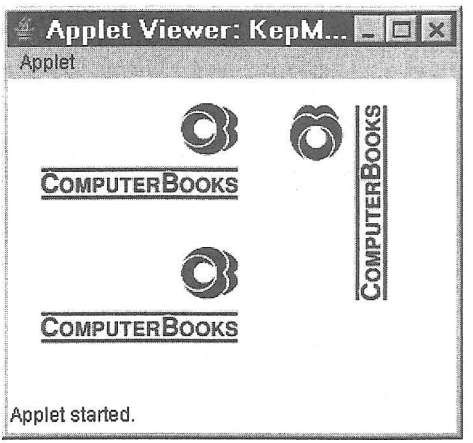


Tervezzünk kisalkalmazást, amely beolvas egy képet, elforgatja 90 fokkal, majd pedig megjeleníti! Az eredeti és a fekete-fehérré alakított kép is látható lesz. (*Grafika\Kepmuveletek*)

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

// Kép pixelenkénti feldolgozása
public class KepMuveletek extends Applet
{
    Image kep=null, kep90=null, kepbw=null;

    public void init () {
       resize(280,200);
       MediaTracker mt = new MediaTracker (this);
       kep = getImage (getDocumentBase(), "CBooks.jpg");
       mt.addImage (kep, 0);
       try {
          mt.waitForAll(); // várakozás a kép betöltéséig
          int wmeret       = kep.getWidth(this);
          int hmeret       = kep. getHeight(this);
          int keppontok[]  = new int [wmeret * hmeret];
          // Az eredeti kép képpontjainak lekérdezése
          PixelGrabber pg  = new PixelGrabber (kep, 0, 0, wmeret,
                                hmeret, keppontok, 0, wmeret);
          // Ha a képpontok előállítása sikeres volt
          if (pg.grabPixels() && ((pg.status() &
             ImageObserver.ALLBITS) !=0)) {
             // A 90-fokkal elforgatott kép
             kep90 = createImage (new MemoryImageSource (hmeret,
                wmeret,forgatas90(keppontok,wmeret,hmeret), 0,hmeret));
             // A fekete-fehér kép
             kepbw = createImage (new MemoryImageSource (wmeret,
                hmeret,feketefeher(keppontok,wmeret,hmeret),0,wmeret));
          }
       }
       catch (InterruptedException e) {
          e.printStackTrace();
       }
    }

    // a képek megjelenítése
    public void paint (Graphics g) {
       g.drawImage (kep, 20, 10, this);        // erdeti kep
       if (kep90 != null)
          g.drawImage (kep90, 170, 15, this); // 90-kal elforgatott kép
       if (kepbw != null)
          g.drawImage (kepbw, 20, 100, this); // fekete-fehér kép
    }
```

```
    // A kép elforgatása 90 fokkal
    private int[] forgatas90 (int keppontok[], int w, int h)
    {
       int forgKeppontok[] = null;
       if ((w * h) == keppontok.length) {
          forgKeppontok = new int [w * h];
          int forgIndex = 0;
          for (int x = w-1; x >= 0; x--)
              for (int y = 0; y < h; y++)
                  forgKeppontok[forgIndex++] = keppontok[y*w + x];
       }
       return forgKeppontok;
    }

    // A kép fekete-fehér változatának előállítása
    private int[] feketefeher (int keppontok[], int w, int h)
    {
       int ujKeppontok[] = null;
       if ((w * h) == keppontok.length) {
          ujKeppontok = new int [w * h];
          int ujIndex=0;
          for (int y = 0; y < h; y++)
              for (int x = 0; x < w; x++) {
                  Color sz = new Color(keppontok[y*w + x]);
                   int bw = (sz.getRed() + sz.getBlue() +
                             sz.getGreen())/3;
              sz = new Color(bw, bw, bw);
              ujKeppontok[ujIndex++] = sz.getRGB();
              }
       }
       return ujKeppontok;
    }
}
```

Az applet futásának eredménye:



12.6 Hangok

A Java már a legelső verziójától kezdve támogatja a (a tömörítetlen *midi*, *au*, *wave*, és *aiff* formátumú) hangállományok (*audio clips*) kisalkalmazásból való – akár egyidejű – lejátszását. A hang megszólaltatásához először az applet

```
public AudioClip getAudioClip(URL url)
public AudioClip getAudioClip(URL url, String név)
```

metódusai egyikével be kell töltenünk a hangfájlt, létrehozva az ***AudioClip*** interfész példányát. Ezt követően az ***AudioClip play()*** metódusával egyszer, a ***loop()*** metódusával pedig ismétlődve (a ***stop()*** hívásáig) játszhatjuk le a hangot.

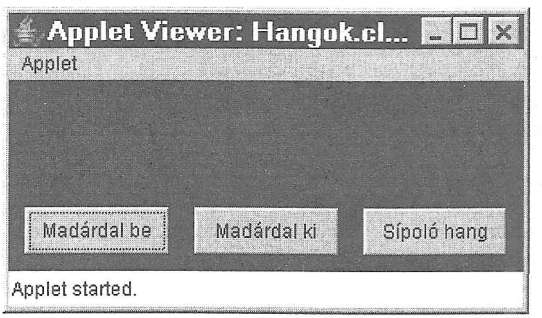
Megjegyezzük, hogy a ***Java Sound API*** általánosabb lehetőségeket is biztosít a hangállományok megszólaltatására, illetve előállítására.

Készítsünk appletet, amely kétféle hangállományt szólaltat meg, gombokkal vezérelve a lejátszást! (*Hangok|Hangok*)

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class Hangok extends Applet implements ActionListener
{
  Button beButton = new Button("Madárdal be");
  Button kiButton   = new Button("Madárdal ki");
  Button sipButton  = new Button("Sípoló hang");

  AudioClip hatterzene, mpjel;

  public void init() {
    hatterzene = getAudioClip(getCodeBase(),"madar.au");
    mpjel = getAudioClip(getCodeBase(), "sip.au");

    setLayout(null);
    beButton.setBounds(10,80,90,30);
    kiButton.setBounds(115,80,90,30);
    sipButton.setBounds(220,80,90,30);
    resize(320,120);
    setBackground(Color.blue);
    add(beButton);
    add(kiButton);
    add(sipButton);
    beButton.addActionListener(this);
    kiButton.addActionListener(this);
    sipButton.addActionListener(this);

    // Hanglejátszás a play() metódus segítségével
    play(getCodeBase(), "sip.au");
    play(getCodeBase(), "sip.au");
  }
```

```
  // Hanglejátszás az AudioClip objektumok metódusaival
  public void actionPerformed(ActionEvent e)
  {
    if(e.getSource()==beButton)     // folyamatos lejátszás
      hatterzene.loop();
    else
    if(e.getSource()==kiButton)     // lejátszás megállítása
      hatterzene.stop();
    else
      mpjel.play();                  // egyszeri lejátszás
  }
}
```

Az applet futásának ablaka:



12.7 Internet

A Java gazdag hálózatkezelési lehetőségei közül csak egyre, a weblapok megjelenítésére térünk ki. Az alábbi példában a ***java.net*** csomag nagyszámú interfésze és osztálya közül az ***URL*** osztályt és a ***MalformedURLException*** kivételt használjuk, valamint megismerkedünk az applet ***getAppletContext()*** metódusával.

A ***getAppletContext()*** metódus az ***AppletContext*** interfész objektumpéldányával tér vissza, melynek ***showDocument()*** metódusával az argumentumként megadott ***URL*** típusú objektumban tárolt honlapra válthatunk. Az elmondottakat szemléltető programrészlet:

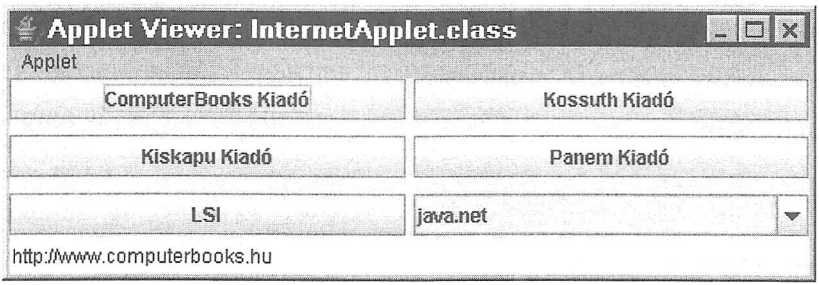
```
try
{
      URL url=new URL("www.computerbooks.hu");
      AppletContext context=getAppletContext();
      context.showDocument(url);
}
catch (MalformedURLException ex)
{
      showStatus("Hibás URL";
}
```

Készítsünk kisalkalmazást, amely nyomógombokkal öt magyar könyvkiadó honlapjára, illetve egy lenyíló listából választva néhány Java-val kapcsolatos webhelyre navigál! (*Internet|InternetApplet*)

```
import java.awt.*;
import java.applet.*;
import java.net.*;
import java.awt.event.*;
import javax.swing.*;

public class InternetApplet extends JApplet implements ActionListener
{
   JComboBox cbUrl;
   JButton kiskapuButton, panemButton, computerbooksButton;
   JButton kossuthButton, lsiButton;
   final String [] webhelyek = {"java.sun.com", "www.java.com",
                                "java.net", "www.javaworld.com",
                                "freewarejava.com", "www.textpad.com",
                                "www.eclipse.org", "www.netbeans.org",
                                     "www.borland.com/us/products/jbuilder",
                                "www.jetbrains.com/idea/",
                                "www.oracle.com/technology/products/jdev"};

   public void init() {
       // A felhasználói felület felépítése
       JPanel panel = new JPanel();
       panel.setLayout(new GridLayout(3,2,5,10));
       Font font=new Font("Times New Roman",Font.PLAIN, 20);

       computerbooksButton=new JButton("ComputerBooks Kiadó");
       panel.add(computerbooksButton);
       computerbooksButton.addActionListener(this);

       kossuthButton=new JButton("Kossuth Kiadó");
       panel.add(kossuthButton);
       kossuthButton.addActionListener(this);

       kiskapuButton=new JButton("Kiskapu Kiadó");
       panel.add(kiskapuButton);
       kiskapuButton.addActionListener(this);

       panemButton=new JButton("Panem Kiadó");
       panel.add(panemButton);
       panemButton.addActionListener(this);

       lsiButton=new JButton("LSI");
       panel.add(lsiButton);
       lsiButton.addActionListener(this);

       cbUrl=new JComboBox(webhelyek);
       panel.add(cbUrl);
       cbUrl.addActionListener(this);
       cbUrl.setFont(new Font("Times new Roman",Font.BOLD, 12));

       setContentPane(panel);
       setSize(500, 100);
   }
```

```
    public void actionPerformed(ActionEvent e) {
        String s="";
        if (e.getSource()==kiskapuButton)
               s="http://www.kiskapu.hu";
        else if (e.getSource()==panemButton)
               s="http://www.panem.hu/";
        else if (e.getSource()==computerbooksButton)
               s="http://www.computerbooks.hu";
        else if (e.getSource()==kossuthButton)
               s="http://www.kossuth.hu";
        else if (e.getSource()==lsiButton)
               s="http://www.lsi.hu";
        else if (e.getSource()==cbUrl)
               s= "http://" + cbUrl.getSelectedItem();
        try {
               URL url = new URL(s);
               AppletContext context=getAppletContext();
               context.showDocument(url);
        }
        catch (MalformedURLException ex) {
               s = "Hibás URL";
        }
        showStatus(s);
    }
}
```

Az applet futásának eredménye:



12.8 Animációk

A 9. fejezetben megismerkedtünk a ***Thread*** osztállyal, melynek segítségével több szálon futó programokat készíthetünk. Az elmondottak grafikus felületen való alkalmazásával látványos animációkat készíthetünk.

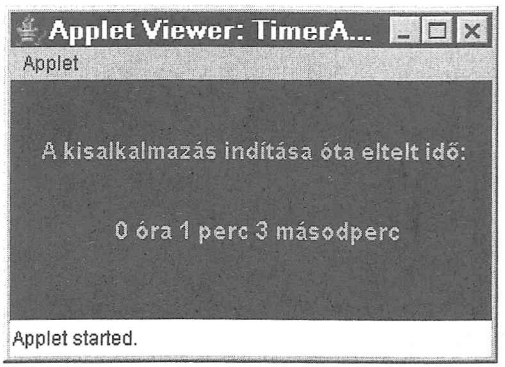
Az animációkban az applet ***start()*** metódusában létrehozzuk a ***Thread*** objektumpéldányát, és elindítjuk a szál futását. A ***stop()*** metódusban leállítjuk a szálat, és töröljük a szál objektumát. Az ilyen appletekben implementáljuk a ***Runnable*** interfész egyetlen ***run()*** metódusát, ahol legtöbbször időzítést, és egyéb, az animációval kapcsolatos tevékenységet valósítunk meg..

Készítsünk appletet, amely megjeleníti a kisalkalmazás indítása óta az eltelt időt! (*Animacio\Timer*)

```
import java.applet.Applet;
import java.awt.*;

public class TimerApplet extends Applet implements Runnable
{
    private String msg1;
    private Font font;
    private int secs;  // másodpercek
    private int mins;  // percek
    private int hrs;   // órák
    private Thread szamlalo = null;
    private volatile boolean stopJelzo=true;

    public TimerApplet() { // konstruktor
        msg1 = new String("A kisalkalmazás indítása óta eltelt idő:");
        font = new Font("Arial", 1, 15);
    }

    public void paint(Graphics gr) { // megjelenítés
        // az időváltozók léptetése
        if(++secs == 60) {
            mins++;
            secs = 0;
        }
        if(mins == 60) {
            hrs++;
            secs = mins = 0;
        }
        gr.setFont(font);
        gr.setColor(Color.green);
        gr.drawString(msg1, 20, 50);
        gr.drawString("    " + hrs + " óra " + mins + " perc "
                          + secs + " másodperc", 50, 100);
        setBackground(Color.black);
    }

    public void init() {
        setSize(300,150);
    }

    public void start() {
       if(szamlalo == null) { // a szál létrehozása és indítása
           szamlalo = new Thread(this);
           stopJelzo = false;
           szamlalo.start();
       }
    }

    public void stop() {
        stopJelzo = true;  // megállásjelző
        szamlalo = null;   // a szál törlése
    }
```

```
    public void run() { // a szál futtatása
        do {
           if (stopJelzo)
               break;  // ha leállt, semmit sem teszünk
            repaint();
            try
            {
               Thread.sleep(1000L); // várakozás 1 másodpercig
            }
            catch(InterruptedException e) { }
        } while(true);
    }
}
```

Az applet futásának eredménye:



Írjunk appletet, amely *madar.au* hangfájlt a háttérben ismételten játssza, és minden másodpercben hangjelzést (*sip.au*) ad! (*Animacio\Zenelo*)

```
import java.awt.*;
import java.applet.*;
public class Zenelo extends Applet implements Runnable
{
      AudioClip hatterzene, mpjel;
      Thread szal;
      boolean futas;

      public void init() { // a hangfájlok betöltése
         hatterzene = getAudioClip(getCodeBase(),"madar.au");
         mpjel = getAudioClip(getCodeBase(), "sip.au");
         futas = false;
         szal = null;
      }

      public void paint(Graphics g) { // szöveg megjelenítése
         g.drawString("Hangfájl lejátszása a háttérben", 10, 10);
      }
```

```
    public void start() { // a szál indítása
        if (szal == null) {
            szal = new Thread(this);
            futas = true;
            szal.start();
        }
     }
    public void stop() {
        if (szal != null) { // a szál leállítása
            if (hatterzene != null) {
                hatterzene.stop(); // a háttérzene leállítása
                hatterzene = null;
            }
            futas = false;
            szal = null;
        }
    }
    public void run() { // várakozás és sipolás másodpercenként
        if (!futas) // ha leállt a szál
            return;
        if (hatterzene != null)
            hatterzene.loop();  // a háttérzene (újra)indítása
        while (szal != null) {
            try {
                Thread.sleep(1000); // 1 mp várakozás
            }
            catch (InterruptedException e) { }
            if (hatterzene != null)
                mpjel.play();  // ha van háttérzene, sípolás
        }
    }
}
```

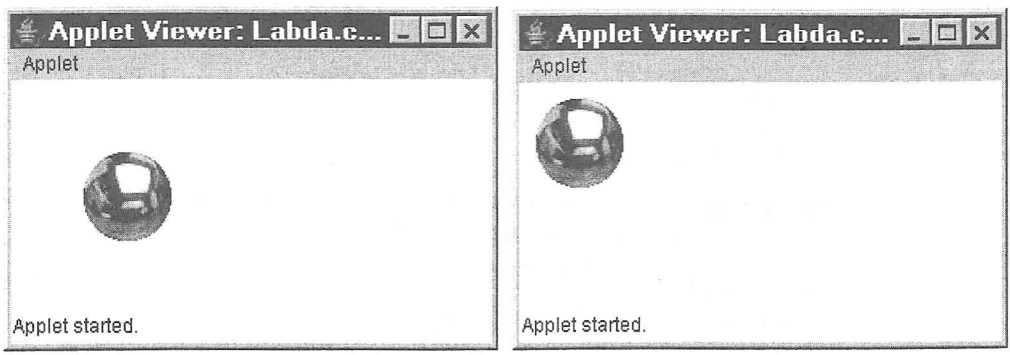
Tervezzünk appletet, amely labdát mozgat az ablakban az ablak széleinek ütközve! A labda mozgásának iránya az egér bármelyik gombjával kattintva megváltoztatható! (*Animacio\Labda*)

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class Labda extends Applet implements Runnable, MouseListener
{
   private int x,y,dx,dy,w,h;
   private Graphics offGraphics;  // háttérgrafika
   private Image offImage;        // háttérkép
   private Image labda;
   private Thread animator=null;
   boolean stopFlag;
```

```
public void init(){
      offImage=createImage(300,300); // a háttérkép létrehozása
      offGraphics=offImage.getGraphics();
      labda=getImage(getDocumentBase(),"labda.jpg");
      x = y = 150;  // indulási pizíció
      dx = dy = 5;  // indulási lépéstáv
      setSize(300,300);
      setBackground(Color.white);
      addMouseListener(this);
}

public void start() { // a szál indítása
      animator=new Thread(this);
      stopFlag=false;
      animator.start();
}

public void run() {
      while(true) {
            if (stopFlag) // ha megállt az animáció
               break;
            try {
               Thread.sleep(100); // 0,1 másodperces várakozás
            }
            catch (InterruptedException ex){}
            w = labda.getWidth(this);  // a labda méretei
            h = labda.getHeight(this);
            if ((x+w+dx)>300 || (x+dx)<0)
               dx=-dx; // x irányváltás
            if ((y+h+dy)>300 || (y+dy)<0)
               dy=-dy; // y irányváltás
            // léptetés
            x += dx;
            y += dy;
            repaint();
      }
}

public void stop() { // a szál törlése
      stopFlag = true;
      animator = null;
}

// a háttérben építjük fel a képet, majd megjelenítjük azt
// példa kettőspuffer használatára
public void paint(Graphics g) {
    int h = getSize().height;
    int w = getSize().width;
    offGraphics.setColor(Color.white);
    offGraphics.fillRect(0,0,w,h);  // a rajzlap törlése
    offGraphics.drawImage(labda,x,y,this);
    g.drawImage(offImage,0,0,this);
}
```

```
   // egérkattintáskor módosítjuk a lépéseket - irányváltás
public void mouseClicked(MouseEvent e){
       int dxx,dyy;
       dxx = e.getX()-x;
       dyy = e.getY()-y;
       dx = 5*Math.abs(dxx)/dxx;
       dy = dx*(dyy/dxx);
}

   public void mouseEntered(MouseEvent e){}
   public void mouseExited(MouseEvent e){}
   public void mousePressed(MouseEvent e){}
   public void mouseReleased(MouseEvent e) {}
}
```

Az applet futásának eredményei:



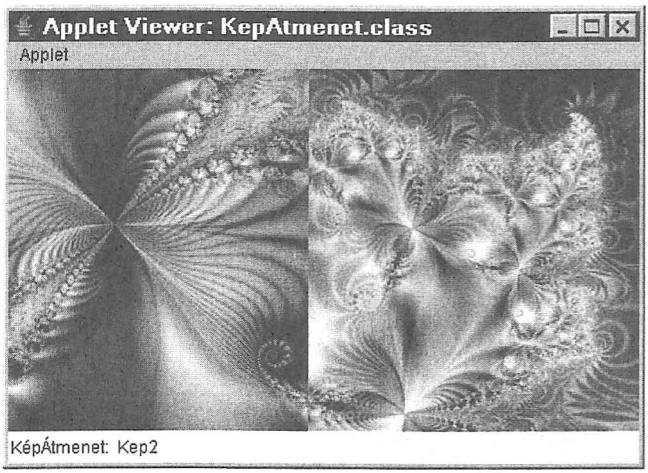
Készítsünk kisalkalmazást, amely két képet jelenít meg felváltva, képváltással! (*Animacio\Kepatmenet*)

```
import java.applet.*;
import java.awt.*;

public class KepAtmenet extends Applet implements Runnable
  {
    Graphics grafika=null;
    Image kep1;
    Image kep2;
    int width;
    int height;
    boolean kepbetoltesKesz = false;
    int varakozasiIdo = 0;

    public void init() {
        grafika = getGraphics();  // az applet adatai: grafika,
        width = getSize().width;  //                    méretek
        height = getSize().height;

        String parameter = getParameter("VARAKOZASI_IDO");
        if (parameter != null) // HTML-paraméter
          varakozasiIdo = Integer.parseInt(parameter);
```

```
        kep1 = getImage(getCodeBase(), "kep1.jpg");
        kep2 = getImage(getCodeBase(), "kep2.jpg");
        Image hatterKep = createImage(500, 300);
        Graphics hatterGrafika = hatterKep.getGraphics();
        hatterGrafika.drawImage(kep2, 0, 0, this);
        hatterGrafika.drawImage(kep1, 0, 0, this);
    }
    public void start() {  // új névtelen szál indítása
        (new Thread(this)).start();
    }

    void varakozik(long vIdo) { // adott ideig (ms) várakozik
        try {
              Thread.sleep(vIdo);
        }
        catch (InterruptedException e) {}
    }

    public void run() {
        while (!kepbetoltesKesz); // vár a képek betöltödéséig
        while (true) {
            showStatus("KépÁtmenet:  Kep2");
            for (int x = 0; x < width; x++) {
                Graphics g2 = grafika.create(); // kép1->kép2
                g2.clipRect(x, 0, 1, height);
                g2.drawImage(kep2, 0, 0, width, height, this);
                varakozik(varakozasiIdo);
            }
            showStatus("KépÁtmenet:  Kep1");
            for (int x = width-1; x >= 0; x--) { // kép2->kép1
                Graphics g2 = grafika.create();
                g2.clipRect(x, 0, 1, height);
                g2.drawImage(kep1, 0, 0, width, height, this);
                varakozik(varakozasiIdo);
            }
        }
    }

    // Akkor hívódik meg, amikor a használni kívánt
    // kép elérhetővé válik
    public boolean imageUpdate(Image img, int infoflags, int x, int y,
                               int w, int h) {
        if (infoflags == ALLBITS && img==kep1) {
            kepbetoltesKesz = true;
            repaint();
            return false;
        }
        else
          return true;
    }
```

```
    public void paint(Graphics g)  // a kép1 megjelenítése
    {
        if (!kepbetoltesKesz)
           showStatus("Képbetöltés folyamatban");
        else
        {
            width = getSize().width;
            height = getSize().height;
            grafika.drawImage(kep1, 0, 0, width, height, this);
        }
    }
}
```

Az applet futásának eredménye:

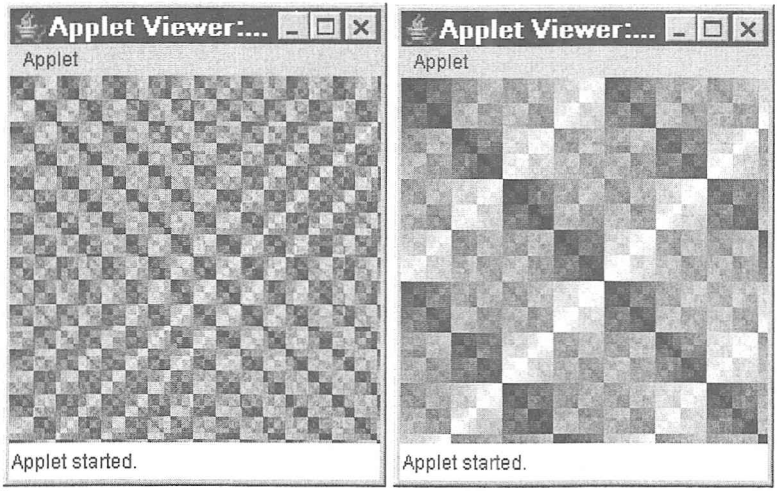


Készítsünk kisalkalmazást, amely egy kaleidoszkóp animációt valósít meg! (*Animacio\Kaleidoszkop*)

```
import java.awt.*;
import java.applet.*;
import java.awt.image.*;

public class Kaleidoszkop extends Applet implements Runnable
{
   Rectangle d;
   Image offImage;
   Thread animator=null;
   boolean stopFlag;

   public void init() {
      setSize(230,230);
      d = getBounds();
      setBackground(Color.blue);
   }
```

```
    public void start() {
       animator = new Thread(this);
       stopFlag = false;
       animator.start();
    }

    public void stop() {
      stopFlag = true;
      animator = null;
    }

    public void run() {
       while(true) {
        try {
            Thread.sleep(300);
        }
        catch (InterruptedException ex){}
        generateImage();
        repaint();
        if (stopFlag)
            break;
       }
    }

    // az offImage kép létrehozása pixelenként
    public void generateImage()
    {
       int pixels[] = new int[d.width * d.height];
       int i = 0;
       int rm = (int)(10*Math.random()),  // RGB színösszetevők
           gm = (int)(10*Math.random()),
           bm = (int)(10*Math.random());
       int r, g, b;
       // a kép létrehozása képpontonként
       for(int y=0; y < d.height; y++)
          for(int x=0; x < d.width; x++) {
            r = (x*rm^y*rm) & 0xff;  // piros
            g = (x*gm^y*gm) & 0xff;  // zöld
            b = (x*bm^y*bm) & 0xff;  // kék
            pixels[i++] = (255 << 24) | (r << 16) | (g << 8) | b;
           }
        offImage = createImage(new MemoryImageSource(d.width, d.height,
                                    pixels, 0, d.width));
    }

    // az offImage kép megjelenítése
    public void paint(Graphics g)
    {
        g.drawImage(offImage,0,0,this);
    }
}
```

Az applet futásának eredményei:



13. Algoritmusok programozása (CD)

Talán sokan összeráncolják a szemöldöküket a fejezet címe láttán, és felteszik a kérdést, mit keres egy ilyen téma az objektum-orientált Java nyelvet ismertető kötetben. Az ellentmondás első látásra valóban szembeötlő, azonban az alábbi gondolatmenettel igyekszünk feloldani azt.

Mint ismert, az objektum-orientált programozás alapjául az osztálynak nevezett programegységek kialakítása, tervezése szolgál. E munka során az adatokat (adatmezőket) és rajtuk végzett műveleteket (metódusokat) egyaránt nagy gonddal kell megválasztanunk, illetve megvalósítanunk. A metódusban tárolt programkód egy jól meghatározott cél érdekében, véges számú lépesben elvégzett tevékenységsorozatot határoz meg. Ez pedig éppen az algoritmusok általános definícióját jelenti, így elmondhatjuk, hogy a metódusok mindegyike valamilyen algoritmust valósít meg.

Az algoritmusok közül úgy kell választanunk, hogy a kitűzött célt a lehető legegyszerűbben és leggyorsabban elérő algoritmusra kell voksolnunk. Sajnos ehhez, a munka kezdetén általában nem rendelkezünk elegendő információval, így fontos szerepet kap a programozói tapasztalat. Tapasztalatot pedig csak úgy szerezhetünk, hogy elég sok kisebb, nagyobb feladatot önállóan megoldunk, illetve a kész megoldásokat megértjük, és egy következő feladat megoldása során felhasználjuk.

A könyvünk zárófejezetének kimondatlanul is a tapasztalatszerzés elősegítése a célja. Ehhez, különböző területekről származó egyszerűbb és bonyolultabb algoritmusok Java megvalósítását csoportosítva tárjuk az *Olvasó* elé. Mivel ez a fejezet, nem kötődik szervesen a Java ismeretekhez, úgy döntöttünk, hogy elektronikus formában teszszük közzé, a CD-melléklet ***\eKonyv*** mappájában.

A fejezetben bemutatott algoritmusokat öt csoportba sorolhatjuk, az alábbiak szerint:

Számelméleti algoritmusok
osztók és prímszámok meghatározása,
törzstényezők előállítása,
a legnagyobb közös osztó és a legkisebb közös többszörös megkeresése.

Rekurzív algoritmusok
bináris keresés,
determináns számítása,
hanoi tornyai.

Keresési algoritmusok
szekvenciális (soros) keresés,
bináris keresés.

Rendezési algoritmusok

rendezés cserével,
rendezés közvetlen beszúrással,
buborékrendezés,
shell-rendezés,
gyorsrendezés (quicksort),
rendezés bináris fa felépítésével,
elemek rendezett listája.

Numerikus módszerek

nemlineáris egyenlet numerikus megoldása,
numerikus integrálás (numerikus kvadratúra),
interpoláció,
lineáris egyenletrendszer megoldása.

Felhívjuk a figyelmet arra, hogy az ***eKonyv\Algoritmusok.pdf*** állomány elolvasásához az ***Adobe® Reader®*** alkalmazást kell a számítógépre telepíteni. Az olvasóprogram legújabb verziója térítésmentesen letölthető az *Adobe Systems Incorporated* cég honlapjáról: *www.adobe.com*, illetve korábbi, magyar nyelvű változatai telepíthetők a CD-mellélet *\Adobe Reader* alkönyvtárából.

F1. Hasznos táblázatok

F1.1 Java kulcsszavak

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

* nem használt

** Java 1.2 verziótól

*** Java 1.4 verziótól

**** Java 5.0 verziótól

F1.2 Primitív adattípusok

Egész típusok

Típus	Ábrázolás	Értékkészlet
byte	8-bites, előjeles, 2-es komplemens	-128 ... 127
short	16-bites, előjeles, 2-es komplemens	-32768 ... 32767
int	32-bites, előjeles, 2-es komplemens	-2147483648 ... 2147483647
long	64-bites, előjeles, 2-es komplemens	-9223372036854775808 ... 9223372036854775807
char	16-bites, előjel nélküli, *Unicode*	'\u0000' ... '\uffff'

Lebegőpontos típusok

Típus	Ábrázolás	Értékkészlet
float	32-bites, IEEE 754	1.40239846e-45 ... 3.40282347e+38
double	64-bites, IEEE 754	4.94065645841246544e-324 ... 1.79769313486231570e+308

Logikai típus

Típus	Ábrázolás	Értékkészlet
`boolean`	`8-bites`	`true` vagy `false`

F1.3 Escape karakterkódok

Escape	*Jelentés*
`\n`	új sor
`\t`	tabulátor
`\b`	visszatörlés
`\r`	kocsi vissza
`\f`	lapdobás
`\\`	fordított perjel
`\'`	aposztróf
`\"`	idézőjel
`\ddd`	oktális
`\xdd`	hexadecimális
`\udddd`	Unicode karakter

F1.4 Operátorok precedenciája

Precedencia	Asszociativitás	Műveleti jel	Megnevezés
1	nincs	`new`	objektum-létrehozás
2	nincs	`new`	tömblétrehozás
3	balról-jobbra	`.`	mezőelérő művelet
4	nincs	`()`	metódushívás
5	nincs	`[]`	tömbelérő művelet
6	nincs	`++, --`	postfix műveletek
7	jobbról-balra	`- , +, !, ~, ++, --, (típus)`	egyoperandusú műveletek
8	balról-jobbra	`*, /, %`	multiplikatív műveletek
9	balról-jobbra	`+, -`	additív műveletek
10	balról-jobbra	`<<, >>, >>>`	biteltoló műveletek
11	balról-jobbra	`<, >, <=, >=, instanceof`	összehasonlító műveletek
12	balról-jobbra	`==, !=`	azonosságvizsgálat

Precedencia	Asszociativitás	Műveleti jel	Megnevezés
13	balról-jobbra	`&`	bitenkénti logikai ÉS
14	balról-jobbra	`^`	bitenkénti logikai kizáró VAGY
15	balról-jobbra	`|`	bitenkénti logikai VAGY
16	balról-jobbra	`&&`	logikai ÉS
17	balról-jobbra	`||`	logikai VAGY
18	jobbról-balra	`?:`	feltételes művelet
19	jobbról-balra	`=, *=, /=, %=, -=, <<=, >>=, >>>=, &=, ^=, |=`	értékadó műveletek

F1.5 Java módosítók

Módosító	Használva	Jelentés
abstract	osztály interfész metódus	Implementálatlan metódust tartalmaz, nem példányosítható. Minden interfész absztrakt, így használata opcionális. Nincs törzse csak fejléce. A metódust tartalmazó osztály absztrakt.
final	osztály metódus adatmező változó	Nem származtatható alosztály belőle. Nem lehet származtatás során lecserélni. Nem lehet megváltoztatni az értékét. A **static final** adatmezők fordítás-idejű konstansok. Nem lehet megváltoztatni az értékét.
native	metódus	Platformfüggő, nincs törzse csak fejléce.
nincs (*package*)	osztály interfész tag	Csak a csomagon belül érhető el. Csak a csomagon belül érhető el. Csak a csomagon belül elérhető adatmező vagy metódus.
private	tag	Csak a tagot definiáló osztályon belül érhető el.
protected	tag	Csak az osztályt tartalmazó csomagból, illetve az osztályból származtatott alosztályokból érhető el.
public	osztály interfész tag	Mindenhonnan elérhető. Mindenhonnan elérhető. Mindenhonnan elérhető, ahonnan a tagot tartalmazó osztály elérhető.

Módosító	Használva	Jelentés
strictfp	osztály metódus	Az osztály minden metódusa automatikusan az IEEE 754 szabványt használja a lebegőpontos kifejezések kiértékelése során. A metódusa az IEEE 754 szabványt használja a lebegőpontos kifejezések kiértékelése során.
static	osztály metódus adatmező inicializáló	Egy belső osztályt külső szinten definiálttá tesz. Osztálymetódus – az osztály nevét használva hívjuk. Egyetlen példányban létező osztálymező – az osztály nevét használva érjük el. Az osztály betöltésekor, a példányok létrehozása előtt fut le.
synchronized	metódus	Statikus metódus esetén zárolja az osztályt, a metódus végrehajtása előtt. Nem statikus metódus esetén pedig a kijelölt objektumpéldányt zárolja A zárolás után egy másik szál nem módosíthatja az osztályt vagy a példányt..
transient	adatmező	Az objektumok adatfolyamból való betöltése/mentése (*serialization*) során használjuk.
volatile	adatmező	Szinkronizálatlan szálakból is elérhető.

F1.6 Az Object osztály metódusai

Metódus	Leírás
`public Object()`	konstruktor,
`protected Object clone() throws CloneNotSupportedException`	új objektummásolat készítése,
`public boolean equals( Object obj )`	két objektum összehasonlítása,
`protected void finalize() throws Throwable`	a garbage collection (szemétgyűjtés) során, az objektumpéldány törlése előtt meghívódó metódus,
`public final Class getClass()`	visszadja az objektum futásidejű osztályát,
`public int hashCode()`	megadja az objektum hash-kódját,

`public final void notify()`	feléleszti azt a szálat, amelyik objektum-monitorra vár,
`public final void notifyAll()`	feléleszti azokat a szálakat, amelyik objektum-monitorra várnak,
`public String toString()`	az objektum szöveges reprezentációjával tér vissza,
`public final void wait() throws InterruptedException`	várakozik, amíg más szál nem jelzi az objektum megváltozását,
`public final void wait( long timeout ) throws InterruptedException`	adott ideig (ms) várakozik, amíg más szál nem jelzi az objektum megváltozását,
`public final void wait( long timeout, int nanos ) throws InterruptedException`	adott ideig (ms,ns) várakozik, amíg más szál nem jelzi az objektum megváltozását.

F2. A Java 2 Platform elemei

F2.1 A Java 2 Platform leggyakrabban használt csomagjai

Csomag	*Tartalom*
java.applet	Appletek készítéséhez szükséges osztályok.
java.awt	A grafikus felhasználói felület kialakításához, rajzoláshoz és képek megjelenítéséhez szükséges osztályok.
java.awt.datatransfer	Alkalmazások közötti adatcsere osztályai.
java.awt.event	Az *AWT* komponensek eseménykezelését megvalósító osztályok és interfészek.
java.awt.font	Betűkészlethet kapcsolódó osztályok és interfészek
java.awt.geom	A *Java 2D* osztályai.
java.awt.image	Képekezelés (létrehozás, módosítás) osztályai.
java.awt.print	A *Printing API* nyomtatást segítő osztályai.
java.beans	A *JavaBeans* arhitektúrán alapuló konponensek (*beans*) fejlesztéséhez kapcsolósdó osztályok.
java.io	A fájlrendszer, az adatfolyamok és a szerializáció (*serialization*) kezeléséhez szükséges ossztályok.
java.lang	A Java programozási nyelvhez kapcsolódó alavető osztályok és interfészek (típusosztályok, *Math*, *Thread*, *String*, *System*, *Runnable*, *Cloneable* stb.)
java.math	Nagypontosságú egész (*BigInteger*) és nagypontosságú decimális aritmetika osztályai (*BigDecimal*).
java.net	Hálózatos alkalmazások készítéséhez szükséges osztályok.
java.security	A biztonsági keretrendszert támogató osztályok és interfészek.
java.sql	*API* a Java nyelven történő adatbázis-kezelés megvalósításához.
java.text	Szövegek, dátumok, számok és üzenetek nyelvfüggetlen kezelését segítő osztályok és interfészek.
java.util	Különböző tároló- (adatstruktúra), dátum- és időosztály, valamint egyéb segédosztályok (sztingtokenizáló, véletlenszám-generátor, bittömb stb.) és interfészek.
java.util.jar	A szabványos *ZIP* formátumon alapuló *JAR* (*Java ARchive*) formátumú állományok írását és olvasását segítő osztályok.
java.util.zip	A szabványos *ZIP* és *GZIP* formátumú fájlok írását és olvasását támogató osztályok.
javax.crypto	Titkosítási műveletek végzését segítő osztályok és interfészek.
javax.imageio	A *Java Image I/O API* fő csomagja, melynek osztályai támogatját a *JPEG*, *PNG*, *BMP*, *WBMP* és *GIF* formátumú képek olvasását és írását (a *GIF* kivételével).
javax.net	Osztályok a *socket* alapú hálózatos alkalmazások készítéséhez.
javax.print	A *Java Print Service API* osztályai és interfészei.

Csomag	*Tartalom*
javax.sound.midi	*MIDI* (*Musical Instrument Digital Interface*) hangadatok kezelését segítő osztályok és interfészek.
javax.sound.sampled	Mintavételezett hangadatok feldolgozását támogató osztályok és interfészek.
javax.sql	A szerveroldali adatbázis-kezelést segítő *API*.
javax.swing	Olyan komponensek készlete, amelyek minden platformon ugyanúgy működnek.
javax.swing.border	A *Swing* komponensek keretét meghatározó osztályok és interfész..
javax.swing.colorchooser	*A JColorChooser* komponens által használt osztályok és interfész.
javax.swing.event	A speciális *Swing* komponensek eseménykezelését megvalósító osztályok és interfészek.
javax.swing.filechooser	*A JFileChooser* komponens által használt osztályok
javax.swing.plaf	A *Swing* komponensek megjelenését szabályzó interfész és absztrakt osztályok.
javax.swing.table	A *javax.swing.JTable* komponenshez kapcsolódó osztályok és interfészek.
javax.swing.text	Osztályok és interfészek, amelyek a szerkeszthető és nem szerkeszthető szövegkomponensekkel állnak kapcsolatban.
javax.swing.text.html	A *HTMLEditorKit* osztály valamint HTML-szerkesztők készítését támogató osztályok.
javax.swing.text.rtf	Az *RTFEditorKit* osztály.
javax.swing.tree	A *javax.swing.JTree* komponenshez kapcsolódó osztályok és interfészek.
javax.swing.undo	A *Visszavonás/Ismét* (*undo/redo*) műveleteket támogató osztályok és interfészek.
javax.xml	Az alapvető *XML*-konstansokat definiáló osztály.
javax.xml.parsers	XML-dokumentumok értelmezését lehetővé tevő osztályok.

F2.2 A java.lang.Math osztály statikus tagjai

Statikus mező

double	***E***	az *e* szám: *2.718281828459045*
double	***PI***	a π szám: *3.141592653589793*

Statikus metódusok

double	***abs***(double x)	$\|x\|$
float	***abs***(float x)	$\|x\|$
int	***abs***(int x)	$\|x\|$
long	***abs***(long x)	$\|x\|$
double	***acos***(double x)	*arc cos x*,értéke *0.0* és π közé esik

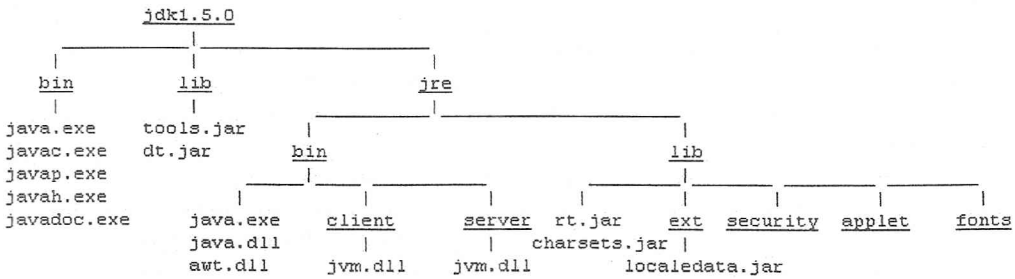
double	***asin***(double x)	*arc sin x*,értéke - *π/2* és *π/2*közé esik
double	***atan***(double x)	*arc tan x*,értéke - *π/2* és *π/2*közé esik
double	***atan2***(double x, double y)	*arc tan y/x*,értéke - *π/2* és *π/2*közé esik
double	***cbrt***(double x)	köbgyök
double	***ceil***(double x)	az argumentumtól nem kisebb legkisebb egész (legközelebbi - végtelen irányában)
double	***cos***(double x)	*cos x*
double	***cosh***(double x)	*ch x*
double	***exp***(double x)	e^x
double	***expm1***(double x)	e^x*-1*
double	***floor***(double x)	az argumentumtól nem nagyobb legnagyobb egész (legközelebbi + végtelen irányában)
double	***hypot***(double x, double y)	***sqrt***$(x^2 + y^2)$ érték
double	***IEEEremainder***(double f1, double f2)	az IEEE 754 szabvány szerinti maradék
double	***log***(double x)	természetes alapú logaritmus
double	***log10***(double x)	tízes alapú logaritmus
double	***logm1***(double x)	***log***(*1.0+x*)
double	***max***(double x, double y)	*x* és *y* közül a nagyobb
float	***max***(float x, float y)	*x* és *y* közül a nagyobb
int	***max***(int x, int y)	*x* és *y* közül a nagyobb
long	***max***(long x, long y)	*x* és *y* közül a nagyobb
double	***min***(double x, double y)	*x* és *y* közül a kisebb
float	***min***(float x, float y)	*x* és *y* közül a kisebb
int	***min***(int x, int y)	*x* és *y* közül a kisebb
long	***min***(long x, long yb)	*x* és *y* közül a kisebb
double	***pow***(double x, double y)	x^y
double	***random***()	véletlen szám, amely ≥ *0.0* és < *1.0*.
double	***rint***(double x)	az argumentumhoz legközelebbi egész
long	***round***(double x)	az argumentumhoz legközelebbi egész
int	***round***(float x)	az argumentumhoz legközelebbi egész
double	***signum*** (double x)	előjel, *-1.0*, ha *x<0*, *0.0*, ha *x=0*, *1.0*, ha *x>0*
float	***signum*** (float x)	előjel, *-1.0*, ha *x<0*, *0.0*, ha *x=0*, *1.0*, ha *x>0*
double	***sin***(double x)	*sin x*
double	***sinh***(double x)	*sh x*
double	***sqrt***(double x)	négyzetgyök
double	***tan***(double x)	*tg* x
double	***tanh***(double x)	*th* x
double	***toDegrees***(double x)	a radiánban megadott *x* szög fok értéke
double	***toRadians***(double x)	a fokban megadott *x* szög radián értéke

F2.3 AWT-események és kezelésük

AWT Eseményosztály	Kiváltó osztály	Figyelő-interfész	Eseménykezelő metódus az interfészben	Adapterosztály
ActionEvent	Button List MenuItem TextField	ActionListener	actionPerformed(e)	---
AdjustmentEvent	Scrollbar	AdjustmentListener	adjustmentValue Changed(e)	---
ComponentEvent	Component	ComponentListener	componentHidden(e) componentMoved(e) componentResized(e) componentShown(e)	ComponentAdapter
ContainerEvent	Container	ContainerListener	componentAdded(e) componentRemoved(e)	ContainerAdapter
FocusEvent	Component	FocusListener	focusGained(e) focusLost(e)	FocusAdapter
ItemEvent	Checkbox CheckboxMenuItem Choice List	ItemListener	itemStateChanged(e)	---
KeyEvent	Component	KeyListener	keyTyped(e) keyPressed(e) keyReleased(e)	KeyAdapter
MouseEvent	Component	MouseListener	mouseClicked(e) mouseEntered(e) mouseExited(e) mousePressed(e) mouseReleased(e)	MouseAdapter
		MouseMotionListener	mouseDragged(e) mouseMoved(e)	MouseMotionAdapter
TextEvent	TextComponent	TextListener	textValueChanged(e)	---
WindowEvent	Window	WindowListener	windowActivated(e) windowClosed(e) windowClosing(e) windowDeactivated(e) windowDeiconified(e) windowIconified(e) windowOpened(e)	WindowAdapter

F2.4 A JDK segédprogramjai

F2.4.1 A JDK Windows alatti telepítésekor kialakuló könyvtárstruktúra



F2.4.2 Gyakrabban használt parancssor programok

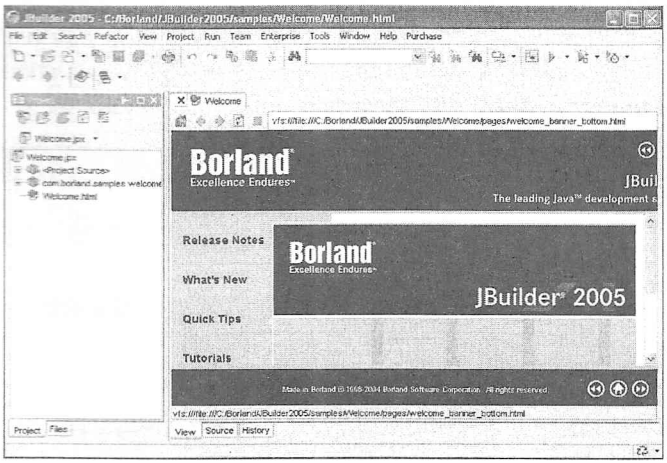
Parancssor	*Leírás*
appletviewer [*opciók*] *URL-cím*	Applet futtatása webböngésző környezet nélkül.
apt [*opciók*] *f1.java*	Java forrásfájl-ellenőrző, kommentáló és fordító.
extcheck [-verbose] vizsgáltfájl.jar	Ellenőrzi, hogy a megadott JAR-fájl tartalma nem ütközik-e a Java 2 SDK elemeivel.
htmlconverter *fájl.html*	Az appletet futtató HTML-állomány átalakítása Java plug-inné.
jar *argumetumok*	Java Archive (JAR) állományok kezelése.
java [*opciók*] *osztályfájl* [*arg1 arg2 ...*] **java** [*opciók*] **-jar** *fájl.jar* [*arg1 arg2 ...*]	Az *osztályfájl* által, illetve a *fájl.jar*-ban (Main-Class: *osztálynév)* kijelölt alkalmazás futtatása
javac [*opciók*] *f1.java f2.javak*	A Java nyelv fordítóprogramja.
javadoc [*opciók*] [*csomagnevek*] *forrásfájlnev(ek)*	A Java API dokumentáció-keszítője.

Parancssor	*Leírás*
javah [*opciók*] *osztálynév*.	C fejléc- és forrásállományok létrehozása natív metódusok készítéséhez.
javap [*opciók*] *osztálynév*	Java disassembler (szétbontó).
Parancssor	*Leírás*
javaws [*opciók*] *URL-cím*	A Java Web Start elindítja a hálózatra telepített alkalmazást/appletet.
jdb [*opciók*] *osztályfájl* [*arg1 arg2 ...*]	Java nyomkövető program.
native2ascii [*opciók*] *inputfájl* [*outputfájl*]	Natív kódolású szöveg unicode kódolásúvá alakítása.
pack200 [*opciók*]. *fájl*.pack[.gz] *jarfájl*.jar	A megadott JAR-állomány átalakítása tömörített pack200 fájllá.
unpack200 [*opciók*]. *fájl*.pack[.gz] *jarfájl*.jar	Tömörített pack200 fájl átalakítása JAR-állománnyá.

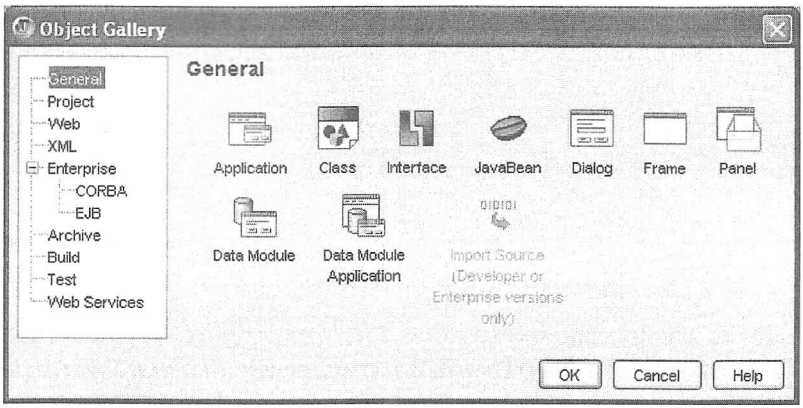
F3. A JBuilder 2005 Foundation használata

A CD-melléklet a „*JBuilder 2005 Foundation*" könyvtára tartalmazza a ***Borland Software Corporation JBuilder 2005*** programcsomagjának szabadon használható, Windows alá telepíthető változatát. A telepítést a „*jb2005_windows\install.exe*" programmal kell kezdenünk. Ezzel minden (IDE, JDK 1.4.2) felkerül a számítógépünkre, amivel a Java alkalmazások és appletek fejlesztését gyakorolhatjuk. Ajánlott azonban a telepítést a dokumentáció (*jb2005_docs\install.exe*) és a példaprogramok (*jb2005_samples\install.exe*) felmásolásával folytatni.

A sikeres telepítés és indítás után bejelentkezik az integrált fejlesztői környezet:



Ez a változat természetesen nem rendelkezik a teljes verzió minden tudásával, de így is az egyik legjobb Java fejlesztői környezet.

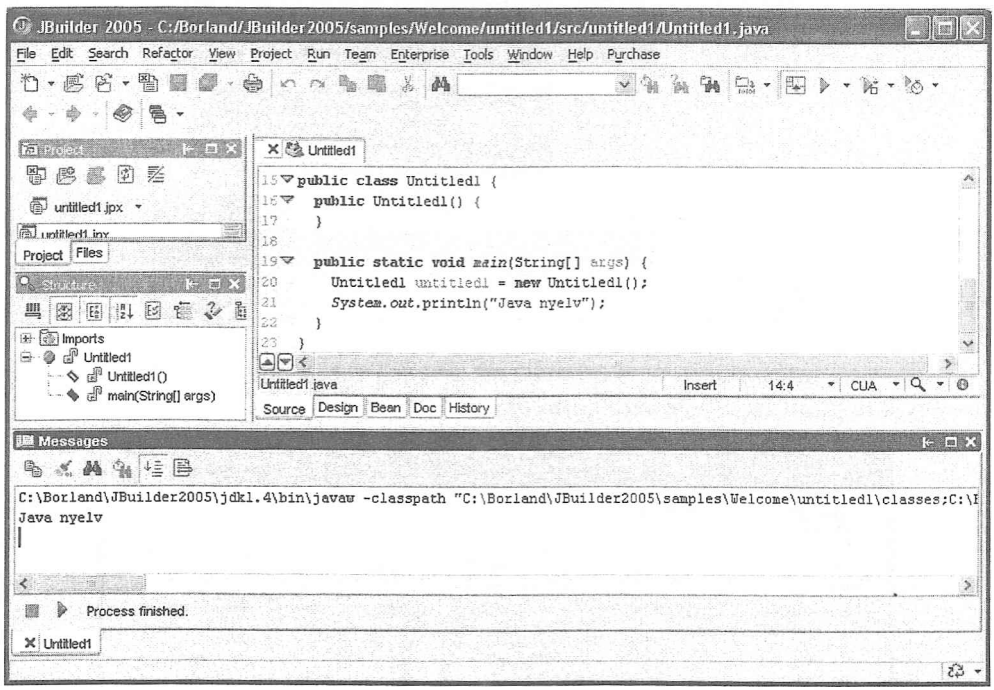


A ***File/New...*** menüválasztás hatására megjelenő párbeszédablakban színes ikonnal rendelkeznek az elérhető lehetőségek. A fentieken túlmenően még *Project*et és *WEB/Appletet* is készíthetünk.

A könyvünk tartalmához kapcsolódva elsősorban osztályt, alkalmazást és appletet kell létrehoznunk. Minden fejlesztés kerete a projekt, melyet a ***File\New Project...*** menüponttal hozhatunk létre. A megjelenő varázsló 3 lépésben kéri a projekt beállításait. Ezt követően adhatjuk a projekthez a szükséges alkalmazástípust.

Szöveges felületű alkalmazás készítése

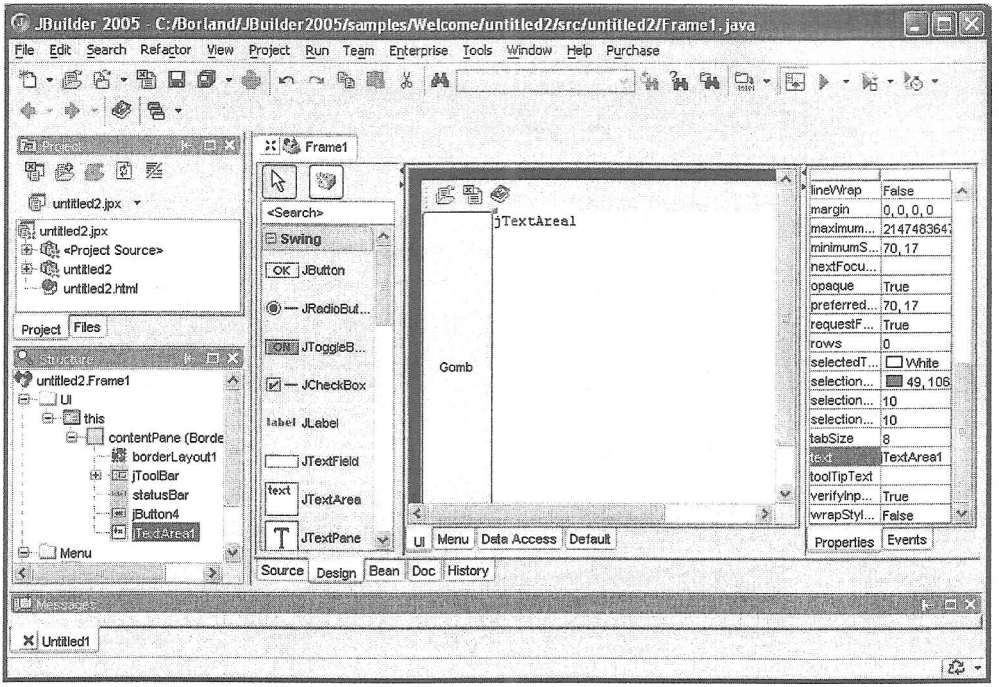
Amennyiben szöveges felületű alkalmazást szeretnénk előállítani csak egy olyan osztályt kell a projekthez adni (***File/New Class...***), amelyik tartalmazza a ***main()*** metódust. A program írása közben szövegérzékeny súgó segíti a munkánkat. A fordítást a ***Project/Make Project ...*** menüpont választásával végezhetjük el. A futtatást a ***Run/Run Project*** paranccsal kérhetjük. Az első futtatás előtt egy párbeszédablakban be kell állítani, hogy melyik osztály indítja a projektet. Szöveges alkalmazások esetén a kiírt szöveg a fejlesztői környezet ***Messages*** ablakában jelenik meg.



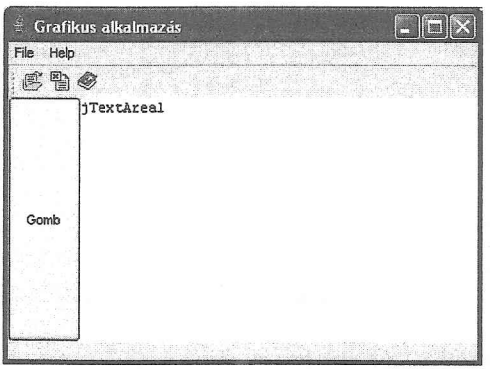
Grafikus felületű alkalmazás készítése

Grafikus felületű alkalmazás fejlesztését a ***File/New/Application*** választással kezdeményezzük. Először lefut a *Projektvarázsló*, majd pedig *Alkalmazásvarázsló* indul el,

melynek második lapján dönthetünk arról, hogy menüsor, eszközsor, állapotsor és névjegy szerepeljen-e az alkalmazás ablakában. Kilépve a varázslóból, érdemes átlépni a tervezőablakba (***Design*** fül), ahol grafikus elemekkel dolgozva folytathatjuk a felhasználói felület kialakítását.

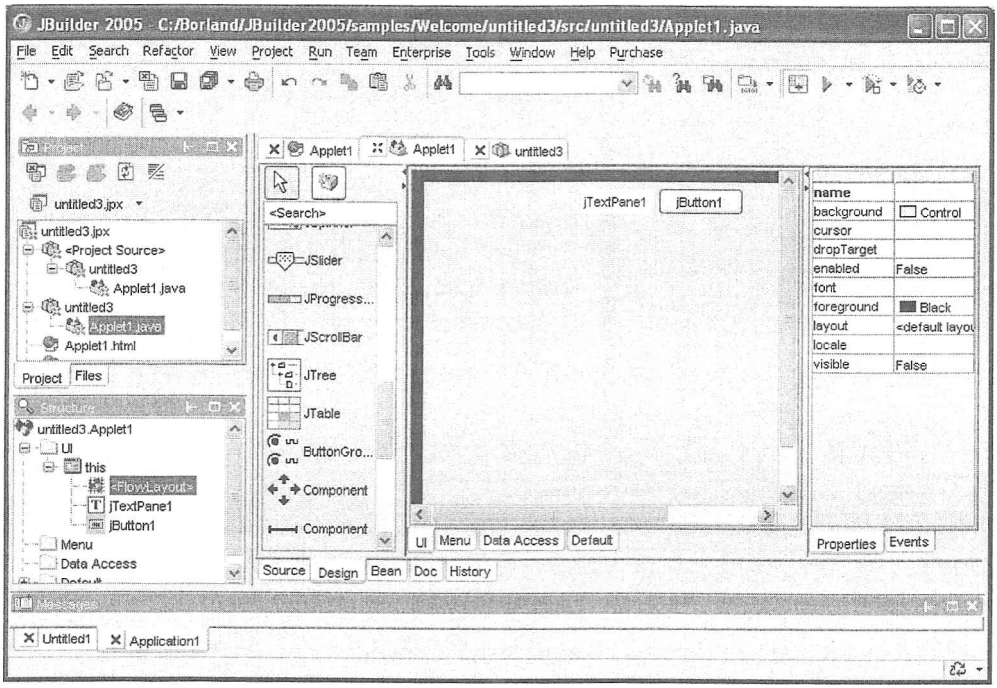


A fordítást és futtatást a szöveges alkalmazásnál elmondottak szerint kell elvégezni. A futtatás eredménye:

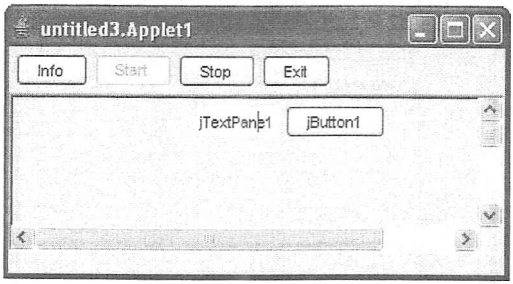


Appletek fejlesztése

A kisalkalmazások készítése nemcsak a Java-források, hanem a megfelelő weboldal kifejlesztését jelenti. Első lépés a *File/New/WEB/Applet* kiválasztása, melyet a Projektvarázsló futása követ. Ezután elindul az *Appletvarázsló*, amely segíti a megfelelő felépítésű kisalkalmazás kialakítását. A tervezést forráskód írásával, illetve a grafikus felület kialakításával egyaránt folytathatjuk.



A fordítás után elindítva a kisalkalmazást, megjelenik annak ablaka:



Megjegyezzük, hogy a ***JBuilder*** telepítése után a ***TextPad*** szövegszerkesztőt is telepíthetjük a CD-melléklet *TextPad* alkönyvtárából. A ***TextPad*** használata akkor javasolt, ha a számítógépünk nem rendelkezik elég erőforrással a ***JBuilder*** integrált grafikus fejlesztői környezet futtatásához.

Irodalomjegyzék

Kris Jamsa:
Java
Kossuth Könyvkiadó, 1996.

Java 2 - útikalauz programozóknak
ELTE TTK Hallgatói alapítvány, 1999.

Daniel J. Berg - J. Steven Fritzinger:
Java felsőfokon
Kiskapu, 1999.

Vég Csaba - dr. Juhász István:
Java - start!
Logos 2000, 1999.

Michael Morgan:
Java 2 for Professional Developers
Sams Publishing, 1999.

David Flanagen:
Java in a Nutshell
O'Reilly; 1999.

Java 2 Complete
Sybex, 1999.

Laura Lemay - Rogers Cadenhead:
Java 2 in 21 days
Sams Publishing, 2000.

Steven Holzner:
Java Black Book
Coriolis, 2000.

Matthew Robison - Pavel Vorobiev:
Swing
Manning, 2000.

Angster Erzsébet:
Objektumorientált tervezés és programozás Java I. kötet
4Kör Bt., 2001.

Angster Erzsébet:
Objektumorientált tervezés és programozás Java II. kötet
4Kör Bt., 2002.

J2EE - útikalauz Java programozóknak
ELTE TTK Hallgatói alapítvány, 2002.

Charlie Calvert - Margie Calvert:
Charlie Calvert's Learn JBuilder
Wordware Publishing, 2003.

Tárgymutató

D

E

F

G

L

M

N

O

Ö

P

R

S

T

U

V

W

Z

Megrendelőlap
info@computerbooks.hu
Telefon. 3751-564, 3753-4591

... *Dr.Kovács T.– Dr.Kovácsné C. J.– Ozsváth M.:*
Adatkezelés az MS ACCESS 2000 alkalmazásával **3.497.-**
... *Kovács L.:* **Adtatbázisok tervezésének és kezelésének módszertana** **4.200.-**
... *Czenky M.* **Adadtmodellezés * SQL és Access alkalmazás * SQL Server ADO ◎** **4.552.-**
... *Dr. Hatvany Béla Csaba*: **ASP.NET vezérlők programozása** **5.990.-**
... *Pintér M.:* **AutoCAD 2004 felhasználói ismeretek** **4.919.-**
... *Móricz Attila:* **Tippek a CD íráshoz -◎** **2.500.-**
... *Kuzmina J.-Dr.Tamás P.-Tóth B.:* **Windows alkalmazások fejlesztése C++Builder 6 rendszerben (e-könyv CD-n)** **3900.-**
... *Benkő L.–Benkő T.né–Tóth B.:* **Programozzunk C nyelven – ◎** **2.968.-**
... *Benkő T.né–Poppe A.*: **Együtt könnyebb a programozás - C – ◎** **3.220.-**
... *Benkő T.né– Tóth B. .*: **Együtt könnyebb a programozás JAVA – ◎** **3.990.-**
... *Tóth B.:* **Programozzunk C++ nyelven ! ◎** **2.994.-**
... *Benkő T.né–Poppe A.*: **Együtt könnyebb a programozás - Objektum-orientált C++ -◎** **3.217.-**
... *Kőhalmi Éva- Kőhalmi Mariann:* **CorelDraw 12** **5.590.-**
... *László József:* **Dinamikus weboldalak programozása -** 💻 **3.150.-**
... *Kovalcsik Géza:* **EXCEL 2000 –◎** **3.400.-**
... *Kovalcsik Géza:* **Az EXCEL programozása –◎** **4.800.-**
... *Kovalcsikné Pintér Orsolya:* **Az EXCEL függvényei A-tól Z-ig** **3.779.-**
... *László József:* **Hangkártya programozása** Pascal és Assembly nyelven – 💾 **1.900.-**
... *Szirmay-Kalos.:* **Háromdimenziós grafika, animáció és játékfejlesztés -◎** **5.496.-**
... *Ács Zsolt*: **Linux operációsrendszer(váltás)** - Linux 9 adminisztrátori ism. ◎ **2.979.-**
... *Czenky M.-Tamás P.- Vágási J.:* **Tanuljuk együtt az informatikát** ECDL elméleti modul – interaktív ◎ melléklettel **3.495.-**
... *Dr.Kovácsné C.J.–Ozsváth M.–G.Nagy J.:*
Mit kell tudni Az EDCL vizsgához?
- Operációs rendszer, Szövegszerkesztés, Táblázatkezelés, Prezentáció, Internet és levelezés **2.995.-**
... *Dr.Kovács T.-Dr.Kovácsné C.J.-Ozsváth M.-G.Nagy J.:*
Mit kell tudni? a PC-ről – Példatár **2.990.-**
... *László József:* **Mindenkinek az Internetről** **1.999.-**
... *Móricz A.:* **Mobiltelefon a kezekben** **990.-**
... *Molnár Mátyás:* **Lotus Notes 6 felhasználóknak –** 6.5 kiegészítéssel **2.997.-**
... *Dr. Kovácsné Cohner J.–Ozsváth M.–G. Nagy J.:* **OFFICE 2000** **2.093.-**
... *Dr. Kovácsné Cohner J.–Ozsváth M.–G. Nagy J.:* **OFFICE XP** **3.498.-**
... *Móricz A.* **Programozási alapfeladatok** - WAP oldalakhoz és WEB lapokhoz **3.398.-**
... *Tóth B-. Benkő T.-né és* társai.: **Programozzunk Turbo Pascal nyelven – ◎** **2.992.-**
... *Tamás P.– Tóth B. és* társai.: **Programozzunk DELPHI 7 rendszerben – ◎** **3.860.-**
... *Benkő T.né.*: **Programozási feladatok és algoritmusok DELPHI rendszerben – ◎** **2.999.-**
... *Juhász Tibor-Kiss Zsolt:* **Tanuljunk programozni** - Visual Basic Script, Objektumok, Web - interaktív ◎ melléklettel **3.925.-**
... *László József:* **PC hardver programozás** - valós és védett módban **4.720.-**
... *Tátrai T.:* **MS Project 2000** **4.500.-**
... *Mudri István dr:* **QuarkXpress 5** **5.500.-**
... *Dr. Hatvany Béla Csaba:* **Bit képek feldolgozása VISUL BASIC**

rendszerben - ◎ - (feladok C++ és Delphi nyelven)	**3.500.-**
... *Tamás P. – Tóth B. és* társai.: **Programozzunk Visual Basic rendszerben –◎**	**3.500.-**
... *Benkő T.né.*: **Programozási feladatok és algoritmusok Visual Basic rendszerben– ◎**	**2.999.-**
... *Stolnicki Gyula:* **SQL programozóknak ◎**	**4.970.-**
....*Nagy S.*: **Útlevél a digitális világhoz** - Nyilvános kulcsú tanúsítványok a gyakorlatban Kriptográfia, digitális aláírások, Hitelesítés szolgáltatás, Webáruházak biztonsága, Elektronikus bank biztonsága, Tanúsítványok használata	**2.680.-**
....*Nagy S.*: **Elektronikus leveleink védelme** Titkosítás, Digitális aláírás, Szabványos tanúsítványok	**2.650.-**
.... *Szász P- Bócz P:* **Világháló lehetőségei interaktív weblapok készítése , HTML 4**	**2.990.-**
...*Mudri István:* **Videoszerkesztés Adobe Premiere Programokkal Premiere Pro, Pro 1.5 Premier Elements**	**4.350.-**
... *Gottdank Tibor*: **WEBszolgáltatások - ◎ ◎**	**4.995.-**
... *Móricz Attila:* **WEBDESIGN a gyakorlatban**	**2.399.-**
... *Móricz A.*: **WINDOWS tippek az újratelepítéshez**	**2.900.-**
... *Gerő Judit:* **WORD 2000 I. kötet** – kezdőknek és haladóknak	**2.600.-**
... *Gerő Judit:* **WORD 2000 II. kötet** – haladóknak	**3.499.-**
... *Gerő J.–Reich G.*: **WORD for WINDOWS 6.0** – magyar & angol	**990.-**

Menedzsment informatikai szakkönyvek:

... *Szerk.:Hetyei J.*: **ERP rendszerek Magyarországon a 21. században**	**6.900.-**
... *Szerk.:Hetyei J.*: **Vezetői döntéstámogató és elektronikus kereskedelmi rendszerek Magyarországon**	**4.900.-**
... *Szerk.:Hetyei J.*: **Pénzintézetek és állami intézmények információs rendszerei Magyarországon**	**5.500.-**

Gazdasági informatika:

... *Lévayné Lakner Mária:* **EXCEL táblázatkezelő a gyakorlatban – 🖫**	**1.200.-**
... *Balogh Gábor:* **Visual Basic és EXCEL programozás – 🖫**	**1.900.-**
... *Csala Péter–Csetényi Athur–Tarlós Béla:* **Informatika alapjai**	**3.999.-**

A Weboldalunkon részletes ismertetést talál könyveinkről:
www.computerbooks.hu

Megrendelő neve

Számlázási cím

Megjegyzés