A JAVA nyelv

Bevezetés

Jellemzői:
- teljesen objektumorientált
- jelölés rendszerében hasonlít a C++ nyelvhez
- interpreteres
- byte kódú program: hordozhatóság
- böngészőkbe épített JAVA futtató: appletek
- kész csomagok használata

A JAVA program osztályok (objektumok) összessége. Az osztályok adattagokból és metódusokból állnak. Kód csak metódusokban fordulhat elő.

Egy példaprogram:

```
public class HelloVilag {
     public static void main(String[] args) {
          System.out.println("Helló világ");
     }
}
```

A fenti példaprogram tetszőleges szövegszerkesztővel megírható és célszerűen HelloVilag.java nevű file-ba elmenthető. A fordítás

javac HelloVilag.java

hatására HelloVilag.class file keletkezik, ami már JVM (java virtual machine) segítségével futtatható is:

java HelloVilag

A példaprogramból kitűnik, hogy egy osztály definíciója található benne, amelyben egyetlen metódus található. Egy osztály akkor futtatható, ha található benne publikus, statikus *main* metódus. A *public* teszi nyilvánossá, mások, pl. JVM számára is láthatóvá a metódust, míg a statikusság szükséges ahhoz, hogy a metódus ne objektum szintű legyen (hiszen akkor csak az osztály példányosítása után lehetne meghívni), hanem osztályszintű.

A JAVA nyelv elemei

Karakterkészlet

Unicode karaktereket használ, tehát bárhol, akár egy azonosítóban is használhatunk bármilyen nyelv betűjét is (pl. jó azonosító a HellóVilág is, vagy a жофи is).

Azonosítók

Betűvel kezdődő és betűvel vagy számmal folytatódó karaktersorozat. A _ és a $ is a betűk közé sorolandó. Természetesen a betűk bármelyik karakterkészletből származhatnak. Az azonosító hossza tetszőleges.
A JAVA kis- és nagybetű érzékeny (*case sensitive*), azaz az Alma != ALMA != alma.
Kulcsszó nem lehet azonosító (pl. *abstract, boolean, break, byte, case, catch*, stb.).

Megjegyzések

// egysoros megjegyzés
/* többsoros megjegyzés */
/** dokumentációs (javadoc) megjegyzés osztály, metódus és tagadat számára */
A dokumentációs megjegyzésben használható néhány javadoc számára szóló jelölés is. (Pl. *@see, @author, @version, @param, @return, @exception.*)

Egyszerű típusok

boolean:	logikai típus (*true* és *false*)
char:	16-bites Unicode karakterek
byte (8), short (16), int (32), long (64)	előjeles egészek
float (32), double (64)	lebegőpontos szám
referenciatípus:	objektumreferencia (nem mutató!)

Literálok

A **boolean** típus literáljai a true, false.
Egész számok a C nyelvben is ismert literáljai vannak: 12 decimális, 012 oktális, 0x12 hexadecimális. Egy egész konstans *int* típusú, 12l vagy 12L *long* típusú, a *short* vagy *byte* típusra nincs ilyen jelölés.
Lebegőpontos számok 12 , 12.3, 12.3e4, 12.3f, 12.3F, van 12.3d is ua. mint 12.3.
Karakter konstansok pl. ’a’, ’\n’, ’\t’.
Szöveg literálok: “szöveg”.
Objektumok: null.

Változódeklaráció

módosítók típus, azonosítólista

pl.: *int x, y;*
String ss;

A módosítókról később.

Kezdeti értékek

Változó deklarációjakor is lehet megadni.

int x = 1;

Osztályok adattagjai automatikusan iniciálódnak.

boolean	false
char	’\u0000’
egész	0
float, double	0.0
objektum	null

A lokális változók nem kapnak automatikusan kezdőértéket. Inicializálatlan lokális változóra hivatkozás, fordítási hibát eredményez.

Tömb típus

Definíció:

int b[]; vagy
int[] b;

Itt *b* egy tetszőleges elemszámú egész számokból álló tömbre hivatkozhat. Tömbobjektumok képzésére a *new* egy speciális formája használatos, ahol szögletes zárójelek között meg kell adni a tömbelemek számát. Pl.

b = new int[15];

A tömbelemek vagy egyféle primitív típusúak, vagy referenciatípusúak lehetnek.
A referenciatömbök létrehozásakor csak a referenciák jönnek létre maguk az objektumok nem! Azokat külön létre kell hozni. Pl.

```
String[] ss;
ss = new String[5];    //referenciatömb létrehozása
ss[0]=new String(“első szöveg”);      //objektumok létrehozása
ss[1]=new String(“második szöveg”);
stb.
```

Mivel minden Jáva objektum a *java.lang.Object* osztályból származik egy *Object* tömb tetszőleges típusú objektumokat tartalmazhat (pontosabban azok referenciáit) (lásd típuskonverzió, illetve értékadás):

Object[] objects = {new Button("Stopp!"), "Hellóka", null};

Tömbinicializáció

Bonyolult (akár *new* operátoros) kifejezéseket is tartalmazhat:

Button[] controls = {stopButton, new Button("Előre"), new Button("Vissza"), null};

Tömbelemekre való hivatkozás

ai[16] = 1997;

A tömbindexek 0-tól indulnak.
Ha hibás indexet adunk meg egy tömbelemre való hivatkozáskor, akkor **ArrayIndexOutOfBounds** kivételt kapunk, ami **RuntimeException** és így nem kötelező kezelni.

Többdimenziós tömbök

Tömbök tömbje. Pl.

```
Sakkfigura[][] sakktabla = new Sakkfigura[8][8];
sakktabla[0][1] = new Sakkfigura("Bástya");
sakktabla[0][2] = new Sakkfigura("Huszár");
```

Részleges tömblétrehozás

```
Button[][] buttons = new Button[10][];
buttons[0] = new Button[6];
buttons[1] = new Button[12];
```

Tömbkonstansok csak inicializáció során használhatók: A *sakktabla[0] = {new Sakkfigura("Bástya"), new ("Huszár"), ... };* utasítást a Jáva fordító nem engedi meg.
Többdimenziós tömb nem feltétlenül rektanguláris.
Többdimenziós tömbök esetén a *new*-nál megengedett, hogy a dimenziók egy része meghatározatlan maradjon. Az első dimenziót kötelező megadni. Meghatározott dimenziót nem előzhet meg meghatározatlan dimenzió.

A *java.lang* csomag nem tartalmaz explicit **Array** vagy más hasonló nevű osztályt, de ennek ellenére a tömbobjektumok is osztályokhoz tartoznak. A *String[]* típusú objektumok osztályának a neve [Ljava.lang.String, a String[][] típusúaké [LLjava.lang.String, a String[][][] típusúaké [LLLjava.lang.String, és így tovább.

```
String[][] sm5x5 = new String[5][5];
System.out.println(sm5x5.getClass().getName());
```

A tömbök öröklése követi a tömb alaptípusának öröklését:

```
Cat[] ac = new Cat[3]; Dog[][] ad new Dog[7][9];
if (ac instanceof Animal[]) {…} // true
if (ad instanceof Animal[][]) {…} // true
if (ad instanceof Dog[]) {…} // false
```

Operátorok

Kiértékelési sorrendet meghatározza:
1. Zárójelezés
2. Operátorok prioritása
3. Azonos prioritású operátorok kiértékelési sorrendje (általában balról jobbra, kivéve az értékadás)

[]	.	(kif)	kif++	kif—		postfix operátorok
++kif	--kif	+kif	-kif	!	~	prefix operátorok
new	(típus)kif					példányosítás, típuskényszerítés
*	/	%				multiplikatív operátorok
+	-					additív operátorok
<<	>>	>>>				léptető operátorok

> > <= >= instanceof	Összehasonlítások
== !=	Egyenlőségvizsgálatok
&	bitenkénti ÉS
^	bitenkénti KIZÁRÓ VAGY
|	bitenkénti VAGY
&&	logikai ÉS
||	logikai VAGY
?:	feltételes kifejezés
= += -= *= /= %= >>= <<= >>>= &= ^= |=	Értékadó

Típuskonverzió

Automatikus konverzió primitív típusokra

Értékadás lehetséges, ha a fogadó változó tartománya nagyobb.
Konstansoknál már fordítási időben kiderül lehet-e szó szűkítésről.

Automatikus konverzió objektumreferenciák esetén

Egy osztály objektumának referenciáját bármely helyen fel lehet használni, ahol valamelyik ősosztályhoz tartozó típus van a kifejezésben. Egy leszármazott mindent tud, amit az őse tudott.

Explicit konverzió

Pl.

```
float f = 12.3;
int i = (int)f;     i==12
```

Explicit konverzió objektumreferenciákra

Objektumok statikus és dinamikus típusa:
Statikus típus a deklarációban szereplő típus. Dinamikus típus a hivatkozott objektum tényleges típusa.

Egy változót értékül lehet adni egy másiknak explicit konverzióval, ha annak dinamikus típusa leszármazottja az értéket fogadó statikus típusának.

```
Konyv konyv1, konyv2;
Konyvsorozat sorozat1, sorozat2;
sorozat1 = new Konyvsorozat();
konyv1 = new Konyv();
konyv2 = sorozat1;       //implicit konverzió, érvényes
```

```
sorozat2 = (Konyvsorozat)konyv2;        //explicit
          konverzió, érvényes mert a dinamikus típusa
          Konyvsorozat
konyv2 = konyv1;          //statikus típus egyezik
sorozat2 = (Konyvsorozat)konyv2;        //explicit
          konverzió, de nem érvényes, ClassCastException
```

Az instanceof operátorral lehet megállapítani egy változó dinamikus típusa leszármazottja-e egy másik típusnak.

```
if (konyv2 instanceof Konyvsorozat) {
     Sorozat2 = (Konyvsorozat)konyv2;
}
```

Szövegkonverzió

Lásd *String* osztály.

Minősítés operátor

Alcsomagok, csomagban levő osztály, osztály egy metódusa vagy adattagja mind a „.” operátorral érhető el.
Pl.

```
java.lang.System.out.println(“Helló Világ”);
```

ahol:
- *java.lang* egy csomag neve,
- *System* a *java.lang* csomag egy osztálya,
- *out* a *System* osztály egy adattagja, egy objektum referencia,
- *println* az out objektum egy metódusa.

Vezérlési szerkezetek

Ez a rész szinte teljesen megegyezik a C++ nyelv szintaktikájával.

Utasítás, blokk

Utasítás lehet kifejezés utasítás és deklarációs utasítás. Az utasításokat pontosvessző zárja.
Kifejezés utasítás csak a következő lehet:
- értékadás,
- ++ és – operátorokkal képzett kifejezések,
- metódushívások,
- példányosítás.

A deklarációs és kifejezés utasítások tetszőleges sorrendben követhetik egymást.

Az utasítások sorozata {} jelek közé zárva a blokk. Utasítás helyére mindig írható blokk.

Elágazások

Kétirányú elágazás

Alakja:

```
if (logikai kif)
     utasítás1
else
     utasítás2
```

Az *else* ág mindig a belső *if* utasításhoz tartozik.

Többirányú elágazás

```
switch (egész kifejezés) {
     case cimke1:
          utasítások
     case cimke2:
          utasítások
     ….
     default:
          utasítások
}
```

Működése a C++ nyelvével megegyező, azaz a megfelelő címke csak belépési pont. A kiugráshoz a *break* utasítás használható.

Ciklusok

A három ciklus a C++ nyelvbelivel teljesen azonos, azaz

```
while (logikai kif.)
     utasítás


do
     utasítás
while (logikai kif.)


for (kif1; kif2; kif3)
     utasítás
```

Feltétlen vezérlésátadás

```
break [címke]
```

ha nem áll mellette címke, akkor a legbelső blokkból lép ki, ha áll, akkor a megcímkézett blokkból. Címkét bármely utasítás elé lehet tenni. Hasznos lehet egymásba ágyazott vezérlési szerkezetek esetén.

```
continue [címke]
```

a megcímkézett vagy a legbelső ciklus magjának hátralevő részét lehet átugrani.

```
return [kifejezés]
```

metódusból való visszatérés.

Osztályok

Osztálydefiníció

[módosítók] ***class*** osztályneve [***extends*** ősosztály] [***implements*** interface1[, interface2, …]] **{**
    adattagok és metódusok definíciója
}

Adattag definíciója ugyanaz mint a változó deklarációnál leírt.
Metódusok definíciója:
[módosítók] típus metódusnév(paraméterek) [***throws*** kivételek] **{**
    metódustörzs
}

Pl.

```
public class Alkalmazott {
     private String nev;
     private int fizetes;

     public void fizetestEmel(int novekmeny) {
          fizetes += novekmeny;
     }

     public boolean tobbetKeresMint(Alkalmazott masik) {
          return fizetes > masik.fizetes
     }
}
```

Az Alkalmazott osztálynak két adattagja van és két metódusa.

Az osztály adattagjaira, metódusaira az osztályon belül minősítetlenül lehet hivatkozni.
A metódusokban a *this* pszeudováltozóval lehet az aktuális példányra hivatkozni.

Hozzáférési kategóriák

- Félnyilvános: csak az azonos csomagban levő osztályok érhetik el. (Csomagokról még később.) Ilyenkor nincs módosító a definícióban.
- Nyilvános: Bármely csomagban levő bármely osztályból elérhető. Módosító a *public*.
- Privát: Más osztályból nem elérhető, de ugyanezen osztály más példányai számára elérhető. Módosító a *private*.
- Leszármazottban elérhető: Az azonos csomagban deklarált osztályok elérhetik, ezenkívül csak az adott osztály leszármazottai számára elérhető. Módosító a *protected*.

Az osztály csak nyilvános vagy félnyilvános lehet.

Egyéb módosítók

Osztályokra ezen kívül még a
abstract: tartalmazhatnak *abstract* metódusokat, nem példányosíthatók.
final: végleges osztály, nem származtatható le belőle.
módosítók alkalmazhatók.

Adattagokra:
final: konstansok
static: osztályszintű adattagok
transient: szerializálásnál
synchronized:
volatile: szálkezelésnél

Metódusokra:
abstract: törzsnélküli (öröklődés)
static: osztályszintű metódus
final: nem felüldefiniálható (öröklődés)
synchronized: szinkronizált (szálkezelés)
native: nem Java-ban megvalósított

Metódusnevek túlterhelése

Egy osztály több metódusát is elnevezhetjük ugyanúgy, ha szignatúrájuk különböző.

Példányosítás

```
Alkalmazott alk = new Alkalmazott();
```

A *new* operátor után az osztály neve áll. Zárójelben a konstruktornak szánt paraméterek találhatók.
Egy objektumra hivatkozhat több referencia is. Pl.

```
Alkalmazott masik;
masik = alk;
```

A *masik* nevű változó ugyanarra az objektumra referencia. Nem másolat.

A
```
final Alkalmazott alk = new Alkalmazott();
```
esetén nem az objektum konstans, csak a referencia. Tehát ezek után
```
alk = new Alkalmazott();      //NEM lehetséges!!!
alk.fizetestEmel(10000);      //lehetséges
```

Konstruktorok

Az objektum inicializálására szolgál. Alakja:
```
módosító Osztálynév(paraméterek) {
     törzs
}
```

Módosító csak hozzáférést szabályozó lehet. A *private* konstruktorú osztályt legfeljebb saját maga példányosíthatja pl. egy osztályszintű metóduson keresztül.
Nincs visszatérőérték típusának megadása, még *void* sincs.
Ha nem definiálunk konstruktort, akkor implicit konstruktor, amely *public*, paraméternélküli, és törzse üres.
Lehetséges konstruktorokra is a metódusnév túlterhelés.
A konstruktorok első utasításában meghívhatjuk ugyanazon osztály egy másik konstruktorát (*this(paraméterek)*), illetve az ősosztály konstruktorát (*super(paraméterek)*).

Inicializáló blokkok

Osztálydefiníción belül elhelyezett kódblokk. Lehet statikus (osztály inicializátor), amely az osztályszintű konstruktort pótolja, és lehet példányszintű, amely példányosításkor a konstruktor hívása előtt hajtódik végre (névtelen osztályokban lehet hasznos, hiszen annak nincs konstruktora).
Egy osztálynak több inicializáló blokkja is lehet.
Korlátok:
Egy inicializáló blokk nem hivatkozhat nála később definiált változókra.
Nem tartalmazhat *return* utasítást.
Osztályinicializátor nem válthat ki ellenőrzött kivételt.
Példányinicializátor csak akkor válthat ki ellenőrzött kivételt, ha névtelen osztály része, vagy ha minden konstruktor deklarálja a kivételt vagy annak valamely ősét. Implicit konstruktor esetén automatikusan deklarálódik a kivétel.

Destruktor jellegű metódusok

Nincs destruktor, az objektumok megszüntetéséért egy szemétgyűjtő algoritmus felelős. Azokat az objektumokat, amelyekre már nem hivatkozik egyetlen referencia sem, a szemétgyűjtő automatikusan kisöpri. Ha mégis szükség van az objektum megszűnésével kapcsolatban valamilyen saját kódra, akkor az *Object* osztályban definiált üres törzsű

```
protected void finalize() throws Throwable
```

metódus felüldefiniálásával kell élnünk. A *finalize* az objektum megszűnésekor még a tárterületének újrafelhasználása előtt meghívódik.
A *finalize* osztályszintű megfelelője a *classfinalize* az osztály megszűnésekor hívódik meg.

Osztályszintű tagok

Az osztályváltozó olyan változó, amely nem egyes példányokhoz, hanem az osztályhoz kapcsolódik. Egy adott osztályváltozóból egy létezik, az osztály minden egyes példánya ezen az egyen osztozik. Deklarációjában a *static* kulcsszót kell használni. Az osztályváltozókra hivatkozás ugyanúgy történhet, mint a példányváltozókra, de lehet rá hivatkozni az

osztály.osztályváltozó

alakban is.

```
public class Szamozott {
     private static int peldanySzam = 0;
     public final int SZAM = peldanySzam++;
}
```

A *Szamozott* osztály osztályváltozója csak egyszer az osztály inicializálódásakor 0 értékkel inicializálódik. A *SZAM* nevű példányváltozó (pontosabban konstans) pedig minden példányosításkor az *peldanySzam* aktuális értékével. Mivel a *peldanySzam* minden egyes alkalommal nő, így gyakorlatilag a *SZAM* az adott példány példányosítási sorszámát adja.

Az osztálymetódus egy metódus, amely az osztályhoz és nem a példányokhoz kötődik. Akkor is végrehajthatók, ha az osztálynak nincsenek példányai. Természetesen csak az osztályváltozókhoz férhet hozzá. Deklarációja szintén a *static* módosítóval történhet. Hivatkozás pedig az osztálynévvel is lehetséges.

```
public class Alkalmazott {
     private static int peldanySzam = 0;
     public final int SZAM =peldanySzam++;
     public static int peldanySzam() {
          return kovSzam;
     }
}
```

esetén az

```
Alkalmazott.peldanySzam()
```

valamint

```
Alkalmazott a = new Alkalmazott();
a.peldanySzam();
```

is lehetséges.

Ilyen, azaz osztályszintű metódus, a *main* metódus is. Ezért lehetséges meghívása még mielőtt példányosítva lenne az osztály.

Öröklődés

A szülő osztály megadása az osztálydefiníció fejlécében, az extends kulcsszóval történhet. Pl.

```
public class Fonok extends Alkalmazott {
     ….
}
```

A Fonok a gyermekosztály, az Alkalmazott a szülő.
Egy osztálynak a C++ nyelvvel ellentétben csak egy szülője lehet. Ha egy osztálynak nem adunk meg szülőt, akkor az az *Object* osztály gyermeke lesz.
A gyermek örökli a szülő tagjait kivéve a konstruktorokat. (A *private* tagokat is örökli, csak nem érheti el közvetlenül.) Az örökölt tagokon kívül definiálhat még saját tagokat is, valamint örökölt tagokat felüldefiniálhat.

A láthatóság öröklése:

Mivel egy altípusnak tudnia kell mindazt, amit az ősének, nem szűkíthető a metódusok láthatósága. Tehát a leszármazottak nem szűkíthetik a felüldefiniált metódusok láthatóságát, de bővíthetik.
Privát (és *final*) metódusokat nem lehet felüldefiniálni, így azok láthatósága nem bővíthető.

Példánymetódusok felüldefiniálása

Szükség lehet arra, hogy egy leszármazottban ugyanaz a funkció másképpen nézzen ki. Pl.

```
public class Alkalmazott {
     …
     private int nyelvekSzama;
     private int fizetes;

     public int fizetes() {
          return fizetes;
     }

     public int potlek() {
          return nyelvekSzama * 5000;
     }
```

```
        public int fizetesPotlekokkal() {
            return fizetes() + potlek();
        }
    }


    public class Fonok extends Alkalmazott {
        …
        int beosztottakSzama = 0;

        public int potlek() {
            return super.potlek() + beosztottakSzama * 1000;
        }
    }
```

A *Fonok* osztály felüldefiniálja a *potlek* metódust. A gyermekosztály *super.potlek()* hívással hivatkozni tud a szülő metódusára.
A felüldefiniált osztályokra mindig vonatkozik az úgynevezett dinamikus kötés, azaz futási időben dől el, hogy melyik metódus hívódik meg a szülő vagy a gyermek metódusa. Pl.

```
Alkalmazott a = new Fonok("Bela");
int p;
p = a.fizetesPotlekokkal();
```

A *fizetesPotlekokkal()* ugyan az *Alkalmazott* osztályban van definiálva, de mivel az „*a*” dinamikus típusa *Fonok*, ezért az ebben levő *potlek()* hívás már a *Fonok*-ben definiált *potlek()* metódust hívja.
Egy metódus felüldefiniálásához a következő feltételeknek kell teljesülnie:

- A felüldefiniáló metódus visszatérési típusának, nevének, és szignatúrájának meg kell egyeznie az eredeti metóduséval.
- A felüldefiniáló metódus csak olyan ellenőrzött kivételeket válthat ki, amilyeneket az eredeti is kiválthat, azaz definiálnia kell minden olyan kivételosztályt vagy annak ősét, amit az eredeti definiál. Ha egy metódus felüldefiniál egy olyan metódust, aminek van ***throws*** cikkelye, akkor a felüldefiniáló *throws* cikkelyében felsorolt kivételosztályoknak azonos típusúnak vagy azokból leszármaztatott típusúnak kell lenni az ősmetódus kivételeivel. A felüldefiniáló metódus nem örökli a *throws* cikkelyt, de nem is kötelező definiálnia. Ha mégis definiálna, akkor az ősmetódus *throws* cikkelyét nem bővítheti
- A felüldefiniáló metódus hozzáférési kategóriája nem lehet szűkebb az eredeti metódusénál. Egy *public* metódust csak *public* metódus, egy *protected* metódust csak *protected* vagy *public* metódus, egy félnyilvános metódust csak félnyilvános, *protected* vagy *public* metódus definiálhat felül.

Ha a feltételeknek nem tesz eleget, akkor nem felüldefiniálásról van szó, azaz a késői kötés nem él.
Statikus metódusokat nem lehet felüldefiniálni csak elfedni.

A konstruktorok nem öröklődnek. Ha egy konstruktor nem hív meg explicite más konstruktort (*this()* vagy super()), akkor egy implicit *super()* hívással kezdődik a konstruktor végrehajtása.

Absztrakt osztályok és metódusok

Absztrakt metódus, ha nincs törzse. Megvalósítást (törzset), majd csak a felüldefiniálás során kap.

```
protected abstract int akarmi();
```

Absztrakt metódusnak nem lehet módosítója a *private, final, static* hiszen ezeket nem lehet felüldefiniálni.

Absztrakt osztály, ha van legalább egy absztrakt metódusa. Absztrakt osztályt nem lehet példányosítani. Szülő osztálya lehet egy az absztrakt metódusokat felüldefiniáló osztálynak. Absztrakt osztály gyermeke lehet absztrakt, ha nem minden absztrakt metódust valósít meg.

Végleges osztályok és metódusok

A végleges, *final* módosítóval rendelkező, metódusok nem definiálhatók felül.
A végleges, *final* módosítóval rendelkező, osztályok nem terjeszthetők ki, azaz nem lehetnek szülőosztályok.

Interfészek

Az interfészek a Java nyelv másik nagy építőköve (az osztályok mellett). Az interfész egy olyan típus, amelyben csak absztrakt metódusok és konstansok szerepelnek.
Alakja:

```
módosító interface Név [extends ősinterfacek] {
     konstansok, absztrakt metódusok
}
```

A módosító *public* lehet, vagy az alapértelmezett *abstract.*
A konstansok módosítója alapértelmezés szerint *public, static, final*. Nem lehet használni a *synchronized, transient, volatile* módosítókat.
A metódusmódosító nem lehet olyan, ami kizárná a metódus megvalósítását az implementáló osztályban vagy értelmetlen. Így pl. nem lehet *native, synchronized, static, final, private, protected*. Lehet viszont *public* és *abstract*, az *abstract* alapértelmezés tehát nem kell kiírni.
Az interfészek között is létezik öröklődés. Interfészek között viszont létezik többszörös öröklődés, vagyis lehet több szülője is. Interfésznek csak interfész lehet a szülője és gyereke is.
Az osztályok implementálhatják az interfészt, azaz az összes absztrakt metódusát definiálják, megadják a törzsét. Egy osztály több interfészt is implementálhat.

Az interfész egy új referencia típust vezet be, bárhol használható, ahol egy osztály. Egy interfész típusú referencia olyan osztályreferenciákat kaphat értékül, amelyek az adott interfészt vagy annak leszármazottját implementálják (az ősinterfészt közvetetten implementálja).

Kivételkezelés

A hibák kezelésének mechanizmusa. Amikor egy metódus futása során valamilyen hiba lép fel, akkor egy kivételobjektum (exception) jön létre, mely információkat tartalmaz a kivétel fajtájáról és a program aktuális állapotáról. Ezek után megtörténik a kivétel kiváltása, azaz megdobódik a kivétel. Kivételt a program szándékosan is megdobhat a

```
throw kivételobjektum;
```

utasítással.
A kivétel kiváltása után a JVM egy olyan helyet keres a programban, ahol a kiváltott kivétel kezelése megtörténhet. A kivétel kezelését az a kivételkezelő blokk fogja végezni, amely megfelelő típusú (típusa megegyezik a kiváltott kivétel típusával vagy annak őse), és amelynek a hatáskörében keletkezett a kivétel. Egymásba ágyazott kivételek esetén kifelé haladva az első megfelelő kezeli le a kivételt. A megfelelő kivételkezelő megtalálását a kivétel elkapásának nevezzük (catching exception). A kivételkezelő nem feltétlenül van abban a metódusban, amelyben a kivétel keletkezett, a hívási fa alján is elhelyezkedhet, azaz pl. a hívó metódusban vagy az azt hívóban. A kivétel lekezelése után a kivételkezelő kódblokk utáni utasításon folytatódik a végrehajtás.

A Java nyelvben a kivételek mind az *Exception* osztályból származnak. Két fajtájuk van azok, amelyeket kötelező lekezelni (pl. *IOException*) és azok, amelyeket nem (pl. *RuntimeException*).

Kivételkezelő kódblokk:

```
try {
     utasítások
}
catch (kivétel) {
     utasítások
}
catch (kivétel) {
     utasítások
}
finally {
     utasítások
}
```

Azokat az utasításokat, amelyekben kivétel megdobódhat *try* blokkban kell elhelyezni. A *catch* blokkok a *try* blokkban keletkező típusuknak megfelelő kivételeket kezelik le. A *finally* blokk pedig mind normál végrehajtás, mind kivétel megdobódása esetén végrehajtódik a végén.
A *catch* ágak sorrendje nem mindegy, hiszen az első olyan *catch* blokk elkapja a kivételt, amely típusa egyező a kiváltott kivétellel vagy őse annak. Tehát pl.

```
try {
     utasítások
}
catch (IOException e) {
```

```
        utasítások
    }
    catch (Exception e) {
        utasítások
    }
```

esetén a első *catch* blokk minden olyan kivételt elkap, ami IOException vagy annak leszármazottja (EOFException, FileNotFoundException, InterruptedIOException, UTFDataFormatException, stb.). A második *catch* blokk pedig minden kivételt elkap (amit nem kapott el előle az előző *catch*), hiszen minden kivétel a *Exception* osztály leszármazottja. Ha a *catch* ágak egyike sem tudja elkapni a kivételt, akkor a beágyazó kivételkezelő blokkban folytatódik a keresés. Ha egyáltalán nem talál megfelelő *catch* blokkot akkor a program befejeződik.
A kivétel csak akkor képes metóduson keresztül is felvándorolni, ha a metódus jelzi, hogy belsejében megdobódhat a kivétel és nincs lekezelve:

```
típus metódusnév(paraméterek) throws kivételosztályok
```

A fordító már fordítási időben leellenőrzi, hogy a kötelezően lekezelendő kivételek le vannak-e kezelve, hogy *try* blokkban nincs-e olyan lokális változó deklarációja illetve inicializálása, amelyre a *try* blokk után is hivatkozunk (hiszen akkor nem biztos, hogy ráfut arra a sorra).

Beágyazott osztályok

A Java 1.1 verziója óta élő lehetőség. Nem csak a program legkülső szintjén lehetséges osztályok definiálása, hanem más osztályokba, utasításokba ágyazva is.
A beágyazott osztályok hozzáférhetnek a befoglaló osztály privát tagjaihoz is. Másfelől a beágyazott osztály definíciójának hatáskörét leszűkíthetjük a befoglaló osztályra, utasításblokkra vagy egyetlen példányosítás pontjára.
A beágyazott osztályoknak négy fajtája van a

- statikus tagosztály,
- nem statikus tagosztály,
- lokális osztály,
- névtelen osztály.

Statikus tagosztály

A befoglaló osztály tagjaként definiáljuk, az egyéb tagok között. Pl.

```
public class Lista {
    private Elem elso;

    private static class Elem {
        Object adat;
        Elem elozo, kovetkezo;

        Elem(Object adat, Elem elozo, Elem kovetkezo) {
            this.adat = adat;
            this.elozo = elozo;
            this.kovetkezo = kovetkezo;
        }
```

```
        }

        public void beszur(Object adat) {
              elso = new Elem(adat, null, elso);
              if (elso.kovetkezo != null)
                    elso.kovetkezo.elozo = elso;
        }

        public void torol(Object adat) {
              Elem elem = keres(adat);
              if (elem != null)
                    torol(elem);
        }

        private void torol(Elem elem) {
              if (elem == elso)
                    elso = elem.kovetkezo;
              if (elem.elozo != null)
                    elem.elozo.kovetkezo = elem.kovetkezo;
              if (elem.kovetkezo != null)
                    elem.kovetkezo.elozo = elem.elozo;
        }

        private Elem keres(Object adat) {
              for (Elem elem=elso; elem != null; elem = elem.kovetkezo)
                    if (elem.adat.equals(adat))
                          return elem;
              return null;
        }
  }
```

A tagosztály statikus voltát a *static* módosító jelzi. Ezen kívül a hozzáférést szabályozó *public, protected, private* módosítója is lehet. Ezek természetesen csak a befoglaló osztályon kívülről történő hivatkozáskor érvényesek, hiszen a befoglaló osztály és a tagosztályok kölcsönösen hozzáférnek egy más tagjaihoz.
Egy statikus tagosztály alkalmazási területei:

- Egy segédosztályt el akarunk rejteni a külvilág elöl (lásd a fenti példa).
- Egy osztály megvalósításakor egy olyan segédosztályra van szükség, amelyeknek hozzá kell férnie az osztály privát tagjaihoz.
- Ki akarjuk fejezni, hogy egy osztály vagy *interface* egy másiknak logikai alárendeltje. Pl. a *java.util* csomagban levő *Map* interfész és annak *Entry* taginterfésze.

Nem csak osztálynak lehetnek tagosztályai, hanem interfésznek is lehetnek taginterfészei. A tagosztályok egymás *private* tagjaihoz is. A tagosztályok öröklődnek.

A tagosztályokra lehet hivatkozni más osztályokból az osztálynév minősítéssel. Akár szülő osztálya is lehet nem tagosztálynak.

Nem statikus tagosztályok

A példányaik a befoglaló osztály egy példányához kötődnek. A tagosztály példányosításakor éppen aktuális példány lesz a befoglaló példány. Ha a példányosítás pontján a befoglaló

osztálynak nincsen aktuális példánya, vagy ha attól eltérő befoglaló példányt akarunk, akkor a *new* operátort minősíteni kell:

```
példányváltozó.new Tagosztály();
```

Az aktuális példányra hivatkozó *this* változót is minősítéssel lehet pontosítani, hogy melyik aktuális példányra a befoglaló vagy a beágyazott példányra vonatkozik.
A nyelv megengedi a többszörös egymásba ágyazást is.
A nem statikus tagosztály is lehet szülő. Természetesen a beágyazónak is elérhetőnek kell lennie.

```
public class Lista {
	…

	private class Felsorolo implements Iterator {
		Elem aktualis = elso;
		Elem torolheto = null;

		public boolean hasNext() {
			return aktualis != null;
		}

		public Object next() throws NoSuchElementException {
			if (aktualis != null) {
				torolheto = aktualis;
				Object adat = aktualis.adat;
				aktualis = aktualis.kovetkezo;
				return adat;
			}
			else {
				throw new NoSuchElementException();
			}
		}

		public void remove() throws IllegalStateException {
			if (torolheto != null) {
				torol(torolheto);
				torolheto = null;
			}
			else {
				throw new IllegalStateException();
			}
		}
	}


	…
	public Iterator felsorol() {
		return new Felsorolo();
	}
}
```

A *Lista* objektum *felsorol()* metódusát meghívva példányosodik a *Felsorolo*. A létrejövő példány a *Lista* osztály aktuális példányához kapcsolódik. Az *aktualis* változó már ennek az *elso* változójának értékét veszi fel.

Lokális osztályok

Olyan osztályok, amelyek hatásköre csak az őket definiáló utasításblokkra terjed ki. Hatókört befolyásoló módosítókat tehát értelemszerűen nem alkalmazhatunk lokális osztályokra.
A lokális osztályok technikai okokból csak olyan lokális változókra, formális paraméterekre hivatkozhatnak, amelyek *final* módosítóval rendelkeznek és már inicializáltak az osztály definíciója előtt. Mivel a lokális osztály példányai tovább élhetnek a lokális változókról másolatokat őriznek (zárványok).
Attól függően, hogy a lokális osztály statikus vagy nem statikus kódblokkba van beágyazva a lokális osztálynál is megkülönböztethetünk statikust és nem statikust.
Ha pl. a fenti példában a *Felsorolo* osztály definícióját áthelyezzük a *Lista* osztály *felsorol()* metódusába, akkor máris egy lokális osztályról van szó.

Névtelen osztályok

Névtelen osztály úgy keletkezik, hogy egy már létező osztályt a példányosító kifejezéshez függesztett osztálytörzzsel kiterjesztünk, vagy egy interfészt hasonló módon implementálunk.

```
new Tipus(par.-ek) {osztálytörzs}
```

A névtelen osztályoknak nem lehet konstruktora, inicializáló blokkja viszont igen.

Csomagok

A Java-ban az osztályok csomagokban helyezkednek el. A csomagok

- lehetőséget nyújtanak egyfajta strukturálásra,
- külön névtérrel rendelkeznek, így elkerülhetőek az egyező típusnevekből eredő problémák,
- a hozzáférési kategóriák egyik eszköze.

A csomagok hierarchikus szerkezetet alkotnak, vagyis egy csomag alcsomagokat tartalmazhat, amelyeknek szintén lehetnek alcsomagjai. Egy alcsomag teljesen egyenrangú a hierarchián fentebb elhelyezkedő csomagokkal. Egy csomag alcsomagja mind hozzáférés, mind névtér szempontjából ugyanúgy idegen csomag, mint bármely más csomag.
Egy adott csomagban elhelyezkedő típus (osztály vagy interfész) hivatkozása

csomag.típus

ha a csomag egy alcsomag, akkor a csomag neve is

csomag.alcsomag

alakú. Pl. a *java* csomagban levő *util* alcsomagban elhelyezkedő *BitSet* osztály teljes megnevezése *java.util.BitSet*.

Egy fordítási egység csak egy csomaghoz tartozhat még akkor is, ha abban esetleg beágyazott osztályok is vannak. A *package* utasítással lehet deklarálni, hogy az adott fordítási egység melyik csomagnak a része A *package* utasításnak a fordítási egység elején kell elhelyezkednie.

```
package gyumolcs.alma;
```

Ha egy fordítási egységben nem deklaráljuk, hogy melyik csomagban helyezkedi el, akkor az egy névtelen csomaghoz fog tartozni. A névtelen csomagok használata nem javasolt mert a csomagok leképzésétől függően problémák adódhatnak belőle.

Importdeklaráció

Az import deklaráció lehetővé teszi egy típusra, hogy teljes megnevezés helyett csak névvel hivatkozhassunk rá. Az import deklarációnak a fordítási egység elején a package utasítás után kell elhelyezkednie.

```
import java.util.BitSet;
```

esetén a programban elég csak *BitSet*-ként hivatkozni rá.
Lehetséges egyszerre egy csomag összes publikus típusát importálni:

```
import java.util.*;
```

Ez nem importálja viszont a csomag alcsomagjaiban levő típusokat (pl. a java.util.zip csomagban levőket).
Minden fordítási egység automatikusan importálja a java.lang csomag típusait.

Csomagok leképzése
A csomagokban levő fordítási egységek tárolására több mód is van:

- filerendszerben
- filerendszer betömörítve
- adatbázisban

A filerendszerben tárolt csomagok esetén könyvtár hierarchiára képződnek le a csomagok. Tehát a *java* csomag fordítási egységei a *java* nevű könyvtárban találhatók, a *java.util* csomag egységei a *java* könyvtárban levő *util* könyvtárban, stb. Magának a java könyvtárnak a szülő könyvtárát, illetve más kiinduló csomagok könyvtárának a szülőkönyvtárát a CLASSPATH nevű környezeti változó kell tárolja. Lehet több szülőkönyvtár is, ilyenkor a CLASSPATH-ban mindet fel kell sorolni pontosvesszővel elválasztva.
A filehierarchia zip file-ba össze is csomagolható. A CLASSPATH-ban ilyenkor a zip file elérési útját kell megadni. Pl.

```
CLASSPATH=/java/myutil.zip;/home/sajatcsomagok
```

esetén a csomagokat a zip file-ban és a */home/sajatcsomagok* könyvtárban keresi.

Java programok fordítása, futtatása

Forrás file-ok

A forrás file neve mindig a file-ban szereplő egyetlen publikus típus (osztály, vagy interfész) nevéből és a *.java* kiterjesztésből áll. A forrás file egy szöveges file.
Betartandó konvenciók:

- Az azonosítók neve mindig kisbetűvel kezdődik.
- Típusok neve mindig nagybetűvel kezdődik.
- Minden névben a szóösszetevők nagybetűvel kezdődnek.
- Konstansok neve csupa nagybetűből áll.
- Blokk-kezdő { mindig az elé tartozó utasítások után, a sor végén áll.
- Blokk-záró } mindig a hozzátartozó blokk-kezdő { előtt álló utasításokkal egy szinten, a sor elején áll.

A forrás file-t fordítóval (javac) lehet lefordítani bájtkódú programmá.

A .class *(bájtkódú) file-ok*

Futás közben egy típusra történő első hivatkozáskor (dinamikusan) töltődik be az adott típushoz tartozó *.class* file. A file kereséséhez használt útvonalak

- alap JDK osztályok útvonala
- JDK-t kiegészítő alaposztályok útvonala
- CLASSPATH-ban megadott vagy a –classpath kapcsolóval megadott útvonalak.

A betöltést a *java.lang.ClassLoader* osztály felhasználásával lehet vezérelni.
A betöltött bájtkód csak azután kaphatja meg a vezérlés, hogy a bájtkód ellenőrző ellenőrizte azt. Ezen ellenőrzés biztosítja, hogy futás közben

- nem lép fel veremtúlcsordulás,
- a JVM regiszterkezelése helye,
- a bájtkód utasítások paraméterei helyesek,
- nincs illegális adatkonverzió.

A Java interpreterek a gyorsabb futtatás érdekében gyakran használják a dinamikus fordítási technikát (JIT: Just in Time). Ez azt jelenti, hogy a dinamikus betöltés után az osztály kódját lefordítják natív kóddá a gyorsabb futás érdekében.

Applet

Az applet egy HTML oldalba ágyazható program. Az appletek alapértelmezett betöltője a *sun.applet.AppletClassLoader*. Az applet bájtkódja hálózaton keresztül is letölthető, ezért a betöltendő típus keresése

- az appletet megjelenítő program keresési útvonalai alapján,
- az applethez megadott archívumokban,
- az applet kódját tartalmazó helyen.

Az applet beágyazása a HTML oldalba az <APPLET> kulcsszóval lehetséges. Ennek paraméterei lehetnek:

- CODEBASE = url az applet kódját tartalmazó URL címe, ha nincs megadva, akkor az appletet tartalmazó HTML dokumentum könyvtárának URL címe kerül felhasználásra
- ARCHIVE=archívum1, az applet kódját és erőforrásfile-jait tartalmazó archívumok nevei. A használata esetén a megadott archívumok egyszer automatikusan letöltődnek, majd dinamikus típusbetöltéskor, vagy kép- és hangfile betöltésekor ezen archívumokban is keresi a file-t. A Java archívum egy zip file.
- CODE=fáljnév az applet kódját tartalmazó, CODEBASE-hez relatív bájtkód-file neve.
- OBJECT=objektumnév az appletet szerializált formában tartalmazó file neve.
- ALT=szöveg megjelenítendő szöveg, ha a böngésző nem képes grafikus megjelenítésre.
- NAME=név a HTML-beni hivatkozási neve az appletnek.
- WIDTH, HEIGHT, ALLIGN, VSPACE, HSPACE az elhelyezkedésével kapcsolatos értelemszerű paraméterek.
- MAYSCRIPT engedi az applet kommunikációját a JavaScriptel.

Az <APPLET> </APPLET> között még lehetnek
<PARAM NAME=név VALUE=érték>
sorok is, amelyekben deklarált paramétereket az applet az *Applet* osztály

```
public String getParameter(String paramnév)
```

metódusával lehet lekérdezni.
Ezenkívül lehet még tetszőleges HTML tag is, amelyek akkor hajtódnak végre, ha a böngésző nem képes appleteket megjeleníteni.

Az applet életciklusa:

A böngésző külön szálon végzi az applet kezelését. Ez a programszál a konstruktor végrehajtása után a következő négy metóduson keresztül vezérli az applet futását:

- *public void init()* az applet inicializálásakor kerül végrehajtásra. Az applet konstruktorának lefutása után egyből meghívódik. Itt érdemes a paramétereket átvenni. Csak ezen metódus végrehajtása után hívódnak meg először a megjelenítés metódusai.
- *public void start()* az applet elindításakor vagy újraindításakor kerül végrehajtásra. Közvetlenül a meghívása előtt az applet aktív állapotba kerül. Kezdetben az *init()* metódus után, majd mindig akkor hívódik meg, amikor az appletet újra kell indítani. Ez akkor következhet be, ha például a böngészőben visszatérünk az appletet tartalmazó HTML laphoz, vagy ha a böngésző visszakerül eredeti méretébe ikonizált állapot után, vagy egyszerűen csak az applet láthatóvá válik.
- *public void stop()* az applet megállításakor kerül végrehajtásra. Közvetlenül a meghívása előtt az applet inaktív állapotba megy át. Mindig akkor hívódik meg, ha az appletnek nem kell tovább futnia. Ez akkor következhet be, ha például a böngészőben elhagyjuk az appletet tartalmazó lapot, vagy ha a böngészőt ikonizáljuk, vagy egyszerűen csak az applet már nem látható. Itt érdemes az applet saját *Thread* objektumait leállítani.
- *public void destroy()* az applet megszüntetésekor kerül végrehajtásra. Mindig a *finalize()* előtt lesz meghívva, amelyet viszont egy külön rendszerszál hajt végre. Itt kell az applet által még lefoglalt erőforrásokat felszabadítani, például a kommunikációs vagy I/O végpontokat lezárni.

A *java.lang* csomag

A java.lang csomag azokat az alapvető típusokat definiálja, amelyekre a programok futtatásához szükség van. Éppen ezért a csomag automatikusan importálódik minden programba.

Az Object osztály

Az *Object* osztály minden osztály közös őse, így metódusait minden osztály örökli.
A metódusainak egy része *final* azaz nem felüldefiniálható. Ilyen a szálkezelésnél használt *wait, notify, notifyAll*, valamint a *getClass* metódus.
Az *Object* osztály felüldefiniálható függvényei közül a legfontosabbak a *toString, equals, clone, hashCode,* és a *finalize*.

A *toString()* az objektumok stringgé konvertálásakor hívódik (hívható) meg. *Object* osztálybeli definíciója szerint:

```
return getClass().getName() + "@" +
Integer.toHexString(hashCode());
```

Az *equals()* metódust feltétlenül felül kell definiálni, ha két (akár eltérő típushoz tartozó) objektum tartalma azonosságának megállapítására akarjuk felhasználni. Az *Object.eqals(Object obj)* implementációja:

```
return (this == obj);.
```

A *clone()* metódus másolatot készít az objektumról. Az *Object* osztálybeli definíciója szerint a mezőket másolja át. Ha egy mező referencia, akkor a referencia másolódik, ilyen esetben célszerű felüldefinálni. A metódus megdobja a *CloneNotSupportedException* kivételt, ha a klónozás nem támogatott az osztály számára. A klónozás akkor támogatott, ha az osztály implementálja a *Clonable* interfészt. (A *Clonable* szintén a *java.lang* csomag része és egyébként teljesen üres interfész csak megjelölő szerepe van.)

A *hashCode()* metódus állítja elő az objektumok hash kódját, ami akkor lehet szükséges, ha az objektumot hashtáblában tároljuk. A hash kódnak egy int típusú számnak kell lennie, amelyre:

- Ha egy objektumra többször is meghívjuk a *hashCode()* eljárást, akkor mindig ugyanazt a számot kell, hogy adja (feltéve, hogy közben nem változott meg az objektum).
- Ha két objektum *equals()* szerint egyenlő, akkor a hash kódjuknak is egyenlőnek kell lennie.
- Nem egyenlő objektumokra lehetőleg eltérő legyen.

A *finalize()*-ről az objektum törlési folyamata során már volt szó.

A Class osztály

Minden Jáva osztályt futásidőben egy *java.lang.Class* típusú objektum reprezentál. Valójában ennek az objektumnak a feladata az osztály instanciáinak a képzése. Egy egyszerű programban többnyire nincs rá szükség. A *Class* osztály *final* minősítésű, azaz nem származtatható le belőle, valamint a konstruktora *private*, azaz nem példányosítható általunk. A funkciói pl.

- Az osztály nevének lekérdezése getName()
- A szülő osztály lekérdezése getSuperClass()
- Osztály vagy interfész eldöntés isInterface()
- Implementált interfészek getInterfaces()
- Dinamikus oszály betöltés forName()
- Dinamikus instancia képzés newInstance()

A *newInstance()* a default konstruktort hívja. Az **InstantiationException** akkor fordul elő, ha egy absztrakt osztály példányát akarjuk létrehozni; az **IllegalAccessException** akkor, ha az osztály konstruktora nem elérhető (privát vagy védett).

```
Cat c;
try {
     c = (Cat)(Class.forName("Cat").newInstance());
```

```
}
catch(ClassNotFoundException e){…}
catch(InstantiationException e){…}
catch(IllegalAccessException e){…}
```

Ezeken a metódusokon kívül még az összes olyan metódus megtalálható az osztályban, amelyekkel a osztály teljes egészében feltérképezhető (metódusai, mezői, láthatóságai, stb.). Ezeket a *java.lang.reflect* csomag segítségével végzi.

Elemi típusok fedő osztályai

Feladatuk elemi típusok osztályba csomagolása. Minden elemi típusra létezik neki megfelelő osztály. Ezek az osztályok mind *final* módosítójúak.
Létezik:

Byte, Short, Integer, Long, Float, Double, Character, Boolean, Void

osztály. A számtípusoknak egy közös őse van a *Number* osztály.

Elemi típusból objektum

Legegyszerűbben a konstruktor segítségével hozhatjuk létre. Pl.

```
int i=23;
Integer j = new Integer(i);
```

Objektumból elemi típus

A *Character* osztálynál:

charValue()

A Boolean osztálynál:

booleanValue()

A numerikus osztályoknál bármelyik elemi típusban meg lehet kapni az értéket. Tehát mindegyiknél létezik:

byteValue()
shortValue()
intValue()
longValue()
floatValue()
doubleValue()

Stringből objektum

A *Character* osztályt kivéve, mindegyiknek van *String* paraméterű konstruktora. Ezenkívül minegyik osztálynak van statikus *valueOf()* metódusa, amely Stringből képes objektumot készíteni

Stringből elemi típus a fedőosztályon keresztül

Használva a Stringből objektumot módszer valamelyikét, majd a megfelelő elemi típus lekérdezést. Pl.

```
int i = new Integer(”35”).intValue();
```

vagy

```
int i = Integer.valueOf(”35”).intValue();
```

A *Byte, Short, Integer, Long* típusoknál lehetőség van arra, hogy egy nem tízes számrendszerben megadott számstringből konvertáljunk a megfelelő elemi típusra a statikus

parseXxxxx(String)

metódussal, ahol Xxxx lehet *Int, Byte, Short, Long*. Pl.

```
int i = Integer.parseInt(”64”,16);
```

esetén az i értéke 100 lesz, mert a 64-t mint 16-os számrendszerbeli számot értelmezi.

Objektumból String

Mint minden objektumtípusnál a fedőosztályok objektumánál is a *toString()* metódus használható erre.

Elemi típusból String

Megoldható

- String konstruktorokkal, lásd String osztály.
- Fedőobjektum készítése után *toString()*
- A fedőosztályok statikus *toString(xxx)* metódusával, ahol xxx a megfelelő elemi típus.

Az Integer és a Long osztályoknak létezik statikus *toString(xxx, int)* metódusa is, amelyben a számrendszer is megadható. Pl.

```
String s = Integer.toString(22, 3)
```

eredménye a ”211”.

Az Integer és a Long osztályoknak létezik statikus *toBinaryString(xxx), toOctalString(xxx), toHexString(xxx)* metódusa is.

Numerikus fedőosztályok konstansai

A *Byte, Short, Integer, Long* osztályok definiálják a MIN_VALUE és a MAX_VALUE konstansokat, amelyek értelemszerűen a legkisebb és a legnagyobb ábrázolható számot jelentik. A *Float* és *Double* osztály MAX_VALUE konstansa a legnagyobb ábrázolható számot, a MIN_VALUE viszont a legkisebb pozitív számot jelenti. A *Float* és *Double* osztályban van még NaN, POSITIVE_INFINITY, NEGATIV_INFINITY nevű konstans is, amelyek a „nem szám”, a pozitív, és a negatív végtelent jelentik. A végtelenekkel műveletet is lehet végezni. A pozitív és a negatív végtelen összege NaN, azaz nem értelmezett. Két pozitív végtelen különbsége szintén NaN.

Property kezelési metódusok

A fedőosztályoknak léteznek statikus *getXxxxx()* metódusai is. Ezek segítségével lehet megfelelő típusú property eredményét lekérdezni. Pl. a *Integer.getInteger(String)* metódussal a paraméterként megadott nevű *int* értékű property értékét lehet megkapni *Integer*-ként. Ha nem létező vagy nem *int* értékű property nevét adom meg, akkor az eredmény *null* konstans lesz.

A String osztály

A String osztály egy *final* osztály, amely csak konstans stringeket képes kezelni. Minden módosítás új String objektumot hozz létre.
A Java minden string literálból is generál egy String objektumot. Ezért string literálok esetén is hívhatók a String tagfüggvények. Pl.

```
int hossz = ”alma”.length();
```

Konstruktorai:

```
String()
String(String)
String(StringBuffer)
String(char[])
String(char[], int tól, int ig)
String(int tól, int ig, char[])
String(byte[])
String(byte[], int u)    deprecated
String(byte[], int tól, int ig)
String(byte[], int u, int tól, int ig)  deprecated
String(byte[], String kód)
String(byte[], int tól, int ig, String kód)
```

Byte tömb esetén az „u” érték a karakter felső byte-ját adja, csak a korábbi verziókkal való kompatibilitás érdekében maradt meg. A byte tömbök esetén a String típusú paraméter a karakter kódolást jelzi. Pl.

```
byte []bytetomb = {65, 66, 67, 68, 69};
String ss = new String(bytetomb, ”ISO8859_1”);
```

esetén *ss* értéke ”ABCDE”.

Leggyakrabban használatos String létrehozási módszer a

```
String s1 = new String(”szöveg”);
```

vagy a

```
String s2 = ”szöveg”;
```

String-ek összefűzése

A String-ekre működik + operátor.

```
String s3 = ”almafának ” + 4 + ” ága van”;
```

esetén az *s3* értéke ”almafának 4 ága van” lesz.

Automatikus konverzió String típusra

Bármely típus képes automatikusan konvertálódni String típusra. Az elemi típusok értelemszerűen (lásd pl. a fenti példa), az objektumok a *toString()* metódusuk segítségével.

Konvertálás tetszőleges típusból String-é

A String osztály statikus *valueOf()* metódusának különböző alakjai segítségével bármely típus konvertálható String típusra. Létezik *boolean, char, double, float, int, long, char[], Object* paraméterű változat is. A *char[]* esetén a *valueOf()* nem készít másolatot a tömbről, hanem

felhasználja az objektumban, így a tömb megváltozása a String objektum tartalmát is megváltoztatja. A *copyValueOf()* használandó, ha ezt el szeretnénk kerülni.

Konvertálás String-ból tetszőleges típusba

Elemi típusokra a fedő osztályok címszónál láthatjuk a megoldást.
A karakter tömbbé konvertálás a *getChars()* vagy a *toCharArray()* metódussal lehetséges. A *getChars()* már meglevő tömb egy részébe másolja a String karaktereinek egy részét.

```
ss.getChars(3, 6, karaktertomb, 2);
```

esetén az *ss* String 3., 4., 5. karakterét másoltuk át a tömb 2., 3., 4. elemébe.
A *toCharArray()* a String egészét képes egy általa létrehozott tömbbe másolni.

```
char sstomb[] = ss.toCharArray()
```

A byte tömbbé konvertálás a *getBytes()*, vagy *getBytes(String kódolás)* metódussal történhet.

Műveletek String-ekkel

Egy karakterének lekérdezése

```
char charAt(int index)
```

Egyezőség vizsgálat

```
boolean equals(String masik)
boolean equalsIgnoreCase(String masik)
```

Összehasonlítás, az eredmény negatív ha az aktuális példány a kisebb, pozitív ha a másik, és 0 ha egyeznek

```
int compareTo(String masik)
```

Részek összehasonlítása

```
boolean regionMatches(int honnan, String masik, int
     masikhonnan, int masikhossz)
boolean regionMatches(boolean noCaseSens, int honnan,
     String masik, int masikhonnan, int masikhossz)
```

Keresés a Stringben, az eredmény a találat kezdetének indexe, illetve -1 na nincs találat.

```
int indexOf(char mit)
int indexOf(char mit, int honnan)
int indexOf(String mit)
int indexOf(String mit, int honnan)
```

Keresés visszafele indexOf() helyett lastIndexOf().
String végének egyezősége

```
boolean endsWith(String veg)
```

String kezdetének egyezősége

```
boolean startsWith(String kezdet)
```

String hossza

```
int length()
```

Konvertálás kisbetűssé, nagybetűssé

```
String toLowerCase(String mit)
String toUpperCase(String mit)
```

Adott karaktereinek lecserélése

```
String replace(char mit, char mire)
```

Szóköz jellegű karakterek eltüntetése a String elejéről és végéről

```
String trim()
```

Rész kimásolása

```
String substring(int honnan)
String substring(int honnan, int mennyit)
```
String hozzáfűzése (mint a + operátor)
```
String concat(String masik)
```

A StringBuffer osztály

Változtatható hosszúságú és tartalmú string kezelésére szolgál.
A StringBuffer egy jellemző adata az aktuális kapacitás. Ez megadja, hogy milyen hosszú szöveget képes tárolni anélkül, hogy növelnie kellene a saját kapacitását.
A String-eket manipuláló műveletek is StringBuffer-ekkel vannak megvalósítva. Például a
`String s = ”alma”+12+”körte”`
hatására az
String s = new StringBuffer().append(”alma”).append(12).append(”körte”)
utasítás hajtódik végre.

Konstruktorai:

- StringBuffer(), üres string, egy alapértelmezett (16) kapacitással
- StringBuffer(int kapacitas), üres string, megadott kezdő kapacitással.
- StringBuffer(String s), ahol s String-el kezdőértékkel jön létre a StringBuffer és a kapacitás s.length()+16 lesz

Metódusai:

Hosszának lekérdezése:
int length()
Kapacitás lekérdezése:
int capacity()
Karakterének lekérdezése:
char charAt(int index)
Kapacitás biztosítás, (if (minkap<régikap) újkap=(régikap+1)*2; else újkap=minkap;):
void ensureCapacity(int minkap)
Rész kimásolása karakter tömbbe:
void getChars(int kezdet, int vég, char cel[], int celkezdet)
A karaktertömb kinyerése:
char[] getValue()
Beszúrás (boolean, char, double, float, int, long, char[], char[] részlet, Object):
StringBuffer insert(int hova, XX mit)
Hozzáfűzés (boolean, char, double, float, int, long, char[], char[] részlet, Object):
StringBuffer append(XX mit)
Megfordítás:
StringBuffer reverse()
Karakterének módosítása:
void setCharAt(int index, char mire)
Hossz beállítása (ha hossz < regihossz akkor levágja a fölösleget):
void setLength(int hossz)
String-re konvertál:
String toString()

A Math osztály

A *Math* osztály egy *final* osztály, amelynek konstruktora *private*, így nem származtatható le belőle és nem lehet példányosítani. Tartalmazza a fontosabb matematikai konstansokat és függvényeket. Minden tagja statikus, így példányosítás nélkül használható.
Konstansok: **E** és **PI**
Metódusok:

abs(),
acos(), asin(), atan(), atan2(), cos(), sin(), tan(),
log(), sqrt(), exp(), pow(),
max(), min(),
random(),
rint(), round(), ceil(), floor(),

Hiba, kivétel osztályok

A java.lang csomagban definiáltak a főbb kivételosztályok.

Szálkezeléssel kapcsolatos osztályok

Majd a szálkezelésnél.

Egyéb rendszerszintű osztályok

ClassLoader: absztrakt osztály, az osztálybetöltéséért felelős.
Process: absztrakt osztály, alprocesszek kezelése
SecurityManager: absztrakt osztály, biztonságkezeléssel kapcsolatos (lásd Biztonság fejezet)
Runtime: a futtató környezettel való kapcsolattartás (parancsvégrehajtás, kilépés, dinamikus library kezelés, szabadmemória, stb.)
Compiler: final osztály, natív kódra fordításhoz
System, *final* osztály, néhány hasznos metódus. Pl.

- *in, out, err* csatornák,
- *arraycopy()*, tömbrészlet másoló
- *exit()*, kilépés
- *gc()*, szemétgyűjtő meghívása
- *getenv()*, környezeti változók lekérdezése
- *getProperties(), getProperty()*, rendszer property-k lekérdezése (pl. java installációs információk, OS információk)
- *load(), loadLibrary()*, dinamikus library betöltése
- *setProperty()*, property beállítás
- stb.

A *java.util* csomag

A java.util csomag hasznos osztályok gyűjteménye. Megtalálható benne:

Stringek részekre bontásához (StringTokenizer)
Véletlenszám generálásához (Random)
Dátumok kezeléséhez (Date, TimeZone, SimpleTimeZone, Calendar, GregorianCalendar)
Objektumok figyeléséhez (Observer, Observable)
Awt eseménykezeléséhez (EventListener, EventObject) (lásd Awt)

szükséges osztályok és interfészek, és egy

Kollekció gyűjtemény rendszer (pl. Vector, HashTable, Array, LinkedList, stb.)

Szövegek részekre bontása, a StringTokenizer osztály

String-ek határolóelemekkel (pl. space, újsor, tabulátor, vagy bármi más) elválasztott elemekre (tokenekre) bontását szolgálja. Megvalósítja az *Enumeration* (lásd gyűjtemények) interfészt.

Konstruktorai:

StringTokenizer(String szöveg)
StringTokenizer(String szöveg, String határolók)
StringTokenizer(String szöveg, String határolók, boolean határolóTokenE)

A *határolók* alapértelmezése a *” \n\t”*, azaz space, újsor, tabulátor.
A *határolóTokenE* alapértelmezése *false*, azaz a határoló nem számít tokennek.

Metódusai:

Hátralevő elemek száma:

int countTokens()

Van-e még token:

boolean hasMoreElements()
boolean hasMoreTokens()

Következő elem:

Object nextElement()
String nextToken()
String nextToken(String EzUtánEzLegyenAzÚjHatárolók)

Példa:

```
StringTokenizer st = new StringTokenizer(
    "a = 123;b = TYW"," =;")
while(st.hasMoreTokens()) {
  String t = st.nextToken();
  …
}
```

A „t” értéke rendre „a”, „123”, „b”, „TYW”.

Véletlenszám generálás, a Random osztály

Véletlenszámok sorozatának generálására használható.

Konstruktorai:

Random()
Random(long inic)

Az inic az inicializáláshoz szükséges szám. Paraméter nélkül a pillanatnyi időt használja fel az inicializálásához.

Metódusai:

Következő véletlenszám:

int **nextInt**()	Integer.MIN_VALUE - MAX_VALUE
long **nextLong**()	Long.MIN_VALUE - MAX_VALUE
float **nextFloat**()	0.0 - 1.0f
double **nextDouble**()	0.0 - 1.0d, (ua. mint a Math.Random())
void **nextBytes**(byte[] számok)	feltölti a tömböt véletlenszámokkal
double **nextGaussian**()	Gauss-eloszlású véletlenszám

Inicializáló szám megadása:

void setSeed(long inic)

Objektumok megfigyelése, az Observer interfész és az Observable osztály

Az *Observer* interfészt implementáló osztályok (megfigyelők) értesülnek arról, hogy, ha egy *Observable* osztály (vagy annak leszármazottja) objektumában bármilyen változás történik.

Az *Observable* objektum a változásokkor meghívhatja a *notifyObserver()* metódusát, amely az összes bejegyzett megfigyelő *update()* metódusát meghívja.

Dátum és időkezelés

A *Date* osztály objektumai egy konkrét dátumot és időpontot reprezentálnak. Leggyakoribb használata az aktuális dátum és idő lekérdezése (Date most = new Date();).
Az időszámításokat kezelő osztályok ősosztálya a *Calendar*. Ennek jelenleg egyetlen leszármazottja van a *GregorianCalendar*. Ezzel az osztállyal lehet egy adott dátumról információkat szerezni (pl. milyen napra esett, stb.), valamint manipulálni a dátumokat (pl. növelni). A *Date* osztálynak is megvannak a dátum kezeléséhez szolgáló metódusai, de azok használata már nem javasolt.
Az időzónákat valamint a nyári időszámítást a *TimeZone* absztrakt osztály leszármazottja a *SimpleTimeZone* használatával kezelhetjük.

Példaprogram:

```
import java.util.*;
```

```
public class datum {
      public static void main(String[] args) {
            //A GNT-től keletre 1 órára eső időzónák nevei
            String[] zon=TimeZone.getAvailableIDs(1*60*60*1000);
            for(int i = 0; i<zon.length;i++)
                  System.out.println(zon[i]);
            //A hivatalosan használt időzónánk: ETC
            SimpleTimeZone ect=new SimpeTimeZone(1, ”ETC”);
            //A nyári időszámítás kezdete: április első vasárnapja 2 óra
            etc.setStartRule(Calendar.APRIL, 1, Calendar.SUNDAY,
                  2*60*60*1000);
            //a vége október utolsó vasárnapján 2 órakor
            etc.setEndRule(Calendar.OCTOBER, -1, Calendar.SUNDAY,
                  2*60*60*1000);

            //Ezzel az időzónával inicializálva a GregorianCalendart,
                  informálódunk az aktuális dátumról
            GregorianCalendar cal = new GregorianCalendar(ect);
            //aktuális dátum lekérdezése és a cal beállítása
            Date most = new Date();
            cal.setTime(most);
            System.out.println(cal.get(Calendar.YEAR));
            System.out.println(cal.get(Calendar.MONTH));
            ….

            //Átállítani valamit pl.
            cal.set(Calendar.MINUTE, 10);
            …
      }
}
```

A GregorianCalendar osztállyal lehet még többek között
dátumot növelni, csökkenteni (*add()*)
időt növelni, csökkenteni (*roll()*)
Date objektumokat összehasonlítani, (*before(), after(), equals()*)
stb.

Helyi jellemzők, erőforrás kezelés

A *Locale* osztály szolgál az környezet azonosítására. Ezt használják a környezetfüggő osztályok, mint például a *GregorianCalendar, Format, NumberFormat, Collator, ResourceBundle*, stb.
Egy *Locale* azonosításához a nyelvkód és az országkód megadása szükséges. A magyar környezetet a

```
Locale magyar = new Locale(”hu”, ”HU”);
```

azonosíthatja.

Erőforrás kezelés

Az erőforrások kezelésére a *ResourceBundle* leszármazottjai a *ListResourceBundle*, *PropertyResourceBundle* használható. Az erőforrásoknak több változata lehetséges az aktuális *Locale*-nak megfelelően.
A *ListResourceBundle* esetén az erőforrások tetszőleges típusúak lehetnek. Definiálásukhoz egy osztályt kell létrehozni egy adott névvel a *ListResourceBundle* leszármazottjaként. Egy

bázisosztály adott *Locale*-ú erőforrásának keresésekor a következő nevű osztályokat keres sorban:

bázisosztálynév_nyelv_ország_változat
bázisosztálynév_nyelv_ország
bázisosztálynév_nyelv
bázisosztálynév_def.nyelv_def.ország_def.változat
bázisosztálynév_def.nyelv_def.ország
bázisosztálynév_def.nyelv
bázisosztálynév

Az első megtalált osztályból fogja az erőforrásokat venni.

Példa:

```
public class MyRes_hu_HU extends java.util.ListResourceBundle {
      public Object[][] getContents() {
            return contents;
      }
      static final Object[][] contents = {
            {”HelpLabel”, ”Súgó”},
            {”ButtonLabel”, ”Nyomógomb”}
      };
}
```

A MyRes osztály pedig:

```
import java.util.*;
public class MyRes {
      public static void main(String[] args) {
            Locale magyar = new Locale(”hu”, ”HU”);
            ResourceBundle myres = ResourceBundle.getBundle(”MyRes”,
magyar);
            //kiírni a kinyert erőforrásokat
            System.out.println(myres.getObject(”HelpLabel”));
      }
}
```

A PropertyResourceBundle nem osztályból, hanem egy Property file-ból nyeri ki az erőforrásokat. Csak String típusú erőforrásokkal tud dolgozni. A file neve ugyanolyan módszerrel képzendő, mint a ListResourceBundle esetén az osztály neve.
A fenti példának megfelelő file a *MyRes_hu_HU.properties* nevű lenne. Tartalma:

HelpLabel = Súgó
ButtonLabel = Nyomógomb

Gyűjtemény keretrendszer

A gyűjtemények kényelmesen kezelhető vektor, halmaz, lista, láncolt lista, stb. jellegű eszközöket szolgáltatnak.
A gyűjteményeket az 1.1-es JDK óta erősen kiterjesztették (sok esetben feleslegessé téve ezáltal az eredetieket), ezért külön először csak az 1.1-esben is meglevő részeket nézzük meg.

1.1-es JKD-beli gyűjtemények

Az Enumeration interfész

Az *Enumeration* interfészt elemek felsorolásához lehet használni. Az elemek a felsorolás során elfogynak, tehát csak egyszer lehet előhozni őket.
Metódusai:
Van-e még elem:
boolean hasMoreElements()
Következő elem:
Object nextElement()

A BitSet osztály

Egy dinamikusan változtatható méretű bit (boolean) tömb. Használhatók rá a logikai műveletek (és, vagy, kizáró vagy), lekérdezhető egy adott indexű bit, beállítható egy adott indexű bit.

A Vector osztály

Egy dinamikusan tömb, amely elemei *Object* típusúak, azaz tetszőleges objektum lehet.
A tömb egy adott elemszámmal inicializálódik, és ha szükséges automatikusan megnöveli a kapacitást egy növekményértékkel (az elemek átmásolódnak egy új, nagyobb tömbbe).
Az elemekhez való hozzáférés szinkronizált.

Konstruktorai:

Vector()
Vector(int kezdőKapacitás)
Vector(int kezdőKapacitás, int növekmény)

Metódusai:

Elemek hozzáadása, törlése:
*final void **addElement**(Object)*
*final void **insertElementAt**(Object, int index)*
*final void **removeElementAt**(int index)*
*final void **removeAllElements**()*
*final boolean **removeObject**(Object)*
Egy adott indexű elem beállítása:
*final void **setElementAt**(Object, int index)*
Elemek közvetlen lekérdezése:
*final boolean **contains**(Object)*
*final Object **elementAt**(int index)*
*final Object **firstElement**()*
*final int **indexOf**(Object)*
final int indexOf(Object, int honnantólKellKeresni)

*final int **lastIndexOf**(Object)*
final int lastIndexOf(Object, int honnantólKellKeresni)
*final Object **lastElement**()*

Elemek kinyerése felsorolás számára:

final Enumeration elements()

Elemek száma:

*final int **size**()*
*final boolean **isEmpty**()*

Kapacitás kezelése:

*final int **capacity**()*
*final void **ensureCapacity**(int minimumkapacitás)*
*final void **setSize**(int újMéret)*
*final void **trimToSize**()*

Elemek másolása egy tömbbe:

*final void **copyInto**(Object[])*

Használat:

Mivel az elemek Object típusúak lehetnek, bármilyen objektumfajta tárolására használhatjuk. Nem használható viszont elemi adattípusok tárolására, csak ha a fedőosztályok segítségével becsomagoljuk őket. Pl. egy int adat hozzáadása:

```
addElement(new Integer(100));
```

Az elemek kinyerésekor az elemeket célszerű visszakonvertálni a valódi típusára. Pl.

```
String a=new String(”Helló”);
Vector v = new Vector();
v.addElement(a);
….
String b = (String)v.elementAt(0);
```

A HashTable osztály

Egy leképzés jellegű gyűjtemény. Elemei kulcs, érték párosok, mind a kettő Object típusúak. Kulcsonként csak egy érték tárolható (mivel az érték is tetszőleges objektum lehet, lehet gyűjtemény is). A kulcs objektum *hashCode()* metódusát használja a kulcs azonosításához (az tekinthető egyenlőnek, ha a hashCode() azonos értéket ad).

Fontosabb metódusai:

Elem felvitele, kiolvasása:

*Object előzőElem **put**(Object kulcs, Object elem)*
*Object **get**(Object kulcs)*

Elem, elemek törlése:

*Object kulcs **remove**(Object kulcs)*
*void **clear**()*

Elemek keresése:

*boolean **contains**(Object elem)*
*boolean **containsKey**(Object kulcs)*
*boolean **isEmpty**()*

Elemek száma:

*int **size**()*

A Properties osztály, a HashTable leszármazottja

Property-k (környezeti információk) tárolására specializálódott HashTable.

Java 2 gyűjtemény keretrendszer

A Java 2 kollekciócsomag tartalmazza a gyűjteményeket, iterátorokat, valamint a gyűjteményeken manipuláló alapvető algoritmusokat megvalósító osztályt (Collections).

Három fő fajtája létezik a gyűjteményeknek:
- halmazok: nem tartalmazhatnak azonos elemeket
- listák: tartalmazhatnak azonos elemeket, felhasználható az elemek indexe
- táblák: kulcs-érték párok tárolására

A gyűjtemények viselkedését interfészek írják le. Minden gyűjtemény fajtának létezik egy vagy több implementációja.

Halmazok

A halmazok viselkedését leíró interfészek:
`Collection`: A halmazok és listák viselkedésének közös őse. Tartalmazza az elemek hozzáadása, kitörlése, tartalmazás vizsgálat, egyéb halmaz műveletek, elemszám lekérdezés, tömbbé alakítás, iterátor készítés, stb. műveleteket.
`Set`: A Collection leszármazottja. Nem definiál újabb viselkedés fajtát, de a viselkedéseket átdefiniálja annak megfelelően, hogy a halmaz nem tartalmazhat azonos elemeket.
`SortedSet`: A Set leszármazottja. A rendezett halmazokkal kapcsolatos plusz viselkedéseket tartalmazza. Pl. első elem lekérdezése, utolsó elem lekérdezése, egy rendezett részhalmaz lekérdezése, az összehasonlító objektum lekérdezése, stb.

Halmaz implementációk:
`HashSet`: Hash táblás tárolással megvalósított rendezetlen halmaz. Implementálja a Set interfészt.
`LinkedHashSet`: Hash táblás és láncolt listás tárolással megvalósított halmaz. A HashSet leszármazottja.
`TreeSet`: Kiegyensúlyozott fás tárolással megvalósított rendezett halmaz. Implementálja a SortedSet interfészt.

Listák

Listák viselkedését leíró interfészek:
`Collection`: A halmazok és listák viselkedésének közös őse. Tartalmazza az elemek hozzáadása, kitörlése, tartalmazás vizsgálat, egyéb halmaz műveletek, elemszám lekérdezés, tömbbé alakítás, iterátor készítés, stb. műveleteket.
`List`: A Collection leszármazottja. Tartalmazza azokat a plusz viselkedéseket, amelyek az elemek index szerinti elérését, elhelyezését támogatja. Pl. elem beszúrása egy adott index

pozícióba, adott indexű elem lekérdezése, átállítása, törlése, egy elem indexének lekérdezése, listiterátor lekérdezése, stb.

Listák implementációi:
`ArrayList`: Egy önátméretező tömb tárolással megvalósított lista. Megvalósítja a List interfészt.
`LinkedList`: Láncolt listás tárolással megvalósított lista. Megvalósítja a List interfészt. Kiegészítő viselkedései pl. első, utolsó elem elérése, törlése, hozzáadás.
`Vector`: Ugyanaz, mint egy ArrayList, csak a metódusai szinkronizáltak. Mivel a Vector már a korábbi JDK-ban is benne volt, megvannak a korábbi metódusai is, de megvalósították a List interfészt is.
`Stack`: A Vector leszármazottja. Veremműveleteket is megvalósít.

Táblák (Map)

Táblák viselkedését leíró interfészek:
`Map`: Tartalma pl. kulcs létezés vizsgálat, érték létezés vizsgálat, adott kulcsú elem kinyerése, adott kulcs-érték pár betétele, adott kulcsú elem kitörlése, elemek számának kinyerése, kulcsok halmazának kinyerése, értékek halmazának kinyerése, stb.
`SortedMap`: A Map leszármazottja. Rendezett kulcs-érték párok tárolására. Plusz funkciói: első, utolsó kulcs kinyerése, rész táblák kinyerése.

Táblák megvalósításai:
`HashMap`: Hash táblás tároláson alapul. Megvalósítja a Map interfészt.
`LinkedHashMap`: Hash táblás és láncolt listás tároláson alapul. A HashMap leszármazottja.
`HashTable`: A HashMap régebbi variációja. Metódusai szinkronizáltak. A Dictionary osztályból származik, amely a korábbi JDK-ban tartalmazta a kulcs-érték párok kezelésének alapjait. A Java 2-be illeszthetőség kedvéért megvalósítja a Map interfészt is.
`Properties`: A HashTable leszármazottja, amely szintén a korábbi JDK-ban keletkezett, és amelyre szintén ráhúzták a Map interfészt.
`TreeMap`: Kiegyensúlyozott fás tároláson alapul. Rendezett kulcs-érték párok tárolására alkalmas. Megvalósítja a SortedMap interfészt.

Iterátorok

Az iterátorok segítségével lehet a gyűjtemény elemein sorban végighaladni.
`Enumerator`: A korábbi JDK iterátora. (lásd korábban)
`Iterator`: Funkciói: következő elem kinyerése, van-e még elem lekérdezése, az utoljára kinyert elem kitörlése a gyűjteményből.
`ListIterator`: Az Iterator interfész leszármazottja. Funkciói: következő elem kinyerése, van-e következő elem lekérdezése, következő elem indexének lekérdezése, előző elem kinyerése, van-e előző elem lekérdezése, előző elem indexének lekérdezése, az utoljára kinyert elem kitörlése, egy elem beszúrása az aktuális pozícióba, az utoljára kinyert elem felülírása egy megadott elemmel.

A Collections osztály

A Collections osztály olyan alapvető algoritmusokat tartalmaz, amelyek gyűjteményekkel dolgoznak. Metódusai statikusak.
Pl. maximum keresés, minimum keresés, rendezés, összekeverés, elemek sorrendjének megfordítása, keresés, bináris keresés, elemek másolása, stb.
Tartalmaz olyan metódusokat is, amelyek segítségével egy meglévő gyűjteményünknek megkaphatjuk a szinkronizált változatát, illetve nem módosítható (csak lekérdezhető) változatát.

A Comparator interfész

A Comparator interfész compare metódusát használják fel a gyűjtemény keretrendszer osztályai az elemek összehasonlítására. A rendezett gyűjtemények létrehozásakor meg kell adni egy Comparator objektumot, ez szolgál a rendezettség eldöntésére (vagy a régebben is használt Comparable interfész compareTo metódusa). A Collections osztály olyan metódusainál, ahol össze kell hasonlítani az elemeket (rendezés, maximumkeresés, stb.), szintén meg kell adni egy Comparator objektumot.

A bemenet és kimenet kezelése. A *java.io* csomag.

A java-ban az adatáramlás a csatorna (Stream) fogalomhoz kapcsolódik.
Csatornák csoportosítása:

- Irány szerint: bemeneti és kimeneti csatorna
- Adattípus szerint: byte- és karaktercsatorna
- Feladatuk alapján:
  forrást illetve nyelőt meghatározó csatorna
  szűrő csatorna: egy meglévő csatorna funkcionalitását bővítik

Osztályhierarchia

A csatornaosztályok absztrakt ősei:

InputStream	byte alapú bemeneti csatorna
OutputStream	byte alapú kimeneti csatorna
Reader	karakter alapú bemeneti csatorna
Writer	karakter alapú kimeneti csatorna

Forrást illetve nyelőt meghatározó csatornák (közvetlen leszármazottai a megfelelő absztrakt ősosztálynak):

Bytetömb forrású illetve célú csatornák
ByteArrayInputStream
ByteArrayOutputStream
Karaktertömb forrású illetve célú csatornák:
CharArrayReader
CharArrayWriter
String forrású illetve célú csatonák:
StringReader
StringWriter

StringBufferInputStream *Deprecated, Nem megfelelően konvertálja a karaktereket byte-okká.*

File forrású illetve célú, byte alapú csatornák:

FileInputStream
FileOutputStream

File forrású illetve célú, karakter alapú csatornák:

FileReader
FileWriter

Több bemenő csatornát összefűző csatorna

SequenceInputStream

Csővezeték csatornák:

PipedInputStream
PipedOutputStream
PipedReader
PipedWriter

Szűrő csatornák

Byte-csatorna felett létrehozott karaktercsatornák

InputStreamReader
OutputStreamWriter

Csatorna bufferelése:

BufferedInputStream
BufferedOutputStream
BufferedReader
BufferedWriter

Szűrőcsatornák őse

FilterInputStream
FilterOutputStream
FilterReader absztrakt
FilterWriter absztrakt

Többféle adattípus olvasását lehetővé tevő szűrőcsatorna:

DataInputStream
DataOutputStream

Input csatorna sorait sorszámozó csatornák:

LineNumberInputStream *Deprecated*
LineNumberReader *(BufferedReader leszármazottja)*

Többféle típus szöveges kiírását lehetővé tevő csatorna (hiba esetén nem kivételt dob, hanem státuszváltozót állít be):

PrintStream
PrintWriter *(Writer leszármazottja)*

Adatvisszatevő csatorna:

PushbackInputStream
PushbackReader

Objektumok kiírására szolgáló csatorna

ObjectInputStream
ObjectOutputStream

Az ősosztályok metódusai

Az *InputStream* és a *Reader* metódusai csak az olvasott adat típusában térnek el egymástól. A *Reader*-ben char típusú adat van byte helyett.
Olvasások:
abstract int read()
int read(byte[] b)
int read(byte[] b, int off, int len)
Csatorna lezárása (erőforrások felszabadítása):
void close()
Hátralevő byte-ok száma:
int available()
A következő n byte kihagyása:
long skip(long n)
Egy pozíció megjelölése (*readlimit* byte olvasásáig őrzi meg a jelölést):
void mark(int readlimit)
Visszaugrás a megjelölt pozícióba:
void reset()
Leellenőrzése, hogy támogatja-e a mark mechanizmust:
boolean markSupported()

Az *OutputStream* és *Writer* metódusai (a *Writer*-ben char van a byte helyett):
Írások:
void write(byte[] b)
void write(byte[] b, int off, int len)
abstract void write(int b)
a *Writer*-ben ezeken kívűl még:
void write(String str)
void write(String str, int off, int len)
Csatorna lezárása:
void close()
Buffer üríttetése:
void flush()

Forrást illetve nyelőt meghatározó csatornák kezelése

A byte tömb, karaktertömb, file forrású illetve célú csatornák számára a konstruktorban kell megadni a forrást (célt). A file esetén a megadás lehet

- String adattípusban megadva a file neve,
- FileDescriptor típussal megadva a file leírója (nyitott file-kat, socket-eket leíró objektum),
- File típusú objektummal (file-okat és directory-kat reprezentáló típus).

A csővezeték csatornák esetén a két csőoldalt össze kell kapcsolni. Ezt lehet bármelyik csatorna konstruktorának segítségével vagy valamelyik csatorna connect() metódusával.

```
PipedInputStream a = new PipedInputStream();
PipedOutputStream b = new PipedOutputStream(a);
```

vagy

```
PipedInputStream a = new PipedInputStream();
PipedOutputStream b = new PipedOutputStream();
```

```
a.connect(b);
```

A SequenceInputStream konstruktorában megadható két InputStream, amelyet összefűz, azaz az első vége esetén olvas a másodikból. Illetve megadható több csatorna is, ha Enumeration-t (felsorolást) készítünk belőlük.

Szűrő csatornák kezelése

A szűrő csatornák közös jellemzője, hogy konstruktorában meg kell adni egy másik csatornát.

Az *InputStreamReader, OutputStreamWriter* csatornák, amelyek byte csatornák karakteres olvasását teszik lehetővé, konstruktorban meg lehet még adni a karakterkódolást is.

A bufferelt csatornák konstruktorában előírható még a buffer mérete is. A *BufferedReader*-ben hasznos metódus a *String*-et visszaadó *readLine()*, amely sorvége karakterig olvas. A *BufferedWriter* hasznos új metódusa pedig a *newLine()*, amely egy újsor jelet ír ki.

A *DataInputStream, DataOutputStream* csatornák implementálják a *DataInput* interfészt, ezért tartalmaznak minden adattípusra olvasó, illetve író metódust. Létezik *readByte()*, *readShort(), readInt(), readLong(), readFloat(), readDouble(), readChar(), readUTF()* (Stringek olvasása). Létezik ugyanezeknek write változata az output csatornában.

A *LineNumberReader* a *BufferedReader* leszármazottja. Az örökölt metódusain kívül képes még az aktuális sor sorszámának lekérdezésére, beállítására.

A *PrintStream* és a *PrintWriter* közös tulajdonsága, hogy különböző típusú adatokat képes írni szöveges alakban a *print()* és a *prinln()* különböző paraméterezésű metódusaival. Létezik például *print(int i), print(long l), print(String s)*, stb. A másik közös jellemző, hogy a műveletek szemben a többi csatornával, nem válthatnak ki *IOException*-t, hanem a *checkError()* metódussal kérdezhető le a művelet sikeressége. A különbség a *PrintStream* és *PrintWriter* között, hogy a *PrintStream* byte-okat ír, míg a *PrintWriter* karaktereket.

A *PushbackInputStream* és *PushbackReader* az adatok csatornára való visszatételére szolgáló metódusokkat definiálja még:

void unread(byte[] b)
void unread(byte[] b, int off, int len)
void unread(int b)

illetve a *PushbackReader* ugyanez csak karakter típussal.

Az *ObjectInputStream* és *ObjectOutputStream* objektumok szerializálására és deszerializálására használható. (Szerializáció: az objektumok aktuális állapotának mentése, a referenciák esetén a teljes hierarchia mentődik.). Implementálja az *ObjectInput* (*ObjectOutput*) interfészt, ami a *DataInput* (*DataOutput*) interfész kiterjesztése egy *readObject()* (*writeObject()*) metódussal.

File kezelés

A File osztály

A File osztály segítségével reprezentálhatunk file- és directory neveket, valamint műveleteket végezhetünk file-okkal, directory-kkal. Objektumot hozhatunk létre nem létező file-ra (directory) is.
Konstruktorai:

File(File parent, String child)
File(String parent, String child)
File(String pathname)

A nevet tehát két részletben is meg lehet adni. Például a /usr/bin –t jelentő File objektum létrehozható:

```
File szulo = new File(”/usr”);
File bin = new File(szulo, ”bin”);
```

vagy

```
File bin = new File(”/usr”, ”bin”);
```

vagy

```
File bin = new File(”/usr/bin”);
```

Osztályszintű változói:
A keresési útvonal megadásokat elválasztó karakter a

static String pathSeparator
static char pathSeparatorChar

A WIN32-es rendszerekben ennek értéke ”;”.
A file nevekben használt elválasztó karakter (Win32-ben: \)

static String separator
static char separatorChar

Metódusai:
A file olvasható-e, írható-e (létezik-e) a program számára:

boolean canRead()
boolean canWrite()

A file létezik-e:

boolean exists()

A file directory-e, file-e, rejtett-e

boolean isDirectory()
boolean isFile()
boolean isHidden()

Abszulut megadás-e:

boolean isAbsolute()

A file (directory) neve:

String getName()

Az útvonal neve:

String getPath()

A teljes név:

String toString()

A szülő directory (null, ha nincs):

String getParent()
File getParentFile()

Az abszulut alak:

File getAbsoluteFile()

String getAbsolutePath()

A canonical alak:

File getCanonicalFile()

String getCanonicalPath()

Az utolsó módosítás ideje:

long lastModified()

A file hossza:

long length()

A file létrehozása, ha még nem létezik:

boolean createNewFile()

A file nevének összehasonlítása egy másikkal:

int compareTo(File pathname)

int compareTo(Object o)

boolean equals(Object obj)

A file törlése, törlése kilépéskor:

boolean delete()

void deleteOnExit()

Directory létrehozása, directory létrehozása beleértve a nem létező szülőket is:

boolean mkdir()

boolean mkdirs()

Átnevezés:

boolean renameTo(File dest)

Utolsó módosítás idejének beállítása

boolean setLastModified(long time)

Csak olvashatóvá tétel:

boolean setReadOnly()

Konvertálás URL objektummá:

URL toURL()

Directory lista:

String[] list()

File[] listFiles()

Adott feltételeknek eleget tevő file-ok listája:

String[] list(FilenameFilter filter)

File[] listFiles(FileFilter filter)

File[] listFiles(FilenameFilter filter)

Osztályszintű metódusok

Temp file készítés, temp file készítés a megadott directory-ban:

static File createTempFile(String prefix, String suffix)

static File createTempFile(String prefix, String suffix, File directory)

A file rendszer gyökerének listája:

static File[] listRoots()

Ha van Security Manager akkor majdnem mindegyik művelet dobhat SecurityException kivételt.

A FileFilter és FileNameFilter interfész

Mindkét interfész arra szolgál, hogy egy *File* objektumról megállapíthassuk, hogy megfelel-e egy kritériumnak (például .TXT-re végződik-e).
Egyetlen metódust deklarál

public boolean accept(File f)

illetve *FileNameFilter* esetén

public boolean accept(File dir, String name)

A FilePermission osztály

File-okhoz, directory-khoz lehet írási, olvasási, törlési, futtatási jogokat rendelni a segítségével.

A RandomAccesFile osztály

Direkt hozzáférésű file. Egy filemutató jelöli az aktuális pozíciót, amelyre a következő írási vagy olvasási művelet vonatkozik. A filemutató állítható.

Konstruktorai:

RandomAccessFile(File file, String mode)
RandomAccessFile(String filename, String mode)

A konstruktor kötelezően lekezelendő *FileNotFoundException* kivételt dob. Megdobhatja még az *IllegalArgumentException* kivételt is, ha a *mode* argumentum nem ”r” vagy ”rw”, valamint a *SecurityException*-t, ha van security menedzser és nem engedélyezett a megadott módú hozzáférés.

Metódusai:

A *RandomAccesFile* implementálja a *DataInput* és *DataOutput* interfészeket, ezért mindenféle adattípust lehet írni és olvasni.
Olvasó metódusok:

readBoolean(), readByte(), readShort(), readInt(), readLong(), readChar(), readDouble(), readFloat(), readUTF(), readLine(), readFully(byte[] b), readFully(byte[] b, int off, int len), valamint read(), read(byte[] b), read(byte[] b, int off, int len)

Író metódusok:

write(byte[] b), write(byte[] b, int off, int len), write(int b), writeBoolean(boolean v), writeByte(int v), writeBytes(String s), writeShort(int v), writeInt(int v), writeLong(long v), writeChar(int v), writeChars(String s), writeDouble(double v), writeFloat(float v), writeUTF(String str)).

Filemutató pozícionálása:

void seek(long pos)

Egyéb:

void close()
FileDescriptor getFD()
long getFilePointer()

long length()
void setLength(long newLength)
int skipBytes(int n)

A StreamTokenizer osztály

Egy input csatorna szöveges elemzésére, feldolgozására használható. Az input csatornát a konstruktor paramétereként kell megadni. Input csatorna lehet byte alapú is, de nem ajánlott (deprecated).
Képes megkülönböztetni számokat (számmal kezdődik), azonosítókat (betűvel kezdődik), szöveges konstansokat (”-el kezdődik), megjegyzéseket (// vagy /* */ között van).
A következő token-t a *nextToken()* metódussal olvastathatjuk a csatornáról. Ennek hatására a *ttype* változó értéke felveszi a beolvasott token típusát jelző konstans értéket:
TT_EOF: csatorna vége
TT_EOL: sor vége
TT_WORD: azonosító
TT_NUMBER: szám
valamint felvehet még ” értéket (int-é konvertálva) a szöveges konstansok esetén, illetve tetszőleges karaktert (int-é konvertálva) egy karakteres token-ek esetén.
Szám esetén az eredmény az *nval* változóban kapjuk, azonosító és szöveges konstans esetén az *sval* változóban. A megjegyzések átugorja, azaz nem tekinti token-nek.

Szálak programozása

Mi a szál?

Vezérlési folyamat egy programon belül. Egy programon belül több szál is létezhet, azaz több egymással párhuzamosan futtatható rész.
A többprogramos operációs rendszerek folyamat (processz) fogalmával szemben a szál nem kernel szintű, hanem felhasználói szintű fogalom. A folyamatok adatai egymástól teljesen elkülönült területen vannak, egymással csak IPC mechanizmus segítségével képesek kommunikálni. Egy program szálai ugyanabban a címtérben futnak, statikus és instancia attribútumai egyaránt közösek. A szálak ütemezése viszont egy újabb problémát vet fel. Ha a szálak teljesen felhasználói szintűek, azaz az operációs rendszer nem is tud róluk, akkor egy szál blokkolódásakor az egész processz blokkolódik. Az operációs rendszernek tehát tudnia kell a szálak létezéséről, hogy ebben az esetben a processz egy másik szála kaphassa meg a vezérlést.
A Jáva szálütemezés preemptiv, azaz egy nagyobb prioritású szál megszakítja az alacsonyabb prioritású szálak futását. Azonos prioritású szálak ütemezése implementációfüggő.
Tipikus alkalmazási területe a szálaknak a kliens-szerver programok, ahol egy szervernek több kérést kell egyszerre kiszolgálnia. Blokkoló csatornaolvasás esetén (csövek, socket-ek olvasása) szintén hasznos, ha addig a program mással foglalkozhat. Az időigényes számítási feladatoknál pedig, hogy ne legyen teljesen ezzel lefoglalva a programunk, ha van esetleg más is amit lehet csinálni.

Appletek futásakor a böngésző szintén többször is használ szálakat. Például a **paint**() metódusát nem az applet hívja, hanem a webböngésző egy másik szála. Valamint szálak végzik a következő feladatokat: képek betöltése, a képernyő frissítése, hangfelvételek lejátszása.
A grafikus felületű Java alkalmazások is több szálon futhatnak (külön szálban az eseményvárás és a megjelenítési funkciók).

Szál létrehozása

Szereplők:
- a szálat képviselő objektum (**Thread**)
- a végrehajtandó kódot tartalmazó tetszőleges objektum (**Runnable**)

A *Thread* maga is implementálja a *Runnable* interfészt egy üres *run()* metódussal. A *Thread* osztályból való leszármaztatással (*run()* metódus felüldefiniálásával) lehet saját szálat létrehozni. A szál futni csak akkor kezd, ha meghívjuk a *Thread* osztály *start()* metódusát.

```
class Animation extends Thread {
  …
  Animation() {
     start();
  }
  public void run() {
     …
  }
}
```

vagy

```
class MyThread extends Thread {
  …
  public void run() {
     …
  }
}

class MyAlk {
  ….
  MyThread a;
  a = new MyThread();
  a.start();
  …
}
```

A *Thread*-ből leszármaztatás hátránya, hogy mivel nincs többszörös öröklés más osztályból való leszármaztatás nem lehetséges.
A másik lehetséges megoldás szerint az osztálynak implementálnia kell a *Runnable* interfészt (ezáltal megvalósítva a *run()* metódust), és példányosítani kell a *Thread* osztályt (vagy egy leszármazottját) a konstruktorának átadva a saját referenciáját:

```
class Animation implements Runnable {
  Thread myThread;
  Animation() {
```

```
    myThread = new Thread(this);
    myThread.start();
  }
  public void run() {
    …
  }
}
```

A szálak vezérlése

Szál állapotai:

- új: a szálobjektum most jött létre és még nincs elindítva a *start()* metódussal
- futtatható: az ütemező hatáskörébe került
  - futó: éppen fut
  - készenléti: futásra kész, de nem ő van soron
- blokkolt, felfüggesztett: nem képes futni, pl. valamilyen I/O-ra vár, vagy engedélyezésre vár
- megszakított: szál megszakítása, blokkolt szálak esetén is azonnal feldolgozódik
- halott: a szál már nem él

Állapotátmenetek:

- új → futtatható (készenléti): *start()* hatására
- futó ←→ készenléti: ütemezés hatására, lásd szálak ütemezése
- futtatható → halott: run() metódus vége, stop() metódus (elavult)
- futtatható → megszakított: *interrup()* hatására, megdobódik az *InterruptedException* kivétel, amelyet a *run()* metódusban le kell kezelni
- blokkolt → megszakított: *interrup()* hatására, megdobódik az *InterruptedException* kivétel, amelyet a *run()* metódusban le kell kezelni
- futtatható (futó) → blokkolt: *wait(), sleep(), join()* illetve I/O hatására (valamint a *suspend()*, de az elavult), lásd szálak blokkolása
- blokkolt → futtatható (készenléti): *notify()*, időletelte (*sleep(), join(idő)*), másik szál befejeződése (*join()*), I/O vége (*resume()* alavult)

A szálak ütemezése

A Java a szálak ütemezésére prioritás osztályos ütemezést használ. Tíz prioritás osztály létezik, ezek értékei Thread.MIN_PRIORITY (1) és Thread.MAX_PRIORITY (10) közé esnek. Az elindított szálak öröklik az elindító prioritását. Az alapértelmezés szerinti prioritás a Thread.NORM_PRIORITY (5).
A magasabb prioritású szálak megszakítják az alacsonyabb prioritású szálakat.
Az azonos prioritású szálak ütemezése Round-Robin módszerrel történik, de nem garantált az időosztásos ütemezés. Ez ugyanis JVM megvalósítás függő (Win32 alatt van, Solaris alatt nincs időosztás). Ha nincs időosztás, akkor egy maximális prioritású futó szál csak akkor enged szóhoz jutni más szálakat, ha blokkolódik vagy a programozó a futási sor végére zavarja a *yield()* metódussal. Szintén problémás lehet a kisebb prioritású szálak sorra kerülése,

ha nincs időosztás. A megoldás szintén lehet a *yield()*, vagy a prioritás növelése, csökkentése a *setPriority()* metódussal. Prioritást nem csak a szál indítása előtt, hanem közben is lehet beállítani.

Szálak leállítása

Ha szeretnénk egy szálat leállítani, akkor a régebbi verziókban a *stop()* meghívása volt a megoldás. Kiderült azonban, hogy nem tökéletes, ezért használata már nem javasolt.
Helyette a *run()* metódust úgy kell szervezni, hogy leálljon, ha szükséges. A *run()* magját olyan *while* ciklusba kell szervezni, amely feltételében egy változó értékét vizsgálja, és a leállításhoz a változó értékét kell átállítani.
Használható lehet még leállításra a megszakítás is, ha a *run()* metódusban az *InterruptedException* lekezelés után vége van az utasítástörzsnek.
Egy szál (amelyben például while(true) ciklus van) tovább élhet még akkor is, ha a létrehozó alkalmazás *main()* metódusa már lefutott. A *System.exit()* hívásra azonban a szálak is lezáródnak.

Szálak blokkolása

Szálak blokkolására szinkronizációs okokból kerülhet sor.
A Java 1.0-tól létező *suspend()* (és párja az elengedésre használt *resume()*) használata nem javasolt, mert programunk holtpontba kerülhet.
Helyette a *wait()* illetve a *sleep()* használható. A *wait()* párja, azaz a blokkolódás feloldása a *notify()*. A *sleep()*-el blokkolt szál pedig a megadott idő letelte után kerül futtatható állapotba.

Szálak összekapcsolása

Szükség lehet arra, hogy egy szál megvárja egy másik szál befejeződését. Erre szolgál a szálak összekapcsolása.
A megvárandó szál *join()* metódusát kell meghívni, hogy az aktuális szál megvárja. Az aktuális szál addig blokkolódik.
A *join()* metódusnak paraméterként megadható egy időérték is, amely idő letelte után nem vár tovább a szál.

Szálak megszakítása

Egy szál megszakításakor a vezérlés a legközelebbi *InterruptedException*-t lekezelő blokkra ugrik. A blokkoló utasítások (*wait(), sleep(), join()*) kiválthatnak ilyen kötelezően lekezelendő kivételt.
A megszakítás az *interrupt()* metódus meghívásával lehetséges.
Használata főleg beragadt (hosszabb ideje blokkolt) szálak kezelésére válhat szükségessé.

Démon szálak

Szerepe ugyanaz, mint a démon processzeké, azaz háttérben futó feladatok ellátására. A démon szálakat ugyanúgy kell készíteni, mint a normál szálakat, csak indítása előtt démonná kell tenni a *setDaemon(true)* metódushívással.
A démon szálak automatikusan állnak le, amikor az összes nem démon szál már befejeződött.

Szálcsoportok

Tetszőleges számú szálat összegyűjthetünk egy csoportba, s így ezeket közösen kezelhetjük. Ha nem sorolunk be egy szálat szálcsoportba, akkor automatikusan besorolódik egy szálcsoportba. Egy szálcsoportnak az elemei lehetnek szálcsoportok is.
Szálcsoport létrehozása a ThreadGroup osztály példányosításával történik. A szál hozzácsatolása egy szálcsoporthoz a szál létrehozásakor történhet azokkal a konstruktorokkal, amelyeknek paraméterként megadhatunk szálcsoportot is.
Hasznosabb szálcsoport kezelő metódusok:

- Egy szál csoportjának a lekérdezése: *getThreadGroup()*
- Egy szálcsoporthoz tartozó futó szálak száma: *activeCount()*
- Egy szálcsoporthoz tartozó futó szálak referenciáinak lekérdezése: *enumerate(Thread[])*
- Egy szálcsoport összes szálának megszakítása: *interrupt()*

Lokális szálváltozók

A Java 2-es verziójától kezdve létező osztály a *ThreadLocal*. Segítségével olyan változókat lehet definiálni, melyek osztály szintű statikus változók, de szálanként más és más értéket tárolnak.
A szálankénti kezdőértéket a ThreadLocal osztály initialValue() metódusának felüldefiniálásával lehet beállítani.
A szálak a változó értékét a *get()*-el kérdezhetik le, és a *set()*-el állíthatják be.

Szinkronizálás

A szálak egyik nagy előnye, hogy megosztott adatokkal és erőforrásokkal dolgoznak. Az osztozás során viszont versenyhelyzetek állnak elő, illetve szinkronizációs gondok merülhetnek fel.
A Java szálak a Brinch-Hansen féle monitor koncepciót használják a kritikus részek kezelésére. Amikor egy szál eljut egy kritikus részhez, magához rendel egy monitort. Ha ekkor egy másik szál is be akarna lépni ebbe a kritikus részbe, akkor annak várnia kell addig, amíg a monitor újra szabad nem lesz.

A Java-ban minden osztályhoz és objektumhoz tartozik egy zár (monitor). A zár elhelyezése egy kódblokkra a *synchronized* kulcsszóval lehetséges.

```
synchronized (obj) {
  …
}
```

Ekkor az *obj* objektum zárolódik, ha egy szál belép a kódblokkba. Azaz más szál nem léphet be olyan *synchronized* blokkba, amelyet az *obj* objektumra definiálunk.
A synchronized szót metódusok fejlécében is használhatjuk. Például

```
public synchronized void fgv() { … }
```

Ez megfelel a

```
public void fgv() {
```

```
        synchronized (this) {
         …
        }
      }
```

Osztályszintű metódusok, illetve ezekben definiált blokkok is szinkronizálhatók. Ekkor az osztályhoz rendelődik a monitor, azaz az osztály statikus mezőihez egyszerre csak egy objektum férhet hozzá. Egy nem statikus metódusban is zárolhatjuk az osztályt a *synchronized (osztály) {…}* blokkal.

A szinkronizált metódusok szinkronizációja nem öröklődik!

A monitor koncepció csak a kritikus szakaszok védelmére való. Szálak kommunikációjának, közös erőforrás használatának vezérlésére használnunk kell a *wait(), notify(), notifyAll()* metódusokat. Ezek a metódusok az *Object* osztály metódusai, azaz minden osztály számára elérhetőek. Használatuk csak *synchronized* blokkban lehetséges! Ha egy szál a *wait()* hatására blokkolódik, akkor elengedi a zárat, hogy ne foglalja feleslegesen. A *notify()* metódus a következő várakozó szálnak jelez, amely feléledvén megpróbálja újra zárolni. A *notifyAll()* az összes várakozó szálat felébreszti.

Egy tipikus példa a termelő-fogyasztó probléma.

```
import java.util.*;
class Producer extends Thread {
    private int queueSize;
    private long delay;
    public Producer(long xDelay,int xQueueSize)
    {
        delay = xDelay; queueSize = xQueueSize; start();
    }
  private Vector q = new Vector();
  public void run() {
    try {
        while(true) {
            sendMessage();
            sleep(delay);
        }
    } catch(InterruptedException e){}
  }
  private synchronized void sendMessage()
    throws InterruptedException {
    while(q.size() == queueSize) wait();
    q.addElement(new Date().toString());
    notifyAll();
  }
  public synchronized String receiveMessage()
    throws InterruptedException {
    notify(); while(q.size() == 0) wait();
```

```
    String s = (String)q.firstElement();
    q.removeElement(s);
    return s;
  }
}


import java.lang.*;
public class Consumer extends java.lang.Thread
{
    private long delay;
    private Producer p;
    private String name;
    private DisplayServiceProvider ds;
    Consumer(String xName,Producer xProducer,
     DisplayServiceProvider xDs,long xDelay) {
        name = xName; p = xProducer; ds = xDs; delay = xDelay;
    }
    public void run() {
        try {
            while(true) {
                String s = p.receiveMessage();
                ds.showString(name + ":" + s);
                sleep(delay);
            }
       } catch(InterruptedException e){}
    }
}
```

Hálózat programozása, a *java.net* csomag

A *java.net* csomag a kommunikációval és a hálózati erőforrások elérésével kapcsolatos osztályokat tartalmazza.
Két részre csoportosítható:

- Socket-ekkel kapcsolatos osztályok

  A Jáva Socket interfészének segítségével elérhetjük az interneten használt alapvető hálózati protokollokat
- URL kezeléssel kapcsolatos osztályok

  Segítséget nyújt a hálózati erőforrások eléréséhez. (Pl.: dokumentumok, programok, egyéb adatállományok)

Socket-ek

A socket-ek alacsony szintű programozási interfészt nyújtanak a hálózati kommunikációhoz.
A programok közti adatcserét csatornákon keresztül oldják meg.
A Jáva különböző típusú socket-eket biztosít a két alapvetően elkülönülő hálózati protokollra:

- kapcsolat-orientált protokoll (*Socket osztály*)
  A kapcsolat felépítése után a két program, mint egy telefonon keresztül kommunikálhat egymással, míg a kapcsolatot az egyik fél meg nem szünteti.
- kapcsolat nélküli protokoll (*DatagramSocket osztály*)
  A kommunikáció csak rövid üzenetek formájában zajlik. Hátránya, hogy nem megbízható: adatcsomagok elveszhetnek, többszöröződhetnek.

A kommunikáció során a szereplőket kliensnek és szervernek nevezzük. Azt a programot nevezzük *kliens*nek, amelyik a kapcsolatot kezdeményezi, *szerver*nek pedig azt, amely a kapcsolatkérést fogadja.

Kapcsolat alapú protokoll

A *Socket* osztály egy példánya reprezentálja a kapcsolat egy végpontját a kliens és a szerver oldalon is.

A szerver alkalmazások a *ServerSocket* osztály egy példányának *accept()* metódusát használják arra, hogy a kliensek kapcsolatkéréseire várjanak. Ez a metódus hozza létre a klienssel való adatcseréhez szükséges *Socket* típusú objektumot. A ServerSocket konstruktorában a port számot kell megadni. (A portszám 0-65535 tartományba eshet. Az ismert alkalmazások, mint például a Telnet, FTP, stb., a 0-1023 tartományból kaptak portszámot, így célszerű egy 1024-nél nagyobb számot választani saját alkalmazásainknak.)
A szerveralkalmazás egyetlen *ServerSocket* típusú objektumot használ a különböző kliensekkel való kapcsolatfelvételre, de ezt követően minden egyes kliens-kapcsolatot már külön *Socket* típusú objektum jelképez. Ha a kliens kiszolgálása sok időt vesz igénybe, célszerű külön szálat indítani rá.

```
try {
```

```
  ServerSocket server = new ServerSocket( 2225 );
  while( !finished ) {
    Socket serverSock = server.accept();
    …
  }
  server.close();
} catch (IOException e) {…}
```

A klienseknek két információra van szükségük, hogy megtaláljanak, és kapcsolatot teremtsenek egy szerveralkalmazással:

- a kiszolgáló gép hálózati címére, és
- a kiszolgáló gép azon portcímére, melyen a szerver a kapcsolat-felvételi kérelmeket várja.

Kapcsolat nyitása a kliens oldalon:

```
try {
   Socket sock = new Socket(”king.iit.uni-miskolc.hu”, 2225);
} catch (UnknownHostException e) {
   System.out.println(”Nem találom a kiszolgálót.”);
} catch (IOException e) {
   System.out.println(”Sikertelen kapcsolatfelvétel.”);
}
```

Ha egyszer egy kapcsolatot sikerült felépíteni, akkor mind kliens mind szerver oldalon kinyerhetjük belőle a kommunikációhoz szükséges folyamokat:

```
InputStream in  = sock.getInputStream();
OutputStream out = sock.getOutputStream();
```

A *ServerSocket* fontosabb metódusai:
Konstruktorok (a *korlát* a kapcsolatok számát korlátozhatja, a *melyikcím* több hálózati interfésszel rendelkező gépek esetén használatos):

public *ServerSocket*(int port) **throws** IOException
public *ServerSocket*(int port, int korlát) **throws** IOException
public *ServerSocket*(int port, int korlát, InetAddress melyikcím) **throws** IOException

A saját host cím lekérdezése:

public InetAddress *getInetAddress*()

A portszám lekérdezése:

public int *getLocalPort*()

Várakozás kapcsolat felvételre:

public Socket *accept*() **throws** IOException

Várakozási idő beállítása, lekérdezése (a várakozási idő letelte után *java.io.InterruptedIOException* generálódik, a 0 érték esetén végtelenségig vár):

public void *setSoTimeout*(int timeout) **throws** SocketException
public int *getSoTimeout*() **throws** IOException

A *Socket* osztály legfontosabb metódusai:
Konstruktorok:

public *Socket*(String host, int port) **throws** UnknownHostException, IOException
public *Socket*(InetAddress address, int port) throws IOException
public *Socket*(String host, int port, InetAddress localAddr, int localPort) **throws** IOException
public *Socket*(InetAddress address, int port, InetAddress localAddr, int localPort) **throws** IOException

Távoli gép címe, helyi gép címe:
public InetAddress *getInetAddress*()
public InetAddress *getLocalAddress*()
Távoli gép portja, helyi gép portja:
public int *getPort*()
public int *getLocalPort*()
Csatornák kinyerése:
public InputStream *getInputStream*() **throws** IOException
public OutputStream *getOutputStream*() **throws** IOException
Késleltetett küldés beállítása, lekérdezése (több adat összegyűjtése egy csomagba):
public void *setTcpNoDelay*(boolean on) **throws** SocketException
public boolean *getTcpNoDelay*() **throws** SocketException
SO_LINGER beállítása lekérdezése (mennyi ideig tartsa még fenn a kapcsolatot, ha a kapcsolat lebontásra kerülne, de még lenne átküldendő adat):
public void *setSoLinger*(boolean on, int linger) **throws** SocketException
public int *getSoLinger*() **throws** SocketException
SO_TIMEOUT beállítása lekérdezése:
public void *setSoTimeout*(int timeout) **throws** SocketException
public int *getSoTimeout*() **throws** SocketException
Socket lezárása:
public void *close*() **throws** IOException

Kapcsolat nélküli protokoll (UDP)

A kommunikációs végpontokat a *DatagramSocket* osztály reprezentálja, az adatcsomagokat a *DatagramPacket* osztály.

A szerveralkalmazásnak létre kell hoznia egy *DatagramSocket* objektumot megadva a portszámot, amelyen a csomagot várja, valamint egy *DatagramPacket* objektumot, amelybe az üzenet érkezni fog. Ezután a *DatagramSocket receive*() metódusával várakozhat az üzenetre.
A kliensalkalmazásnak létre kell hoznia egy *DatagramSocket* objektumot, majd össze kell állítania egy elküldendő csomagot, azaz létrehoznia egy *DatagramPacket* objektumot. Az elküldendő csomagnak tartalmaznia kell az üzenetet, az üzenet hosszát, címzett gép címét, címzett gép portszámát. Ezután a *DatagramSocket send*() metódusával elküldhető a csomag. Természetesen a szerepek cserélődhetnek, tehát a szerveralkalmazás is állíthat össze és küldhet csomagot a kliensnek.

Az összeköttetés-mentes kapcsolattartás lehetőséget ad arra is, hogy ne csak egy gép számára küldhessük el az üzenetet. Ennek támogatására hozták létre a Java-ban a *MulticastSocket* osztály. Ennek segítségével csoportba foghatok össze több gépet és az elküldött üzenet mindegyik számára elmegy.

Internet címek kezelése, az InetAddress osztály

Az IP címeket reprezentáló osztály. Példányosítása nem a szokásos módon történik, hanem létezik három statikus metódusa, amely képes ilyen példányokat előállítani:
adott nevű gép:

public static InetAddress *getByName*(String host) **throws** UnknownHostException
public static InetAddress[] *getAllByName*(String host) **throws** UnknownHostException

lokalis gép:

public static InetAddress *getLocalHost*() **throws** UnknownHostException

Egy már létező InetAddress objektumról meg lehet tudni a
címet:

public byte[] *getAddress*()
public String *getHostAddress*()

a gép nevét:

public String *getHostName*()

multicast cím-e:

public boolean *isMulticastAddress*()

URL-ek kezelése

URL-ek

Az URL-ek Interneten elhelyezett objektumokra (bármilyen típusú adatforrásra) mutatnak. Meghatározzák egy adattömeg helyét az Interneten, valamint információt tartalmaznak az adatok kinyeréséhez használható protokollról is.
Az URL-ek többnyire sztring-reprezentációban jelennek meg.
Legáltalánosabb formája:

protokoll://kiszolgáló/hely/objektum

A Java nyelvben az URL-eket az *URL* osztály egy példánya reprezentálja. Az URL-hez való kapcsolódást egy URLConnection valósítja meg felhasználva a megfelelő protokollkezelő osztályt (URLStreamHandler leszármazott), és a tartalom értelmezéséhez a megfelelő tartalomkezelő osztályt (ContentHandler leszármazott).

Az URL osztály

Egy *URL* objektum alkalmas valamely URL-lel kapcsolatos összes információ kezelésére, valamint segítséget nyújt az általa meghatározott objektum kinyeréséhez.
URL objektum létrehozása:
karaktersorozatból:

```
URL homePage =new URL( ”http://www.iqsoft.hu/doc/homepage.html” );
```

részekből:

```
URL homePage = new URL(”http”,”www.iqsoft.hu”,”doc/homepage.html”);
```

Az *URL* objektum függetlenül attól, hogy az általa reprezentált objektum létezik-e vagy sem megkonstruálódik.
A konstrukció során az *URL* objektum nem teremt hálózati kapcsolatot a kiszolgáló géppel.

A konstrukció alatt a Java elemzi az URL specifikációját és kiemeli belőle a használni kívánt protokollt, majd megvizsgálja, hogy ismer-e hozzátartozó protokollkezelőt. Abban az esetben, ha nem ismer *MalformedURLException* kivétel generálódik.

Az URL objektum lehetőséget biztosít az URL tartalmának, mint objektumoknak a kinyerésére. Az objektumként való letöltés a **tartalomkezelők** (Content Handler) használatával valósul meg. Egy URL objektumkénti letöltését az URL *getContent*() metódusának meghívásával kezdeményezhetjük.
A *getContent*() meghívásával az URL a következőket teszi:
- felveszi a kapcsolatot a kiszolgáló géppel,
- letölti az adatokat a kliens gépre a megadott protokoll segítségével,
- megállapítja az adatok formátumát, és
- meghívja az adatformátumhoz tartozó tartalomkezelőt az objektum előállítására

Ha nincs megfelelő tartalomkezelő, akkor a *getContent*() egy *InputStream*-t ad vissza, melynek segítségével byte folyamként kezelhetjük az URL-t.
A getContent() által visszaadott referencia *Object* típusú tehát cast-olni kell a kapott objektum tényleges típusára. A cast-olás során *ClassCastException* keletkezhet.

Tartalom- és protokollkezelők
A tartalomkezelő tevékenysége a protokollkezelő objektum kimenetére épül.
A protokollkezelő az adatok hálózati átvitelért, míg a tartalomkezelő a protokollkezelő által dekódolt adatokból való objektumkonstrukcióért felelős, így működésük közt nincs átfedés.
A tartalomkezelő ugyanolyan inputból ugyanazt az objektumot készíti el, számára lényegtelen, hogy az ő bemenete milyen protokollon keresztül érkezik meg.
Így teljesen más protokollon alapuló URL-ek is használhatják ugyanazt a tartalomkezelőt (ha azt a tartalom indokolja).

A Protokollkezelő működése

A protokollkezelő két részből áll:
- URLStreamHandler (folyamkezelő) objektum
- URLConnection (URL kapcsolat) objektum

A *getContent()* metódushívás esetén az URL kiemeli az URL specifikációból a protokoll komponenst, és az alapján megkeresi a megfelelő *URLStreamHandler* objektumot. Az *URLStreamHandler* befejezi az URL specifikáció feldolgozását, és létrehoz egy *URLConnection* objektumot.

A protokollkezelők elhelyezése
A protokollkezelőket a csomagok hierarchiájában kell elhelyezni. Minden egyes protokollkezelőnek saját csomagot kell definiálni. Ez a csomag fogja tartalmazni a protokollkezelőhöz tartozó *URLStreamHandler* és *URLConnection* osztályokat. Ezeket a csomagokat a `net.www.protocol` csomagban kell elhelyezni. A protollhozhoz tartozó *URLStreamHandler* osztály neve kötelezően *Handler*, az *URLConnection* osztály elnevezésére nincs megszorítás.

Az *URLConnection* osztály

Az *URLConnection* osztály egy példánya a protokollkezelő kapcsolatát reprezentálja valamely hálózati erőforráshoz.

Sok hasznos metódusa mellett a legfontosabb a *getInputStream()* metódus, mely visszaadja a protokollkezelő által készített folyamot.
Metódusai segítségével elemezhetjük az erőforrás tartalmát.

Tartalomkezelők

A tartalomkezelő a protokollkezelő által készített nyers folyamot olvassa, és az abból nyert adatok alapján készíti el a visszaadandó objektumot.
Az URL a protokollkezelőt kérdezi meg a reprezentált objektum típusáról (*MIME típus*), így dönti el, hogy melyik tartalomkezelőt kell meghívnia. (Sok esetben a protokollkezelő az URL által képviselt objektumot tároló állomány kiterjesztéséből állapítja meg az objektum típusát.)

A Java programokban (*application*) a MIME típus és a megfelelő osztály összerendelésének feladata ránk vár egy *ContentHandlerFactory* osztály implementálásával, és üzembe helyezésével.
A Java programkákban (*applet*) nincs szükségünk saját *ContentHandlerFactory* definiálására, mert minden egyes böngésző program a sajátját használja.

Az *URL*, *URLStreamHandler*, *URLConnection* és *ContentHandler* osztályok szorosan együttműködnek.

Grafikus felület programozása, a java.awt csomag

Grafikus felhasználói felület készítésére a Java 1.2 verziójáig a java.awt csomag szolgált. A javax.swing csomag elkészülte óta inkább a swing osztályait használják a felület felépítése. Ennek ellenére a java.awt csomag ismerete nem felesleges, hiszen a swing nem leváltotta, hanem kiegészítette a java.awt-t. A swing grafikus komponensei az AWT komponenseire épűlnek, eseménykezelésként az AWT eseménykezelési modelljét használja.

Az AWT az Abstract Window Toolkit szavak rövidítése. Az abstract jelző arra utal, hogy az AWT a megjelenítéshez mindig az adott operációs rendszer grafikus felületének objektumait használja. Tehát, amikor létrehozok például java.awt.Button objektumot, akkor az adott operációs rendszer grafikus készletének nyomógomb objektuma jön létre.
Emiatt a grafikus programok is hordozhatóak lettek, de tényleges kinézetük függ az operációs rendszertől. A módszer másik hátránya az, hogy az AWT csak olyan grafikus elemekkel dolgozhat, amely minden operációs rendszer grafikus készletének a metszete.
Az AWT egy másik lényeges elve szintén arra szolgál, hogy a programok különböző hardveren is ugyanúgy nézzenek ki. Ez az elrendezés szervezők használata. Az elrendezés szervezők a grafikus komponensek különféle szabályok szerinti, automatikus elrendezését és méretezését végzik.

A grafikus felületű programok készítésének két nagyon fontos eleme: a felhasználói felület felépítése a komponensekből és az eseménykezelés.

Komponensek

A komponensek három kategóriába sorolhatók: konténerek, felületelemek (nyomógomb, lista, stb.), valamint menük. A konténerek olyan komponensek, amelyek más komponenseket (konténereket, felületelemeket) tartalmazhatnak.
A felületelemek és a konténerek közös őse a *Component* osztály.

Konténerek:
A konténerek közös ősosztálya a Container osztály. Ez a Component osztály leszármazottja, amely kiegészíti a komponensi funkciókat a konténer funkciókkal. Ennek gyermekei:

- **Window**: a képernyőablakok alaposztálya, két leszármazottja a
  - **Dialog**: dialógusablakok készítésére (leszármazottja a **FileDialog**)
  - **Frame**: normális kerettel, fejléccel, stb.-vel rendelkező ablak
- **Panel**: egy belső konténer (konténeren belüli konténer) megvalósítására szolgál (leszármazottja az **Applet**)
- **ScrollPane**: egy komponenssel rendelkező annak tartalmát görgetni képes konténer.

Felületelemek:

- **Label**: Címke, azaz statikus szöveg
- **Button**: Nyomógomb
- **Canvas**: Rajzoló felület
- **Checkbox**: Kijelölő négyzet, azaz kétállapotú négyzet
- **CheckboxGroup**: Választó karikák (rádió gombok), több *Checkbox*, melyekből egyszerre csak egy lehet kiválasztott (Object leszármazott)
- **Choice**: Legördülő lista
- **List**: Lista doboz
- **Scrollbar**: Görgető sáv
- **TextField**: Egysoros beviteli sor (*TextComponent* leszármazott)
- **TextArea**: Többsoros beviteli sor (*TextComponent* leszármazott):

Menük:
A menük őse a MenuComponent. Ennek leszármazottai:

- **Menubar**: egy alkalmazás fő menüje
- **MenuItem**: menüpontokat testesít meg. Ezek lehetnek:
  - **Menu**: újabb menüpontokat tartalmazhat (Ennek a leszármazottja a **PopupMenu**)
  - **CheckboxMenuItem**: a pipálható menüelem

Elrendezés szervezők

A komponensek elhelyezkedésének és méretének automatikus vezérlésére szolgálnak. A komponensek pakolási stratégiáit a konténer *setLayout()* metódusával adhatjuk meg. A setLayout() paramétereként meg kell adni a megfelelő elrendezési stratégiát megvalósító objektumot. A null paraméter azt jelenti, hogy nem használok automatikus elrendezést.

A méretek meghatározásához az elrendezés szervező felhasználja a komponens getPreferredSize(), getMinimumSize(), getMaximumSize() metódusait.

Elrendezés szervező osztályok:

FlowLayout
GridLayout
CardLayout
BorderLayout
GridBagLayout

A *FlowLayout* a komponensek méretét a legkedvezőbbre állítja és sorban egymás mellé helyezi őket egy megadott tájolás (középre, balra, jobbra) szerint. Ha nem fér több a sorba, akkor folytatja a következő sorban.

A *GridLayout* megadott sor és oszlopszámú rácsot definiál a konténerre, és a komponenseket átméretezi rács méretűvé és ezekbe a rácsokba rakja sorfolytonosan.

A *CardLayout* egy kártyapakliszerű elhelyezést biztosít, azaz a komponensek egymáson helyezkednek el. Különböző metódusokkal vezérelhető a komponensek sorrendje (show(), next(), previous(), first(), last()).

A *BorderLayout* a komponenseket a konténer égtájszerinti oldalához, illetve középre igazítja (north, south, west, east, center). Az északi és déli komponensek felveszik a legkedvezőbb magasságukat és vízszintesen kitöltik a teret. A keleti és nyugati komponensek felveszik a legkedvezőbb szélességüket és magasságukkal kitöltik az északi és déli komponensek közti teret. A középső komponens a maradék helyet tölti ki.

A *GridBagLayout* a *GridLayout*-hoz hasonlóan rácsokra osztja a konténert, de egy komponens több rácselemet is elfoglalhat. A komponensek elhelyezkedésének leírásához többnyire *GridBagConstraints* típusú objektumokat használunk.

Eseménykezelés

Az események forrásuk alapján két csoportra oszthatók: beviteli események (billentyűleütés, egérmozgatás, stb.) és komponensek által kiváltott események. Az események a megfelelő eseményosztályok példányai.
Az események lekezeléséhez eseményfigyelőkre van szükség. Az eseményfigyelők olyan objektumok, amelyek megvalósítják a megfelelő Listener interfészt.
Ahhoz, hogy egy eseményfigyelő megkapja az eseményt, regisztrálni kell az eseményforrást jelentő komponensnél.

Esemény osztályok

ComponentEvent: a komponensekkel kapcsolatos eseményeket jelenti:
- COMPONENT_MOVED
- COMPONENT_RESIZED
- COMPONENT_SHOWN
- COMPONENT_HIDDEN

ContainerEvent: a gyermek komponens hozzáadás és elvétel események:
- COMPONENT_ADDED
- COMPONENT_REMOVED

FocusEvent: Az input fókusz megnyerésével, elvesztésével kapcsolatos események:
- FOCUS_GAINED
- FOCUS_LOST

InputEvent: Bevitellel kapcsolatos események. Két fajtájuk van a
- **KeyEvent**: A billentyűzetről érkező események:
  - KEY_PRESSED
  - KEY_RELEASED
  - KEY_TYPED
- **MouseEvent**: Az egérről érkező események:
  - MOUSE_CLICKED
  - MOUSE_PRESSED
  - MOUSE_RELEASED
  - MOUSE_EXITED
  - MOUSE_ENTERED
  - MOUSE_MOVED
  - MOUSE_DRAGGED

PaintEvent: A komponens megjelenítésével kapcsolatos események:
- PAINT
- UPDATE

WindowEvent: Az ablakkal történt események:
- WINDOW_CLOSED
- WINDOW_OPENED
- WINDOW_CLOSING
- WINDOW_ICONIFIED
- WINDOW_DEICONIFIED
- WINDOW_ACTIVATED
- WINDOW_DEACTIVATED

AdjustmentEvent: A Scrollbar eseménye:
- ADJUSTMENT_VALUE_CHANGED

ActionEvent: A nyomógombok és menüelemek aktiválásának eseménye:
- ACTION_PERFORMED

ItemEvent: A List és Choice objektumok eseményei:

ITEM_STATE_CHANGED

TextEvent: A TextField és TextArea objektumok eseménye:
TEXT_VALUE_CHANGED

Mindegyik eseményosztály szolgáltat metódusokat arra, hogy az eseményobjektumból kinyerhessük a még szükséges információkat. Például a KeyEvent-ből lekérdezhető a lenyomott billentyű.

Eseményfigyelő interfészek

Minden esemény osztályhoz létezik neki megfelelő eseményfigyelő interfész. Az egyetlen kivétel a MouseEvent, amelyhez két Listener is létezik, különválasztva a MOUSE_DRAGGED, MOUSE_MOVED eseményeket a többitől. Létezik ComponentListener, ContainerListener, FocusListener, KeyListener, MouseListener, MouseMotionListener, WindowListener, AdjustmentListener, ItemListener, ActionListener, TextListener.
Az eseményfigyelő interfészek minden eseményhez tartalmaznak metódus deklarációt.
Például a KeyListener metódusai:

void *keyTyped*(KeyEvent e)
void *keyPressed*(KeyEvent e)
void *keyReleased*(KeyEvent e)

Eseményfigyelők regisztrálása

Minden komponens rendelkezik az általa kiválthatott eseményekhez regisztráló metódussal

void **addXxxxListener**(XxxxListener l)

alakban. Ezek segítségével kell bejegyezni azokat az objektumokat, amelyeket értesíteni kell az esemény bekövetkezéséről. A bejegyzett objektum megfelelő metódusa meghívódik az esemény előfordulásakor. Például *ActionEvent* esetén meghívódik az eseményfigyelő objektum (*ActionListener* interfészt megvalósításával definiált) *actionPerformed*() metódusa,

Esemény adapter osztályok

Ha egy Listener interfésznek több metódusa is van, létezik hozzá az interfészt üres törzsű metódusokkal megvalósító adapter osztály. Ez lehetővé teszi az interfész részleges implementálását.
Adaptert célszerűen beágyazott osztályként használunk.

Komponensek megjelenítése

A komponenseket megjelenítetni a show() illetve a setVisible() metódusokkal lehet. Egy konténer megjelenésekor a benne levő komponensei is megjelennek.

A komponensek megjelenéséhez az AWT szükség esetén kéri a komponenseket, hogy rajzolják ki magukat. Induláskor az AWT a **paint**()-et hívja, ami során az egész komponenst ki kell rajzolni; később az **update**() hívódik.
A **repaint**() hívása kényszeríti, hogy a komponens *update*()-jét hívja meg.

Csak a **Canvas** és **Panel** (Applet) objektumok felületére ajánlatos rajzolni. Nem érdemes nehézsúlyú komponensek paint()-jét felüldefiniálni mivel a peer elrontja rajzainkat.

A Component osztály

Eseménykezelő metódusai:

void *addXxxxxListener*(XxxxxListener l)
void *removeXxxxxListener*(XxxxxListener l)
protected void *processXxxxxxEvent*(XxxxxEvent e)

ahol Xxxxx lehet *Component, Focus, InputMethod, Key, Mouse, MouseMotion, PropertyChange*.

protected void *disableEvents*(long eventsToDisable)
protected void *enableEvents*(long eventsToEnable)
void *dispatchEvent*(AWTEvent e)
protected void *processEvent*(AWTEvent e)

Méret, elhelyezkedés, szín, font, cursor:

Rectangle getBounds()
int getHeight()
int getWidth()
int getX()
int getY()
Point getLocation()
Point getLocationOnScreen()
Dimension getSize()
float getAlignmentX()
float getAlignmentY()
void setBounds(int x, int y, int width, int height)
void setBounds(Rectangle r)
void setLocation(int x, int y)
void setLocation(Point p)
void setSize(Dimension d)
void setSize(int width, int height)

Dimension getMaximumSize()
Dimension getMinimumSize()
Dimension getPreferredSize()

Color getBackground()
Color getForeground()
void setBackground(Color c)
void setForeground(Color c)
ColorModel getColorModel()

Font getFont()
FontMetrics getFontMetrics(Font font)
void setFont(Font f)

Cursor getCursor()
void setCursor(Cursor cursor)

Megjelenítés:
void paint(Graphics g)
void paintAll(Graphics g)
void repaint()
void repaint(int x, int y, int width, int height)
void repaint(long tm)
void repaint(long tm, int x, int y, int width, int height)
void update(Graphics g)
void show() Deprecated. Helyette setVisible(boolean).
void setVisible(boolean b)
boolean imageUpdate(Image img, int flags, int x, int y, int w, int h)
boolean isDisplayable()
boolean isShowing()
boolean isValid()
boolean isVisible()
boolean isDoubleBuffered()
void invalidate()
void validate()
boolean isEnabled()
void setEnabled(boolean b)

Fókusz:
void requestFocus()
void transferFocus()
boolean hasFocus()

Image:
int checkImage(Image image, ImageObserver observer)
int checkImage(Image image, int width, int height, ImageObserver observer)
Image createImage(ImageProducer producer)
Image createImage(int width, int height)

Egyéb:
String getName()
void setName(String name)
Container getParent()
Toolkit getToolkit()
Graphics getGraphics()
Locale getLocale()
void setLocale(Locale l)
boolean isLightweight()
boolean isOpaque()

boolean prepareImage(Image image, ImageObserver observer)
Component getComponentAt(int x, int y)
Component getComponentAt(Point p)
boolean contains(int x, int y)
boolean contains(Point p)

Példaprogram:
Egy ablakban egy Helló Világ feliratú nyomógomb, melyre rákattintva kilép.

```
import java.awt.*;
import java.awt.event.*;

public class Hello extends Frame {
  private Button gomb;

  class Gombra implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      System.exit(0);
    }
  }

  class Bezar extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }

  public Hello() {
    super();
    setTitle(”Üdvözlet”);
    setBounds(50,50,200,200);
    addWindowListener(new Bezar());

    Button gomb = new Button(”Helló Világ”);
    gomb.addActionListener(new Gombra());
    add(gomb);
  }


  public static void main(String[] args) {
    Hello h = new Hello();
    h.setVisible(true);
  }
}
```