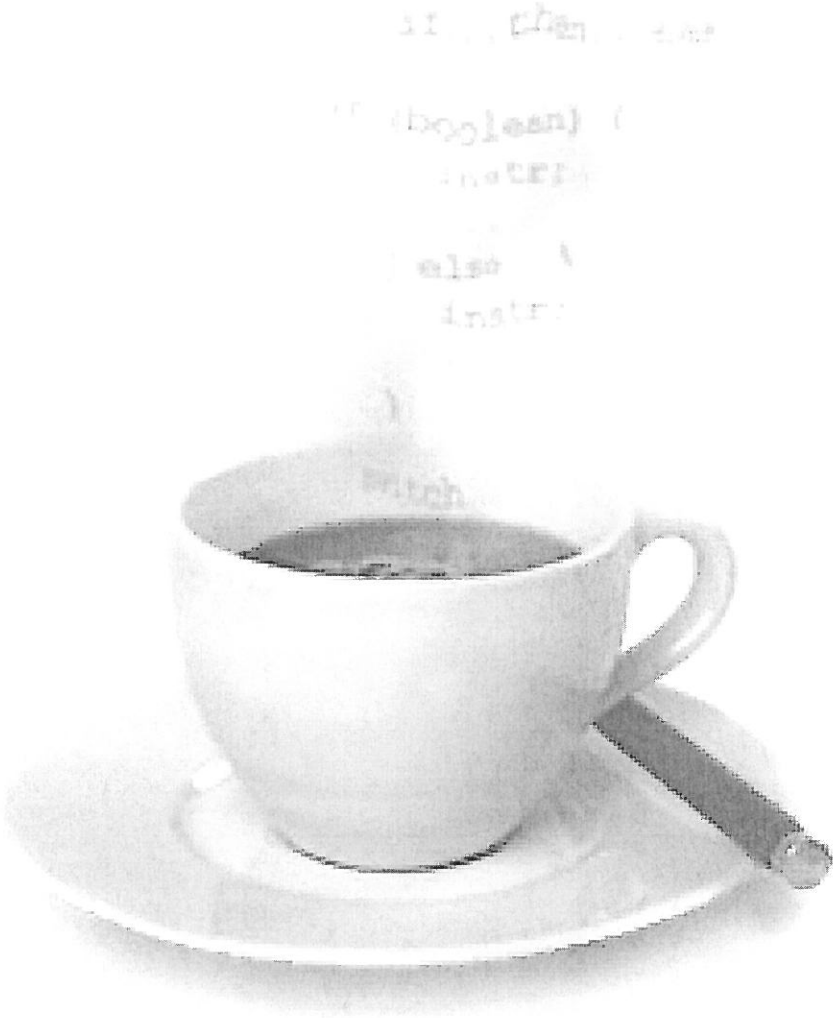
Nagy Gusztáv

Java programozás



1.3. verzió

2007. február

Jogi nyilatkozat

E jegyzet hivatalos honlapjáról (http://nagygusztav.hu) tölthető le a mindenkori legfrissebb verzió.

Bevezetés

Felhasználás

Ezzel a jegyzettel arra vállalkozok, hogy a Kecskeméti Főiskola GAMF Karán tanuló műszaki informatikus hallgatók „kezébe" olyan írásos anyagot adjak, amely az előadások és gyakorlatok mellett további segítséget ad a Java nyelv alapjainak és alapvetőbb osztályainak alapos megismerésére.

A jegyzet használatához nem nélkülözhetetlen, de erősen javasolt az előadások látogatása és a gyakorlatokon való aktív részvétel. A jegyzet alapvetően a tanórák menetéhez kapcsolódó lineáris feldolgozásra készült, de a fejezetek egy része nem épít a közvetlen megelőző fejezetekre.

Az egyes fejezetek tananyagának elsajátításához az elméleti rész átolvasása után az ellenőrző kérdések alapos átgondolását, valamint a gyakorló feladatok megoldását javaslom. Enélkül a tantárgy teljesítése a hallgatók többsége számára nem lesz eredményes.

A jegyzet feltételezi a *C++* programozási nyelv minimum közép szintű ismeretét. Az anyag elsajátítása ennek hiányában sem lehetetlen, de több időt kell ráfordítani.

A jegyzet alapos áttanulmányozása és a Java programozás komolyabb gyakorlása után akár a Sun **Java programozói minősítés**ének megszerzésére is lehet készülni.

Köszönet

A jegyzet elkészítéséhez elsősorban a *Sun Microsystems Java Tutorial*-ját használtam. További információkért a *http://java.sun.com* oldalt, valamint az ELTE pillanatnyilag nehezen elérhető, de nagyon népszerű *„Java Útikalauz programozóknak"* című könyvét ajánlom.

Szeretnék köszönetet mondani a *Sun*-nak a Java megalkotásáért, a hallgatóknak, akik a jegyzet alapjául szolgáló *Java Tutorial* nyers fordításának elkészítésében részt vettek, és végül, de nem utolsó sorban kollégámnak, Gurka Dezsőné Csizmás Editnek a jegyzet alapos átolvasásáért és építő ötleteiért.

Tervek

A tavaszi félév folyamát ellenőrző és gyakorló feladatokkal szeretném a jegyzetet bővíteni. Ennek párhuzamosan távoktatási jegyzetté alakítás is meg fog történni.

Kecskemét, 2007. február

a szerző

Tartalomjegyzék

1. Első lépések

Ez a fejezet a Java programozás alapjaihoz mutat utat. Bemutatja, hogyan tudjuk a Javát beszerezni, és hogyan tudjuk elkészíteni, fordítani és futtatni az egyszerű Java programokat. Végül megismerhetjük azt a háttértudást, amely a programok működésének megértéséhez szükséges.

1.1. Az első csésze kávé

A Java programok készítéséhez szükségünk lesz a következőkre:

Java fejlesztőkörnyezet (Java SE Development Kit)

A JDK-t (*Java SE Development Kit*) a

http://java.sun.com/javase/downloads/index.jsp

címről tölthetjük le, majd értelemszerűen telepítsük is. (Fontos, hogy a **JDK**-t, és ne a **JRE**-t töltsük le!)

Megjegyzés: Természetesen újabb verzió megjelenése esetén azt érdemes letölteni és telepíteni.

A telepítés a Windowsban megszokott egyszerűséggel történik, általában elegendő a *Next* gombra kattintani.

Dokumentáció

A Java fejlesztőkörnyezeten kívül érdemes beszerezni (bár a fordításhoz közvetlenül nem szükséges) az API (*Application Programming Interface*) dokumentációt is (szintén az előző letöltési oldalról indulva). Ez a Java platformon használható több ezer osztály igen részletes dokumentációját tartalmazza. A tömörített ZIP állomány tartalmát (*docs* könyvtár) a fejlesztőkörnyezet gyökérkönyvtárába (pl. *C:\Program Files\ jdk1.6.0_01*) érdemes kicsomagolni.

Szövegszerkesztő

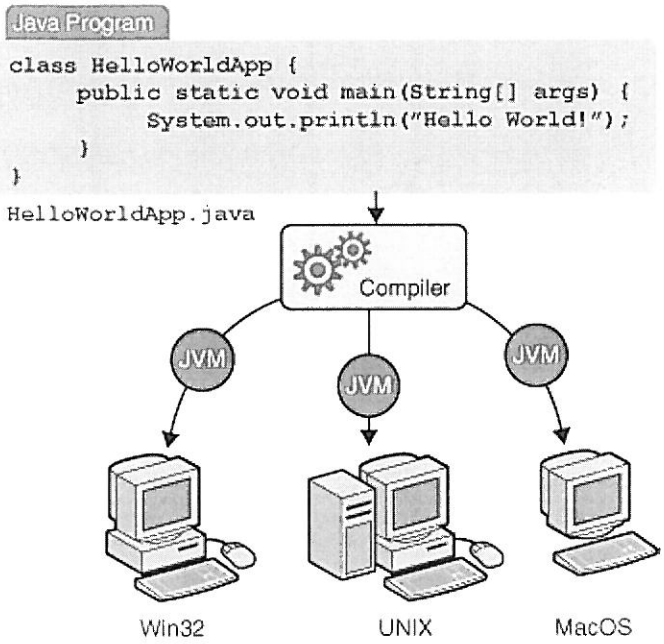
Bármilyen editor megfelel a jegyzettömbtől (Notepad.exe) az összetett programozói editorokig.

Nem érdemes a tanulás legelején olyan integrált fejlesztőkörnyezet (*IDE – Integrated Development Environment*) alkalmazni, amelyik bizonyos nyelvi elemeket elrejt a programozó elől (pl. *JBuilder*), ugyanakkor a rendszer rengeteg szolgáltatása között elveszik a kezdő felhasználó. Elegendő a kódkiemelést biztosító, esetleg a gépelést könnyítő szerkesztő használata is. A nyelvtani alapok, fordítás, futtatási ciklus begyakorlása után már praktikus lehet az átállás egy komolyabb támogatást nyújtó eszközre.

A Jegyzet 25. fejezetében három népszerű editor használatához találnak tippeket. (E fejezet példái a parancssori fordítás-futtatás kissé nehézkes módszerét mutatják be.)

1.1.1 Az első alkalmazás létrehozása

A fejlesztés menetét jól mutatja a következő ábra: a forrásállományból a fordítás hatására előálló bájtkódot (*bytecode*) különböző (Java Virtuális Gépet, JVM-et tartalmazó) operációs rendszeren tudjuk futtatni.



Az első program (*HelloWorldApp*) egyszerűen kiírja a képernyőre a *Hello World!* üzenetet. A következő lépések szükségesek:

Hozzunk létre egy forrásállományt

A forrásállomány egyszerű szöveges állomány a Java nyelv szintaxisa szerint. A Java forrásállomány kiterjesztése *.java*.

Megjegyzés: *Unicode* (egész pontosan UTF) kódolású forráskód is használható!

(A **Unicode** kódolás két bájton tárol egy-egy karaktert, így a legtöbbet használt nyelvek legtöbb betűje és írásjele ábrázolható vele.)

Az első programunk:

```
public class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Mentsük el a programot *HelloWorldApp.java* néven.

Megjegyzés: A Java nyelvben és az állománynévben is különbséget kell tenni kis-, és nagybetű között, függetlenül a Windowsban megszokott lazaságtól.

Fordítsuk le a forrásállományt bájtkódra

A *javac* (bin\javac.exe) fordító a szövegből olyan utasításokat állít elő, amelyeket a JVM (*Java Virtual Machine*, Java virtuális gép) végre tud hajtani. A bájtkódú programállomány kiterjesztése *.class*.

- Nyissunk meg egy parancssor ablakot (*Start menü / Futtatás / cmd.exe*), majd állítsuk be az aktuális könyvtárat a Java *bin* alkönyvtárára (pl. *cd "C:\Program Files\jdk1.6.0_01\bin"*).
- Indítsuk el a *javac* fordítót: *javac HelloWorldApp.java*. (Bizonyos esetekben szükség lehet a forrásállomány teljes elérési útjának megadására.)

Futtassuk a programot tartalmazó bájtkód állományt

A Java értelmező (bin\java.exe) a számítógépre telepített Java VM számára értelmezi a bájtkódú program utasításait, a VM pedig futtatja azokat.

Gépeljük be (kiterjesztés nélkül):

```
java HelloWorldApp
```

Ha mindent jól csináltunk, megjelenik a konzol ablak következő sorában a program üdvözlete.

1.2. Bevezetés a Java technológiába

A Java technológia egyaránt programozási nyelv és platform.

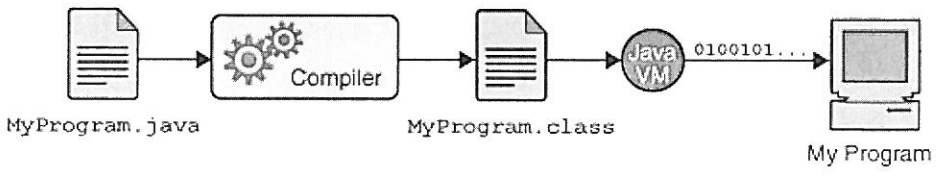
1.2.1 A Java programozási nyelv

A Java egy magas szintű nyelv a következő főbb jellemzőkkel:

- egyszerű
- objektumorientált
- előfordított
- értelmezett
- robusztus
- biztonságos
- semleges architektúrájú
- hordozható
- nagy teljesítményű
- többszálú
- dinamikus

Megjegyzés: Valószínűleg a felsorolás egy része most még nem sokat mond. Mire azonban a jegyzet végére érnek, és a Java nyelvű fejlesztésben legalább alap szintű gyakorlatuk lesz, ez a lista sokkal többet fog mondani.

A legtöbb programozási nyelv esetén fordítást vagy értelmezést hajtunk végre, mielőtt a program futna a gépünkön. A Java esetén a kettőnek egy különös keverékét használjuk. Először a forrásprogramot (*myProgram.java*) a fordító (*compiler, bin\javac.exe*) egy közbülső nyelvre fordítva Java bájtkódot (*myProgram.class*) állít elő, és ezt a platformfüggetlen kódot értelmezi és futtatja a Java VM (*interpreter, bin\java.exe*). A fordítás egy alkalommal történik, az értelmezés pedig minden alkalommal, ahányszor a program végrehajtódik. A következő ábra ennek működését illusztrálja.



A Java bájtkódot gépi kóddá alakítja a Java VM. Minden Java értelmező, akár a fejlesztőkörnyezet, akár egy böngészőben futó applet, tartalmaz Java VM-et a futtatáshoz.

A Java bájtkód segítségével megoldható az, hogy csak egyszer kell megírni egy Java programot, majd tetszőleges (megfelelő verziójú) Java VM-et tartalmazó gépen futtatni lehet. A Java programunkat bármelyik operációs rendszeren telepített fordítóval le lehet fordítani, mindenütt használható lesz.

Megjegyezés: A fordítást és értelmezést is alkalmazó hibrid megoldás manapság egyre nagyobb népszerűségnek örvend. A Microsoft .Net platformja sok architektúrális elemet vett át a Javától, a web hagyományos értelmező megoldásai ma már sokszor előfordítással is kombinálhatók.

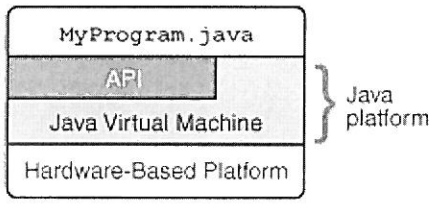
1.2.2 Java platform

A platform hardver vagy szoftverkörnyezet, ahol a programok futnak. A legtöbb platform a hardvert és az operációs rendszert jelenti. A Java platform annyiban különbözik a legtöbb más platformtól, hogy teljesen szoftverplatform, és más hardver alapú platformokra épül. A Java platform két komponensből áll:

- Java VM
- Java API

A Java API igen sok (több ezer) használatra kész szoftverkomponenst tartalmaz: csomagokba szervezett osztályokat és interfészeket.

A következő ábra bemutatja a Java platform működését.



A natív kód olyan kódot jelent, amelyik a hardveren közvetlenül futtatható. A platformfüggetlen Java kód valamivel lassabb, mint a natív kód. Azonban jó fordítóval, optimalizált értelmezővel, és JIT bájtkód fordítóval a különbség elég kicsi lehet. A mai futtatókörnyezetek már tartalmazzák a JIT (*Just in time*) fordítót, amivel az első futtatás előtt natív kódra fordul a bájtkód, így a további futások során már közvetlenül a natív kód futtatható.

Megjegyzés: Az előzőek következménye, hogy egy Java alkalmazás első futtatása több ideig tarthat, de a további futtatásoknál ez az időveszteség nem fog jelentkezni.

1.3. Mire jó a Java technológia?

A legtöbb Java platformra készült program asztali alkalmazás vagy **applet**. Ha a weben szörfözünk, találkozhatunk appletekkel. Az applet olyan program, amely bizonyos megszorításokkal futtatható Javát ismerő böngészőben. Kezdetben ezt látványos grafikai effektusok készítésére használták. Mára ez a felhasználás visszaszorult, és viszonylag ritkán találkozhatunk appletekkel.

Az eredeti célja szerint a Java magas szintű programozási nyelv és erős szoftverplatform kíván lenni. A gazdag osztálykönyvtár segítségével nagyon sokféle programot készíthetünk.

Az asztali **alkalmazás** olyan program, amely közvetlenül a Java platformon (pl. nem böngészőben) futtatható. Az alkalmazások speciális fajtája szerverként fut, hálózati klienseket kiszolgálva. Például lehet webszerver, proxy-szerver, levelező szerver vagy nyomtató szerver.

Szintén speciális program a **szervlet** (*servlet*). Szerver oldalon fut, de nem önállóan, hanem egy szerver-futtatókörnyezet részeként. Pl. egy portált ki lehet szolgálni néhány szervlet együttesével, vagy akár egyetlen szervlettel. Ebben az esetben a szervlet a webszerver részeként fut. A szervletek hasonlóak az appletekhez, mivel futásidejű kiterjesztései a (szerver) alkalmazásoknak.

A mobil telefonon, kézi számítógépen futó alkalmazást **midletnek** hívjuk.

Hogyan nyújtja az API ezt a sokféle támogatást? Szoftverkomponensek csomagjaiként, amelyek sokféle feladatot ellátnak. A Java platform minden teljes implementációja (például a midlet futtatókörnyezet nem teljes) rendelkezik a következő tulajdonságokkal:

- **Alap összetevők**: objektumok, sztringek, szálak, számok, I/O, adatstruktúrák, dátum és időkezelés, stb.
- **Appletek**: a szokásos felhasználások
- **Hálózatok**: URL, TCP, UDP, socket-ek, IP címzés
- **Nemzetközi programozás**: Segítség az egész világon használható alkalmazások írásához. A programok könnyedén tudnak alkalmazkodni a helyi sajátosságokhoz, és többféle nyelven kommunikálni a felhasználókkal
- **Biztonság**: alacsony és magas szintű védelem, beleértve az elektronikus aláírást, titkos-, és nyilvános kulcsú titkosítást, hozzáférés-szabályozást és azonosítást
- **Szoftver komponensek**: a JavaBeans használatával könnyen összeilleszthető komponenseket fejleszthetünk
- **Objektum szerializáció**: lehetővé teszi a könnyűsúlyú perzisztenciát és az RMI-t
- **JDBC**: relációs adatbázis-kezelők széles köréhez nyújt egységes elérési felületet

A Java platform ezen felül tartalmaz API-t a 2D és 3D grafikához, szerverekhez, telefóniához, beszédfeldolgozáshoz, animációhoz stb.

1.4. Még egyszer a *HelloWorld* programról

1.4.1 Megjegyzések a Java nyelvben

```
/*
 * A HelloWordApp program kiírja a köszöntő szöveget
 */
public class HelloWorldApp {
    public static void main(String[] args) {
        // Kiírja: "Hello World!"
        System.out.println("Hello World!");
    }
}
```

A Java nyelv a megjegyzések három típusát támogatja. Hagyományos (C stílusú) megjegyzés:

```
/* szöveg */
```

A fordító a begépelt szöveget figyelmen kívül hagyja a /*-tól a */-ig.

```
/** dokumentáció */
```

Ez egy dokumentációs megjegyzés, a fordító figyelmen kívül hagyja, mint az előző típust is, de a *javadoc* eszköz (bin\javadoc.exe) segítségével automatikusan lehet generálni *hypertext* (HTML) dokumentációt, ami felhasználja a dokumentációs megjegyzéseket is.

Megjegyzés: a letölthető Java dokumentáció is ez alapján készült.

```
// szöveg
```

A fordító figyelmen kívül hagyja a sort a //-től a sor végéig.

1.4.2 Osztálydefiníció

A következő kód mutatja az osztálydefiníciós blokkot.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Az osztály (*class*) alapvető építőeleme az objektumorientált nyelveknek. Az osztály az adatok és viselkedések összességéből álló példányok sablonját adja meg. Amikor az osztályból példányt hozunk létre, akkor tulajdonképpen egy olyan objektum jön létre, amelyik úgy épül fel, és úgy viselkedik, mint az osztály egyéb példányai.

Az adatok az objektumpéldányok változóiként írhatók le, a viselkedések pedig a metódusokkal.

A valós életből egy hagyományos példa a téglalap osztály. Az osztály tartalmaz változókat a pozíció, valamint a szélesség és magasság leírására, és tartalmaz metódust a terület kiszámítására.

A Java nyelvben a legegyszerűbb osztálydefiníció a következő:

```
class name {
    . . .
}
```

A *class* kulcsszóval és az osztály nevével kezdődik az osztálydefiníció, majd kapcsos-zárójelek között változók és metódusok következnek. A korábbi példa alkalmazásunkban nincs változó, és csak egyetlen metódus van *main* néven.

1.4.3 A *main* metódus

Minden Java alkalmazásnak tartalmaznia kell *main* metódust a következő deklarációval:

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

A *main* metódus deklarációja három módosítót tartalmaz:

- *public*: jelzi, hogy a metódust más osztálybeli objektumokból is meg lehet hívni
- *static*: jelzi, hogy a *main* osztálymetódus
- *void*: jelzi, hogy a metódusnak nincs visszatérési értéke

Hogyan hívódik meg a *main* metódus?

A Java nyelvben a *main* metódus hasonló a C nyelv *main* függvényéhez. Amikor a Java értelmező végrehajt egy alkalmazást, először meghívja az osztály *main* metódusát. Az alkalmazásunk további metódusait (közvetlenül vagy közvetve) a *main* fogja meghívni. Ha *main* nélkül próbálunk egy alkalmazást futtatni, az értelmező megtagadja a végrehajtást, és hibaüzenetet kapunk.

Megjegyzés: A *main* függvényre csak hagyományos alkalmazások esetén van szükség. Beágyazott Java program (*applet, midlet, servlet*) esetén a program az őt tartalmazó futtatókörnyezet (böngésző, szervlet konténer) speciális közreműködésével fog futni.

A *main* metódus paramétere

A *main* metódusnak egy paramétere van, amely sztringek tömbje. Ez a tömb a parancssori paramétereket tartalmazza. A példaprogramunkban nem foglalkoztunk az esetleges parancssori paraméterekkel.

Megjegyzés: A C nyelvtől eltérően az *args* tömb nem tartalmazza a program nevét. Ez ugyanis az önelemzés (*reflection*) módszerével könnyen megállapítható. Másrészt a Java nyelvben a tömb objektum, tudja a méretét, így nincs szükség további paraméterre.

1.4.4 Osztályok és objektumok használata

A példaalkalmazásunk egy nagyon egyszerű Java program, ezért nincs szüksége arra, hogy más osztályokat alkalmazzunk, a képernyőre írást kivéve. Összetettebb programok használni fognak más segítő osztályokat.

A *HelloWorld* alkalmazás egyedül a *System* osztályt használja. Ez az osztály rendszerfüggetlen hozzáférést tesz lehetővé a rendszerfüggő szolgáltatásokhoz.

Példánkban egy osztályváltozó (*out*) példány metódusát (*println*) hívjuk meg.

Osztálymetódusok és változók használata

A *System.out* a *System* osztály *out* változójának a teljes neve. (A *System* osztályból soha nem fog példányosítani az alkalmazásunk, csupán az osztály nevével tudunk hivatkozni a változóra. Ez azért van, mert az *out* osztályváltozó, és az osztályhoz van kapcsolva, nem egy példányhoz.

Példánymetódusok és változók használata

Azokat a metódusokat és változókat, amelyek nem az osztályhoz, hanem egy konkrét objektumpéldányhoz vannak kapcsolva, példánymetódusoknak vagy példányváltozóknak nevezzük.

Az *out* osztályváltozó, a *PrintStream* osztály egy példányára hivatkozik, és megvalósítja a standard kimenetet. Amikor a *System* osztály betöltődik, a *PrintStream* osztály példánya jön létre, és *System.out* változó néven hivatkozhatunk rá. Most már van egy példányunk, meg tudjuk hívni a példánymetódusát:

```
System.out.println("Hello World!");
```

Ahogy látszik, a példánymetódusok és változók használata hasonló az osztálymetódusok és változók működéséhez.

A Java fordító megengedi egy lépésben a többszörös hivatkozást:

```
System.out.println("Hello World!");
```

1.5. Ellenőrző kérdések

- Mi a különbség a gépi kód és a Java bájtkód között?
- Mi a Java platform két fő összetevője?
- Mi a Java VM?
- Mi a legfontosabb feltétele annak, hogy egy adott gépen lehessen Java programot futtatni?
- Mi a Java API?
- Milyen parancs indítja el a Java fordítóprogramot?
- Milyen parancs indítja el a Java VM-t?
- Mi az a Java nyelvi szerkezet, amivel a konzolra lehet írni?
- Mire szolgálnak a megjegyzések a programban?
- Mi a különbség a /* ...*/, /** ...*/ és a //... megjegyzés-szintaktika között?
- Melyik metódus fog először elindulni egy Java program esetén? Írjon egy egyszerű példát az alkalmazására!

Igaz vagy hamis? Indokolja!

- A natív kód az ember számára könnyen értelmezhető programkód.
- A Java fordító gépfüggetlen közbenső kódot, bájtkódot generál.
- Az *interpreter* a tárgykódot visszaalakítja forráskóddá.
- A forráskód egy szöveg, melyet a fordítóprogram értelmez, illetve fordít le.
- A forráskód a számítógép által értelmezhető, közvetlenül futtatható kód.

Mit tartalmaz a *main* függvény paraméterének 0. eleme?

- A program nevét
- A paraméterek számát
- Az első paramétert

2. Objektumorientált paradigma

Az objektumorientált programozás alapfogalmaival korábban már bizonyára minden olvasó találkozott. A téma rendkívüli fontossága miatt egy rövid bevezetést is olvashatnak ebben a fejezetben.

A jegyzet példái általában elég egyszerűek, de érdemes minél előbb megismerkedni egy olyan jelölésmóddal, amivel az objektumorientált programunkat előre megtervezhetjük. Leíró eszközként a leginkább elterjedt UML (Unified Modeling Language[1]) jelöléseivel fog találkozni a jegyzetben a tisztelt olvasó.

2.1. Az objektum

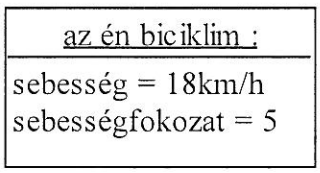
Az objektumok az objektumorientált technológia alapjai. Néhány példa a hétköznapi életből: kutya, asztal, tv, bicikli. Ezek a valódi objektumok két jellemzővel rendelkeznek: állapottal és viselkedéssel. Például a kutya állapotát a neve, színe, fajtája, éhessége stb. jellemzi, viselkedése az ugatás, evés, csaholás, farok-csóválás stb. lehet. A bicikli állapotát a sebességfokozat, a pillanatnyi sebesség, viselkedését a gyorsulás, fékezés, sebességváltás adhatja.

A programbeli objektumok modelljei a valódi objektumoknak. Az objektum állapotát egy vagy több változóval, a viselkedését az objektumhoz rendelt metódussal (függvénnyel) írjuk le.

Definíció: Az **objektum** változókból és kapcsolódó metódusokból felépített egység.

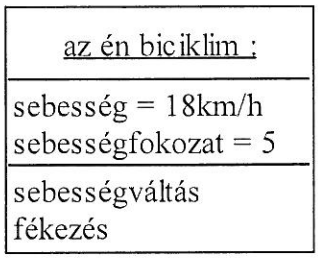
A valós élet objektumait leírhatjuk program objektumokkal. Ha szükség van arra, hogy valódi kutyákat ábrázoljunk egy animációs programban, akkor használhatunk program objektumokat az elvont fogalmak modellezésére. Például egy hétköznapi eseményt modellezhet egy billentyűleütés vagy egérkattintás.

Egy biciklit modellező objektum változókkal írja le a pillanatnyi állapotot: a sebesség 18 km/h, és a sebességfokozat 5-ös. Ezeket a változókat példányváltozóknak nevezzük, mert ezek egy konkrét bicikli állapotát írják le. Az objektumorientált terminológiában egy önálló objektumot példánynak is nevezünk. A következő ábra bemutat egy biciklit modellező objektumot az UML objektumdiagramja (*Object diagram*) segítségével.



A bicikli tud fékezni, sebességfokozatot váltani is. Ezeket a metódusokat példánymetódusoknak hívjuk, mivel egy konkrét bicikli (példány) állapotában képesek változást elérni.

[1] http://www.uml.org/



Az objektum tulajdonságait szokás a külvilágtól elrejteni, és csak a metódusokon keresztül befolyásolni.

Definíció: Az objektum változók becsomagolását, védőőrizetbe helyezését **egységbezárás**nak nevezzük.

Időnként – gyakorlati megfontolásból – egy objektum megmutat néhány változóját és elrejti néhány metódusát. A Java nyelvben az objektum meg tudja határozni, hogy négy hozzáférési szint közül melyiket választja az egyes változók és metódusok számára. A hozzáférési szint határozza meg, hogy más objektumok és osztályok hozzá tudjanak-e férni az egyes változókhoz és objektumokhoz.

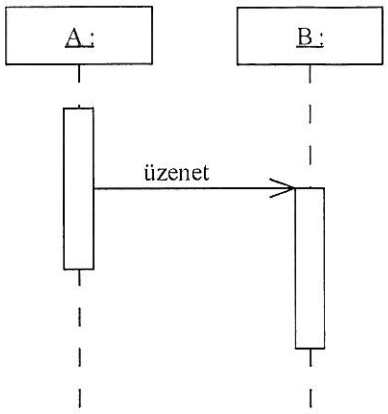
Az egységbezárás tiszta alkalmazása esetén két egyszerű, de nagyon hasznos előnye lesz a szoftverfejlesztőnek:

- **Modularitás**: Az objektum forráskódja **független marad** más objektumok forráskódjától. Ezen kívül az objektum könnyen tud illeszkedni a rendszer különböző részeihez.
- **Információ elrejtés**: Az objektum a **publikus interfészén** keresztül nyújt kommunikációs lehetőséget a többi objektum felé. Az objektum gondoskodik a saját adatairól, és csak a metódusain keresztül ad változtatási lehetőséget a külső objektumoknak. A külső objektumoknak igazából nem is kell tudnia arról, hogy az objektum állapota milyen belső változókkal van reprezentálva, csak a kívánt viselkedést kell kérnie a metódusokon keresztül.

2.2. Az üzenet

Amíg csak egy objektumunk van, addig nem sok haszna van a programnak. Általában egy objektum csak egy kis részét jelenti egy nagyobb alkalmazásnak. Ezért kölcsönhatás van az objektumok között. A biciklink egy összetett szerkezet, magában mégis használhatatlan, kapcsolatba kell kerülnie más objektummal, pl. velünk a pedálon keresztül.

A program objektumok hatnak egymásra és kommunikálnak egymással üzeneteken keresztül. Amikor az *A* objektum meghívja a *B* objektum egy metódusát, tulajdonképpen egy üzenetet küld neki. Ezt az UML szekvencia diagramja a következő módon ábrázolja (az idő fentről lefelé halad):



Néha a fogadó objektum több információt igényel, hogy pontosan tudja, mi a dolga. Például amikor sebességet váltunk a biciklin, megadjuk a kívánt sebességváltás irányát is. Ezt az információt paraméterként adjuk az üzenethez.

Az üzenet három része összefoglalva:

- Melyik objektum az üzenet címzettje
- A végrehajtandó metódus neve
- Az esetleges paraméterek

Ez a három összetevő elegendő, hogy a meghívott objektum végrehajtsa a kívánt metódust. Az üzenetek két fontos előnnyel járnak:

- Egy objektum viselkedését meghatározzák a metódusai, üzenetküldéssel megvalósítható az összes lehetséges kapcsolat két objektum között.
- Nem szükséges, hogy az objektumok ugyanabban a folyamatban, vagy akár ugyanazon gépen legyenek, az üzenetküldés és fogadás ettől függetlenül lehetséges.

Üzenetek a gyakorlatban

Az elméleti objektumorientált szemléletben megkülönböztetünk aszinkron és szinkron üzenetküldéses rendszert. Bár a hétköznapi modellt az aszinkron megközelítés jellemzi, gyakorlati, megvalósíthatósági okokból a programnyelvek többnyire a szinkron üzenetküldéses modellen alapulnak.

2.3. Az osztály

A valódi világban gyakran sok objektummal találkozunk ugyanabból a fajtából. Például a biciklink nagyon sok más biciklire jelentősen hasonlít. Az objektumorientált szóhasználatban azt mondjuk, hogy egy konkrét bicikli a biciklik osztályának egy példánya. A biciklik rendelkeznek állapottal (aktuális sebességfokozat, fordulatszám stb.) és viselkedéssel (sebességváltás, fékezés). Ennek ellenére minden bicikli konkrét állapota független az összes többi bicikli állapotától.

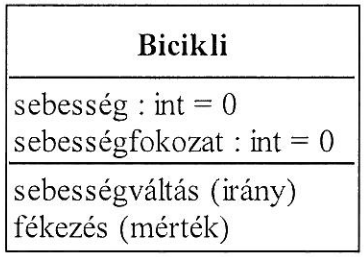
Definíció: **Osztályozásnak** nevezzük azt a folyamatot, amelynek során a hasonló objektumokat közös csoportokba, más néven osztályokba soroljuk.

Amikor a biciklik készülnek, a gyártók nyereséget szeretnének előállítani, ezért a biciklik nagy mennyiségben, közös tervrajz alapján sorozatgyártásban készülnek. Nagyon rossz lenne a hatásfok, ha minden biciklihez egyedi tervrajzot kellene készíteni.

Az objektumorientált programokban is hasonló a helyzet: közös tervezésre ad lehetőséget, hogy sok objektum hasonló jellemzőkkel rendelkezik: téglalapok, alkalmazottak, videofelvételek, stb. A kerékpárgyártókhoz hasonlóan nekünk is előnyös az, ha sok hasonló objektumot közös tervrajz alapján készíthetünk el. Az objektumok tervrajzait hívjuk osztályoknak.

Definíció: Az **osztály** bizonyos fajta objektumok közös változóit és metódusait leíró tervrajz.

A bicikli osztály legszükségesebb példányváltozói az aktuális sebesség és a sebességfokozat lehetnek. Az osztály tartalmazza a példánymetódusokat is: a sebességváltást és a fékezést, ahogy a következő UML osztálydiagramon (*Class diagram*) látszik:



A következő kód az így megtervezett osztály kódját tartalmazza:

```
class Bicycle {
        int cadence = 0;
        int speed = 0;
        int gear = 1;
        void changeCadence(int newValue) {
             cadence = newValue;
        }

        void changeGear(int newValue) {
             gear = newValue;
        }

        void speedUp(int increment) {
             speed = speed + increment;
        }

        void applyBrakes(int decrement) {
             speed = speed - decrement;
        }

        void printStates() {
             System.out.println("cadence:"+cadence+"
speed:"+speed+" gear:"+gear);
        }
}
```

Miután létrehoztuk a *Bicikli* osztályt, az osztály alapján akármennyi bicikli objektumot létre tudunk hozni. Amikor példányosítunk egy osztályból, a futtatórendszer elegendő memóriát foglal az objektum példányváltozóinak. Minden példány kap egy másolatot a definiált változókról:

az én biciklim : Bicikli
sebesség : 18km/h sebességfokozat : 5
sebességváltás (irány) fékezés (mérték)

a te biciklid : Bicikli
sebesség : 10km/h sebességfokozat : 2
sebességváltás (irány) fékezés (mérték)

Nézzük meg egy bicikliket példányosító kódot:

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

          // Create two different Bicycle objects
          Bicycle bike1 = new Bicycle();
          Bicycle bike2 = new Bicycle();

          // Invoke methods on those objects
          bike1.changeCadence(50);
          bike1.speedUp(10);
          bike1.changeGear(2);
          bike1.printStates();

          bike2.changeCadence(50);
          bike2.speedUp(10);
          bike2.changeGear(2);
          bike2.changeCadence(40);
          bike2.speedUp(10);
          bike2.changeGear(3);
          bike2.printStates();
     }
}
```

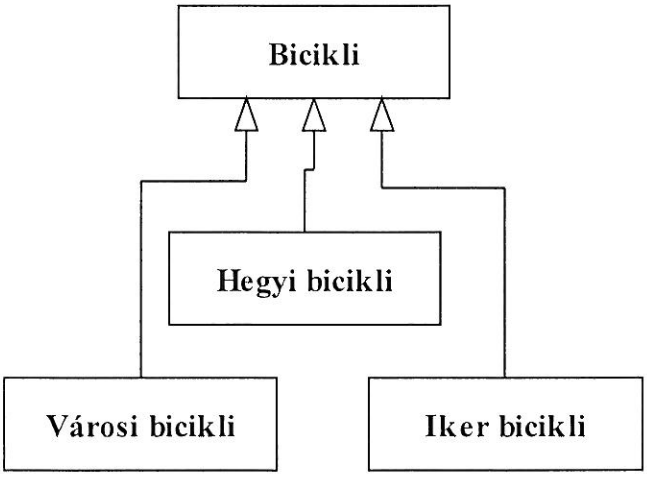
A példányváltozók mellett az osztályok definiálhatnak osztályváltozókat is. Az **osztályváltozók** az összes objektumpéldány számára megosztott információkat tartalmaznak. Például képzeljük el, hogy az összes kerékpár ugyanannyi sebességfokozattal rendelkezik. Ebben az esetben felesleges példányváltozót alkalmazni, minden példány ugyanazt a másolatot tárolná. Ilyenkor osztályváltozóban érdemes az adatot tárolni, amit minden példány el tud érni. Ha egy objektum megváltoztatja az értékét, az összes objektum számára is megváltozik.

Az osztálynak lehet **osztálymetódusa** is.

Az objektumok használatának előnye a modularitás és az információelrejtés. Az osztályok használatának előnye az **újrafelhasználhatóság**. A bicikligyárak újra és újra fel tudják használni a gyártás során az egyszer elkészített tervrajzokat. A programozók ugyanazokat az osztályokat, ugyanazokat a kódokat újra és újra felhasználják a példányosítás során.

2.4. Az öröklődés

Az objektumorientált rendszerekben egyes objektumok között további összefüggéseket figyelhetünk meg. Bizonyos feltételeknek megfelelő objektumok egy másik osztályba sorolhatók. Például a hegyi vagy éppen a városi biciklik a biciklik speciális fajtái. Az objektumorientált szóhasználatban ezeket leszármazott osztálynak (leszármazott osztálynak) nevezzük. Hasonlóan, a bicikli osztály ősosztálya (szülő osztálya, bázisosztálya) a városi biciklik osztályának. Ezt az összefüggést mutatja a következő ábra:



Az objektumorientált tervezés folyamán használt **általánosítás** és **specializálás** fogalmak az osztályhierarchia kialakítása során használatosak. Az őstől a gyermek felé speciálisabb osztályokat látunk, visszafele pedig egyre általánosabbakat.

Minden gyermekosztály örökli az ősosztály állapotát és a metódusait, de nincs ezekre korlátozva. A gyermekosztályok hozzáadhatnak változókat és metódusokat ahhoz, amit az ősosztálytól örökölt. A gyermekosztályok felül tudják írni az örökölt metódusokat, vagy speciálisabb megvalósítást tud adni azoknak.

Például a Hegyi bicikli a Bicikli leszármazottja:

```
class MountainBike extends Bicycle {

    // új adattagok és metódusok helye

}
```

Az öröklődésnél nem vagyunk behatárolva egy szintre. Az öröklési fa, vagy más néven az osztályhierarchia több szintű öröklést is lehetővé tesz, bár egy átlagos felhasználói program esetén legtöbbször 4-5 szint elegendő.

Az öröklődés a következő előnyökkel jár:

- A leszármazott osztályok tudják specializálni az ősosztálytól örökölt viselkedést. Az öröklődés segítségével az egyes osztályokat újra fel lehet használni.
- A programozók meg tudnak valósítani olyan viselkedéseket, amelyek az ősosztályban még nem voltak konkrétan leírva. (Az ilyen osztályokat absztrakt, elvont osztályoknak nevezzük.) Az absztrakt ősosztályok csak részben valósítják meg a szükséges viselkedéseket, és akár más programozók fogják azt a leszármazottakban megvalósítani.

Megjegyzés: Az objektumorientált szemlélet megismerése után sok fejlesztő számára erős a kísértés, hogy olyankor is az öröklődést alkalmazza, amikor inkább más technikákat (pl. kompozíció, aggregáció) érdemes alkalmazni.

Javában az *Object* osztály az osztályhierarchia legfelső eleme, minden más osztály belőle származik (közvetlenül vagy közvetve). Az *Object* típusú változó bármilyen objektumra tud hivatkozni.

Az *Object* osztálynak olyan megosztott viselkedései (metódusai) vannak, amelyek lehetővé teszik a Java VM-en való futást. Például minden osztály örökli a *toString* metódust, hogy az objektum sztringként is megmutatható legyen.

2.5. Publikus interfész

Általánosságban egy eszköznek vagy rendszernek szokás az **interfész**éről beszélni: azt írja le, hogy külső dolgok hogyan tudnak kapcsolódni hozzá. Ilyen értelemben két magyar ember között egy interfész a magyar nyelv.

Egy tetszőleges osztály esetén tehát a publikus interfész alatt az osztály kívülről is látható (publikus) felületét értjük. (Ez többnyire a publikus konstruktorokra és metódusokra korlátozódik.) Az üzenetküldés fogalmára visszautalva az osztály publikus interfésze azt határozza meg, hogy más objektumok milyen üzenetet küldhetnek az objektumnak, illetve milyen módon hozhatnak létre az osztálynak egy példányát.

Java interfész

Érdemes itt megemlíteni, hogy ez az elméleti fogalom nem egyezik meg a Java interfész fogalmával. A Java nyelven belül az interfész egy típus, mint ahogy az osztály is típus. Az osztályhoz hasonlóan az interfész is definiál metódusokat, de attól eltérően soha nem valósít meg metódust. Az interfészt megvalósító osztály fogja annak metódusait megvalósítani.

Az interfészek hasznosak a következő esetekben:

- Hasonlóságok megfogalmazása anélkül, hogy mesterkélt osztályhierarchiát építenénk fel
- Olyan metódusok definiálása, amelyeket több osztályban meg kell valósítani
- Többszörös öröklődés modellezése a más nyelvekből ismert veszélyek nélkül

2.6. Ellenőrző kérdések

- Mi az objektum?
- Mi az üzenet? Hogyan valósul meg a Java nyelvben?
- Mi az osztály?
- Mi az információ-elrejtés?
- Mit értünk egy osztály publikus interfészén?
- Mi az *Object* osztály szerepe?
- Mitől objektumorientált egy program?

Igaz vagy hamis? Indokolja!

- Az absztrakció az objektumok közötti hasonlóságok figyelése, összegyűjtése.
- Az osztályozás a világ objektumainak rendszerezése.
- Az általánosítás a világ objektumainak leegyszerűsítése.
- A specializálás egy szűkebb kategória meghatározása az objektumok különbözősége alapján.
- Az objektumorientált program egymással kommunikáló objektumok összessége, ahol minden objektumnak megvan a jól meghatározott feladatköre.

- Az objektum példányokat tulajdonságaik és viselkedésük alapján osztályokba soroljuk.
- Csak akkor küldhető üzenet egy objektumnak, ha a küldő és a fogadó objektum kapcsolatban állnak egymással.
- Ha két objektum állapota megegyezik, akkor a két objektum azonos.
- Az osztály meghatározza objektumainak viselkedését.
- Ha két objektum ugyanahhoz az osztályhoz tartozik, és ugyanaz az állapota, akkor ugyanarra az üzenetre ugyanúgy reagál.

2.7. Gyakorló feladat

Modellezze egy nyelviskola tanfolyam-szervezési feladatkörét!

Kik vesznek részt a folyamatban? Hogyan osztályozhatjuk őket? Milyen öröklési kapcsolatok vannak az osztályok között? Milyen műveleteket képesek végezni az objektumok?

Modellezze egy cég (pl. egy 5-10 fős Kft.) tevékenységét!

A munka során mik (és kik) tekinthetők objektumnak? Milyen jellemzőkkel és viselkedési lehetőségekkel (itt speciálisabban munkaképességekkel) rendelkeznek? Van-e osztályozási lehetőség az objektumok között?

3. Változók

A *BasicsDemo* program összeadja 1-től 10-ig az egész számokat, majd kiírja az eredményt.

```
public class BasicsDemo {
    public static void main(String[] args) {
        int sum = 0;
        for (int current = 1; current <= 10; current++) {
            sum += current;
        }
        System.out.println("Sum = " + sum);
    }
}
```

A program a következőt írja ki:

```
Sum = 55
```

Ez a kis program használja az alapvető nyelvi elemeket: változókat, operátorokat és vezérlési szerkezeteket.

Az objektum az állapotát változókban tárolja.

Definíció: A **változó** olyan adatelem, amely azonosítóval van ellátva.

Egy változó nevét és típusát egyértelműen meg kell adni a programunkban, ha használni akarjuk azt. A változó neve csak érvényes azonosító lehet: tetszőleges hosszúságú *Unicode* karakterekből álló sorozat, de az első helyen csak betű szerepelhet. A változó típusa meghatározza, hogy milyen értékeket vehet fel a változó, és milyen műveleteket hajthatunk végre rajta. A változó nevét és típusát a változódeklarációban adjuk meg, ami általában ehhez hasonló:

```
type name;
```

A változó rendelkezik hatókörrel (érvényességi tartománnyal) is. A hatáskört a változódeklaráció helye egyértelműen meghatározza.

Az alábbi példa különböző változók deklarációját mutatja:

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

        byte largestByte = Byte.MAX_VALUE;
        short largestShort = Short.MAX_VALUE;
        int largestInteger = Integer.MAX_VALUE;
        long largestLong = Long.MAX_VALUE;

        float largestFloat = Float.MAX_VALUE;
        double largestDouble = Double.MAX_VALUE;

        char aChar = 'S';
        boolean aBoolean = true;
```

```
        System.out.println("The largest byte value is "
                           + largestByte);
        System.out.println("The largest short value is "
                           + largestShort);
        System.out.println("The largest integer value is "
                           + largestInteger);
        System.out.println("The largest long value is "
                           + largestLong);

        System.out.println("The largest float value is "
                           + largestFloat);
        System.out.println("The largest double value is "
                           + largestDouble);

        if (Character.isUpperCase(aChar)) {
            System.out.println("The character " + aChar
                               + " is upper case.");
        } else {
            System.out.println("The character " + aChar
                               + " is lower case.");
        }

        System.out.println("The value of aBoolean is "
                           + aBoolean);
    }
}
```

A program kimenete:

```
The largest byte value is 127
The largest short value is 32767
The largest integer value is 2147483647
The largest long value is 9223372036854775807
The largest float value is 3.40282e+38
The largest double value is 1.79769e+308
The character S is upper case.
The value of aBoolean is true
```

3.1. Adattípusok

Minden változó rendelkezik adattípussal. A változó adattípusa határozza meg, hogy milyen értékeket vehet fel a változó, és milyen műveletek végezhetők vele. A *MaxVariablesDemo* példaprogramban az

```
int largestInteger;
```

deklarál egy *largestInteger* nevű változót *int* adattípussal. Az *int* típus csak egész számot tud tárolni.

A Java nyelvben az adattípusoknak két típusa van: primitív és referencia típusok. A primitív adattípusok egy egyszerű értéket képesek tárolni: számot, karaktert vagy logikai értéket. A változó neve (*variableName*) közvetlenül egy értéket (*value*) jelent.

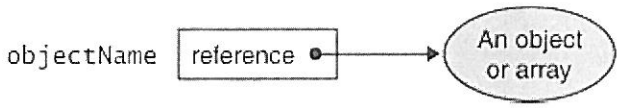
A következő táblázat az összes primitív típust tartalmazza. A példaprogramunk minden típusból deklarál egyet.

Típus	Leírás	Méret/formátum
(egészek)		
byte	bájt méretű egész	8-bit kettes komplemens
short	rövid egész	16-bit kettes komplemens
int	egész	32-bit kettes komplemens
long	hosszú egész	64-bit kettes komplemens
(valós számok)		
float	egyszeres pontosságú lebegőpontos	32-bit IEEE 754
double	dupla pontosságú lebegőpontos	64-bit IEEE 754
(egyéb típusok)		
char	karakter	16-bit Unicode karakter
boolean	logikai érték	*true* vagy *false*

Lehetőségünk van egyből kezdőértéket is adni a változónknak:

```
int anInt = 4;
```

A tömbök, az osztályok és az interfészek referencia-típusúak. A referencia-változó más nyelvek mutató vagy memóriacím fogalmára hasonlít. Az objektum neve (*objectName*) nem egy közvetlen értéket, hanem csak egy referenciát (*reference*) jelent. Az értéket közvetetten, a referencián keresztül érhetjük el:




```
int [] intArray = new int intArray[10];
```

3.2. Változó nevek

A program változónevekkel azonosítja a változóértékeket.

A Java nyelvben a következők érvényesek a nevekre:

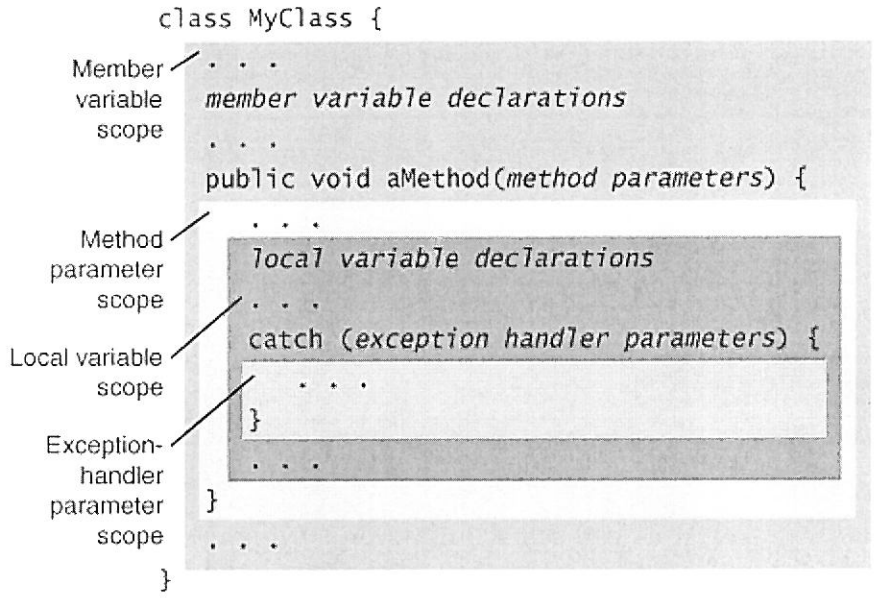
- Valódi azonosító legyen, tetszőlegesen hosszú Unicode karaktersorozat, de az első karakter csak betű lehet.
- Nem lehet foglalt szó, logikai literál (*true* vagy *false*) vagy *null*.
- Egyedinek kell lenni az érvényességi tartományban, viszont más tartománybeli változóval megegyezhet.

Konvenció (tehát nem kötelező, de szokás), hogy a változóneveket kisbetűvel kezdjük, az osztályneveket pedig naggyal. Ha a név több szóból áll össze, a közbülső szavak kezdőbetűit mindig naggyal írjuk. A többszavas nevek ragasztására más nyelvekben használt _ karakter Javában nem használatos.

3.3. Érvényességi tartomány

A változó érvényességi tartománya a programnak az a területe, ahol a változó használható egyszerű névként. Ezen kívül meghatározza, hogy a rendszer mikor foglal le és szabadít fel memóriát a változók számára.

A változódeklaráció helye határozza meg az érvényességi tartományt. A következő ábrán látható négy kategóriát különböztetünk meg:



A tagváltozó (*member variable*) az osztály vagy objektum része. Az osztályon belül, de a metódusokon kívül lehet deklarálni. A tagváltozó az osztály egészében látható.

A lokális változók (*local variable*) egy kódblokkon belül vannak. A láthatóságuk a deklaráció helyétől az őket közvetlenül körülvevő blokk végéig tart.

A metódusok formális paraméterei (*method parameters*) az egész metóduson belül láthatók.

A kivételkezelő paraméterek (*exception handler parameters*) hasonlók a formális paraméterekhez.

Figyeljük meg a következő példát:

```
if (...) {
    int i = 17;
    ...
}
System.out.println("The value of i = " + i);     //error
```

Az utolsó sor kívül van az *i* lokális változó érvényességi körén, ezért a fordítás hibával leáll.

3.4. Változók inicializálása

A lokális és tagváltozókat lehet inicializálni (kezdőértékkel ellátni) a deklarációnál. A változó adattípusa meghatározza a lehetséges kezdőérték típusát is.

```
byte largestByte = Byte.MAX_VALUE;
short largestShort = Short.MAX_VALUE;
int largestInteger = Integer.MAX_VALUE;
long largestLong = Long.MAX_VALUE;

float largestFloat = Float.MAX_VALUE;
double largestDouble = Double.MAX_VALUE;

char aChar = 'S';
boolean aBoolean = true;
```

A Java fordító nem engedi meg, hogy inicializálatlan lokális változót használjunk, vagyis az első használat előtt mindenképpen inicializálni kell azt. Tagváltozók esetén a 0 érték alapértelmezett, tehát az inicializálás elmaradása esetén az érték (típustól függő) 0 vagy *null* lesz. Ennek ellenére jó szokás tagváltozók 0 kezdőértékét is explicit megadni.

A metódusok és konstruktorok formális paramétereinek, valamint a kivételkezelő paramétereknek nem lehet kezdőértéket adni, az értékét a híváskor kapják meg.

Megjegyzés: A C++ nyelvvel szemben tehát a paraméterek esetén nem adható meg konstans kezdőérték. A következő kód csak C++-ban működik, Javában nem:

```
void move(int x = 1, int y = 1) {...}
```

Változók inicializálásához a legcélszerűbb ugyanolyan típusú literált megadni, mint a változó. Itt is igaz az az alapelv, hogy az adatvesztéssel járó implicit konverzió nem megengedett, tehát a következő kód hibás:

```
int i = 3.2;
```

Megjegyzés: Az előző példában szereplő 3.2 literál *double* típusú, tehát pl. *float* változó esetén sem lenne helyes az inicializálás. *3.2f* lenne az alkalmas *float* literált.

3.5. Végleges változók

Változót lehet véglegesen is deklarálni. A végleges változó értékét nem lehet megváltoztatni az inicializálás után. Más nyelvekben ezt konstans változóknak is hívják.

A végleges változók deklarációjánál a *final* kulcsszót kell használni:

```
final int aFinalVar = 0;
```

A végleges lokális változót nem kötelező a deklarációnál inicializálni, de addig nem használhatjuk a változót, amíg nem történik meg az inicializálás:

```
final int blankFinal;
. . . // nem használható
blankFinal = 0;
```

3.6. Ellenőrző kérdések

- Mi az összefüggés a változó neve és értéke között?
- Mit jelent, hogy egy változót deklarálni kell?
- Hol kell lennie egy változó deklarációjának?
- Hogyan kell leírni egy egész típusú változó deklarációját?
- Mi a legnagyobb és legkisebb érték, amit egy egész típusú változóban tárolhatunk?

- Mi a különbség a Java értékadó utasítása és a matematikai egyenlőség között?
- Mi a lebegőpontos típus?

Mit tapasztalunk, ha fordítani és futtatni próbáljuk a következő programot?

```
public class Test {
    public static void main (String args []) {
        int age;
        age = age + 1;
        System.out.println("The age is " + age);
    }
}
```

- Lefordul, majd lefut kimenet nélkül
- Kiírja a „*The age is 1*" szöveget
- Lefordul, majd elindul, és futási hibával leáll
- Nem fordul le

Melyik a helyes forma, ha egy *a* betűt tartalmazó karakterliterált szeretnénk létrehozni?

- *'a'*
- *"a"*
- *new Character(a)*
- *\000a*

Milyen értékeket vehet fel egy byte típusú változó?

- 0 – 65 535
- (–128) – 127
- (–32 768) – 32 767
- (–256) – 255

Melyik nem fordul le?

- *int i = 32;*
- *float f = 45.0;*
- *double d = 45.0;*

Melyik sor fordul le hiba és figyelmeztetés nélkül?

(Minden helyes választ jelöljön meg!)

- *float f=1.3;*
- *char c="a";*
- *byte b=257;*
- *boolean b=null;*
- *int i=10;*

Melyik korrekt változónév?

(Minden helyes választ jelöljön meg!)

- *2variable*
- *variable2*
- *_whatavariable*
- *_3_*
- *#myvar*

Mit tapasztalunk, ha fordítani és futtatni próbáljuk a következő programot?

```
public class Question01 {
    public static void main(String[] args){
        int y=0;
        int x=z=1;                          // A
        System.out.println(y+","+x+","+z);    // B
    }
}
```

- Kiírja: 0, 1, 1
- Kiírja: 0, 0, 1
- Fordítási hiba az A jelű sorban
- Fordítási hiba a B jelű sorban
- Fordítási hiba mindkét (A és B) sorban

4. Operátorok

Az operátorok egy, kettő vagy három operanduson hajtanak végre egy műveletet. Az egyoperandusú operátorokat unáris operátoroknak hívjuk. Például a ++ operátor az operandusát 1-el növeli. A kétoperandusú operátorokat bináris operátoroknak hívjuk. Például az = operátor a jobb oldali operandus értékét a baloldali operandusba másolja. Végül a háromoperandusú operátor három operandust vár. Javában egy háromoperandusú operátor van, a *?:* feltételes operátor.

Az unáris operátorok lehetővé teszik a prefix és postfix jelölés is:

```
operator op          //prefix
op operator          //postfix
```

Az összes bináris operátor infix jelölést alkalmaz, vagyis az operátor az operandusok között szerepel:

```
op1 operator op2     //infix
```

A háromoperandusú operátor szintén infix jelölést tesz lehetővé. Az operátor mindkét komponense az operandusok között szerepel:

```
op1 ? op2 : op3      //infix
```

A művelet végrehajtása után a kifejezés értéke rendelkezésre áll. Az érték függ az operátortól és az operandusok típusától is. Aritmetikai operátorok esetén a típus alá van rendelve az operandusoknak: ha két *int* értéket adunk össze, az érték is *int* lesz.

Megjegyzendő, hogy a Javában a kifejezések kiértékelési sorrendje rögzített, vagyis egy művelet operandusai mindig balról jobbra értékelődnek ki (ha egyáltalán kiértékelődnek, lásd rövidzár kiértékelés), még a művelet elvégzése előtt.

4.1. Aritmetikai operátorok

A Java programozási nyelvben sokféle aritmetikai operátor áll rendelkezésre lebegőpontos és egész számokhoz. Ezek az operátorok a + (összeadás), - (kivonás), * (szorzás), / (osztás) és % (maradékképzés). A következő táblázat összefoglalja a Java nyelv kétoperandusú aritmetikai operátorait.

operátor	használat	Leírás
+	op1 + op2	op1 és op2 összeadása, valamint sztring összefűzés
-	op1 - op2	op2 és op1 különbsége
*	op1 * op2	op1 és op2 szorzata
/	op1 / op2	op1 és op2 (egész) hányadosa
%	op1 % op2	op1 és op2 egész osztás maradéka

Az *ArithmeticDemo* példaprogram definiál két egész és két dupla-pontosságú lebegőpontos számot, és öt aritmetikai operátort mutat be. Ezen kívül használja a + operátort sztringek összefűzésére. Az aritmetikai operátorok félkövérek:

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

        int i = 37;
        int j = 42;
        double x = 27.475;
        double y = 7.22;

        System.out.println("Variable values...");
        System.out.println("    i = " + i);
        System.out.println("    j = " + j);
        System.out.println("    x = " + x);
        System.out.println("    y = " + y);

        System.out.println("Adding...");
        System.out.println("    i + j = " + (i + j));
        System.out.println("    x + y = " + (x + y));

        System.out.println("Subtracting...");
        System.out.println("    i - j = " + (i - j));
        System.out.println("    x - y = " + (x - y));

        System.out.println("Multiplying...");
        System.out.println("    i * j = " + (i * j));
        System.out.println("    x * y = " + (x * y));

        System.out.println("Dividing...");
        System.out.println("    i / j = " + (i / j));
        System.out.println("    x / y = " + (x / y));

        System.out.println("Computing the remainder...");
        System.out.println("    i % j = " + (i % j));
        System.out.println("    x % y = " + (x % y));

        System.out.println("Mixing types...");
        System.out.println("    j + y = " + (j + y));
        System.out.println("    i * x = " + (i * x));
    }
}
```

A program kimenete:

```
Variable values...
    i = 37
    j = 42
    x = 27.475
    y = 7.22

Adding...
    i + j = 79
    x + y = 34.695

Subtracting...
    i - j = -5
    x - y = 20.255

Multiplying...
    i * j = 1554
    x * y = 198.37

Dividing...
    i / j = 0
    x / y = 3.8054
```

```
Computing the remainder...
    i % j = 37
    x % y = 5.815

Mixing types...
    j + y = 49.22
    i * x = 1016.58
```

4.1.1 Implicit konverzió

Amikor egy aritmetikai operátor egyik operandusa egész, a másik pedig lebegőpontos, akkor az eredmény is lebegőpontos lesz. Az egész érték implicit módon lebegőpontos számmá konvertálódik, mielőtt a művelet végrehajtódna. A következő táblázat összefoglalja az aritmetikai operátorok értékét az adattípusok függvényében. A szükséges konverziók még a művelet végrehajtása előtt végre fognak hajtódni.

long	az egyik operandus sem lebegőpontos, és legalább az egyik *long*
int	az egyik operandus sem lebegőpontos, és nem *long*
double	legalább az egyik operandus *double*
float	legalább az egyik operandus *float*, és a másik nem *double*

A + és - operátorok unáris (egyoperandusú) operátorokként is használhatók:

+op	*int* értékké konvertálja a *byte*, *short* és *char* értéket
-op	aritmetikai negálás

A ++ operátor növeli az operandus értékét, a -- pedig csökkenti eggyel. Mindkettőt írhatjuk az operandus elé (prefix) és után (postfix) is. A prefix forma esetén először történik az érték növelése vagy csökkentése, majd a kifejezés értéke is a megváltozott érték lesz. A postfix használat esetén fordítva történik a végrehajtás: először értékelődik ki az operandus, majd utána hajtódik végre a ++ vagy -- művelet.

A következő *SortDemo* program mindkét operátort használja:

```
public class SortDemo {
    public static void main(String[] args) {
        int[] arrayOfInts = { 32, 87, 3, 589, 12, 1076,
                              2000, 8, 622, 127 };

        for (int i = arrayOfInts.length; --i >= 0; ) {
            for (int j = 0; j < i; j++) {
                if (arrayOfInts[j] > arrayOfInts[j+1]) {
                    int temp = arrayOfInts[j];
                    arrayOfInts[j] = arrayOfInts[j+1];
                    arrayOfInts[j+1] = temp;
                }
            }
        }
```

```
        for (int i = 0; i < arrayOfInts.length; i++) {
            System.out.print(arrayOfInts[i] + " ");
        }
        System.out.println();
    }
}
```

A program a rendezett számsorozatot fogja megjeleníteni.

4.2. Relációs operátorok

A relációs operátorok összehasonlítanak két értéket, és meghatározzák a köztük lévő kapcsolatot. Például a *!= true*-t ad, ha a két operandus nem egyenlő. A következő táblázatban összegyűjtöttük a relációs operátorokat:

Operátor	Alkalmazás	Leírás
>	op1 > op2	*true*-t ad vissza, ha op1 nagyobb, mint op2
>=	op1 >= op2	*true*-t ad vissza, ha op1 nagyobb vagy egyenlő, mint op2
<	op1 < op2	*true*-t ad vissza, ha op1 kisebb, mint op2
<=	op1 <= op2	*true*-t ad vissza, ha op1 kisebb vagy egyenlő, mint op2
==	op1 == op2	*true*-t ad vissza, ha op1 és op2 egyenlők
!=	op1 != op2	*true*-t ad vissza, ha op1 és op2 nem egyenlők

A következő példában (*RelationalDemo*) definiálunk három *int* típusú számot, és relációs operátorok használatával összehasonlítjuk őket. Az összehasonlító műveleteket félkövér betűtípussal emeltük ki:

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

        int i = 37;
        int j = 42;
        int k = 42;

        System.out.println("Variable values...");
        System.out.println("    i = " + i);
        System.out.println("    j = " + j);
        System.out.println("    k = " + k);

        System.out.println("Greater than...");
        System.out.println("    i > j = " + (i > j));//false
        System.out.println("    j > i = " + (j > i));//true
        System.out.println("    k > j = " + (k > j));//false;

        System.out.println("Greater than or equal to...");
        System.out.println("    i >= j = " + (i >= j));//false
        System.out.println("    j >= i = " + (j >= i));//true
        System.out.println("    k >= j = " + (k >= j));//true
```

```
        System.out.println("Less than...");
        System.out.println("    i < j = " + (i < j));//true
        System.out.println("    j < i = " + (j < i));//false
        System.out.println("    k < j = " + (k < j));//false

        System.out.println("Less than or equal to...");
        System.out.println("    i <= j = " + (i <= j));//true
        System.out.println("    j <= i = " + (j <= i));//false
        System.out.println("    k <= j = " + (k <= j));//true

        System.out.println("Equal to...");
        System.out.println("    i == j = " + (i == j));//false
        System.out.println("    k == j = " + (k == j));//true

        System.out.println("Not equal to...");
        System.out.println("    i != j = " + (i != j));//true
        System.out.println("    k != j = " + (k != j));//false
    }
}
```

A fenti program kimenete:

```
Variable values...
    i = 37
    j = 42
    k = 42

Greater than...
    i > j = false
    j > i = true
    k > j = false

Greater than or equal to...
    i >= j = false
    j >= i = true
    k >= j = true

Less than...
    i < j = true
    j < i = false
    k < j = false

Less than or equal to...
    i <= j = true
    j <= i = false
    k <= j = true

Equal to...
    i == j = false
    k == j = true

Not equal to...
    i != j = true
    k != j = false
```

4.3. Logikai operátorok

A relációs operátorokat gyakran használják logikai operátorokkal együtt, így összetettebb logikai kifejezéseket hozhatunk létre. A Java programozási nyelv hatféle logikai operátort – öt bináris és egy unáris – támogat, ahogy azt a következő táblázat mutatja:

Operátor	Alkalmazás	Leírás
&&	op1 && op2	Logikai és: *true*-t ad vissza, ha op1 és op2 egyaránt *true*; op2 feltételes kiértékelésű
||	op1 || op2	Logikai vagy: *true*-t ad vissza, ha op1 vagy op2 *true*; op2 feltételes kiértékelésű
!	!op	Logikai nem: *true*-t ad vissza, ha op *false*
&	op1 & op2	Bitenkénti és: *true*-t ad vissza, ha op1 és op2 egyaránt *boolean* és *true*; op1 és op2 mindig kiértékelődik; ha mindkét operandus szám, akkor bitenkénti *és* művelet
|	op1 | op2	Bitenkénti vagy: *true*-t ad vissza, ha op1 és op2 egyaránt *boolean* vagy op1 vagy op2 *true*; op1 és op2 mindig kiértékelődik; ha mindkét operandus szám, akkor bitenkénti *vagy* művelet
^	op1 ^ op2	Bitenkénti nem: *true*-t ad vissza, ha op1 és op2 különböző — vagy egyik, vagy másik, de nem egyszerre mindkét operandus *true*

A következő példa az *&&* operátort használja a két rész-kifejezés logikai értékének összekapcsolására:

```
0 <= index && index < NUM_ENTRIES
```

4.3.1 Rövidzár kiértékelés

Bizonyos esetekben a logikai operátor második operandusa nem értékelődik ki. Például a következő esetben:

```
(numChars < LIMIT) && (...)
```

Az *&&* operátor csak akkor ad vissza *true*-t, ha mindkét operandus *true*. Tehát, ha *numChars* nagyobb vagy egyenlő, mint LIMIT, akkor *&&* bal oldala *false*, és a kifejezés visszaadott eredménye a jobb oldali operandus kiértékelése nélkül születik meg. Ilyen esetekben a fordító nem értékeli ki a jobb oldali operandust. Ilyenkor a jobb oldali kifejezés miatt közvetett mellékhatások léphetnek fel, például ha adatfolyamból olvasunk, értékeket aktualizálunk, vagy számításokat végzünk a jobb oldali operandusban. Ehhez hasonlóan, ha a || operátor bal oldali operandusa igaz, felesleges a jobboldalt kiértékelni, nem is fog megtörténni.

Ha mindkét operandus logikai, az *&* operátor hasonlóan viselkedik, mint az *&&*. Azonban *&* mindig kiértékelődik és *true*-t ad vissza, ha mindkét operandusa *true*. Ha az operandusok *boolean*-típusúak, | azonos műveletet végez, mint ||.

A fenti működés miatt nem érdemes olyan kódot készíteni, amelyik a jobboldali operandusának kiértékelése során a kiértékelésen túl mást is tesz. Például veszélyes, nehezen áttekinthető lesz a következő feltételes kifejezés:

```
if (a < b && b++ < f(c) ) {...}
```

Ha a bal oldali operandus (a<b) hamis, akkor sem a ++ operátor, sem az f függvényhívás nem fog végrehajtódni.

4.4. Bitléptető és bitenkénti logikai operátorok

A léptető operátorok bit műveleteket végeznek, a kifejezés első operandusának bitjeit jobbra, vagy balra léptetik. A következő táblázat a Java nyelvben használt léptető operátorokat mutatja be:

Operátor	Alkalmazás	Leírás
<<	op1 << op2	op1 bitjeit op2 értékével balra lépteti; jobbról nullákkal tölti fel
>>	op1 >> op2	op1 bitjeit op2 értékével jobbra lépteti; balról a legnagyobb helyértékű bitet tölt fel
>>>	op1 >>> op2	op1 bitjeit op2 értékével jobbra lépteti; balról nullákkal tölt fel.

Mindegyik operátor az első operandus bitjeit lépteti az operátor által megadott irányba a második operandus értékével. Például a következő kifejezésben a 13-as egész szám bitjeit léptetjük 1-el jobbra:

```
13 >> 1;
```

A 13-as szám kettes számrendszerbeli értéke: 1101. A léptetés eredménye: 1101 egy pozícióval jobbra - 110, (decimálisan 6). A bal oldali biteket 0-val töltöttük fel.

A következő táblázatban a Java programozási nyelvben használatos bitenkénti logikai operátorokat és funkciójukat láthatjuk:

Operátor	Alkalmazás	Leírás
&	op1 & op2	Bitenkénti *és*, ha mindkét operandus szám; feltételes *és* ha mindkét operandus logikai
|	op1 | op2	Bitenkénti *vagy*, ha mindkét operandus szám; feltételes *vagy*, ha mindkét operandus logikai
^	op1 ^ op2	Bitenkénti *kizáró vagy* (*xor*)
~	~op2	Bitenkénti negáció

És

Ha az operandus szám, az *&* operátor bitenkénti *és* műveletet hajt végre páronként (helyérték szerint) az operandus bitjein. Az *és* művelet akkor ad vissza 1-et, ha a kifejezés mindkét bitje 1.

Ha az *és* műveletet két decimális számon hajtjuk végre, például a 12-n és 13-n (12&13) akkor az adott számok kettes számrendszerbeli alakján bitenként kell végrehajtanunk az *és* műveletet. Az eredmény így 12-lesz decimálisan.

Ha mindkét operandus 1, az és művelet eredményként is 1-et ad. Ellenkező esetben 0-t.

Vagy

Ha mindkét operandus szám, akkor a | operátor a vagy műveletet hajtja végre. A vagy művelet 1-et ad eredményül, ha két bit közül bármelyik értéke 1.

Kizáró vagy

Ha mindkét operandus szám, akkor a ^ operátor a kizáró vagy (*xor*) műveletet hajtja végre. Kizáró vagy esetén a kifejezés eredménye akkor egy, ha a két operandus *bit* különböző, ellenkező esetben az eredmény 0.

Negáció

Végül a negációs operátor (~) az operandus bitjeit egyenként az ellenkezőjére fordítja: ha az operandus bitje 1, akkor az eredmény 0, ha az operandus bitje 0, akkor az eredmény 1. Például: ~1011 (11) = 0100 (4).

4.4.1 Bitmanipulációk a gyakorlatban

Egyebek között a bitenkénti műveletekkel hasznosan kezelhetők a logikai bitek is. Tegyük fel például, hogy az adott programban vannak logikai bitek, melyek a különböző összetevők állapotát határozzák meg a programban. Ez célravezetőbb, mint különböző *boolean* változók definiálása. Bitmanipuláció segítségével beállíthatók, és változtathatók az adott bitek értékei.

Először definiáljunk konstansokat, melyek a program különböző bitjeit határozzák majd meg. Ezek a konstansok a kettes számrendszer különböző helyi értékei, így biztosíthatjuk, hogy később nem lesznek összekeverhetők. Később ezek a konstansok a bit változók értékeit segítik majd kinyerni. A következő példában a biteket 0 értékűnek inicializáljuk, ami azt jelenti, hogy minden érték hamis.

```
static final int VISIBLE = 1;
static final int DRAGGABLE = 2;
static final int SELECTABLE = 4;
static final int EDITABLE = 8;

int flags = 0;
```

A *VISIBLE* szimbolikus konstans jelzi, ha valami láthatóvá válik a programban. Ezt a bitet állítja egyesre a következő sor:

```
flags = flags | VISIBLE;
```

A láthatóságot a következő képen tesztelhetjük:

```
if ((flags & VISIBLE) == VISIBLE) {
    ...
}
```

Itt látható a teljes program (*BitwiseDemo*), mely magában foglalja a fenti kódot:

```
public class BitwiseDemo {

    static final int VISIBLE = 1;
    static final int DRAGGABLE = 2;
    static final int SELECTABLE = 4;
    static final int EDITABLE = 8;
```

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

        flags = flags | VISIBLE;
        flags = flags | DRAGGABLE;

        if ((flags & VISIBLE) == VISIBLE) {
            if ((flags & DRAGGABLE) == DRAGGABLE) {
                System.out.println("Flags are Visible "
                                   + "and Draggable.");
            }
        }

        flags = flags | EDITABLE;

        if ((flags & EDITABLE) == EDITABLE) {
           System.out.println("Flags are now also Editable.");
        }
    }
}
```

A fenti program kimenete:

```
Flags are Visible and Draggable.
Flags are now also Editable.
```

4.5. Értékadó operátorok

Az alap értékadó (=) operátort használhatjuk arra, hogy egy értéket hozzárendeljünk egy változóhoz. A *MaxVariablesDemo* program az "="-t használja, hogy inicializálja a változóit:

```
byte largestByte = Byte.MAX_VALUE;
short largestShort = Short.MAX_VALUE;
int largestInteger = Integer.MAX_VALUE;
long largestLong = Long.MAX_VALUE;

float largestFloat = Float.MAX_VALUE;
double largestDouble = Double.MAX_VALUE;

char aChar = 'S';
boolean aBoolean = true;
```

A Java programozási nyelv azt is megengedi a rövidített értékadó operátorok segítségével, hogy aritmetikai, értéknövelési, valamint bitenkénti műveletvégzést összekössük az értékadással.

Például, ha egy változó értékét akarjuk növelni, akkor:

```
i=i+2;
```

Ezt le lehet rövidíteni a += rövidített operátor segítségével:

```
i += 2;
```

A fenti két értékadás megegyezik.

A következő táblázat tartalmazza a rövidített értékadó operátorokat és a hosszabb alakjukat.

Operátor	Használat	Egyezik
+=	op1 += op2	op1 = op1 + op2

-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2
&=	op1 &= op2	op1 = op1 & op2
|=	op1 |= op2	op1 = op1 | op2
^=	op1 ^= op2	op1 = op1 ^ op2
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

4.6. Egyéb operátorok

A Java nyelv támogatja még a következő táblázatban foglalt operátorokat.

Operátor	Leírás
?:	Feltételes operátor
[]	Tömbök deklarálására, létrehozására és elemeinek hozzáférésére használt operátor.
.	Minősített hivatkozás
(params)	Vesszővel elválasztott paramétereket foglalja keretbe.
(type)	Átkonvertálja az értéket egy meghatározott típussá.
new	Új objektum létrehozása.
instanceof	Megállapítja, hogy az első operandus típusa-e a második operandus.

4.7. Ellenőrző kérdések

- Mit jelent a logikai kifejezés?
- Hogyan kell logikai típusú változókat deklarálni?
- Mire használható a != operátor?
- Mire használható a || operátor?
- Az *és* és *vagy* operátorok jobb oldali operandusa mikor kerül kiértékelésre?

- Mi a különbség a >> és >>> operátorok között?

Mit ír ki a következő kódrészlet?

```
System.out.println(4/3);
```

- 6
- 0
- 1
- 7

5. Kifejezések, utasítások, blokkok

A változók és az operátorok, amelyekkel az előző fejezetben ismerkedhettünk meg, a programok alapvető építőkövei. Változók, literálok és operátorok kombinációiból hozzuk létre a kifejezéseket — kódszegmenseket, amelyek számításokat végeznek, és értékeket adnak vissza. Néhány kifejezés utasításokba szervezhető — ezek komplett futtatási egységek. Ezeket az utasításokat kapcsos zárójelek közé csoportosítva — { és } — kapjuk meg az utasításblokkokat.

5.1. Kifejezések

A kifejezések hajtják végre a program feladatát. A kifejezéseket többek között változók kiszámítására, értékük beállítására és a program futásának ellenőrzésére használjuk. A kifejezéseknek kétféle feladata van: végrehajtani a számításokat, amelyeket a kifejezés alkotóelemi határoznak meg, és visszaadni a számítás végeredményét.

Definíció: A **kifejezés** változók, operátorok és metódushívások olyan sorozata (a nyelv szintaxisát figyelembe véve) amely egy értéket ad vissza.

Ahogy az előzőekben már említettük, az operátorok egy értéket adnak vissza, így az operátor használatával egy kifejezést kapunk. A *MaxVariablesDemo* program részlete néhányat bemutat a program kifejezései közül:

```
...
char aChar = 'S';
boolean aBoolean = true;

System.out.println("The largest byte value is "
    + largestByte);
...

if (Character.isUpperCase(aChar)) {
    ...
}
```

A kifejezések kiértékelése során végrehajtásra kerülnek műveletek, és a kifejezés visszaad egy értéket, mint az a következő táblázatban is látható:

Kifejezés	Művelet	Visszaadott érték
aChar = 'S'	Az 'S' karaktert adja értékül az *aChar* karakter típusú változónak	*aChar* értéke az értékadás után ('S')
"The largest byte value is " + largestByte	Összefűzi a sztringet és a *largestByte* értékét sztringgé konvertálva	Az eredmény az összefűzött sztring
Character.isUpperCase(aChar)	*isUpperCase* metódus hívása	A metódus visszatérési értéke: *true*

A kifejezés által visszaadott érték adattípusa függ a kifejezésben használt alkotóelemektől. Az *aChar = 'S'* kifejezés egy karaktert ad vissza, mert az értékadó operátor ugyan-

olyan típusú értékkel tér vissza, mint amilyenek az operandusai, és az *aChar* valamint az *'S'* karakter típusúak. A többi kifejezésnél már láthattuk, hogy egy kifejezés egy *boolean* értékkel, egy sztringgel és egyéb értékekkel is visszatérhet.

A Java nyelv lehetőséget biztosít **összetett kifejezés**ek és utasítások létrehozására kisebb kifejezésekből, amíg a kifejezés egyik részében használt adattípusok megegyeznek a többi rész adattípusaival. Itt láthatunk egy példát összetett kifejezésre:

```
x * y * z
```

Ebben a különleges példában a sorrend, amely szerint a kifejezés kiértékelődik, nem fontos, mivel a szorzás eredménye nem függ a tagok sorrendjétől, a végeredmény mindig ugyanaz, nem számít, milyen sorrendben végezzük el a szorzásokat. Azonban ez nem minden kifejezésre igaz. Például a következő kifejezés különböző eredményt ad attól függően, hogy az összeadás vagy az osztást megvalósító operátor hajtódik-e végre elsőként:

```
x + y / 100
```

Zárójelezéssel meghatározhatjuk, hogy egy kifejezés hogyan értékelődjön ki. Például így tehetjük egyértelművé a végrehajtást az előző példa esetében:

```
(x + y)/ 100
```

Ha határozattan nem jelezzük a sorrendet, amely szerint akarjuk, hogy az összetett kifejezés kiértékelődjön, akkor a sorrendet az **operátorok precedenciája** fogja meghatározni. A magasabb precedenciával rendelkező operátorok hajtódnak végre elsőként. Például az osztás operátor magasabb precedenciájú, mint az összeadás. Így a következő két kifejezés megegyezik egymással:

```
x + y / 100
x + (y / 100)
```

Ha összetett kifejezést írunk, akkor kifejezetten figyelnünk kell a zárójelezésre és arra, hogy mely operátoroknak kell kiértékelődniük elsőként. Ennek a gyakorlása segíthet a forráskód jobb olvashatóságának és karbantarthatóságának elérésében.

A következő táblázat a Java platformban használt operátorok precedencia-szintjeit mutatja be. Az operátorok a táblázatban precedencia-szint szerint vannak rendezve: legfelül a legnagyobb precedenciával rendelkező található. A magasabb precedenciával rendelkező operátorok előbb hajtódnak végre, mint az alacsonyabbal rendelkezők. Az azonos szinten elhelyezkedő operátorok azonos precedenciával rendelkeznek. Ha azonos precedenciájú operátorok szerepelnek egy kifejezésben, akkor szabályozni kell, hogy melyik értékelődjön ki elsőként. Minden bináris operátor, kivéve az értékadó operátorokat balról jobbra hajtódnak végre. Az értékadó operátorok jobbról balra hajtódnak végre.

Operátor precedencia szintek

postfix	*expr++ expr--*
unáris	*++expr --expr +expr -expr* ~ !
multiplikatív	* / %
additív	+ -
léptetés	<< >> >>>
relációs	< > <= >= *instanceof*
egyenlőség	== !=
bitenkénti és	&
bitenkénti kizáró vagy	^
bitenkénti vagy	|
logikai és	&&
logikai vagy	||
feltételes	? :
értékadás	= += -= *= /= %= &= ^= |= <<= >>= >>>=

5.2. Utasítások

Az utasítások nagyjából a beszélt nyelv mondatainak felelnek meg. Az utasítás egy konkrét futási egységet hoz létre. A következő kifejezéstípusok szervezhetők utasításokba, amelyek pontosvesszővel végződnek (;):

- értékadó kifejezések
- ++ és -- használata
- metódushívások
- objektumot létrehozó kifejezések

Az utasításoknak ezt a fajtáját **kifejezés utasítások**nak nevezzük. Itt láthatunk pár példát a kifejezés-utasításokra:

```
aValue = 8933.234;                          //értékadó utasítás
aValue++;                                   //növelés
System.out.println(aValue);                 //metódus hívás
Integer integerObject = new Integer(4); //objektum létrehozás
```

A kifejezés-utasításokon kívül még kétféle típust kell megemlítenünk. A **deklarációs utasítás** létrehoz egy változót. Sok példát láthattunk már deklarációs utasításokra.

```
double aValue = 8933.234;                   //deklarációs utasítás
```

A **végrehajtás-vezérlő utasítás** szabályozza, hogy az utasítások milyen sorrendben hajtódnak végre. A *for* ciklus és az *if* utasítás jó példák a végrehajtás-vezérlő szerkezetre.

5.3. Blokkok

A blokk nulla vagy több utasítás kapcsos zárójelek között, amely használható bárhol, ahol az önálló utasítások megengedettek. A következő részlet két blokkot mutat be *MaxVariablesDemo* programból, mindegyik tartalmaz egy önálló utasítást:

```
if (Character.isUpperCase(aChar)) {
    System.out.println("The character " + aChar
                       + " is upper case.");
} else {
    System.out.println("The character " + aChar
                       + " is lower case.");
}
```

5.4. Összefoglalás

Egy kifejezés változók, operátorok és metódushívások sorozata (a nyelv szintaxisát figyelembe véve), amely egy értéket ad vissza. Írhatunk összetett kifejezéseket is egyszerűbbekből összeállítva mindaddig, amíg a magában foglalt, az operátorokhoz szükséges adattípusok megfelelőek. Ha összetett kifejezést írunk, akkor kifejezetten figyelnünk kell a zárójelezésre és arra, hogy mely operátoroknak kell kiértékelődniük elsőként.

Ha nem használjuk a zárójelezést, akkor a Java platform az operátorok precedenciája szerint értékeli ki az összetett kifejezést. A korábbi táblázat mutatja be a Java platformban megtalálható operátorok precedenciáját.

Az utasítás egy konkrét utasítási egységet valósít meg, amelyet pontosvessző zár le (;). Az utasításoknak három fajtája van: kifejezés utasítások, deklarációs utasítások, végrehajtás-vezérlő utasítások.

Nulla vagy több utasításból a kapcsos zárójelek segítségével blokkokat alakíthatunk ki: { és }. Habár nem szükséges, ajánlott az alkalmazása akkor is, ha a blokk csak egy utasítást tartalmaz.

5.5. Ellenőrző kérdések

- Mit jelent a programblokk?
- Hogyan kell egy programblokkot leírni Jávában?
- Mondjon példát kifejezés-utasításra!
- Kell e pontosvesszőt (;) írni egy kifejezés végére?
- Azonos precedenciájú operátorok esetén mikor fog jobbról balra haladni a kiértékelés?

A következő sorok közül melyik fog hiba nélkül lefordulni?

- *float f=1.3;*
- *char c="a";*

- *byte b=257;*
- *boolean b=null;*
- *int i=10;*

5.6. Gyakorló feladatok

Írjon programot, amely kiszámítja és kiírja az 5 egység sugarú gömb térfogatát!

Hozzon létre egy *r* változót 5-ös kezdőértékkel. Számítsa ki a térfogat értékét a következő képlet alapján.

$$V = \frac{4r^3\pi}{3}$$

Írjon programot, amely kiszámítja és kiírja a másodfokú egyenlet két megoldását!

(Feltételezzük, hogy két megoldás létezik, a gyök alatti kifejezés esetleges negatív voltával most nem kell foglalkozni.)

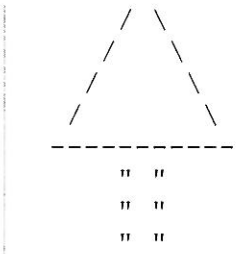
$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Hozzon létre *a*, *b* és *c* valós típusú változókat tetszőleges kezdőértékkel! Hozzon létre *x1* és *x2* nevű változókat, és végezze el a számításokat! (A négyzetgyökvonásra a *Math.sqrt()* metódust lehet alkalmazni. A metódus negatív paraméter esetén NaN speciális értéket adja.) Írja ki az eredményeket a konzolra!

Módosítsa a programot úgy, hogy a diszkrimináns (a gyökjel alatti kifejezés) értékét egy *d* nevű változóba számítsa ki először, majd ennek felhasználásával számítsa az *x1* és *x2* értékét.

(Példa adatok a teszteléshez: a = 1, b = -5, c = 6 esetén x1=2, x2=3)

Írjon programot, amelyik „kirajzolja" a karácsonyfát!



6. Vezérlési szerkezetek

6.1. A *while* és a *do-while* ciklusok

A *while* ciklus utasításblokk végrehajtására használható, amíg a feltétel igaz. A *while* ciklus szintaxisa:

```
while (feltétel) {
    utasítások
}
```

A *while* ciklus először kiértékeli a feltételt, amely művelet egy *boolean* értéket ad vissza. Ha a kifejezés értéke igaz, a *while* ciklus végrehajtja *while* blokkjában szereplő utasításokat. A *while* ciklus addig értékeli ki a kifejezést és hajtja végre az utasításblokkot, amíg a kifejezés hamis értékű nem lesz.

A következő *WhileDemo* nevű példaprogram a *while* ciklust használja fel, amely megvizsgálja a sztring karaktereit, hozzáfűzi a sztring minden karakterét a sztring puffer végéhez, amíg 'g' betűvel nem találkozik.

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

        String copyFromMe = "Copy this string until you " +
                            "encounter the letter 'g'.";
        StringBuffer copyToMe = new StringBuffer();

        int i = 0;
        char c = copyFromMe.charAt(i);

        while (c != 'g') {
            copyToMe.append(c);
            c = copyFromMe.charAt(++i);
        }
        System.out.println(copyToMe);
    }
}
```

Az érték, amelyet az utolsó sor ír ki:

```
Copy this strin
```

A Java nyelv egy a *while* ciklushoz hasonló utasítást is biztosít — *a do-while* ciklust. A *do-while* szintaxisa:

```
do {
    utasítás(ok)
} while (feltétel);
```

Ahelyett, hogy a feltételt a ciklus végrehajtása előtt értékelné ki, a *do-while* ezt a ciklusmag lefutása után teszi meg. Így a *do-while* magjában szereplő utasítások minimum egyszer végrehajtódnak.

Itt látható az előző program *do-while* ciklussal megvalósítva, ami a *DoWhileDemo* nevet kapta:

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

        String copyFromMe = "Copy this string until you " +
                            "encounter the letter 'g'.";
        StringBuffer copyToMe = new StringBuffer();

        int i = 0;
        char c = copyFromMe.charAt(i);

        do {
            copyToMe.append(c);
            c = copyFromMe.charAt(++i);
        } while (c != 'g');
        System.out.println(scopyToMe);
    }
}
```

Az érték, amelyet az utolsó sorban kiír:

```
Copy this strin
```

6.2. A *for* ciklus

A *for* utasítás jó módszer egy értéktartomány bejárására. A *for* utasításnak van egy hagyományos formája, és a Java 5.0-tól kezdődően egy továbbfejlesztett formája is, amit tömbökön és gyűjteményeken való egyszerű bejárásnál használhatunk. A *for* utasítás általános formája a következőképpen néz ki:

```
for (inicializálás; feltétel; növekmény) {
    utastás(ok)
}
```

Az inicializálás egy olyan kifejezés, amely kezdőértéket ad a ciklusnak – ez egyszer, a ciklus elején fut le. A feltétel kifejezés azt határozza meg, hogy meddig kell a ciklust ismételni. Amikor a kifejezés hamisként értékelődik ki, a ciklus nem folytatódik. Végezetül a növekmény egy olyan kifejezés, amely minden ismétlődés után végrehajtódik a ciklusban. Mindezen összetevők opcionálisak. Tulajdonképpen ahhoz, hogy egy végtelen ciklust írjunk, elhagyjuk mindhárom kifejezést:

```
for ( ; ; ) {
    ...
}
```

A *for* ciklusokat gyakran arra használjuk, hogy egy tömb elemein vagy egy karakterláncon végezzünk iterációt. Az alábbi példa, *ForDemo*, egy *for* utasítást használ arra, hogy végighaladjon egy tömb elemein és kiírja őket.

```
public class ForDemo {
    public static void main(String[] args) {
        int[] arrayOfInts = { 32, 87, 3, 589, 12,
                              1076, 2000, 8, 622, 127 };

        for (int i = 0; i < arrayOfInts.length; i++) {
            System.out.print(arrayOfInts[i] + " ");
        }
        System.out.println();
    }
}
```

A program futási eredménye:

```
32 87 3 589 12 1076 2000 8 622 127
```

Megjegyezzük, hogy egy lokális változó is deklarálható a *for* ciklus inicializáló kifejezésében. Ennek a változónak az érvényessége a deklarációjától a *for* utasítás által vezérelt blokk végéig terjed, tehát mind a lezáró és a növekmény kifejezéseiben is használhatók. Ha a *for* ciklust vezérlő változóra nincs szükség a cikluson kívül, a legjobb, ha a változót az értékadó kifejezésben deklaráljuk. Az *i*, *j* és *k* neveket gyakran a *for* ciklusok vezérlésére használjuk, ezeknek a *for* ciklus értékadó kifejezésén belül való deklarálása leszűkíti élettartamukat, és csökkenti a hibalehetőségeket.

Gyűjteményeken és tömbökön való bejárás a kibővített *for* ciklussal

Az 5.0-ban egy újfajta *for* utasítást hoztak létre kifejezetten gyűjteményekhez és tömbökhöz. Az utasítás általános alakja:

```
for (elemtípus elem : tároló) {
    utasítás(ok)
}
```

Itt egy kis kódrészlet, ami ugyanazt a feladatot végzi, mint az előző kódrészlet.

```
public class ForEachDemo {
    public static void main(String[] args) {
        int[] arrayOfInts = { 32, 87, 3, 589, 12,
                              1076, 2000, 8, 622, 127 };

        for (int element : arrayOfInts) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```

A kibővített *for* utasítás igazán akkor előnyös, amikor gyűjteményekre alkalmazzuk (osztályok, amik a *Collection* interfészt implementálják). Itt látható egy régi típusú *for* utasítás, ami iterátor segítségével egy gyűjteményen halad végig:

```
void cancelAll(Collection<TimerTask> c) {
    for (Iterator<TimerTask> i = c.iterator(); i.hasNext(); )
        i.next().cancel();
}
```

Most nem kell aggódnunk a különös *<TimerTask>* kódrészlet miatt. Később fogunk magyarázatot adni a részletekre. A lényeg az, hogy elkerülhetjük ezt a kibővített *for* ciklus használatával:

```
void cancelAll(Collection<TimerTask> c) {
    for (TimerTask t : c)
        t.cancel();
}
```

Amikor iterációkat ágyazunk egybe, a kibővített *for* utasítás még jobb, mivel fölösleges kódrészeket kerülhetünk el. Például:

```
for (Suit suit : suits) {
    for (Rank rank : ranks)
        sortedDeck.add(new Card(suit, rank));
}
```

A kibővített *for* utasítás sajnos nem működik mindenhol. Hogyha pl. tömbindexekhez akarunk hozzáférni, a kibővített *for* nem fog működni. Amikor csak lehet, a továbbfejlesztett *for*-t használjuk, több programhibát is kiküszöbölhetünk, és a forráskódunk rendezettebb lesz.

6.3. Az *if-else* szerkezet

Az *if* utasítás lehetővé teszi a programunk számára, hogy valamilyen kritérium szerint kiválasztva futtasson más utasításokat. Például tegyük fel azt, hogy a programunk hibakereső (*debugging*) információkat ír ki egy *DEBUG* nevű, *boolean* típusú változó értéke alapján. Ha a *DEBUG* igaz, a program kiírja az információt, az *x* változó értékét. Különben a program futása normálisan folytatódik. Egy ilyen feladatot implementáló programrész a következőképpen nézhet ki:

```
if (DEBUG) {
    System.out.println("DEBUG: x = " + x);
}
```

Ez az *if* utasítás legegyszerűbb formája. Az *if* által vezérelt blokk végrehajtódik, ha a feltétel igaz. Általában az *if* egyszerű alakja így néz ki:

```
if (feltétel) {
    kifejezések
}
```

Mi van akkor, ha az utasítások más változatát akarjuk futtatni, ha a feltétel kifejezés hamis? Erre az *else* utasítást használhatjuk. Vegyünk egy másik példát. Tegyük fel azt, hogy a programunknak különböző műveleteket kell végrehajtania attól függően, hogy a felhasználó az *OK* gombot vagy más gombot nyom meg a figyelmeztető ablakban. A programunk képes lehet erre, ha egy *if* utasítást egy *else* utasítással együtt használunk.

```
if (response == OK) {
    //code to perform OK action
} else {
    //code to perform Cancel action
}
```

Az *else* blokk akkor kerül végrehajtásra, ha az *if* feltétele hamis. Az *else* utasítás egy másik formája az *else if* egy másik feltételen alapulva futtat egy utasítást. Egy *if* utasításnak lehet akárhány *else if* ága, de *else* csak egy. Az alábbi *IfElseDemo* program egy teszt pontszámot alapul véve egy osztályzatot határoz meg: 5-ös 90%-ért vagy afölött, 4-es 80%-ért vagy afölött és így tovább:

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

```
        int testscore = 76;
        int grade;

        if (testscore >= 90) {
            grade = 5;
        } else if (testscore >= 80) {
            grade = 4;
        } else if (testscore >= 70) {
            grade = 3;
        } else if (testscore >= 60) {
            grade = 2;
        } else {
            grade = 1;
        }

        System.out.println("Grade = " + grade);
    }
}
```

Ennek a programnak a kimenete:

```
Grade = 3
```

Megfigyelhetjük, hogy a *testscore* értéke több kifejezés feltételének is eleget tehet az alábbi *if* utasítások közül: 76 >= 70 és 76 >= 60. Azonban, ahogy a végrehajtó rendszer feldolgoz egy olyan összetett *if* utasítást, mint ez, amint egy feltétel kielégül, lefutnak a megfelelő utasítások (*grade = 3*), és a vezérlés kikerül az *if* utasításból anélkül, hogy a további feltételeket kiértékelné.

A Java programozási nyelv támogat egy háromoperandusú operátort, ami egyszerű esetekben az *if* utasítás helyett alkalmazható. Az operátor általános alakja:

```
logikai kifejezés ? kifejezés-ha-igaz : utasítás-ha-hamis
```

Idézzük fel ezt az utasítást a *MaxVariablesDemo* programból:

```
if (Character.isUpperCase(aChar)) {
    System.out.println("The character " + aChar
                       + " is upper case.");
} else {
    System.out.println("The character " + aChar
                       + " is lower case.");
}
```

Itt látható, hogyan használhatjuk ezt az operátort:

```
System.out.println("The character " + aChar + " is " +
    (Character.isUpperCase(aChar) ? "upper" : "lower")
     + "case.");
```

A *?:* operátor kiértékelésének eredménye az *upper* karaktersorozat lesz, ha az *isUpperCase* metódus igaz értéket ad vissza, egyébként pedig a *lower* karaktersorozat. Az eredmény össze lesz fűzve a megjeleníteni kívánt üzenet más részeivel. Ha megszokjuk ezt a szerkezetet, bizonyos esetekben könnyebben olvashatóbbá és tömörebbé teheti a kódunkat.

Megjegyzés: Érdemes megfigyelni, hogy miért is lehetett itt az *if-else* szerkezetet kiváltani a háromoperandusú operátorral: az *if* és *else* ágon is ugyanazt akartuk tenni egy bizonyos kifejezéssel: ki akartuk írni. Ha ez a közös felhasználás nem áll fenn, akkor maradnunk kell az *if-else* utasításnál.

Megjegyzés: Általában nem feltétlenül szükséges, mégis sok alkalommal zárójelezzük az egyes operandusokat, és időmként az egész operátor-kifejezést is. (Az előző példa az egész kifejezést zárójeleni.)

6.4. A *switch-case* szerkezet

Akkor használhatjuk a *switch* utasítást, ha egy egész szám értéke alapján akarunk végrehajtani utasításokat. A következő *SwitchDemo* példaprogram egy *month* nevű egész típusú változót deklarál, melynek értéke vélhetőleg a hónapot reprezentálja egy dátumban. A program a *switch* utasítás használatával a hónap nevét jeleníti meg a *month* értéke alapján.

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

        int month = 8;

        switch (month) {
            case 1:  System.out.println("January"); break;
            case 2:  System.out.println("February"); break;
            case 3:  System.out.println("March"); break;
            case 4:  System.out.println("April"); break;
            case 5:  System.out.println("May"); break;
            case 6:  System.out.println("June"); break;
            case 7:  System.out.println("July"); break;
            case 8:  System.out.println("August"); break;
            case 9:  System.out.println("September"); break;
            case 10: System.out.println("October"); break;
            case 11: System.out.println("November"); break;
            case 12: System.out.println("December"); break;
            default: System.out.println("Not a month!");
                     break;
        }
    }
}
```

A *switch* utasítás kiértékeli kifejezést, ez esetben a *month* értékét, és lefuttatja a megfelelő *case* utasítást. Ezáltal a program futási eredménye az *August* lesz. Természetesen ezt *az if* utasítás felhasználásával is megoldhatjuk:

```
int month = 8;

if (month == 1) {
    System.out.println("January");
} else if (month == 2) {
    System.out.println("February");
}

...
```

Annak eldöntése, hogy az *if* vagy a *switch* utasítást használjuk, programozói stílus kérdése. Megbízhatósági és más tényezők figyelembevételével eldönthetjük, melyiket használjuk. Míg egy *if* utasítást használhatunk arra, hogy egy értékkészlet vagy egy feltétel alapján hozzunk döntéseket, addig a *switch* utasítás egy egész szám értéke alapján hoz döntést. Másrészt minden *case* értéknek egyedinek kell lennie, és a vizsgált értékek csak konstansok lehetnek.

Egy másik érdekesség a *switch* utasításban a minden *case* utáni *break* utasítás. Minden egyes *break* utasítás megszakítja az épp bezáródó *switch* utasítást, és a vezérlés szála a *switch* blokk utáni első utasításhoz kerül. A *break* utasítások szükségesek, mivel nélkülük a *case* utasítások értelmüket vesztenék. Vagyis egy explicit *break* nélkül a vezérlés folytatólagosan a rákövetkező *case* utasításra kerül (átcsorog). Az alábbi *SwitchDemo2* példa azt illusztrálja, hogyan lehet hasznos, ha a *case* utasítások egymás után lefutnak.

```
public class SwitchDemo2 {
    public static void main(String[] args) {

        int month = 2;
        int year = 2000;
        int numDays = 0;

        switch (month) {
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:
                numDays = 31;
                break;
            case 4:
            case 6:
            case 9:
            case 11:
                numDays = 30;
                break;
            case 2:
                if ( ((year % 4 == 0) && !(year % 100 == 0))
                     || (year % 400 == 0) )
                    numDays = 29;
                else
                    numDays = 28;
                break;
            default:
                numDays = 0;
                   break;
        }

        System.out.println("Number of Days = " + numDays);
    }
}
```

A program kimenete:

```
Number of Days = 29
```

Technikailag az utolsó *break* nem szükséges, mivel a vezérlés amúgy is befejeződne, kilépve a *switch* utasításból. Azonban javasolt ekkor is a *break* használata, mivel így a kód könnyebben módosítható és kevésbé hajlamos a hibára. Később még látni fogjuk a ciklusok megszakítására használt *break*-et.

Végül a *switch* utasításban használhatjuk a *default* utasítást, hogy mindazokat az értékeket is kezelhessük, amelyek nem voltak egy *case* utasításban sem kezelve.

Megjegyzés: A *default* utasításnak nem kell feltétlenül az utolsónak lenni, bár így a leglogikusabb és így is szokás általában elhelyezni. Ugyanígy a *case* ágak sorrendjének is csak akkor lehet jelentősége, ha van olyan ág, amelyiket nem zártunk le *break*kel.

6.4.1 A *switch* utasítás és a felsorolt típus

A felsorolt adattípus az 5.0-ban bevezetett újdonság, amiről később olvashat majd. Ez a rész csak azt mutatja be, hogyan használhatjuk őket egy *switch* utasításban. Szerencsére ez pont olyan, mint a *switch* használata az egész típusú változók esetén.

Az alábbi *SwitchEnumDemo* kódja majdnem megegyezik azzal a kóddal, amit korábban a *SwitchDemo2*-ben láttunk. Ez az egész típusokat felsorolt típusokkal helyettesíti, de egyébként a *switch* utasítás ugyanaz.

```
public class SwitchEnumDemo {
    public enum Month { JANUARY, FEBRUARY, MARCH, APRIL,
                        MAY, JUNE, JULY, AUGUST, SEPTEMBER,
                        OCTOBER, NOVEMBER, DECEMBER }

    public static void main(String[] args) {
        Month month = Month.FEBRUARY;
        int year = 2000;
        int numDays = 0;

        switch (month) {
            case JANUARY:
            case MARCH:
            case MAY:
            case JULY:
            case AUGUST:
            case OCTOBER:
            case DECEMBER:
                numDays = 31;
                break;
            case APRIL:
            case JUNE:
            case SEPTEMBER:
            case NOVEMBER:
                numDays = 30;
                break;
            case FEBRUARY:
                if ( ((year % 4 == 0) && !(year % 100 == 0))
                     || (year % 400 == 0) )
                    numDays = 29;
                else
                    numDays = 28;
                break;
            default:
                numDays=0;
                break;
        }

        System.out.println("Number of Days = " + numDays);
    }
}
```

Ez a példa csak egy kis részét mutatta be annak, amire a Java nyelvi felsorolások képesek. A továbbiakat később olvashatja el.

6.5. Vezérlésátadó utasítások

Kivételkezelő utasítások

A Java programozási nyelv egy kivételkezelésnek nevezett szolgáltatást nyújt, hogy segítse a programoknak a hibák felderítését és kezelését. Amikor egy hiba történik, a program „dob egy kivételt". Ez azt jelenti, hogy a program normális végrehajtása megszakad, és megkísérel találni egy kivételkezelőt, vagyis egy olyan kódblokkot, ami a különféle tí-

pusú hibákat le tudja kezelni. A kivételkezelő blokk megkísérelheti a hiba kijavítását, vagy ha úgy tűnik, hogy a hiba visszaállíthatatlan, akkor szabályosan kilép a programból.

Alapvetően három utasítás játszik szerepet a kivételkezelésekben:

- a *try* utasítás tartalmaz egy utasítás blokkot, amiben a kivétel dobása elképzelhető
- a *catch* utasítás tartalmaz egy olyan utasításblokkot, ami le tudja kezelni az azonos típusú kivételeket. Az utasítások akkor hajtódnak végre, ha *kivételtípus* típusú kivétel váltódik ki a *try* blokkban
- a *finally* egy olyan utasítás blokkot tartalmaz, ami végrehajtódik akkor is, ha a *try* blokkban hiba történt, és akkor is, ha hiba nélkül futott le a kód.

Az utasítások általános alakja:

```
try {
    utasítás(ok)
} catch (kivételtípus kivételobjektum) {
    utasítás(ok)
} finally {
    utasítás(ok)
}
```

A kivételkezelés módszerének részletes ismertetésére később kerül sor.

Feltétel nélküli vezérlésátadás

A Java programnyelv háromféle feltétel nélküli vezérlésátadást támogat:

- a *break* utasítást
- a *continue* utasítást
- a *return* (visszatérés) utasítást

A *break* és a *continue* utasításokat használhatjuk címkével vagy anélkül. A címke egy azonosító, ami az utasítás előtt helyezkedik el. A címkét egy kettőspont (:) követi. A következő programrészletben láthatunk egy példát a címke alkalmazására:

```
statementName: someJavaStatement;
```

A *break* utasítás

A *break* utasításnak két alakja van: címke nélküli és címkés. A címke nélküli *break* utasítást korábban a *switch*-nél már használtuk. Ahol a címke nélküli *break* utasítással fejeztük be a sort, ott befejezi a *switch* utasítást, és átadja a vezérlést a *switch* után következő utasításnak. A címke nélküli *break* utasítás használható még a *for*, *while* vagy *do-while* ciklusokból való kilépésre is. A *BreakDemo* példaprogram tartalmaz egy *for* ciklust, ami egy bizonyos értéket keres egy tömbön belül:

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

        int[] arrayOfInts = { 32, 87, 3, 589, 12, 1076,
                              2000, 8, 622, 127 };
        int searchfor = 12;

        int i = 0;
        boolean foundIt = false;
```

```
        for ( ; i < arrayOfInts.length; i++) {
            if (arrayOfInts[i] == searchfor) {
                foundIt = true;
                break;
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor
                               + " at index " + i + '.');
        } else {
            System.out.println(searchfor
                               + "not in the array");
        }
    }
}
```

A *break* utasítás befejezi a *for* ciklust, ha megvan az érték. A vezérlés átadódik a for lezárása után lévő utasításnak, ami a print utasítást tartalmazó *if* a program végén.

A program kimenete:

```
Found 12 at index 4.
```

Megjegyzés: A *for* ciklusból *break*-kel való kilépést sokszor használjuk egy adott érték keresése érdekében. Érdemes megfigyelni, hogy ekkor a ciklus után meg kell állapítani (*if* utasítás), hogy miért is fejeződött be a ciklus: a találat miatt, vagy mert a végére értünk.

A *break* utasítás címke nélküli alakját használhatjuk a legbelső *switch*, *for*, *while* vagy *do-while* befejezésére. A címkézett alak befejez egy olyan külső utasítást, ami a *break* címkéje által van azonosítva. Másként fogalmazva: egyszerre több utasításból is képes kiugrani. A következő (*BreakWithLabelDemo*) program hasonló az előzőhöz, de itt kétdimenziós tömbben keressük az értéket. Kettő egymásba ágyazott *for* ciklus vizsgálja át a tömböt. Amikor az érték megvan, egy címkézett *break* befejezi a *search*-ként címkézett utasítást, ami a külső *for* ciklus:

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

        int[][] arrayOfInts = { { 32, 87, 3, 589 },
                                { 12, 1076, 2000, 8 },
                                { 622, 127, 77, 955 }
                              };
        int searchfor = 12;

        int i = 0;
        int j = 0;
        boolean foundIt = false;

    search:
        for ( ; i < arrayOfInts.length; i++) {
            for (j = 0; j < arrayOfInts[i].length; j++) {
                if (arrayOfInts[i][j] == searchfor) {
                    foundIt = true;
                    break search;
                }
            }
        }
```

```
        if (foundIt) {
            System.out.println("Found " + searchfor +
                               " at " + i + ", " + j + '.');
        } else {
            System.out.println(searchfor
                               + "not in the array.");
        }
    }
}
```

A program kimenete:

```
Found 12 at 1, 0.
```

Ez a szintaxis egy kicsit zavaró lehet. A *break* utasítás befejezi a címkézett utasítást, és nem a címkének adja át a vezérlést. A vezérlés annak az utasításnak adódik át, ami közvetlen a (befejezett) címkézett utasítás után van.

A *continue* utasítás

A *continue* utasítás arra használható, hogy átugorjuk a ciklusmag hátralevő részét egy *for*, *while* vagy *do-while* ciklusnak. A címke nélküli alakja átugrik a legbelső ciklusmag végére és kiértékeli a logikai kifejezés értékét, ami a ciklust vezérli. A következő *ContinueDemo* program végigfut egy *StringBuffer*-en, megvizsgálva az összes betűt. Ha a vizsgált karakter nem 'p', a *continue* utasítás átugorja a ciklus hátralevő részét és vizsgálja a következő karaktert. Ha ez egy 'p', a program megnöveli a számláló értékét és átalakítja a 'p'-t nagybetűssé.

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

        StringBuffer searchMe = new StringBuffer(
            "peter piper picked a peck of pickled peppers");
        int max = searchMe.length();
        int numPs = 0;

        for (int i = 0; i < max; i++) {
            //interested only in p's
            if (searchMe.charAt(i) != 'p')
                continue;

            //process p's
            numPs++;
            searchMe.setCharAt(i, 'P');
        }

        System.out.println("Found " + numPs
                           + " p's in the string.");
        System.out.println(searchMe);
    }
}
```

Ennek a programnak a kimenete:

```
Found 9 p's in the string.
Peter PiPer Picked a Peck of Pickled PePPers
```

A *continue* utasítás címkés alakja átugorja a címkézett ciklus ciklusmagjának hátralevő részét. A következő *ContinueWithLabelDemo* példaprogram egymásba ágyazott ciklusokat használ egy szövegrész keresésére egy másik szövegben. Két egymásba ágyazott ciklus szükséges: egy, hogy ismételje a szövegrészt, és egy, hogy addig ismételje, amíg át

nem vizsgálta a szöveget. Ez a program a *continue* címkézett alakját használja, hogy átugorjon egy ismétlést a külső ciklusban:

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

        String searchMe = "Look for a substring in me";
        String substring = "sub";
        boolean foundIt = false;

        int max = searchMe.length() - substring.length();

    test:
        for (int i = 0; i <= max; i++) {
            int n = substring.length();
            int j = i;
            int k = 0;

            while (n-- != 0) {
                if (searchMe.charAt(j++)
                        != substring.charAt(k++)) {
                    continue test;
                }
            }

            foundIt = true;
                    break test;
        }

        System.out.println(foundIt ? "Found it" :
                                     "Didn't find it");
    }
}
```

Ennek a programnak a kimenete:

```
Found it
```

Megjegyezés: Ahogy a korábbi példából is láthatjuk, a címke nélküli *continue* mindig kiváltható a ciklus átszervezésével, ilyen esetben ritkán is alkalmazzuk.

A *return* (visszatérés) utasítás

Ez az utasítás az utolsó a feltétlen vezérlésátadó utasítások közül. A *return*-t az aktuális metódusból vagy konstruktorból való kilépésre használjuk. A vezérlés visszaadódik annak az utasításnak, ami az eredeti hívást követi. A *return* utasításnak két formája van: ami visszaad értéket, és ami nem. Hogy visszatérjen egy érték, egyszerűen tegyük az értéket (vagy egy kifejezést, ami kiszámítja azt) a *return* kulcsszó után:

```
return ++count;
```

A visszaadott érték adattípusa meg kell, hogy egyezzen a függvényben deklarált visszatérési érték típusával. Ha a függvényt *void*-nak deklaráltuk, használjuk a *return* azon alakját, ami nem ad vissza értéket:

```
return;
```

6.6. Ellenőrző kérdések

- Mi az *if* utasítás?
- Mit jelent, ha az *if* utasításnak *else* ága van?

- Mi a *while* utasítás?
- Mi a ciklusmag?
- Mit jelent, hogy a *while* elöltesztelő ciklus?
- Hogyan lehet hátultesztelő *while* ciklust írni?
- Mi a *for* ciklus?
- Elől-vagy hátultesztelő a *for* ciklus?
- Mit jelent az egymásba ágyazott ciklus?
- Mit nevezünk címkének, hogyan címkézhetünk utasításokat?
- Ismertesse a többszörös elágazás készítésére alkalmas utasítást!
- Ismertesse a *break* utasítás használatát!
- Mire használhatjuk a *continue* utasítást?

Lefordul-e hiba nélkül a következő kódrészlet? Ha nem, indokolja!

```
int i=0;
if(i) {
    System.out.println("Hello");
}
```

Lefordul-e hiba nélkül a következő kódrészlet? Ha nem, indokolja!

```
boolean b=true;
boolean b2=true;
if(b==b2) {
    System.out.println("So true");
}
```

Lefordul-e hiba nélkül a következő kódrészlet? Ha nem, indokolja!

```
int i=1;
int j=2;
if(i==1 || j==2)
    System.out.println("OK");
```

Lefordul-e hiba nélkül a következő kódrészlet? Ha nem, indokolja!

```
int i=1;
int j=2;
if(i==1 &| j==2)
    System.out.println("OK");
```

6.7. Gyakorló feladatok

Írjon programot,

amely két egész szám legnagyobb közös osztóját számolja ki.

Algoritmus: A nagyobbik számot csökkentsük a kisebbikkel, amíg a két szám egyenlő nem lesz. Ha már a két szám egyenlő, megkaptuk a legnagyobb közös osztót. (Figyelem: lehet, hogy a két szám már eleve egyenlő!)

Írjon programot,

amely év, hónap és nap számértékek alapján kiszámolja, hogy az év hányadik napjáról van szó. Figyelni kell a szökőévekre is!

Írjon programot,

amely kiírja egy egész szám számjegyeinek összegét.

Ötlet: az utolsó jegy maradékképzéssel könnyedén megkapható, utána pedig osztással el lehet távolítani az utolsó számjegyet.

Pl. a szám 123. Ekkor az utolsó számjegy 3 (a 10-es osztás maradéka), majd az utolsó jegy eldobása a 10-el való osztással oldható meg: 123/10 = 12.

7. Objektumok használata

Egy tipikus Java program sok objektumot hoz létre, amik üzenetek küldésével hatnak egymásra. Ezeken keresztül tud egy program különböző feladatokat végrehajtani. Amikor egy objektum befejezi a működését, az erőforrásai felszabadulnak, hogy más objektumok használhassák azokat.

A következő *CreateObjectDemo* program három objektumot hoz létre: egy *Point* és két *Rectangle* objektumot:

```
public class CreateObjectDemo {
    public static void main(String[] args) {
        Point originOne = new Point(23, 94);

        Rectangle rectOne =
            new Rectangle(originOne, 100, 200);
        Rectangle rectTwo = new Rectangle(50, 100);

        System.out.println("Width of rectOne: " +
            rectOne.width);
        System.out.println("Height of rectOne: " +
            rectOne.height);
        System.out.println("Area of rectOne: "
            + rectOne.area());

        rectTwo.origin = originOne;
        System.out.println("X Position of rectTwo: "
            + rectTwo.origin.x);
        System.out.println("Y Position of rectTwo: "
        + rectTwo.origin.y);
        rectTwo.move(40, 72);
        System.out.println("X Position of rectTwo: "
            + rectTwo.origin.x);
        System.out.println("Y Position of rectTwo: "
            + rectTwo.origin.y);
    }
}
```

Ez a program létrehoz, megváltoztat és információt ír ki különböző objektumokról. Kimenete:

```
Width of rectOne: 100
Height of rectOne: 200
Area of rectOne: 20000

X Position of rectTwo: 23
Y Position of rectTwo: 94
X Position of rectTwo: 40
Y Position of rectTwo: 72
```

7.1. Objektumok létrehozása

Az objektum alapját egy osztály szolgáltatja, osztályból hozunk létre (példányosítunk) objektumot. A következő sorok objektumokat hoznak létre, és változókhoz rendelik őket:

```
Point originOne = new Point(23, 94);
Rectangle rectOne = new Rectangle(originOne, 100, 200);
Rectangle rectTwo = new Rectangle(50, 100);
```

Az első sor egy *Point* osztályból, a második és harmadik a *Rectangle* osztályból hoz létre objektumot.

Minden sor a következőket tartalmazza:

- **Deklaráció**: Az = előtti részek a deklarációk, amik a változókhoz rendelik hozzá az objektum típusokat.
- **Példányosítás**: A *new* szó egy Java operátor, ami létrehoz egy objektumot.
- **Inicializáció**: A *new* operátort egy konsturktorhívás követi. Pl. a *Point (23,94)* meghívja a *Point* egyetlen konstruktorát. A konstruktor inicializálja az új objektumot.

Változó deklarálása objektum hivatkozásként

Egy változót a következőképpen deklarálunk:

```
type name
```

Ez közli a fordítóval, hogy a *name* tagot használjuk egy adatra hivatkozáshoz, aminek a típusa *type*. A Java nyelv a változó típusokat két fő kategóriára osztja: egyszerű típusok és hivatkozás (referencia) típusok.

Az egyszerű típusok (*byte, short, int, long, char, float, double, boolean*) mindig egyszerű értékeket tartalmaznak az adott típusból.

A hivatkozás típusok azonban némileg összetettebbek. A következő módok bármelyike szerint deklarálhatók:

- A deklarált típus megegyezik az objektum osztályával:

```
MyClass myObject = new MyClass();
```

- A deklarált típus egy szülő osztálya az objektum osztályának:

```
MyParent myObject = new MyClass();
```

- A deklarált típus egy interfész, amit az objektum osztálya implementál:

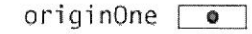
```
MyInterface myObject = new MyClass();
```

Így is deklarálható egy változó:

```
MyClass myObject;
```

Ha ezt használjuk, a *myObject* értéke automatikusan *null* lesz, amíg egy objektum ténylegesen létre nem lesz hozva és hozzárendelve. A változó deklaráció önmagában nem hoz létre objektumot. Ehhez a *new* operátort kell használni.

Amíg egy változó nem tartalmaz hivatkozást objektumra, *null* hivatkozást tartalmaz. Ha az *originOne* változót ilyen módon deklaráljuk, akkor a következőképen illusztrálható (változó neve a hivatkozással, ami nem mutat sehova):



Objektum példányosítása

A *new* operátor egy példányt hoz létre egy osztályból, és memóriaterületet foglal az új objektumnak.

A *new* operátor után szükség van egy osztályra, ami egyben egy konstruktor hívást is előír. A konstruktor neve adja meg, hogy melyik osztályból kell példányt létrehozni. A konstruktor inicializálja az új objektumot.

A *new* operátor egy hivatkozást ad vissza a létrehozott objektumra. Gyakran ezt a hivatkozást hozzárendeljük egy változóhoz. Ha a hivatkozás nincs hozzárendelve változóhoz, az objektumot nem lehet majd elérni, miután a *new* operátort tartalmazó utasítás végrehajtódott. Az ilyen objektumot **névtelen objektumnak** is szoktuk nevezni.

Megjegyzés: A névtelen objektumok nem olyan ritkák, mint ahogy azt gondolhatnánk. Pl. egy tömbbe vagy tárolóba helyezett objektum is névtelen, hiszen nincs saját, névvel ellátott hivatkozása.

Objektum inicializálása

A *Point* osztály kódja:

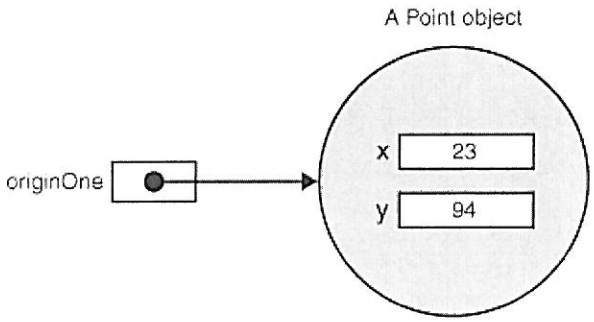
```
public class Point {
    public int x = 0;
    public int y = 0;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Ez az osztály egy konstruktort tartalmaz. A konstruktornak ugyanaz a neve, mint az osztálynak, és nincs visszatérési értéke. A *Point* osztály konstruktora két egész típusú paramétert kap: (*int x, int y*). A következő utasítás a 23 és 94 értékeket adja át paraméterként:

```
Point originOne = new Point(23, 94);
```

Ennek a hatását mutatja a következő ábra:



A *Rectangle* osztály négy konstruktort tartalmaz:

```
public class Rectangle {
    public int width = 0;
    public int height = 0;
    public Point origin;

    public Rectangle() {
        origin = new Point(0, 0);
    }

    public Rectangle(Point p) {
        origin = p;
    }

    public Rectangle(int w, int h) {
        this(new Point(0, 0), w, h);
    }
```

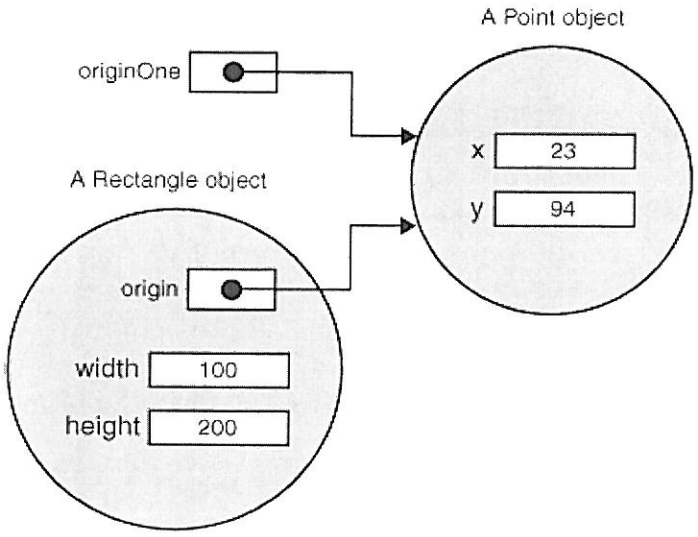
```
    public Rectangle(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }
    public void move(int x, int y) {
        origin.x = x;
        origin.y = y;
    }
    public int area() {
        return width * height;
    }
}
```

Akármelyik konstruktorral kezdeti értéket adhatunk a téglalapnak, különböző szempontok szerint: a koordinátái (*origin*); szélessége és magassága; mind a három; vagy egyik sem.

Ha egy osztálynak több konstruktora van, mindnek ugyanaz a neve, de különböző számú vagy különböző típusú paraméterekkel rendelkezik. A Java platform a konstruktort a paraméterek száma és típusa alapján különbözteti meg. A következő kódnál a *Rectangle* osztálynak azt a konstruktorát kell meghívnia, ami paraméterként egy *Point* objektumot és két egészet vár:

```
Rectangle rectOne = new Rectangle(originOne, 100, 200);
```

Ez a hívás inicializálja a téglalap *origin* változóját az *originOne*-nal (*originOne* értékét veszi fel az *origin* változó), ami egy *Point* objektumra hivatkozik, a *width* értéke 100-zal lesz egyenlő, a *height* értékét 200-zal. Most már két hivatkozás van ugyanarra a *Point* objektumra; egy objektumra több hivatkozás is lehet:



A következő sor két egész típusú paraméterrel rendelkező konstruktort hívja meg, amik a *width* és *height* értékei. Ha megnézzük a konstruktor kódját, látjuk, hogy létrehoz egy új *Point* objektumot, aminek az *x* és *y* értéke is nulla lesz:

```
Rectangle rectTwo = new Rectangle(50, 100);
```

Ez a konstruktor nem vár paramétereket, paraméter nélküli konstruktor:

```
Rectangle rect = new Rectangle();
```

Ha egy osztály nem deklarál egy konstruktort se, akkor a Java platform automatikusan szolgáltat egy paraméter nélküli (alapértelmezett, default) konstruktort, ami „nem csinál semmit". Így minden osztálynak van legalább egy konstruktora.

7.2. Hivatkozás egy objektum tagjaira

Ha létrehozunk egy objektumot, valószínűleg használni is akarjuk valamire. Ennek két módja lehet:

- módosítani vagy megnézni a tagváltozóit
- meghívni a metódusait

Általános formája:

```
objectReference.variableName
```

Amikor egy változó az érvényességi körén belül van – itt az osztályon belül –, használható az egyszerű hivatkozás is.

A hivatkozás első része egy objektum referencia kell, hogy legyen. Lehet használni egy objektum nevét, vagy egy kifejezést, ami egy objektum-hivatkozással tér vissza. A *new* operátor egy hivatkozással tér vissza, ezzel hozzáférhetünk egy új objektum változóihoz:

```
int height = new Rectangle().height;
```

Miután ez végrehajtódik, többé nem lehet hivatkozni a létrejött *Rectangle* objektumra, mert nem tároltuk el a hivatkozást egy változóban.

A tagváltozók hozzáférhetősége

Konvenció szerint egy objektum tagváltozóit más objektum vagy osztály közvetlenül nem módosíthatja, mert lehet, hogy értelmetlen érték kerülne bele.

Ehelyett, hogy a változtatást megengedje, egy osztály biztosíthat metódusokat, amiken keresztül más objektumok megnézhetik, illetve módosíthatják a változóit. Ezek a metódusok biztosítják, hogy a megfelelő típus kerüljön a változóba. A másik előnyük, hogy az osztály megváltoztathatja a változó nevét és típusát anélkül, hogy hatással lenne a klienseire.

Azonban néha szükség lehet arra, hogy közvetlen hozzáférést biztosítsunk a változókhoz. Ezt úgy tehetjük meg, hogy a *public* szót írjuk eléjük, a *private*-tel pedig tilthatjuk a külső hozzáférést.

7.3. Metódushívás

Itt is használható ugyanaz a forma, mint a változóknál. A metódus neve utáni zárójelekben adhatók meg a paraméterek. Ha nincsenek, üres zárójelet kell írnunk.

```
objectReference.methodName(argumentList);
objectReference.methodName();
```

Az objektum referencia itt is lehet változó vagy kifejezés:

```
new Rectangle(100, 50).area();
```

A kifejezés egy hivatkozást ad vissza a *Rectangle* objektumra. A pont után írva a metódus nevét, az végrehajtódik az új *Rectangle* objektumon.

Néhány eljárásnak van visszatérési értéke, ezért ezek kifejezésekben is használhatóak. A visszaadott értéket változóhoz rendelhetjük:

```
int areaOfRectangle = new Rectangle(100, 50).area();
```

Metódusok hozzáférhetősége

Ugyanúgy működik, mint a változókhoz való hozzáférés. A metódusokhoz való hozzáférést is a *public* kulcsszóval engedélyezhetjük más objektumoknak, a *private*-tel pedig tilthatjuk.

7.4. Nem használt objektumok eltávolítása

A Java platform lehetővé teszi annyi objektum létrehozását, amennyit csak akarunk (korlát csak az, hogy mennyit tud kezelni a rendszerünk), és nem kell aggódnunk a megszüntetésük miatt. A futtatókörnyezet törli az objektumokat, ha többet már nem használjuk. Ez a szemétgyűjtés.

Egy objektum akkor törölhető, ha már nincs rá több hivatkozás. Meg lehet szüntetni egy objektumot úgy is, hogy a hivatkozását *null* értékre állítjuk. Egy objektumra több hivatkozás is lehet, ezeket mind meg kell szüntetni, hogy az objektum a szemétgyűjtőbe kerülhessen.

A szemétgyűjtő

A szemétgyűjtő periódikusan felszabadítja a már nem használt objektumok által foglalt memóriát. Automatikusan végzi a dolgát, bár néha szükség lehet rá, hogy közvetlen meghívjuk. Ezt a *System* osztály *gc* metódusával tehetjük meg. Olyankor lehet rá szükség, ha egy kódrész sok szemetet hoz létre, vagy egy következő kódrésznek sok memóriára van szüksége. Általában elegendő hagyni, hogy magától fusson le.

Megjegyzés: Egyes fejlesztők a Java egyik gyenge pontjának tartják a memóriakezelését. Aki úgy gondolja, hogy tud jobb megoldást a Sun programozói által létrehozottnál, akár a sajátját is használhatja.

7.5. Takarítás

Mielőtt egy objektum a szemétgyűjtőbe kerülne, a gyűjtő lehetőséget ad az objektumnak, hogy kitakarítson maga után úgy, hogy meghívja az objektum *finalize* metódusát.

A *finalize* metódus az *Object* osztály tagja, ami minden osztály szülőosztálya, a hierarchia tetején áll. Egy osztály felülírhatja a *finalize* metódust, ha szükséges, de ekkor a metódus legvégén meg kell hívnia a *super.finalize()* függvényt.

7.6. Ellenőrző kérdések

- Mikor lehet egy objektumot paraméter nélkül létrehozni? Mikor nem?
- Hogy lehet egy objektum adattagjainak kezdőértékét beállítani? Mi történik, ha ezt nem tesszük meg?
- Mi történik azokkal az objektumokkal, amelyekre már nem hivatkozik a futó program?
- Mikor fut le a szemétgyűjtő algoritmus? Hogyan működik?

- Hogy néz ki Javában a destruktor?

Mi a hiba a következő programban?

```
public class SomethingIsWrong {
    public static void main(String[] args) {
        Rectangle myRect;
        myRect.width = 40;
        myRect.height = 50;
        System.out.println("myRect's area is " +
                           myRect.area());
    }
}
```

A következő kód létrehoz egy *Point* és egy *Rectangle* objektumot.

Hány referencia hivatkozik az objektumokra a következő kódrészlet lefutása után? Ha lefut a szemétgyűjtés, melyik objektum fog megszűnni?

```
...
Point point = new Point(2,4);
Rectangle rectangle = new Rectangle(point, 20, 20);
point = null;
...
```

7.7. Gyakorló feladat

Írjon programot, amely a felhasználótól beolvassa egy téglalap, majd tetszőleges számú pont koordinátáit, és minden egyes pontról megállapítja, hogy belül van-e a téglalapon.

Továbbfejlesztés (matematikailag nehezebb):

Számolja ki, hogy a pontok a téglalap melyik oldalához (vagy csúcsához) vannak a legközelebb!

8. Karakterek és sztringek

8.1. A *Character* osztály

Egy *Character* típusú objektum egyetlen karakter értéket tartalmaz. A *Character* objektumot az egyszerű *char* változó helyett használjuk, amikor objektum szükséges, például: amikor átadunk egy karakter értéket egy metódusnak, ami megváltoztatja az értéket, vagy amikor egy karakter értéket helyezünk el egy adattárolóban, mint például egy *ArrayList-ben*, ami objektumokat tud csak tárolni, primitív értékeket nem.

Megjegyzés: a burkoló (csomagoló) osztályokról a következő fejezetben lesz szó.

A következő példaprogram (*CharacterDemo*) létrehoz néhány *Character* objektumot, és megjelenít róluk néhány információt. A *Character* osztályhoz tartozó kód az alábbiakban látható:

```
public class CharacterDemo {
    public static void main(String args[]) {
        Character a = new Character('a');
        Character a2 = new Character('a');
        Character b = new Character('b');

        int difference = a.compareTo(b);

        if (difference == 0) {
            System.out.println("a is equal to b.");
        } else if (difference < 0) {
            System.out.println("a is less than b.");
        } else if (difference > 0) {
            System.out.println("a is greater than b.");
        }
        System.out.println("a is "
            + ((a.equals(a2)) ? "equal" : "not equal")
            + " to a2.");
        System.out.println("The character " + a.toString()
            + " is " + (Character.isUpperCase(a.charValue()) ?
                 "upper" : "lower")
            + "case.");
    }
}
```

A program kimenete a következő lesz:

```
a is less than b.
a is equal to a2.
The character a is lowercase.
```

A fenti példában a *Character.isUpperCase(a.charValue())* függvény adja az *a* nevű *Character* objektum kódját. Ez azért van, mert az *isUppercase* metódus *char* típusú paramétert vár. Ha a JDK 5.0-t vagy újabb fejlesztőkörnyezetet használunk, akkor ennek a metódusnak megadhatjuk a *Character* típusú objektumot is:

```
Character.isUpperCase(a)
```

A *CharacterDemo* program a *Character* osztály alábbi konstruktorait, illetve metódusait hívja meg:

- *Character(char)*: A *Character* osztály egyetlen konstruktora, amely létrehoz egy *Character* objektumot, melynek értékét a paraméterben adjuk meg, létrehozás után a *Character* objektum értéke nem változhat.
- *compareTo(Character)*: Összehasonlít két *Character* objektumban tárolt értéket, azt az objektumot, ami meghívta (a példában *a*), és azt, ami a paraméterben van (*b*). Visszaad egy egész számot, ami jelzi, hogy az objektum értéke kisebb, egyenlő, vagy nagyobb, mint a paraméterben megadott érték.
- *equals(Object)*: 2 *Character* objektumot hasonlít össze. *True* értékkel tér vissza, ha a két érték egyenlő.
- *toString()*: *Stringgé* konvertálja az objektumot, a sztring 1 karakter hosszú lesz, és a *Character* objektum értékét tartalmazza.
- *charValue()*: Megadja az objektum értékét egyszerű *char* értékként.
- *isUpperCase(char)*: Meghatározza, hogy az egyszerű *char* érték nagybetű-e.

A *Character* osztály néhány további fontosabb tagfüggvénye :

- *boolean isUpperCase(char)*
  boolean isLowerCase(char)
- *char toUpperCase(char)*
  char toLowerCase(char)
- *boolean isLetter(char)*
  boolean isDigit(char)
  boolean isLetterOrDigit(char)
- *boolean isWhitespace(char)*
- *boolean isSpaceChar(char)*
- *boolean isJavaIdentifierStart(char)*
  boolean isJavaIdentifierPart(char)

8.2. *String*, *StringBuffer* és *StringBuilder* osztály

A Java platform a kezdetektől fogva biztosított két osztályt, melyekkel tárolhatunk, illetve manipulálhatunk sztringeket, ezek a *String* és a *StringBuffer*. A *String* osztályban olyan sztringeket tárolunk, melyek értéke nem fog változni. A *StringBuffer* osztályt akkor használjuk, ha a szövegen szeretnénk változtatni, ezt elsősorban dinamikus karakterlánc készítésekor (pl. fájlból olvasás) használjuk. A *StringBuffer*-ek használata biztonságos több szálas környezetben. A *StringBuilder* osztályt a JDK 5.0-tól vezették be, ami gyorsabb, mint a *StringBuffer*, de csak egy szálon használható biztonságosan.

A következő irányelvek alapján döntsünk, hogy melyik osztályt használjuk:

- Ha a szöveg nem fog változni, használjuk a *String*-et.
- Ha a szöveg változni fog, és csak egy szálon keresztül fogjuk elérni, használjuk a *StringBuilder*-t.
- Ha a szöveg változni fog és több szálon keresztül fogjuk elérni *StringBuffer*-t használjuk.

A következő példaprogram neve *StringsDemo*, amely megfordítja egy sztring karaktereit. A program használja a *String* és *StringBuilder* osztályokat is. Ha a JDK 5.0-ás válto-

zatánál régebbit használ, a *StringBuilder* előfordulásait le kell cserélni *StringBuffer*-re, és a program működni fog.

```
public class StringsDemo {
    public static void main(String[] args) {
        String palindrome = "Dot saw I was Tod";
        int len = palindrome.length();
        StringBuilder dest = new StringBuilder(len);
        for (int i = (len - 1); i >= 0; i--) {
            dest.append(palindrome.charAt(i));
        }
        System.out.println(dest.toString());
    }
}
```

A program kimenete a következő lesz:

```
doT saw I was toD
```

Megjegyzés: Érdemes még a példán megfigyelni, hogy a *StringBuilder* (és *StringBuffer*) osztály példányosításakor az előre látható méretet meg lehet adni. Ez gyorsabb futást eredményez.

Sztring objektumok létrehozása

A sztringet gyakran egy sztring konstansból, egy karaktersorozatból készítjük. A *StringsDemo* program is ezt a módszert használja, hogy létrehozzon egy sztringet, amire a *palindrome* változóval hivatkozik:

```
String palindrome = "Dot saw I was Tod";
```

String-eket úgy is előállíthatunk, min bármilyen más Java objektumot: a *new* kulcsszó és a konstruktor segítségével. A *String* osztály több konstruktort szolgáltat, amelyekkel beállíthatjuk a *String* kezdőértékét, különböző forrásokat használva, mint például karakter tömböt, byte tömböt, *StringBuffer*t, vagy *StringBuilder*t.

A következő táblázat a *String* osztály konstruktorait tartalmazza:

String()	Üres *String*-et hoz létre.
String(byte[]) *String(byte[], int, int)* *String(byte[], int, int, String)* *String(byte[], String)*	Bájttömb tartalma alapján jön létre. Lehetőség van egy egész kezdőindex és hossz megadására részintervallum figyelembevételéhez, illetve meg lehet adni karakterkódolást is.
String(char[]) *String(char[], int, int)*	Karaktertömb egésze vagy csak egy része alapján jön létre.
String(String)	Másik *String* másolatát hozza létre.
String(StringBuffer)	*StringBuffer* tartalma alapján jön létre.
String(StringBuilder)	*StringBuilder* tartalma alapján jön létre.

Egy példa arra, amikor karaktertömbből készítünk sztringet:

```
char[] helloArray = { 'h', 'e', 'l', 'l', 'o' };
String helloString = new String(helloArray);
System.out.println(helloString);
```

A kódrészlet utolsó sora ezt írja ki:

```
hello
```

Ha *StringBuffer*-t, vagy *StringBuilder*-t hozunk létre, mindig használnunk kell a *new* operátort. Mivel a két osztálynak azonosak a konstruktorai, a következő táblázat csak a *StringBuffer* osztály konstruktorait tartalmazza:

StringBuffer()
StringBuffer(CharSequence)
StringBuffer(int)
StringBuffer(String)

A *StringsDemo* programban egy *dest* nevű *StringBuildert* hozunk létre, azt a konstruktort használva, amely a puffer kapacitását állítja be:

```
String palindrome = "Dot saw I was Tod";
int len = palindrome.length();
StringBuilder dest = new StringBuilder(len);
```

A kód létrehoz egy *StringBuilder*t, melynek kezdeti kapacitása megegyezik a *palindrome* nevű *String* hosszával. Ez csak a memóriafoglalást biztosítja a *dest* számára, mert ennek mérete pontosan elegendő lesz a bemásolandó karakterek számára. A *StringBuffer* vagy *StringBuilder* kapacitásának inicializálásával egy elfogadható méretre minimalizálhatjuk a memóriafoglalások számát, amivel sokkal hatékonyabbá tehetjük programunkat, mert a memóriafoglalás elég költséges művelet.

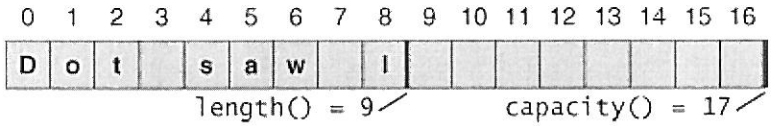
Megjegyzés: Ha egy *StringBuilder* vagy *StringBuffer* objektum méret növelése során a szabad kapacitása elfogy, akkor egy új, kétszer akkora memóriaterület kerül lefoglalásra, ahová a régi tartalom átmásolásra kerül. Ebből is látszik, hogy ha tudjuk, érdemes pontosan (vagy legalább becsülten) megadni a szükséges kapacitást.

A Stringek hossza

Azon metódusokat, amelyeket arra használunk, hogy információt szerezzünk egy objektumról, olvasó (vagy hozzáférő) metódusoknak nevezzük. Egy ilyen *String*-eknél, *StringBuffer*-eknék és *StringBuilder*-eknél használható metódus a *length*, amely visszaadja az objektumban tárolt karakterek számát. Miután az alábbi két sor végrehajtódik, a *len* változó értéke 17 lesz:

```
String palindrome = "Dot saw I was Tod";
int len = palindrome.length();
```

A *length* mellett *StringBuffer* és *StringBuilder* osztályok esetén használhatjuk még a *capacity* metódust is, amely a lefoglalt terület méretét adja vissza, és nem a használt területet. Például a *dest* nevű *StringBuilder* kapacitása a *StringDemo* programban nem változik, míg a hossza minden karakter beolvasása után nő egyel. A következő ábra mutatja a *dest* kapacitását és hosszát, miután kilenc karakter hozzá lett fűzve.



A *StringBuffer* vagy *StringBuilder* hossza a benne tárolt karakterek száma, míg kapacitása az előre lefoglalt karakterhelyek száma. A *String* osztály esetén a *capacity* metódus nem használható, mert a *String* tartalma nem változhat meg.

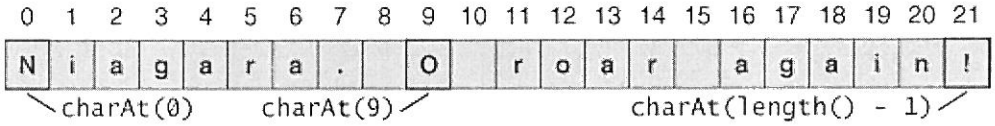
Stringek karaktereinek olvasása

A megfelelő indexű karaktert megkapjuk a *String*-en, *StringBuffer*-en vagy a *StringBuilder*-en belül, ha meghívjuk a *charAt* függvényt. Az első karakter indexe 0, az utolsó karakteré pedig a *length()-1*.

Például az alábbi forráskódban a 9. indexű karaktert kapjuk meg a *String*-ben:

```
String anotherPalindrome = "Niagara. O roar again!";
char aChar = anotherPalindrome.charAt(9);
```

Az indexelés 0-val kezdődik, tehát a 9-es indexű karakter az 'o', mint ahogy a következő ábra is mutatja:



Használjuk a *charAt* függvényt, hogy megkapjuk a megfelelő indexű karaktert.

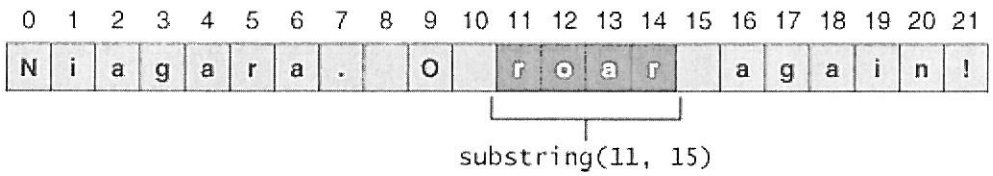
Az ábra is mutatja, hogy hogyan lehet kiszámítani egy *String*-ben az utolsó karakter indexét. Ki kell vonni a *length*() függvény visszatérési értékéből 1-et.

Ha több, mint egy karaktert szeretnénk megkapni a *String*-ből, *StringBuffer*-ből vagy *StringBuilder*-ből, akkor a *substring* függvényt kell használni. A *substring*-nek két fajtája van, amit az alábbi táblázat is mutat:

String substring(int) *String substring(int, int)*	Visszatérési érték egy új *String*, ami az eredeti sztring részének másolata. Az első paraméter az első karakter indexe (ahonnan kérjük a karaktereket), a második int paraméter pedig az utolsó karakter indexe (ameddig kérjük)-1. A *substring* hosszát megkapjuk, ha a második paraméter értékből kivonjuk az első paraméter értékét. Ha nem adjuk meg a második paramétert, akkor az eredeti *String* végéig történik a másolás.

Az alábbi forráskód a Niagara tükörmondatból ad vissza egy részletet, ami a 11. indextől a 15. indexig tart, ez pedig a *„roar"* kifejezés:

```
String anotherPalindrome = "Niagara. O roar again!";
String roar = anotherPalindrome.substring(11, 15);
```



Karakter vagy String keresése Stringben

A *String* osztály két függvényt nyújt, amelyek pozícióval térnek vissza: *indexOf* és a *lastIndexOf*. A következő táblázat e két függvény alakjait mutatja be:

int indexOf(int)	Visszaadja az első (utolsó) előforduló karakter indexét.

int lastIndexOf(int)

int indexOf(int, int) *int lastIndexOf(int, int)*	Visszaadja az első (utolsó) előfordulókarakter indexét, az indextől előre (visszafele) keresve.
int indexOf(String) *int lastIndexOf(String)*	Visszaadja az első (utolsó) előforduló *String* indexét.
int indexOf(String, int) *int lastIndexOf(String, int)*	Visszaadja a String első (utolsó) előfordulásának indexét, a megadott indextől előre (visszafele) keresve.

A *StringBuffer* és a *StringBuilder* osztályok nem támogatják az *indexOf* és a *lastIndexOf* függvényeket. Ha használni szeretnénk ezeket a metódusokat, először *String*-gé kell konvertálnunk az objektumot a *toString* függvény segítségével.

Megjegyzés: A metódusok az alábbi *Filename* osztályban nem minden hibát kezelnek, és feltételezik, hogy az paraméter tartalmazza a teljes könyvtár útvonalat, és a fájlnevet kiterjesztéssel.

```
public class Filename {
    private String fullPath;
    private char pathSeparator, extensionSeparator;

    public Filename(String str, char sep, char ext) {
        fullPath = str;
        pathSeparator = sep;
        extensionSeparator = ext;
    }

    public String extension() {
        int dot = fullPath.lastIndexOf(extensionSeparator);
        return fullPath.substring(dot + 1);
    }

    public String filename() {
        int dot = fullPath.lastIndexOf(extensionSeparator);
        int sep = fullPath.lastIndexOf(pathSeparator);
        return fullPath.substring(sep + 1, dot);
    }

    public String path() {
        int sep = fullPath.lastIndexOf(pathSeparator);
        return fullPath.substring(0, sep);
    }
}
```

A következő program létrehoz egy *Filename* objektumot, és meghívja a metódusait:

```
public class FilenameDemo {
    public static void main(String[] args) {
        final String FPATH = "/home/mem/index.html";
        Filename myHomePage = new Filename(FPATH,
                                           '/', '.');
        System.out.println("Extension = " +
            myHomePage.extension());
        System.out.println("Filename = " +
            myHomePage.filename());
        System.out.println("Path = " +
            myHomePage.path());
    }
}
```

A program kimenete:

```
Extension = html
Filename = index
Path = /home/mem
```

Ahogy a fenti példa is mutatja, az *extension* metódus a *lastIndexOf* függvényt használja, hogy megtalálja az utolsó pontot a fájlnévben. Ha a fájlnévben nincs pont, a *lastIndexOf* visszatérési értéke -1, és a *substring* metódus *StringIndexOutOfBoundsException* kivételt dob.

Ha a pont karakter (.) az utolsó karakter a *String*-ben, akkor ez egyenlő a *String* hosszával, ami egyel nagyobb, mint a legnagyobb index a *String*-ben (mivel az indexelés 0-val kezdődik).

Sztringek és rész-sztringek összehasonlítása

A *String* osztálynak van néhány függvénye a sztringek és a rész-sztringek összehasonlítására. Az alábbi táblázat ezeket a függvényeket mutatja be:

boolean endsWith(String) *boolean startsWith(String)* *boolean startsWith(String, int)*	Visszatérési értéke igaz, ha a *String* a paraméterben megadott szóval kezdődik, vagy végződik. Az *int* paraméterben az eltolási értéket adhatjuk meg, hogy az eredeti *String*-ben hanyadik indextől kezdődjön a keresés.
int compareTo(String) *int compareTo(Object)* *int compareToIgnoreCase(String)*	Két *String*-et hasonlít össze ABC szerint, és egy egész számmal tér vissza, jelezve, hogy ez a *String* nagyobb (eredmény>0), egyenlő (eredmény=0), illetve kisebb (eredmény<0), mint a paraméter. A *CompareToIgnoreCase* nem tesz különbséget a kis-és nagybetűk között.
boolean equals(Object) *boolean equalsIgnoreCase(String)*	Visszatérési értéke igaz, ha a *String* ugyanazt a karaktersorozatot tartalmazza, mint a paramétere. Az *equalsIgnoreCase* függvény nem tesz különbséget kis- és nagybetűk között; így *'a'* és *'A'* egyenlő.

Sztringek módosítása

A *String* osztály sokféle metódust tartalmaz a *String*-ek módosításához. Természetesen a *String* objektumokat nem tudja módosítani, ezek a metódusok egy másik *String*-et hoznak létre, ez tartalmazza a változtatásokat. Ezt követhetjük az alábbi táblázatban.

String concat(String)	A *String* végéhez láncolja a *String* paramétert. Ha az paraméter hossza 0, akkor az eredeti *String* objektumot adja vissza.
String replace(char, char)	Felcseréli az összes első paraméterként megadott ka-

	raktert a második paraméterben megadottra. Ha nincs szükség cserére, akkor az eredeti *String* objektumot adja vissza.
String trim()	Eltávolítja az elválasztó karaktereket a *String* elejéről és a végéről.
String toLowerCase() *String toUpperCase()*	Konvertálja a *String*-et kis, vagy nagybetűsre. Ha nincs szükség konverzióra, az eredeti *String*-et adja vissza.

Íme egy rövid program (*BostonAccentDemo*), ami a *replace* metódussal egy *String*-et fordít Bostoni dialektusra:

```
public class BostonAccentDemo {
    private static void bostonAccent(String sentence) {
        char r = 'r';
        char h = 'h';
        String translatedSentence = sentence.replace(r, h);
        System.out.println(translatedSentence);
    }
    public static void main(String[] args) {
        String translateThis =
            "Park the car in Harvard yard.";
        bostonAccent(translateThis);
    }
}
```

A *replace* metódus kicseréli az összes *r*-t *h*-ra a mondatokban.

A program kimenete:

```
Pahk the cah in Hahvahd yahd.
```

A *StringBuffer*-ek módosítása

A *StringBuffer*-ek tartalma módosítható. A következő táblázat összefoglalja a *StringBuffer*-ek módosításához használható metódusokat. Azonos metódusokat tartalmaz a *StringBuilder* osztály is, de *StringBuilder*-eket is ad vissza, ezért ezeket külön nem soroljuk fel.

StringBuffer append(boolean) *StringBuffer append(char)* *StringBuffer append(char[])* *StringBuffer append(char[], int, int)* *StringBuffer append(double)* *StringBuffer append(float)* *StringBuffer append(int)* *StringBuffer append(long)* *StringBuffer append(Object)* *StringBuffer append(String)*	Hozzáfűzi a megadott paramétert a *StringBuffer*-hez. Az adat *String*-gé konvertálódik, mielőtt a hozzáfűzés megtörténne.
StringBuffer delete(int, int) *StringBuffer deleteCharAt(int)*	Törli a megadott karaktereket a *StringBuffer*-ből.
StringBuffer insert(int, boolean)	A *StringBuffer*-hez ad egy új paramétert.

StringBuffer insert(int, char) *StringBuffer insert(int, char[])* *StringBuffer insert(int, char[], int, int)* *StringBuffer insert(int, double)* *StringBuffer insert(int, float)* *StringBuffer insert(int, int)* *StringBuffer insert(int, long)* *StringBuffer insert(int, Object)* *StringBuffer insert(int, String)*	Az első egész típusú paraméter jelzi az adat indexét, ahova a beillesztés történik. Az adat *String*-gé konvertálódik, mielőtt a beillesztés megtörténik.
StringBuffer replace(int, int, String) *void setCharAt(int, char)*	Kicseréli a megadott karaktereket a *StringBuffer*-ben.
StringBuffer reverse()	Felcseréli a karakterek sorrendjét a *StringBuffer*-ben.

Az *append* metódusra már láttunk példát a *StringsDemo* programban, a fejezet elején. Az alábbi *InsertDemo* program bemutatja az *insert* metódus használatát. Beilleszt egy *String*-et a *StringBuffer*-be:

```
public class InsertDemo {
    public static void main(String[] args) {
        StringBuffer palindrome = new StringBuffer(
            "A man, a plan, a canal; Panama.");
        palindrome.insert(15, "a cat, ");
    System.out.println(palindrome);
    }
}
```

A program kimenete:

```
A man, a plan, a cat, a canal; Panama.
```

Az általunk megadott index utáni helyre kerül beillesztésre az adat. A *StringBuffer* elejére való beillesztéshez a 0 indexet kell használni. A végéhez való beillesztés esetén az index értéke megegyezik a *StringBuffer* jelenlegi hosszával, vagy használjuk a hozzáfűzést (*append*) is.

Ha a művelet miatt túlságosan megnő a *StringBuffer* mérete, akkor az több memóriát fog lefoglalni. Mivel a memória lefoglalás költséges, ezért lehetőleg úgy kell elkészíteni a kódot, hogy megfelelően be legyen állítva a *StringBuffer* mérete.

A *String*-ek és a fordító

A fordító felhasználja a *String* és a *StringBuffer* osztályokat a háttérben, hogy a *String*-literálokat és különböző összefűzéseket kezelje. A *String*-et idézőjelek között adhatjuk meg:

```
"Hello World!"
```

String-literálokat bárhol használhatunk *String* példányként. Példaként, a *System.out.println* paraméterének *String*-literált adunk meg:

```
System.out.println("Might I add that you look lovely today.");
```

Használhatunk *String* metódust közvetlenül a *String*-literálból hívva:

```
int len = "Goodbye Cruel World".length();
```

Használhatjuk a *String*-literált *String* inicializálásra:

```
String s = "Hola Mundo";
```

A következő példa egyenértékű az előzővel, de nem olyan hatékony. Két azonos *String*-et készít, használata kerülendő:

```
String s = new String("Hola Mundo"); //ne használjuk
```

Használható a + operátor a *String*-ek összefűzésére:

```
String cat = "cat";
System.out.println("con" + cat + "enation");
```

Az előző példa alapján a fordító a *StringBuffer*-eket használja az összefűzés végrehajtására:

```
String cat = "cat";
System.out.println(new StringBuffer().append("con").
append(cat).append("enation").toString());
```

Használható a + operátor az összefűzésre:

```
System.out.println("You're number " + 1);
```

A fordító konvertálja a nem *String* értéket (a példában *int* 1-et) *String* objektummá, mielőtt az összefűzést elvégzi.

8.3. Sztringek darabolása

A *java.util.StringTokenizer* osztály hasznos lehet, ha egy *String*-et adott elválasztó karakter(ek) mentén szét kell bontani. A következő egyszerű példa bemutatja a használat módját:

```
StringTokenizer st = new StringTokenizer("this is a test");
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

A kód a következő eredményt írja ki:

```
this
is
a
test
```

A *StringTokenizer* objektum nyilván tartja, hogy a feldolgozás a *String* melyik pontján jár. A konstruktornak megadhatunk a szövegen kívül egy elválasztó-karaktereket tartalmazó *String*-et is, ekkor az alapértelmezett " \t\n\r\f" elválasztók helyett ezt fogja az objektum figyelembe venni.

8.4. Ellenőrző kérdések

- Mi a karakter?
- Hányféle jelet képes tárolni a Java *char* típusa?
- Hogy hívják a Java által támogatott karaktertípust?
- Mi a karaktersorozat (sztring?)

- Mit jelent, hogy a *String* nem megváltoztatható?
- Hogyan lehet egy *String*-nek kezdőértéket adni?
- Mire való a *String indexOf* metódusa?
- Mire való *String substring* metódusa?
- Mi a különbség a *StringBuilder*, a *StringBuffer* és a *String* között?

Melyik kifejezés értéke lesz logikailag igaz?

- *"john" == "john"*
- *"john".equals("john")*
- *"john" = "john"*
- *"john".equals(new Button("john"))*

Melyik fordul le?

- *"john" + " was " + " here"*
- *"john" + 3*
- *3 + 5*
- *5 + 5.5*

Mit ír ki a következő kódrészlet?

```
String s = new String("Bicycle");
int iBegin=1;
char iEnd=3;
System.out.println(s.substring(iBegin, iEnd));
```

- *Bic*
- *ic*
- *icy*
- fordítási hiba miatt nem indul el

Ha a "Java" tartalmú s *String*-ben keressük a 'v' betű pozícióját (a 2-t), akkor melyik metódushívással kapjuk ezt meg?

- *mid(2,s);*
- *s.charAt(2);*
- *s.indexOf('v');*
- *indexOf(s,'v');*

A következő deklarációk esetén melyik művelet érvényes?

```
String s1 = new String("Hello")
String s2 = new String("there");
String s3 = new String();
```

- *s3=s1 + s2;*
- *s3=s1 - s2;*

- *s3=s1 & s2;*
- *s3=s1 && s2*

8.5. Gyakorló feladatok

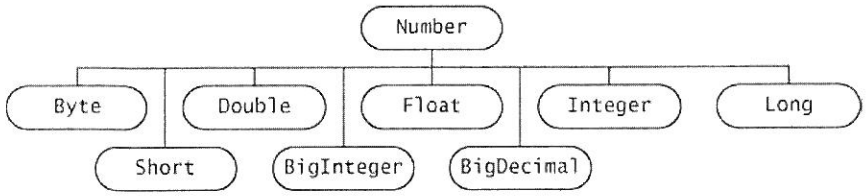
Írjon programot, amely az „Indul a görög aludni" *String* tartalmát

- megfordítja és kiírja,
- szavanként megfordítja és kiírja,
- törli az összes 'g' betűt és kiírja.

Figyelem, a megfordított *String* objektumot elő kell állítani!

9. Számok

A következő ábra bemutatja a Jáva platformon rendelkezésre álló szám osztályok hierarchiáját.



Ezen kívül a platform tartalmazza a *Boolean*, *Void* és *Character* osztályokat, amelyek a szám osztályokkal együtt **csomagoló** (vagy **burkoló**) osztályoknak hívunk.

Talán meglepő, hogy miért nélkülözhetetlenek a csomagoló osztályok, de ezt majd látni fogjuk a későbbiekben.

9.1. A csomagoló osztályok néhány használata

- Tudunk használni olyan tároló objektumokat, amelyekbe különböző típusú számok helyezhetők bele. Pl. az *ArrayList* osztály *Object*-eket tud tárolni, így akár az *Object* különböző leszármazott objektumait (vagyis csomagoló objektumokat) el tudunk helyezni benne.
- Az előző ponthoz hasonlóan akár más típusú objektumokkal is tudunk együtt tárolni számokat.
- Az osztályok definiálnak hasznos változókat, mint például *MIN_VALUE* vagy *MAX_VALUE*, ezek biztosítják az általános információt az adattípusról. Az osztályok definiálnak hasznos metódusokat, amelyek értéket konvertálnak más típussá, például *String*-gé, vagy egy másik szám típussá.

Továbbá a *BigInteger* és *BigDecimal* kiterjeszti a hagyományos adattípusokat, és tetszőleges pontosság alkalmazását engedi meg. A többi osztály a *java.lang* csomag része, a *BigDecimal* és a *BigInteger* a *java.math* csomagban van.

Következzen egy példa (*NumberDemo*), amely létrehoz két *Float* és egy *Double* objektumot, és utána használja a *compareTo* és az *equalTo* metódusokat összehasonlításra.

```
public class NumberDemo {
    public static void main(String args[]) {
        Float floatOne = new Float(14.78f - 13.78f);
        Float floatTwo = Float.valueOf("1.0");
        Double doubleOne = new Double(1.0);

        int difference = floatOne.compareTo(floatTwo);
```

```
        if (difference == 0) {
            System.out.println(
                        "floatOne is equal to floatTwo.");
        } else if (difference < 0) {
            System.out.println(
                        "floatOne is less than floatTwo.");
        } else if (difference > 0) {
            System.out.println(
                        "floatOne is greater than floatTwo.");
        }

        System.out.println("floatOne is "
                           + ((floatOne.equals(doubleOne)) ?
                           "equal" : "not equal")
                           + " to doubleOne.");
    }
}
```

Ennek a programnak a kimenete kicsit meglepő lehet:

```
floatOne is equal to floatTwo.
floatOne is not equal to doubleOne.
```

A következő táblázat bemutatja azon példánymetódusokat, melyeket a *Number* osztály összes leszármazott osztálya tartalmaz, beleszámítva az összehasonlító és egyenlőségvizsgáló metódusokat, melyek az előző példában szerepeltek.

byte byteValue() *short shortValue()* *int intValue()* *long longValue()* *float floatValue()* *double doubleValue()*	Konvertálja a szám objektum értékét egy egyszerű adattípussá (*short, long, float, doube*)
int compareTo(Integer) *int compareTo(Object)*	Összehasonlítja a szám objektumot a paraméterrel. A metódus nullánál kisebb, nulla, vagy nullánál nagyobb értékkel tér vissza attól függően, hogy az objektum értéke kisebb, egyenlő vagy nagyobb a hasonlított objektummal.
boolean equals(Object)	Meghatározza, hogy a szám objektum egyenlő-e a paraméterrel. Különböző típusú szám objektumok – a matematikai értéküktől függetlenül – sosem egyenlőek.

A szám csomagoló osztályok több hasznos publikus és statikus konstanst tartalmaznak. Ezekre úgy tudunk hivatkozni kifejezésekben, hogy az osztály nevét és a konstanst egy ponttal választjuk el, pl. *Integer.MIN_VALUE*.

A következő táblázat felsorol több hasznos konstanst a *Float* és *Double* osztályokból

Float.NaN *Double.NaN*	Nem szám. Ez az alapértelmezett érték, ha egy kifejezés értéke matematikailag nem értelmezhető.

Float.NEGATIVE_INFINITY *Double.NEGATIVE_INFINITY*	Negatív végtelen érték
Float.POSITIVE_INFINITY *Double.POSITIVE_INFINITY*	Pozitív végtelen érték

9.2. Szövegből számmá konvertálás

Néha szükség van arra, hogy a *String*ként rendelkezésre álló adatot számként kell értelmeznünk, például a felhasználó által bevitt adatot esetén.

A numerikus csomagoló osztályok mindegyike biztosítja a *valueOf* metódust, amely *String*-et konvertál adott objektummá.

A következő *ValueOfDemo* példaprogram kap két *String*-et parancssorból, konvertálja azokat szám objektummá, majd elvégez két aritmetikai műveletet az értékekkel.

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

        if (args.length == 2) {

            float a = Float.valueOf(args[0]).floatValue();
            float b = Float.valueOf(args[1]).floatValue();

            System.out.println("a + b = " + (a + b) );
            System.out.println("a - b = " + (a - b) );
            System.out.println("a * b = " + (a * b) );
            System.out.println("a / b = " + (a / b) );
            System.out.println("a % b = " + (a % b) );
        } else {
            System.out.println("This program requires" +
                "two command-line arguments.");
        }
    }
}
```

A következőkben a program kimenetét láthatjuk, amely a *4.5* és a *87.2*-t használta parancssori paraméternek:

```
a + b = 91.7
a - b = -82.7
a * b = 392.4
a / b = 0.0516055
a % b = 4.5
```

9.3. Számból szöveggé konvertálás

Néha szükség van a számból szövegbe konvertálásra, mert szükség van a *String* formátumban lévő értékkel való műveletekre.

Az összes osztály örökli a *toString* metódust az *Object* osztálytól. A csomagoló osztályok felülírják ezt a metódust, hogy biztosítsák az ésszerű konverziót.

A következő *ToStringDemo* program használja a *toString* metódust, és számot konvertál szöveggé. Utána a program néhány *String* metódust mutat be, amivel megszámolja a szám tizedes pont előtti és utáni szemjegyeinek számát.

```
public class ToStringDemo {
    public static void main(String[] args) {
        double d = 858.48;
        String s = Double.toString(d);

        int dot = s.indexOf('.');
        System.out.println(s.substring(0, dot).length()
                           + " digits before decimal point.");
        System.out.println(s.substring(dot+1).length()
                           + " digits after decimal point.");
    }
}
```

A program kimenete:

```
3 digits before decimal point.
2 digits after decimal point.
```

9.4. Számok formázott konvertálása

A szám csomagoló osztályok ugyan rendelkeznek a *toString* metódussal, de az így létrejövő szöveges forma nem minden esetben megfelelő. Például pénzösszegek valós számként való tárolásánál általában elegendő a két tizedes jegy használata. Formázott konverzióhoz használható a *NumberFormat* és leszármazottja, a *DecimalFormat* osztályok a *java.text* csomagból.

A következő kódrészlet *Double* objektumot formáz. A *getNumberInstance* metódus egy *NumberFormat* objektumot ad vissza. Ennek *format* metódusa egy *Double* értéket vár, és formázott *String*-et állít elő:

```
Double amount = new Double(345987.246);
NumberFormat numberFormatter;
String amountOut;
numberFormatter = NumberFormat.getNumberInstance();
amountOut = numberFormatter.format(amount);
System.out.println(amountOut);
```

A kód utolsó sora a következő írja ki:

```
345,987.246
```

Megjegyzés: Az eredmény függ a helyi beállításoktól.

Pénznem formátum

Gazdasági alkalmazás esetén pénzösszegek kezelésére is szükség van. Ennek alkalmazását mutatja a következő kódrészlet:

```
Double currency = new Double(9876543.21);
NumberFormat currencyFormatter;
String currencyOut;

currencyFormatter = NumberFormat.getCurrencyInstance();
currencyOut = currencyFormatter.format(currency);
System.out.println(currencyOut);
```

Az utolsó sor kiírja hogy:

```
876,543.21
```

A százalék formátum

Százalék formában való kiíráshoz szintén a helyi beállításokat figyelembe vevő objektumot kapunk a *getPercentInstance* metódus segítségével:

```
Double percent = new Double(0.75);
NumberFormat percentFormatter;
String percentOut;

percentFormatter = NumberFormat.getPercentInstance();
percentOut = percentFormatter.format(percent);
System.out.println(percentOut);
```

A *printf* metódus lehetőségei

A JDK 5.0 tette lehetővé a *java.io.PrintStream printf* metódusának alkalmazását, amely a kimenet formázását nagyban leegyszerűsítette. A metódus deklarációja:

```
public PrintStream printf(String format, Object... args)
```

Az első paraméter azt határozza meg, hogy a további paraméterek milyen formátumban kerüljenek kiírásra. A sokféle lehetőség szerencsére alaposan dokumentálva van az *API* specifikációban.

Nézzük a következő példát:

```
Calendar c =...;
String s =
    String.format("Duke's Birthday: %1$tm %1$te,%1$tY",c);
```

A formátum meghatározás egyszerű. Három van belőlük: *%1$tm*, *%1$te*, és *%1$tY*, amelyek mindegyike elfogadja a *c* nevű *Calendar* objektumot. A formátum meghatározás a következőket jelenti:

- *%* jelzi a formátum meghatározás kezdetét
- *1$* az első további paraméter jelzése
- *tm* jelenti a hónapot
- *te* jelenti a hónap napját
- *tY* jelenti az évet

Sablon készítése

A *DecimalFormat* osztályban meghatározhatjuk egy *String* minta segítségével a speciális kiírási formátumot. A minta fogja meghatározni, hogy hogyan nézzen ki a megformázott szám. A következő példa készít egy formázót, ami átadja a *String* mintát a *DecimalFormat* konstruktorának. A *format* metódus átvesz egy *Double* értéket, és formázott *String*-gel tér vissza.

```
DecimalFormat myFormatter = new DecimalFormat(pattern);
String output = myFormatter.format(value);
System.out.println(value + " " + pattern + " " + output);
```

A fenti kódsorok kimenete a lenti táblázatban látható.

value értéke	*pattern* értéke	eredmény	magyarázat
123456.789	###,###.###	123,456.789	A kettős kereszt (#) utal a számjegyre, a vessző a csoportosítás helyét jelzi, és a pont jelzi a tizedespont helyét.
123456.789	###.##	123456.79	A *value*-nak három számjegye van a tizedes pont után, de a *pattern*-nek csak kettő. A *format* metódus ezt úgy oldja meg, hogy felkerekíti a számot.
123.78	000000.000	000123.780	A *pattern* itt a kettős kereszt (#) helyett kezdő és soron következő nullákat definiál.
12345.67	$###,###.###	$12,345.67	Az első karakter a *pattern*-ben a dollár jel ($). Ez annyit jelent, hogy rögtön a balról első számjegy elé kerül.
12345.67	\u00A5###,###.###	¥12,345.67	A *pattern* beilleszti a Japán pénzt, a yen-t (¥) az Unicode értékével : 00A5.

A formázott szimbólumok módosítása

Erre a feladatra a *DecimalFormatSymbols* osztályt lehet használni. Ezzel az osztállyal meg tudjuk változtatni a *format* metódus által már formázott szimbólumokat. Ezek a szimbólumok többek között magukba foglalják a tizedesvesszőt, az ezres csoportosítást, a mínuszjelet, és a százalék jelet.

A következő példa demonstrálja a *DecimalFormatSymbols* osztályt alkalmazva egy szokatlan számalakra. A szokatlan formátum eredménye a következő metódusok hívása: *setDecimalSeparator*, *setGroupingSeparator*, *setGroupingSize*.

```
DecimalFormatSymbols unusualSymbols =
    new DecimalFormatSymbols(currentLocale);
unusualSymbols.setDecimalSeparator('|');
unusualSymbols.setGroupingSeparator('^');

String strange = "#,##0.###";
DecimalFormat weirdFormatter =
                new DecimalFormat(strange, unusualSymbols);
weirdFormatter.setGroupingSize(4);

String bizarre = weirdFormatter.format(12345.678);
System.out.println(bizarre);
```

Futtatás során, a példa furcsa alakban jeleníti meg a számot:

```
1^2345|678
```

9.5. Aritmetika

A Java programozási nyelv támogatja az alapvető aritmetikai számításokat az aritmetikai operátorokkal együtt: +, -, *, /, és %. A *java.lang* csomagban a Java platform biztosítja a *Math* osztályt. Ez olyan metódusokat és változókat biztosít, amik segítségével már magasabb rendű matematikai számítások is elvégezhetők, mit például egy szög szinuszának kiszámítása, vagy egy szám bizonyos hatványra emelése.

A *Math* osztály metódusai osztálymetódusok, tehát közvetlenül az osztály nevével kell őket meghívni, például így:

```
Math.round(34.87);
```

A *Math* osztály metódusai közül elsőként különböző alapvető matematikai függvényeket nézzük meg, mint például egy szám abszolút értékének kiszámítása vagy egy szám kerekítése valamelyik irányban. A következő táblázat ezeket a metódusokat tartalmazza:

double abs(double) *float abs(float)* *int abs(int)* *long abs(long)*	A paraméterként kapott paraméter abszolút értékével tér vissza.
double ceil(double)	A legkisebb *double* értékkel tér vissza, ami nagyobb vagy egyenlő a paraméterrel, és egyenlő egy egész számmal. (felfelé kerekít)
double floor(double)	A legnagyobb *double* értékkel tér vissza, ami kisebb vagy egyenlő a paraméterrel, és azonos egy egész számmal. (lefelé kerekít)
double rint(double)	A paraméterhez legközelebb álló *double* értékkel tér vissza, és azonos egy egész számmal. (a legközelebbi egészhez kerekít)
long round(double) *int round(float)*	A legközelebbi *long* vagy *int* értéket adja vissza, ahogy azt a metódus visszatérési értéke jelzi.

A következő *BasicMathDemo* program néhány alkalmazási módot illusztrál a fentiekre:

```
public class BasicMathDemo {
    public static void main(String[] args) {
        double aNumber = -191.635;

        System.out.println("The absolute value of " + aNumber
            + " is " + Math.abs(aNumber));
        System.out.println("The ceiling of " + aNumber +" is "
            + Math.ceil(aNumber));

        System.out.println("The floor of " + aNumber + " is "
            + Math.floor(aNumber));
        System.out.println("The rint of " + aNumber + " is "
            + Math.rint(aNumber));
    }
}
```

A program kimenetei:

```
The absolute value of -191.635 is 191.635
The ceiling of -191.635 is -191
The floor of -191.635 is -192
The rint of -191.635 is -192
```

További két alapvető metódus található a *Math* osztályban. Ezek a *min* és a *max*. Az alábbi táblázat mutatja be a *min* és *max* metódusok különböző formáit, amik két számot hasonlítanak össze, és a megfelelő típusú értékkel térnek vissza.

double min(double, double) *float min(float, float)* *int min(int, int)* *long min(long, long)*	A két paraméterből a kisebbel térnek vissza.
double max(double, double) *float max(float, float)* *int max(int, int)* *long max(long, long)*	A két paraméterből a nagyobbal térnek vissza.

A *MinDemo* program mutatja be a *min* alkalmazását két érték közül a kisebb kiszámítására:

```
public class MinDemo {
    public static void main(String[] args) {
        double enrollmentPrice = 45.875;
        double closingPrice = 54.375;
        System.out.println("A vételár: $"
            + Math.min(enrollmentPrice, closingPrice));
    }
}
```

A program valóban az alacsonyabb árat adta vissza:

```
A vételár: $45.875
```

A *Math* osztály következő metódusai a hatványozással kapcsolatosak. Ezen kívül megkaphatjuk az **e** értékét a *Math.E* használatával.

double exp(double)	A szám exponenciális értékével tér vissza.
double log(double)	A szám természetes alapú logaritmusával tér vissza.
double pow(double, double)	Az első paramétert a második paraméternek megfelelő hatványra emeli.
double sqrt(double)	A paraméter négyzetgyökével tér vissza.

A következő *ExponentialDemo* program kiírja az *e* értékét, majd meghívja egyenként a fenti táblázatban látható metódusokat egy számra:

```
public class ExponentialDemo {
    public static void main(String[] args) {
        double x = 11.635;
        double y = 2.76;
```

```
        System.out.println("Az e értéke: " +
                           Math.E);
        System.out.println("exp(" + x + ") is " +
                           Math.exp(x));
        System.out.println("log(" + x + ") is " +
                           Math.log(x));
        System.out.println("pow(" + x + ", " + y + ") is " +
                           Math.pow(x, y));
        System.out.println("sqrt(" + x + ") is " +
                           Math.sqrt(x));
    }
}
```

Az *ExponentialDemo* kimenete:

```
Az e értéke: 2.71828
exp(11.635) is 112984
log(11.635) is 2.45402
pow(11.635, 2.76) is 874.008
sqrt(11.635) is 3.41101
```

A *Math* osztály egy sor trigonometrikus függvényt is kínál, ezek a következő táblázatban vannak összefoglalva. A metóduson áthaladó szögek radiánban értendők. Radiánból fokká, és onnan visszakonvertálásra két függvény áll rendelkezésünkre: *toDegrees* és *toRadians*. Előbbi fokká, utóbbi radiánná konvertál. A *Math.PI* függvény meghívásával a PI értékét kapjuk meg a lehető legpontosabban.

double sin(double)	Egy *double* szám szinuszával tér vissza.
double cos(double)	Egy *double* szám koszinuszával tér vissza.
double tan(double)	Egy *double* szám tangensével tér vissza.
double asin(double)	Egy *double* szám arc szinuszával tér vissza.
double acos(double)	Egy *double* szám arc koszinuszával tér vissza.
double atan(double)	Egy *double* szám arc tangensével tér vissza.
double atan2(double)	Derékszögű koordinátákat konvertál (b, a) polárissá (r, théta).
double toDegrees(double) *double toRadians(double)*	A paramétert radiánná vagy fokká konvertálják, a függvények adják magukat.

A következő *TrigonometricDemo* program használja a fenti táblázatban bemutatott összes metódust, hogy különböző trigonometrikus értékeket számoljon ki a 45 fokos szögre:

```
public class TrigonometricDemo {
    public static void main(String[] args) {
        double degrees = 45.0;
        double radians = Math.toRadians(degrees);
```

```
        System.out.println("The value of pi is " +
                           Math.PI);
        System.out.println("The sine of " + degrees +
                           " is " + Math.sin(radians));
        System.out.println("The cosine of " + degrees +
                           " is " + Math.cos(radians));
        System.out.println("The tangent of " + degrees +
                           " is " + Math.tan(radians));
    }
}
```

A program kimenetei:

```
The value of pi is 3.141592653589793
The sine of 45.0 is 0.8060754911159176
The cosine of 45.0 is -0.5918127259718502
The tangent of 45.0 is -1.3620448762608377
```

Megjegyezzük, hogy a *NaN* akkor jelenik meg, amikor az eredmény matematikailag nem definiált. A *Double* és a *Float* osztályok is tartalmazzák a *NaN* konstanst. Összehasonlítva a visszatérési értéket az egyik ilyen konstanssal, a programunk el tudja dönteni, hogy a visszaadott érték érvényes-e. Ilyen módon a programunk elfogadhatóan tud reagálni, ha egy metódus nem definiált értéket kap.

Az utolsó *Math* metódus, amiről szót ejtünk, a *random*. A metódus egy kvázi-véletlen 0.0 és 1.0 közé eső számmal tér vissza. Pontosabban leírva:

$0.0 \leq Math.random() < 1.0$

Hogy más intervallumban kapjunk meg számokat, műveleteket hajthatunk végre a függvény által visszaadott értéken. Például, ha egy egész számot szeretnénk kapni 1 és 10 között, akkor a következőt kell begépelnünk:

```
int number = (int)(Math.random() * 10 + 1);
```

Megszorozva ezt az értéket 10-el a lehetséges értékek intervalluma megváltozik:

$0.0 \leq szám < 10.0.$

1-et hozzáadva az intervallum ismét megváltozik:

$1.0 \leq szám < 11.0.$

Végül, az érték egésszé konvertálásával egy konkrét számot (int) kapunk 1 és 10 között.

A *Math.random* használata tökéletes, ha egy egyszerű számot kell generálni. Ha egy véletlen számsorozatot kell generálni, akkor egy hivatkozást kell létrehozni a *java.util.Random*–ra, és meghívni ennek az objektumnak a különböző metódusait.

9.6. Ellenőrző kérdések

- Mi a Number osztály szerepe az osztályhierarchiában?
- Hogyan tudunk egy begépelt szöveget (*String*) *int* értékként megkapni?
- Hogyan tudunk egy egész számot (*int*) szöveggé (*String*) konvertálni?
- Hogyan tudunk Javában összetett matematikai műveletek (szinusz, exponenciális) végrehajtani?

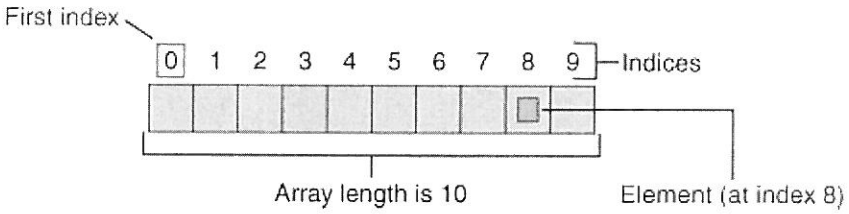
Mit ír ki a következő kódrészlet?

```
Integer ten = new Integer(10);
Long nine = new Long (9);
System.out.println(ten + nine);
int i = 1;
System.out.println(i + ten);
```

- 19, majd 20
- 19, majd 11
- 10, majd 1
- fordítási hiba miatt nem indul el

10.Tömbök

A tömb egy olyan változó, amely több azonos típusú adatot tartalmaz. A tömb (futásidei) hossza a létrehozásakor kerül megállapításra, és attól kezdve a tömb egy állandó méretű adatszerkezet.



A tömb egy eleme egyike a tömbben található tagoknak, mely a tömbben elfoglalt helye (indexe) alapján érhető el.

Ha különböző típusú adatokat akarunk tárolni egy szerkezeten belül, vagy olyan szerkezetre van szükség, melynek mérete dinamikusan módosítható, akkor használjunk a tömb helyett olyan gyűjtemény implementációkat, mint az *ArrayList*.

Fontos megjegyezni, hogy a tömbök objektumok, eltérően a C nyelvtől. Ez sok mindenben befolyásolja a működését.

10.1. Tömbök létrehozása és használata

Következzen az *ArrayDemo* nevű program, mely létrehoz egy tömböt, adatokkal tölti fel, majd kiírja tartalmát:

```
public class ArrayDemo {
    public static void main(String[] args) {
        int[] anArray;
        anArray = new int[10];

        for (int i = 0; i < anArray.length; i++) {
            anArray[i] = i;
            System.out.print(anArray[i] + " ");
        }

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

A program kimenete:

```
0 1 2 3 4 5 6 7 8 9
```

Tömbök deklarálása

Ez a sor egy példaprogramból való, és egy tömb típusú változót deklarál:

```
int[] anArray;
```

Mint minden másfajta változó deklarálásakor, a tömb deklarálása is két részből áll: a tömb típusa és a tömb neve. A tömb típusa a *tömb[]* formátumban írható, ahol a típus a tömb által tartalmazott elemek típusa, a [] pedig azt jelzi, hogy tömbről van szó. A tömb minden eleme azonos típusú! A fenti példa *int[]* tömböt használ, tehát az *anArray* nevű tömb *int* típusú egészek tárolására lesz alkalmas. Néhány más típust tárolni képes tömb létrehozása:

```
float[] anArrayOfFloats;
boolean[] anArrayOfBooleans;
Object[] anArrayOfObjects;
String[] anArrayOfStrings;
```

Így is írható a deklaráció:

```
float anArrayOfFloats[];
```

Ez a forma nem ajánlott, mert a zárójelek jelzik, hogy tömbről van szó, így azok a típussal tartoznak össze, nem pedig a tömb nevével.

A tömb változók deklarálásával – mint bármely más nem primitív változóéval – sem jön létre tényleges tömb, és nem foglal le helyet a memóriában az elemek számára. A példakódban explicit módon kell létrehozni a tömböt és elnevezni *anArray*-nek.

Tömbök létrehozása

Tömb létrehozható explicit módon a Java *new* operátora segítségével. A példaprogram következő részében 10 egész tárolására szolgáló tömbhöz elegendő memóriát foglalunk le, és elnevezzük a már korábban deklarált *anArray*-nek.

```
anArray = new int[10];
```

Általában tömb létrehozásakor használni kell a *new* operátort, majd a tömb elemeinek típusát és végül a tömb méretét kell megadni szögletes zárójelek között:

```
new elemtípus[tömbméret];
```

Ha a *new* operátort kihagytuk volna a példaprogramból, a fordító leállt volna következő hibaüzenettel:

```
ArrayDemo.java:4: Variable anArray may not have been
initialized.
```

Tömb kezdőérték beállítása

Használható a tömbök létrehozására és inicializálására a következő rövid formula:

```
boolean[] answers = {true, false, true, true, false};
```

Ilyekor a tömb nagyságát a {} közé írt elemek száma határozza meg.

Tömbelemek elérése

Most, hogy már megtörtént a memóriafoglalás a tömb számára, a program értékeket rendel a tömb elemeihez.

```
for (int i = 0; i < anArray.length; i++) {
    anArray[i] = i;
    System.out.print(anArray[i] + " ");
}
```

A kód ezen része azt mutatja be, hogy ahhoz, hogy hivatkozni tudjuk a tömb bármely elemére beírás vagy kiolvasás céljából, a tömb nevéhez egy []-et kell írni. A zárójelben (változó vagy egyéb kifejezés segítségével) írt érték az elérni kívánt tömbelem indexét jelöli.

Megjegyzés: A tömb (mivel objektum), tudja, hogy mekkora a mérete, és milyen index használható az indexelés során. Érvénytelen index esetén (a C nyelvvel szemben) a hiba futás közben egyértelműen kiderül: a futtatókörnyezetet egy *ArrayIndexOutOfBoundsExeption* típusú kivételt dob.

Tömb méretének meghatározása

A tömb méretének meghatározásához használható:

```
Tömbnév.length;
```

Figyelem! Azok, akiknek új a Java programnyelv, gyakran üres zárójelet raknak a *length* után. Ez nem működik, mert a *length* nem metódus! A *length* egy csak olvasható adattag, melyet a Java platform nyújt minden tömb számára.

A *for* ciklus a programunkban bejárja az *anArray* minden elemét, értéket adva nekik. A ciklus *anArray.length*–et használ a ciklus végének megállapításához.

10.2. Objektum tömbök

A tömbök tárolhatnak referencia típusú elemeket a primitív típusokhoz hasonlóan. Ezeket a tömböket is nagyrészt ugyanúgy kell létrehozni, mint a primitív típust tárolókat. A következő *ArrayOfStringsDemo* nevű program három *String* objektumot tárol, és kiírja őket kisbetűsítve.

```
public class ArrayOfStringsDemo {
    public static void main(String[] args) {
        String[] anArray = { "String One",
                             "String Two",
                             "String Three" };

        for (int i = 0; i < anArray.length; i++) {
            System.out.println(anArray[i].toLowerCase());
        }
    }
}
```

A program kimenete:

```
string one
string two
string three
```

A JDK 5.0–ás és későbbi verziói esetén lehetőség van a tömb elemeinek bejárásához egy újabb szintaktika alkalmazására. Ezt – más nyelvekben használt nevük alapján – *for-each* ciklusnak is szokás hívni. Figyelni kell azonban arra, hogy a ciklust ugyanúgy a *for* kulcsszó vezeti be, mint a hagyományos *for* ciklust.

```
String[] anArray = {"String One","String Two","String Three"};
for (String s : anArray) {
    System.out.println(s.toLowerCase());
}
```

A következő *ArrayOfIntegersDemo* nevű program feltölti a tömböt Integer objektumokkal.

```
public class ArrayOfIntegersDemo {
    public static void main(String[] args) {
        Integer[] anArray = new Integer[10];

        for (int i = 0; i < anArray.length; i++) {
            anArray[i] = new Integer(i);
            System.out.println(anArray[i]);
        }
    }
}
```

A program kimenete:

```
0
1
2
3
4
```

A következő programrészlet létrehoz egy tömböt, de nem rak bele semmit:

```
Integer[] anArray = new Integer[5];
```

Ez egy potenciális hibázási lehetőség, melyet az újdonsült Java programozók gyakran elkövetnek, amikor objektum tömböket használnak. Miután a fenti kódsor végrehajtódik, a tömb létrejött és képes 5 *Integer* objektum tárolására, bár a tömbnek még nincs eleme. Üres. A programnak explicit módon létre kell hoznia az objektumokat, és belerakni a tömbbe. Ez nyilvánvalónak tűnik, habár sok kezdő azt gondolja, fenti kódrészlet létrehozza az üres tömböt és még 5 üres objektumot is a tömbbe. Ha a tömbelemek létrehozása nélkül próbálunk azokra hivatkozni, akkor a futtatókörnyezetet *NullPointerException*-t fog dobni.

A probléma előfordulása még veszélyesebb, ha a tömb létrehozása a konstruktorban, vagy más kezdőérték állítással történik, és máshol használjuk a programban.

10.3. Tömbök tömbjei

Tömbök tartalmazhatnak tömböket. A következő példaprogram létrehoz egy tömböt, és kezdeti értékadásnál négy másodlagos tömböt, használ:

```
public class ArrayOfArraysDemo {
    public static void main(String[] args) {
        String[][] cartoons =
        {
            { "Flintstones", "Fred", "Wilma",
              "Pebbles", "Dino" },
            { "Rubbles", "Barney", "Betty",
               "Bam Bam" },
            { "Jetsons", "George", "Jane",
              "Elroy", "Judy", "Rosie", "Astro" },
            { "Scooby Doo Gang", "Scooby Doo",
              "Shaggy", "Velma", "Fred", "Daphne" }
        };

        for (int i = 0; i < cartoons.length; i++) {
            System.out.print(cartoons[i][0] + ": ");
            for (int j = 1; j < cartoons[i].length; j++) {
                System.out.print(cartoons[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

A program kimenetele:

```
Flintstones: Fred Wilma Pebbles Dino
Rubbles: Barney Betty Bam Bam
Jetsons: George Jane Elroy Judy Rosie Astro
Scooby Doo Gang: Scooby Doo Shaggy Velma Fred Daphne
```

Vegyük észre, hogy mindegyik másodlagos tömb különböző hosszúságú. A melléktömbök nevei *cartoons[0]*, *cartoons[1]*, és így tovább.

Mint az objektumok tömbjeinél, létre kell hoznunk a másodlagos tömböket a tömbön belül. Ha nem használunk kezdeti paraméterinicializálást, akkor a következőhöz hasonló kódot kell írnunk:

```
public class ArrayOfArraysDemo2 {
    public static void main(String[] args) {
        int[][] aMatrix = new int[4][];

        for (int i = 0; i < aMatrix.length; i++) {
            aMatrix[i] = new int[5];
            for (int j = 0; j < aMatrix[i].length; j++) {
                aMatrix[i][j] = i + j;
            }
        }

        for (int i = 0; i < aMatrix.length; i++) {
            for (int j = 0; j < aMatrix[i].length; j++) {
                System.out.print(aMatrix[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

A program kimenetele:

```
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
```

Meg kell adni a tömb hosszúságát, amikor létrehozzuk. Tehát egy tömbnek, ami tartalmaz másodlagos tömböket, meg kell adni a hosszúságát, amikor létrehozzuk, de nem kell megadni a másodlagos tömbök hosszúságát is.

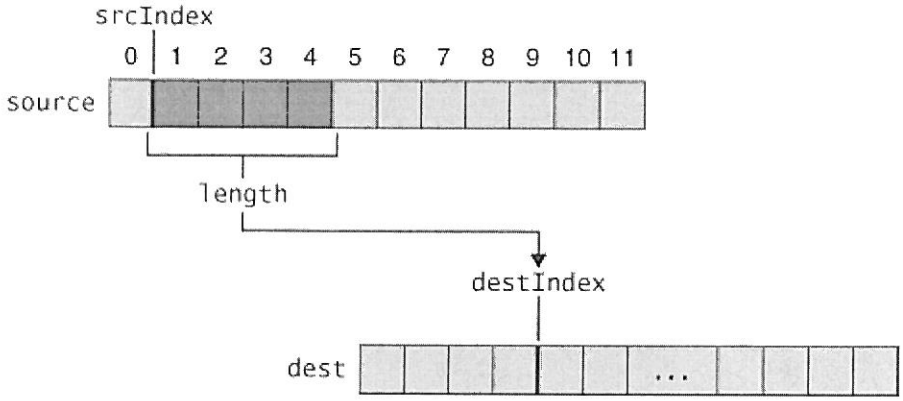
10.4. Tömbök másolása

Használhatjuk a *System* osztály *arraycopy* metódust, hogy adatokat másoljunk hatékonyan egyik tömbből a másikba. Az *arraycopy* metódus öt paramétert vár:

```
public static void arraycopy(Object source,
        int srcIndex, Object dest, int destIndex, int length)
```

A két *Object* paraméter rámutat a kiinduló és a cél tömbre. A három *int* paraméter jelzi a kezdő helyet a forrás és a céltömbön belül, és az elemek számát, amennyit másolni akarunk.

A következő kép illusztrálja, hogyan megy végbe a másolás:



A következő *ArrayCopyDemo* program használja az *arraycopy* metódust, ami az elemeket a *copyFrom* tömbből a *copyTo* tömbbe másolja.

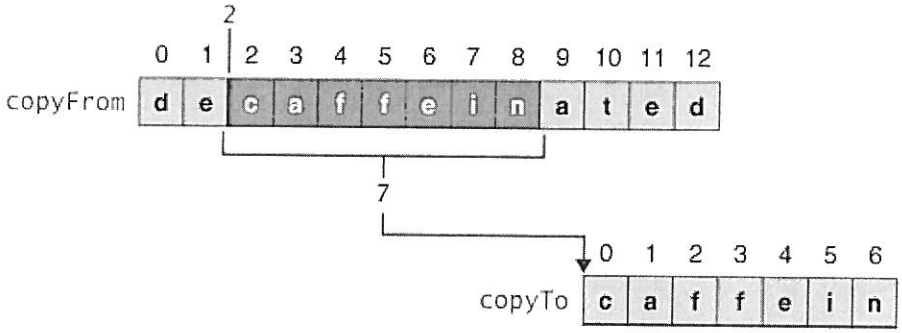
```
public class ArrayCopyDemo {
    public static void main(String[] args) {
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                'i', 'n', 'a', 't', 'e', 'd' };
        char[] copyTo = new char[7];

        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        System.out.println(new String(copyTo));
    }
}
```

A program kimenetele:

```
caffein
```

A következő képen lehet látni az *arraycopy* metódus működését:



Az eredménytömböt létre kell hozni, mielőtt meghívódik az *arraycopy* metódus, és elég nagyméretűnek kell lennie, hogy beleférjenek a másolandó tömb elemei.

Tömbkezeléshez további szolgáltatásokat nyújt a *java.util.Arrays* osztály is.

10.5. Ellenőrző kérdések

- Mi a tömb?
- Mi lehet Javában egy tömb elemtípusa?
- Mit jelent az, hogy Javában a tömböt tömbreferenciával érhetjük el?
- Mit tárol a tömb *length* példányváltozója?
- Mit jelent, hogy egy tömb objektum-referenciákat tárol?

- Mit jelent a többdimenziós tömb?

Melyik fordul le?

- *String temp [] = new String {"j" "a" "z"};*
- *String temp [] = { "j " " b" "c"};*
- *String temp = {"a", "b", "c"};*
- *String temp [] = {"a", "b", "c"};*

Hogyan tudhatjuk meg a *myarray* tömb méretét?

- *myarray.length();*
- *myarray.length;*
- *myarray.size*
- *myarray.size();*

10.6. Gyakorló feladatok

Írjon programot, amely egy egész számokat tartalmazó tömb elemeit permutálja (cserélgeti):

- fordítsa meg a tömböt
- kettesével cserélje meg az elemeket

Írjon programot, amely gyümölcsnevek tömbjét hozza létre!

Ezután keresse meg és írja ki, melyik gyümölcs nevében van a legtöbb magánhangzó.

Ötlet: érdemes egy azonos méretű másik tömböt létrehozni, amiben a darabszámokat tároljuk.

Írjon programot, amely a következő tartalmú mátrixot hozza létre, majd ki is írja azt a képernyőre:

```
2 3 4 1
1 1 2 1
2 2 1 2
2 3 1 1
```

Ezután transzponálja (a főátlóra tükrözze) a mátrixot, és így is írja ki. (Segítségként: a következő eredményt kell a képernyőn kapni:)

```
2 1 2 2
3 1 2 3
4 2 1 1
1 1 2 1
```

Megjegyzés: Az ideális megoldás a helyben tükrözés, de ha ez nem megy, lehet próbálkozni azzal is, hogy a transzponált egy másik tömbben jöjjön létre.

Írjon programot, amely az 5x5-ös egységmátrixot hozza létre!

Az egységmátrixban főátlóbeli elemek 1-et, míg az ezen kívüli elemek 0-t tartalmaznak.

Írjon programot, amely egy „háromszögmátrixot" reprezentál!

A főátló elemei legyenek 2-esek, a főátló alatti elemek 1-esek, a főátló feletti elemek pedig ne szerepeljenek a mátrixban (tekinthetjük nulláknak is).

Írja ki a képernyőre is ilyen formában.

11. Osztályok létrehozása

Ez a fejezet az osztályok fő alkotóelemeit mutatja be.

Az osztály definíciója 2 fő alkotóelemből áll:

- az osztály deklarációból,
- és az osztály törzsből.

A következő *Bicycle* osztály példáján bemutatjuk az osztály elemeit.

```
public class Bicycle {
```

Az osztály deklaráció az osztály kódjának az első sora. Minimálisan az osztály deklaráció meghatározza az osztály nevét. Az osztálytörzs az osztály deklarációt követi, és kapcsos zárójelek között áll. Az osztály törzs tartalmazza mindazt a kódot, amely hozzájárul az osztályból létrehozott objektumok életciklusához: konstruktorok, új objektumok inicializálására, változó deklarációk, amelyek megadják az osztály és objektumának állapotát, és eljárásokat az osztály és objektumai viselkedésének meghatározására.

```
    private int cadence;
    private int gear;
    private int speed;
```

Az osztály három tagváltozót definiál az osztálytörzsön belül. A következő konsrtruktor a tagváltozók kezdőértékeinek beállítására biztosít lehetőséget.

```
    public Bicycle(int startCadence, int startSpeed,
                   int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }
```

Végül négy metódus teszi teljessé az osztályt:

```
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }
}
```

11.1. Osztályok deklarálása

Több alkalommal láthatta, hogy az osztálydefiníciók a következő formában állnak:

```
class MyClass {
    // tagváltozók, konstruktor és metódus deklarációk
}
```

A kód első sorát osztálydeklarációnak nevezzük. A megelőző osztálydeklaráció egy minimális osztálydeklaráció; csak azokat az elemeket tartalmazza, amelyek feltétlenül szükségesek. Néhány aspektusa ennek az osztálynak ugyan nincs specifikálva, alapértelmezettek. A legfontosabb az, hogy a *Myclass* osztály közvetlen ősosztálya az *Object* osztály. Több információ is megadható az osztályról, ideértve az ősosztálya nevét, azt, hogy implementál-e interfészt vagy nem, hogy lehetnek-e leszármazott osztályai és így tovább, mindezt az osztálydeklaráción belül.

A következő táblázat bemutatja az összes lehetséges elemet, mely előfordulhat egy osztálydeklarációban előfordulásuk szükséges sorrendjében.

public	(opcionális) az osztály nyilvánosan hozzáférhető
abstract	(opcionális) az osztályt nem lehet példányosítani
final	(opcionális) az osztály nem lehet őse más osztálynak
class NameOfClass	az osztály neve
extends Super	(opcionális) az osztály őse
implement Interfaces	(opcionális) az osztály által implementált interfészek
{ *osztálytörzs* *}*	az osztály működését biztosítja

11.2. Tagváltozók deklarálása

A *Bicycle* a következő kód szerint definiálja tagváltozóit:

```
private int cadence;
private int gear;
private int speed;
```

Ez a kód deklarálja a tagváltozókat, de más változókat, mint például a lokális változókat nem. A tagváltozók deklarációja az osztálytörzsben, bármilyen konstruktoron és eljáráson kívül történik meg. Az itt deklarált tagváltozók *int* típusúak. Természetesen a deklarációk komplexebbek is lehetnek.

A *private* kulcsszó mint privát tagokat vezeti be a tagokat. Ez azt jelenti, hogy csak a *Bicycle* osztály tagjai férhetnek hozzájuk.

Nem csak típust, nevet és hozzáférési szintet lehet meghatározni, hanem más attribútumokat is, ideértve azt, hogy a változó-e, vagy konstans. A következő táblázat tartalmazza az összes lehetséges tagváltozó deklarációs elemet.

hozzáférési szint	(opcionális) A következő négy hozzáférési szint szerint vezérelhető, hogy más osztályok hogyan férhessenek hozzá a tagváltozókhoz: *public*, *protected*, csomag szintű, vagy *private*.
static	(opcionális) Osztályváltozót, és nem példányváltozót deklarál.
final	(opcionális) a változó értéke végleges, meg nem változtatható (konstans) Fordítási idejű hibát okoz, ha a program megpróbál megváltoztatni egy *final* változót. Szokás szerint a konstans értékek nevei nagybetűvel íródnak. A következő változó deklaráció a PI változót határozza meg, melynek értéke π (.3.141592653589793), és nem lehet megváltoztatni: *final double PI = 3.141592653589793;*
transient	(opcionális) a változót átmenetiként azonosítja
volatile	(opcionális) Megakadályozza, hogy a fordító végrehajtson bizonyos optimalizálásokat a tagon.
típusnév	A változó típusa és neve Mint más változóknak, a tagváltozóknak is szükséges, hogy típusa legyen. Használhatók egyszerű típusnevek, mint például az *int*, *float* vagy *boolean*. Továbbá használhatók referencia típusok, mint például a tömb, objektum vagy interfész nevek. Egy tagváltozó neve lehet minden megengedett azonosító, mely szokás szerint kisbetűvel kezdődik. Két vagy több tagváltozónak nem lehet megegyező neve egy osztályon belül.

11.3. Metódusok deklarálása

A következő példa a *setGear* metódus, amely a sebességváltást teszi lehetővé:

```
public void setGear(int newValue) {
    gear = newValue;
}
```

Mint az osztályt, a metódust is két nagyobb rész határoz meg: a metódus deklarációja és a metódus törzse. A metódus deklaráció meghatározza az összes metódus tulajdonságát úgy, mint az elérési szint, visszatérő típus, név, és paraméterek. A metódus törzs az a rész, ahol minden művelet helyet foglal. Olyan instrukciókat tartalmaz, amelyre a metódus végrehajtásához van szükség.

A metódus deklaráció kötelező elemei: a metódus neve, visszatérő típusa, és egy zárójelpár: (). A következő táblázat megmutat minden lehetséges részt a metódus deklarációjában.

hozzáférési szint	(opcionális) A metódus hozzáférési szintje
static	(opcionális) Osztály metódust deklarál
abstract	(opcionális) Jelzi, hogy a metódusnak nincs törzse
final	(opcionális) Jelzi, hogy a metódus nem írható felül a leszármazottakban
native	(opcionális) Jelzi, hogy a metódust más nyelven készült
synchronized	(opcionális) A metódus kér egy monitort a szinkronizált futáshoz
returnType methodName	Az metódus visszatérő típusa és neve
(paramList)	A paraméterlista a metódushoz
throws exceptions	(opcionális) A metódus le nem kezelt kivételei

A metódus neve

A metódus neve szokás szerint kis betűvel kezdődik, hasonlóan a változók neveihez.

Általában az osztályon belül egyedi neve van a metódusnak.

Ha a metódus neve, paraméterlistája és visszatérési értéke megegyezik az ősében definiált metódussal, akkor felülírja azt.

Javában az is megengedett, hogy ugyanazzal a névvel, de különböző paraméterlistával hozzunk létre metódusokat.

Nézzük a következő példát:

```
public class DataArtist {
    ...
    public void draw(String s) {
        ...
    }
    public void draw(int i) {
        ...
    }
    public void draw(float f) {
        ...
    }
}
```

Azonos paraméterlistával, de különböző típusú visszatérési értékkel nem lehet metódusokat létrehozni.

11.4. Konstruktorok

Minden osztályban van legalább egy konstruktor. A konstruktor inicializálja az új objektumot. A neve ugyanaz kell, hogy legyen, mint az osztályé. Például a *Bicycle* nevű egyszerű osztálynak a konstruktora is *Bicycle*:

```
public Bicycle(int startCadence, int startSpeed,
               int startGear) {
    gear = startGear;
    cadence = startCadence;
    speed = startSpeed;
}
```

Ebben a példában a konstruktor a paraméter alapján tudja létrehozni a kívánt méretű tömböt.

A konstruktor nem metódus, így nincs visszatérő típusa. A konstruktor a *new* operátor hatására hívódik meg, majd visszaadja a létrejött objektumot. Egy másik konstruktor csupán a 0 kezdőértékeket állítja be:

```
public Bicycle() {
    gear = 0;
    cadence = 0;
    speed = 0;
}
```

Mindkét konstruktornak ugyanaz a neve (*Bicycle*), de a paraméterlistájuk különböző. A metódusokhoz hasonlóan a konstruktorokat is megkülönbözteti a Java platform a paraméterek száma és típusa alapján. Ezért nem írhatunk két ugyanolyan paraméterlistával rendelkező konstruktort. (Különben a fordító nem lenne képes őket megkülönböztetni, így fordítási hibát adna.)

Amikor létrehozunk egy osztályt, meg kell adnunk, hogy az egyes példányok milyen konstruktorokkal legyen létrehozhatók. A korábban bemutatott *Rectangle* osztály négy konstruktort tartalmaz, és így lehetővé teszi a különböző inicializálási lehetőségek közti választást.

Nem kell konstruktorokat írni az osztályainkhoz, ha úgy is el tudja látni a feladatát. A rendszer automatikusan létrehoz egy paraméter nélküli konstruktort, különben nem tudnánk példányt létrehozni.

A konstruktor deklarációjánál használhatjuk a következő hozzáférési szinteket:

- **private**
  Csak ez az osztály használhatja ezt a konstruktort. Ha minden konstruktorok privát, akkor az osztályban lehet egy publikus osztály metódus, mely létrehoz és inicializál egy példányt.
- **protected**
  Az osztály és leszármazott osztályai használhatják ezt a konstruktort.
- **public**
  Minden osztály használhatja ezt a konstruktort.
- **nincs megadva**
  Csak az osztállyal azonos csomagban elhelyezkedő osztályokból lesz elérhető ez a konstruktor.

11.5. Információátadás metódus vagy konstruktor számára

A metódus vagy konstruktor deklaráció megmutatja a paraméterek számát és típusát. Például a következő metódus kiszámolja a kölcsön havi törlesztő részleteit:

```
public double computePayment(double loanAmt, double rate,
                             double futureValue,
                             int numPeriods) {
    double I, partial1, denominator, answer;
    I = rate / 100.0;
    partial1 = Math.pow((1 + I), (0.0 - numPeriods));
    denominator = (1 - partial1) / I;
    answer = ((-1 * loanAmt) / denominator)
              - ((futureValue * partial1) / denominator);
    return answer;
}
```

Ebben a metódusnak négy paramétere van: a kölcsön összege, kamata, végösszege, fizetés gyakorisága. Az első három dupla pontosságú lebegő pontos szám, míg a negyedik egész szám.

Ahogy ennél a metódusnál is látható, a metódus vagy konstruktor paraméterlistája változó deklarációk vesszővel elválasztott listája, ahol minden változó deklaráció típus-név párokkal van megadva. Amint láthatjuk a *computePayment* metódus törzsében a paraméternevekkel egyszerűen hivatkozunk az értékükre.

Paraméterek típusa

A metódusok vagy konstruktorok paraméterei típussal rendelkeznek. A típus lehet primitív (*int*, *float*, stb.), amint láthattuk a *computePayment* metódusnál, vagy referencia típusú (osztályok, tömbök esetén). A következő metódus egy tömböt lap paraméterként, majd létrehoz egy új *Polygon* objektumot, és inicializálja a *Point* objektumok tömbjéből. (Feltételezzük, hogy a *Point* egy osztály, amelyik *x*, *y* koordinátákat tartalmaz.)

```
public static Polygon polygonFrom (Point[] listOfPoints) {
    ...
}
```

A Java nyelv nem engedi, hogy metódust egy másik metóduson belül helyezzünk el.

Paraméter nevek

Amikor metódusban vagy konstruktorban paramétert deklarálunk, eldönthetjük, hogy milyen nevet adunk a paraméternek. Ezt a nevet használhatjuk a metódus törzsében a paraméter értékek elérésére.

A paraméter nevének egyedinek kell lennie a hatáskörén belül. Vagyis nem lehet ugyanaz, mint egy másik paraméter, vagy lokális változó vagy bármely paraméter neve egy *catch* záradékban ugyanazon a metóduson vagy konstruktoron belül. Lehet viszont ugyanaz a név, mint az osztály egy tagváltozója. Ebben az esetben, azt mondjuk, hogy a paraméter elfedi a tagváltozót. Ilyen elfedett tagváltozót is el lehet érni, bár kicsit körülményes. Például nézzük meg a következő *Circle* osztályt és a *setOrigin* metódust:

```
public class Circle {
    private int x, y, radius;
    public void setOrigin(int x, int y) {
        ...
    }
}
```

A *Circle* osztálynak három tagváltozója van: *x*, *y* és *radius*. A *setOrigin* metódus két paramétert vár, mindkettőnek ugyanaz a neve, mint a tagváltozónak. A metódus mindkét paramétere elrejti az azonos nevű tagváltozót. Ha a metódus törzsében használjuk az *x*

vagy *y* nevet, akkor a paraméterekre és nem a tagváltozókra hivatkozunk. A tagváltozó eléréséhez minősített nevet kell használni, amiről később olvashat.

Paraméter-átadás érték szerint

A paraméter-átadás érték szerint történik. Amikor meghívunk egy metódust vagy egy konstruktort, akkor a metódus megkapja az érték másolatát. Amikor a paraméter referencia típusú, akkor a referencián keresztül ugyanazt az objektumot érhetjük el, mivel csak a referencia értéke másolódott át: meghívhatjuk az objektumok metódusait és módosíthatjuk az objektum publikus változóit.

Nézzük meg a következő *getRGBColor* metódust, amelyik megkísérli visszaadni a paraméterein keresztül a tagváltozói értékeit:

```
public class Pen {
    private int redValue, greenValue, blueValue;
    ...
    public void getRGBColor(int red, int green, int blue) {
        red = redValue;
        green = greenValue;
        blue = blueValue;
    }
}
```

Ez így nem működik. A *red*, *green* és *blue* változó létezik ugyan, de a hatásköre a *getRGBColor* metóduson belül van. Amikor a metódusból visszatér a program, ezek a változók megszűnnek.

Írjuk újra a *getRGBColor* metódust, hogy az történjen, amit szerettünk volna. Először is, kell egy új *RGBColor* típus, hogy megőrizze a *red*, *green* és *blue* színek értékét:

```
public class RGBColor {
    public int red, green, blue;
}
```

Most már átírhatjuk a *getRGBColor* metódust, hogy paraméterként *RGBColor* objektumot várjon. A *getRGBColor* metódus visszatér az aktuális toll színével a beállított *red*, *green* és *blue* tagváltozó értékével az *RGBColor* paraméter segítségével:

```
public class Pen {
    private int redValue, greenValue, blueValue;
    ...
    public void getRGBColor(RGBColor aColor) {
        aColor.red = redValue;
        aColor.green = greenValue;
        aColor.blue = blueValue;
    }
}
```

Az így visszaadott *RBColor* objektum a *getRGBColor* metóduson kívül is megtartja értékét, mivel az *aColor* egy objektumra hivatkozik, amely a metódus hatáskörén kívül létezik.

Megjegyzés: Természetesen az is egy – bizonyos értelemben egyszerűbb – megoldás lehetett volna, hogy a három érték lekérdezéséhez három lekérdező metódust készítünk. Így nem lett volna szükség egy új típus bevezetésére. E megoldás a következő pont elolvasása után el is készíthető.

11.6. A metódusok visszatérési értéke

A metódusok visszatérési értékének típusa a metódus deklarációjakor adható meg. A metóduson belül a *return* utasítással lehet a visszaadott értéket előállítani. A *void*-ként deklarált metódusok nem adnak vissza értéket, és nem kell, hogy tartalmazzanak *return* utasítást. Minden olyan metódus, amely nem *void*-ként lett deklarálva, kötelezően tartalmaz *return* utasítást. Sőt a fordító azt is kikényszeríti, hogy minden lehetséges végrehajtási ág *return* utasítással végződjön.

Tekintsük a következő *isEmpty* metódust a *Stack* osztályból:

```
public boolean isEmpty() {
    if (top == 0) {
        return true;
    } else {
        return false;
    }
}
```

A visszaadott érték adattípusa meg kell, hogy egyezzen a metódus deklarált visszatérési értékével; ezért nem lehetséges egész számot visszaadni olyan metódusból, aminek logikai a visszatérési értéktípusa. A fenti *isEmpty* metódus deklarált visszatérési érték típusa logikai (*boolean*), és a metódus megvalósítása szerint igaz (*true*), vagy hamis (*false*) logikai érték kerül visszaadásra a teszt eredményének megfelelően.

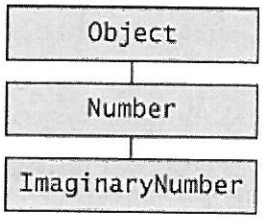
Megjegyzés: A fenti kód helyett általában tömörebb írásmódot szokás alkalmazni. Ennek ellenére ebben a jegyzetben a jobb érthetőség kedvéért időnként a hosszabb formát alkalmazzuk. A következő kód az előzővel teljesen ekvivalens:

```
public boolean isEmpty() {
    return top == 0;
}
```

Az *isEmpty* metódus elemi (vagy primitív) típust ad vissza. Egy metódus referencia adattípust is visszaadhat. Például ha a *Stack* osztály definiálja a *pop* metódust, amely az *Object* referencia adattípust adja vissza:

```
public Object pop() {
    if (top == 0) {
        throw new EmptyStackException();
    }
    Object obj = items[--top];
    items[top] = null;
    return obj;
}
```

Amikor egy metódus visszatérési típusa egy osztály, (mint ahogy a fenti *pop* esetén), a visszaadott objektum típusa meg kell, hogy egyezzen a visszatérési típussal, vagy leszármazottja kell, hogy legyen annak. Tegyük fel, hogy van egy olyan osztályhierarchia, amelyben *ImaginaryNumber* a *java.lang.Number* leszármazott osztálya, ami viszont az *Object* leszármazott osztálya. Ezt mutatja az alábbi ábra:



Ezek után tegyük fel, hogy egy metódus úgy kerül deklarálásra, hogy egy *Number*-t ad vissza:

```
public Number returnANumber() {
    ...
}
```

A *returnANumber* metódus visszaadhat egy *ImaginaryNumber* típust, de nem adhat vissza *Object* típust. *ImaginaryNumber* egy *Number*, mivel a *Number* leszármazott osztálya. Ugyanakkor az *Object* nem feltétlenül *Number* is egyben, – lehet akár *String*, vagy akár egész más típusú is.

Interfész neveket is lehet visszatérési típusként használni. Ebben az esetben a visszaadott objektumnak a megadott interfészt (vagy leszármazottját) kell implementálnia.

11.7. A *this* kulcsszó használata

A példa metóduson, vagy a konstruktoron belül a *this* az *aktuális objektumra* való hivatkozást (referenciát) jelenti – azaz arra az objektumra, amelynek a metódusa vagy a konstruktora meghívásra kerül. Az aktuális objektum bármelyik tagja hivatkozható egy példány metódusból, vagy egy konstruktorból a *this* használatával. A leggyakoribb használat oka az, hogy egy változó tag egy paraméter által kerül elfedésre a metódus vagy a konstruktor számára.

Például a következő konstruktor a *HSBColor* osztály számára inicializálja az objektum tagváltozóit a konstruktornak átadott paramétereknek megfelelően. A konstruktor mindegyik paramétere elfed egy-egy objektum tagváltozót, ezért az objektum tagváltozóira a konstruktorban a *this* megadással kell hivatkozzon:

```
public class HSBColor {
    private int hue, saturation, brightness;

    public HSBColor (int hue, int saturation, int brightness){
        this.hue = hue;
        this.saturation = saturation;
        this.brightness = brightness;
    }
}
```

A konstruktoron belül a *this* kulcsszó használható arra is, hogy egy ugyanabba az osztályba tartozó másik konstruktort meghívjunk. Ezt a metódust **explicit konstruktor hívás**nak nevezzük. Az alábbi *Rectangle* osztály másként kerül implementálásra, mint korábbi megoldás.

```
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this(0, 0, 0, 0);
    }

    public Rectangle(int width, int height) {
        this(0, 0, width, height);
    }
```

```
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    ...
}
```

Ez az osztály konstruktorok halmazát tartalmazza. Mindegyik konstruktor inicializálja a négyszög (*Rectangle*) tagváltozóinak egy részét, vagy az összeset. A konstruktorok konstans kezdőértéket adnak minden olyan tagváltozónak, amelynek nem ad értéket valamelyik paraméter. Például a paraméter nélküli konstruktor a négy paraméteres konstruktort hívja, és nullákat ad kezdőértékeknek.

Ahogy az eddigiekben is, a fordítóprogram a paraméterek száma és típusa alapján határozza meg, hogy melyik konstruktort kell meghívni.

Amennyiben használjuk, úgy az explicit konstruktorhívás a konstruktor első utasítása kell, hogy legyen.

11.8. Egy osztály tagjai elérhetőségének felügyelete

Egy elérési szint meghatározza, hogy lehetséges-e más osztályok számára használni egy adott tagváltozót, illetve meghívni egy adott metódust. A Java programozási nyelv négy elérési szintet biztosít a tagváltozók és a metódusok számára. Ezek a *private*, *protected*, *public*, és amennyiben nincsen jelezve, a csomag szintű elérhetőség. Az alábbi tábla az egyes szintek láthatóságát mutatja:

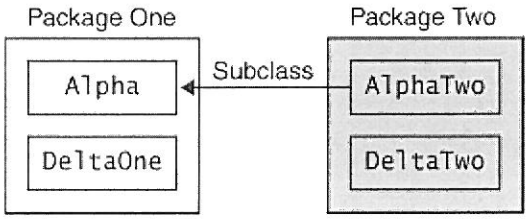
	osztály	csomag	leszármazott	összes
private	I	N	N	N
nincs	I	I	N	N
protected	I	I	I	N
public	I	I	I	I

Az első oszlop azt mutatja, hogy maga az osztály elérheti-e az adott jelzővel megjelölt tagokat. Ahogy az látható, az osztály mindig elérheti saját tagjait. A második oszlop azt mutatja, hogy az eredeti osztállyal azonos csomagban lévő más osztályok – függetlenül a szülői kapcsolatoktól – elérhetik-e a tagokat. Egy csomagban lévő csoportok osztályokkal és interfészekkel állnak kapcsolatban, továbbá elérési védelmet és tárterület felügyeletet biztosítanak. (A csomagokról egy későbbi fejezetben lesz szó.) A harmadik oszlop azt jelzi, hogy az osztály leszármazottai elérhetik-e a tagokat – függetlenül attól, hogy melyik csomagban vannak. A negyedik oszlop pedig azt jelzi, hogy az összes osztály elérheti-e a tagokat.

Az elérési szintek két módon hatnak. Az első mód az, amikor külső forrásból származó osztályokat (például a Java platform osztályait) használjuk, ekkor az elérési szintek meghatározzák, hogy ezen osztályok melyik tagjait tudjuk elérni. A másik mód az, hogy

amennyiben saját osztályokat írunk, meghatározhatjuk az elérési szintet minden tagváltozóhoz, metódushoz, illetve konstruktorhoz, amelyek az osztályban szerepelnek.

Tekintsük az osztályok egy halmazát, és nézzük, hogyan működnek az elérési szintek. A következő ábra az alábbi példában szereplő négy osztályt és a közöttük fennálló kapcsolatokat mutatja:



Osztály elérési szint

Nézzük az *Alpha* osztály listáját. Ennek tagjait más osztályok is el akarják érni. Az *Alpha* elérési szintenként egy tagváltozót és egy metódust tartalmaz. Az *Alpha* egy *One* nevű csomagban van:

```
package One;

public class Alpha {
    private   int iamprivate = 1;
              int iampackage = 2;  //csomag elérés
    protected int iamprotected = 3;
    public    int iampublic = 4;

    private void privateMethod() {
        System.out.println("iamprivate Method");
    }

    void packageMethod() { // csomag elérés
        System.out.println("iampackage Method");
    }

    protected void protectedMethod() {
        System.out.println("iamprotected Method");
    }

    public void publicMethod() {
        System.out.println("iampublic Method");
    }

    public static void main(String[] args) {
        Alpha a = new Alpha();
        a.privateMethod();    // megengedett
        a.packageMethod();    // megengedett
        a.protectedMethod(); // megengedett
        a.publicMethod();     // megengedett

        System.out.println("iamprivate: "
            + a.iamprivate);     // megengedett
        System.out.println("iampackage: "
            + a.iampackage);     // megengedett
        System.out.println("iamprotected: "
            + a.iamprotected"); // megengedett
        System.out.println("iampublic: "
            + a.iampublic);      // megengedett
    }
}
```

Ahogy az látható, az *Alpha* úgy hivatkozik valamennyi tagváltozójára és valamennyi metódusára, ahogy az az előző táblázat osztály oszlopában szerepelt. A program kimenete a következő lesz:

```
iamprivate Method
iampackage Method
iamprotected Method
iampublic Method
iamprivate: 1
iampackage: 2
iamprotected: 3
iampublic: 4
```

Egy tag elérési szintje azt határozza meg, hogy melyik osztályok érhetik el az illető tagot, és nem azt, hogy melyik példányok érhetik el. Például egy osztály valamennyi példánya elérheti egy másik publikus tagjait.

Az *Alpha* osztályhoz hozzávehetünk egy példány metódust, ami összehasonlítja az aktuális *Alpha* objektumot (*this*) egy másik objektummal az *iamprivate* változói alapján:

```
package One;

public class Alpha {
    ...
    public boolean isEqualTo(Alpha anotherAlpha) {
        if (this.iamprivate == anotherAlpha.iamprivate) {
        //megengedett
            return true;
        } else {
            return false;
        }
    }
}
```

Csomag elérési szint

Tekintsük a következő *DeltaOne* osztályt, amely az *Alpha*-val azonos csomagba tartozik. Az előző táblázat csomag oszlopa meghatározza azokat a változókat és metódusokat, amelyeket ez az osztály használhat.

```
package One;

public class DeltaOne {
    public static void main(String[] args) }
        Alpha a = new Alpha();

        //a.privateMethod();  // nem megengedett
        a.packageMethod();    // megengedett
        a.protectedMethod();  // megengedett
        a.publicMethod();     // megengedett
```

```
        //System.out.println("iamprivate: "
        //  + a.iamprivate);   // nem megengedett
        System.out.println("iampackage: "
            + a.iampackage);   // megengedett
        System.out.println("iamprotected: "
            + a.iamprotected); // megengedett
        System.out.println("iampublic: "
            + a.iampublic);    // megengedett
    }
}
```

A *DeltaOne* nem hivatkozhat az *iamprivate* változóra, és nem hívhatja meg a *privateMethod* metódust, ugyanakkor elérheti az *Alpha* többi tagját. Amennyiben a kommentezett sorok elől eltávolítjuk a // jelölést, és így próbáljuk meg lefordítani az osztályt, úgy a fordítóprogram hibát fog jelezni. A megjegyzésekkel futtatva a következő eredményt kapjuk:

```
iampackage Method
iamprotected Method
iampublic Method
iampackage: 2
iamprotected: 3
iampublic: 4
```

Leszármazott osztály elérési szint

A következő, *AlphaTwo* nevű osztály az *Alpha* leszármazott osztálya, de egy másik csomagban található. Az előző táblázat leszármazott oszlopa jelzi, hogy melyik tagváltozókat és metódusokat lehet használni:

```
package two;

import one.*;

public class AlphaTwo extends Alpha {
    public static void main(String[] args) {
        Alpha a = new Alpha();

        //a.privateMethod();   // nem megengedett
        //a.packageMethod();   // nem megengedett
        //a.protectedMethod(); // nem megengedett
        a.publicMethod()       // megengedett

        //System.out.println("iamprivate: "
        //   + a.iamprivate);    // nem megengedett
        //System.out.println("iampackage: "
        //   + a.iampackage);    // nem megengedett
        //System.out.println("iamprotected: "
        //   + a.iamprotected);  // nem megengedett
        System.out.println("iampublic "
            + a.iampublic);      // megengedett

        AlphaTwo a2 = new AlphaTwo();
        a2.protectedMethod();   // megengedett
        System.out.println("iamprotected: "
            + a2.iamprotected); // megengedett
    }
}
```

Vegyük észre, hogy *AlphaTwo* nem hívhatja a *protectedMethod* metódust, és nem érheti el az *iamprotected* tagot az *Alpha* példányban (ez az ősosztály), de hívhatja a *protected-*

Method metódust és elérheti az *iamprotected*-et az *AlphaTwo* példányában (vagy *AlphaTwo* leszármazott osztály példányában). Más szóval a *protected* elérési szint csak azt teszi lehetővé egy leszármazott osztály számára, hogy egy védett (*protected*) tagot csak akkor tudjon elérni, ha az illető tag hivatkozása ugyanolyan típusú osztályban, vagy leszármazott osztályban van. Az *AlphaTwo* futásának eredménye a következő lesz:

```
iampublic Method
iampublic: 4
iamprotected Method
iamprotected: 3
```

Nyilvános elérési szint

Végül a következő *DeltaTwo* nem kapcsolódik az osztály hierarchiában *Alpha*-hoz, és más csomagban is van, mint *Alpha*. Az előző táblázat utolsó oszlopa szerint *DeltaTwo* csak az *Alpha* nyilvános (*public*) tagjait érheti el.

```
package two;

import one.*;

public class DeltaTwo {
    public static void main(String[] args) {
        Alpha alpha = new Alpha();

        //alpha.privateMethod();    // nem megengedett
        //alpha.packageMethod();    // nem megengedett
        //alpha.protectedMethod(); // nem megengedett
        alpha.publicMethod();       // megengedett

        //System.out.println("iamprivate: "
        //    + a.iamprivate);      // nem megengedett
        //System.out.println("iampackage: "
        //    + a.iampackage);      // nem megengedett
        //System.out.println("iamprotected: "
        //    + a.iamprotected);    // nem megengedett
        System.out.println("iampublic: "
            + a.iampublic);         // megengedett
    }
}
```

A *DeltaTwo* outputja a következő lesz:

```
iampublic Method
iampublic: 4
```

Ha más programozók is használnak egy kész osztályt, szükséges lehet annak biztosítása, hogy a téves használat ne vezessen hibákhoz. Az elérési szintek segíthetnek ebben. A következő javaslatok segítenek annak meghatározásában, hogy egy adott taghoz melyik elérési szint a legmegfelelőbb.

- Használjuk a leginkább korlátozó, még észszerű elérési szintet az egyes tagokra. Hacsak valami különösen nem mond ellene, használjuk a *private* szintet.
- Kerüljük a publikus tagváltozókat, kivéve a konstansok estében. A publikus tagváltozók használata oda vezethet, hogy valamelyik speciális implementációhoz fog kapcsolódni a program, és ez hibákat, tévedéseket fog eredményezni. Továbbá, ha egy tagváltozót csak a hívó metódus tud megváltoztatni, akkor ezt a változást jelezni lehet a többi osztály vagy objektum felé. Ugyanakkor a változás jelzése lehetetlen, ha

egy tagváltozó publikus elérésű. A publikus elérés biztosítása ugyanakkor teljesítmény nyereséget eredményezhet.

Megjegyzés: Ebben az oktatási anyagban sok példa használ publikus tagváltozókat. A példák és elvi kódrészletek nem szükségszerűen felelnek meg azoknak a szigorú tervezési szabályoknak, amik egy API számára előírások.

- Korlátozzuk a védett (*protected*) és a csomag (*package*) elérésű tagváltozók számát.
- Ha egy tagváltozó *JavaBeans* tulajdonság, akkor az kötelezően *private* elérésű.

11.9. A példányok és az osztály tagok

Az osztályokról és az osztály tagokról már volt szó a nyelvi alapismeretek részben. Ez a rész bemutatja, hogy hogyan deklarálhatunk osztályt és osztálypéldányt. A következő osztály (*AClass*) deklarál egy példányváltozót, egy példánymetódust, egy osztályváltozót, egy osztály metódust, és végül a *main* metódust, ami szintén osztály metódus.

```
public class AClass {

    public int instanceInteger = 0;
    public int instanceMethod() {
        return instanceInteger;
    }

    public static int classInteger = 0;
    public static int classMethod() {
        return classInteger;
    }

    public static void main(String[] args) {
        AClass anInstance = new AClass();
        AClass anotherInstance = new Aclass();

        anInstance.instanceInteger = 1;
        anotherInstance.instanceInteger = 2;
        System.out.println(anInstance.instanceMethod());
        System.out.println(
                anotherInstance.instanceMethod());

        //System.out.println(instanceMethod());     //illegal
        //System.out.println(instanceInteger);      //illegal

        AClass.classInteger = 7;
        System.out.println(classMethod());

        System.out.println(anInstance.classMethod());

        anInstance.classInteger = 9;
        System.out.println(anInstance.classMethod());
        System.out.println(anotherInstance.classMethod());
    }
}
```

Itt látható a program kimenete:

```
1
2
7
7
9
9
```

Ha nincs egyéb meghatározás, akkor egy osztályon belül deklarált tag példány tag lesz. Így az *instanceInteger* és az *instanceMethod* mindketten példány tagok. A futtató rendszer a program összes példányváltozójából objektumonként készít egy példányt. Így az *anInstance* és az *anotherInstance* objektumok tartalmaznak egy-egy *instanceInteger* tagváltozót.

Hozzá tudunk férni a példányokhoz és meghívhatunk egy példánymetódust egy hivatkozáson keresztül. Ha az *illegal* felirattal megjelölt sorok elejéről kitöröljük a //-t, és megpróbáljuk lefordítani a programot, akkor egy hibaüzenetet kapunk.

Egy osztálytag a *static* módosítóval kerül deklarálásra. A *main* metóduson kívül az *AClass* deklarál egy osztályváltozót és egy osztálymetódust, melyeket *classInteger*-nek és *classMethod*-nak hívnak. A futtató rendszer osztályonként lefoglal egy osztályváltozót, függetlenül az osztály által lefoglalt példányok számától. A rendszer lefoglalja a memóriát az osztályváltozónak, legkésőbb akkor, amikor az először felhasználásra kerül.

Az osztály minden példányában elérhetőek az osztály osztályváltozói. Hozzáférhetünk az osztályváltozóhoz a példányokon keresztül, valamint az osztályon keresztül is. Hasonlóképpen egy osztály metódus is elérhető az osztályban vagy egy példányon keresztül. Megjegyezzük, hogyha a program megváltoztatja a *classVariable* értékét, akkor az megváltozik az összes osztálypéldányban is.

11.9.1 A példányok és az osztály tagok inicializálása

Osztály vagy példányváltozónak a deklarációnál adhatunk legegyszerűbben kezdőértéket:

```
public class BedAndBreakfast {
    public static final int MAX_CAPACITY = 10;
    private boolean full = false;
}
```

Ez jól működik egyszerű adattípusok esetében. Akkor is működik, ha tömböket vagy osztályokat készítünk. De vannak korlátai is:

- Az inicializálás csak kifejezést tartalmazhat, nem lehet pl. egy *if-else* utasítás.
- Az inicializáló kifejezés nem hívhat olyan függvényt, amely futásidejű kivételt dobhat. Ha olyan függvényt hív, amely futásidejű kivételt dob, mint pl. a *NullPointerException*, nem tudjuk a kivételt elkapni.

Ha ezek a korlátok gátolnak abban, hogy tagváltozót inicializáljunk a deklarációban, az inicializáló kód máshová is elhelyezhető. Egy osztályváltozó inicializálásához tegyük a kódot a statikus inicializáló blokkba, ahogy a következő példa mutatja. Egy példány inicializálásához tegyük a kódot a konstruktorba.

Statikus inicializáló blokk használata

Itt egy példa a statikus inicializáló blokkra:

```
import java.util.ResourceBundle;
```

```
class Errors {
    static ResourceBundle errorStrings;
    static {
        try {
            errorStrings =
               ResourceBundle.getBundle("ErrorStrings");
        } catch (java.util.MissingResourceException e) {
            //error recovery code here
        }
    }
}
```

A statikus inicializáló blokk a *static* kulcsszóval kezdődik, és mint általában, itt is kapcsos zárójelek közé tesszük a kódot. Az *errorStrings* a statikus inicializáló blokkban kerül inicializálásra, mert a *getBundle* metódus dobhat kivételt.

Egy osztály tartalmazhat bármennyi statikus inicializáló blokkot, amelyek bárhol lehetnek az osztály törzsében. A futtatórendszer garantálja, hogy a forráskódban elfoglalt helyük sorrendjében kerülnek meghívásra az inicializáló blokkok, még mielőtt a legelső alkalommal használná az így inicializált osztályváltozókat a program.

Példányváltozók inicializálása

Példányváltozók inicializálása az osztály konstruktorában is történhet. Ha az előző példában szereplő *errorStrings* példányváltozó lenne, akkor az inicializáló kódot az osztály konstruktorába tehetjük, az alábbi példa szerint:

```
import java.util.ResourceBundle;

class Errors {
    ResourceBundle errorStrings;
    Errors() {
        try {
            errorStrings =
              ResourceBundle.getBundle("ErrorStrings");
        } catch (java.util.MissingResourceException e) {
            //error recovery code here
        }
    }
}
```

11.10. Ellenőrző kérdések

- Mi az információelrejtés előnye, és hogyan valósul meg Javában?
- Mi az üzenet? Hogyan valósul meg Javában?
- Mi az osztály? Hogyan hozzuk létre Javában?
- Mi az objektum? Hogyan hozunk létre Javában?
- Mi a metódus aláírása (szignatúrája)?
- Mi a *void* típus?
- Mik játszódnak le egy metódus híváskor?
- Hogyan adja vissza a metódus a visszatérési értékét?
- Mi történik a metódus által deklarált változókkal?

- Mi az objektum, mi az osztály és mi a kapcsolatuk?
- Mi a különbség a példányváltozó és az osztályváltozó között?
- Mi az objektumreferencia?
- Mit jelent a *null*?
- Hogyan hívhatjuk meg egy objektum metódusait az objektumreferencián keresztül?
- Mi a konstruktor?
- Hogyan hívjuk meg a konstruktort?
- Mi az alapértelmezett konstruktor?
- Mikor generál alapértelmezett konstruktort a fordító maga?
- Hogyan hivatkozhatunk egy objektum példányváltozóira az objektumreferencián keresztül?
- Mit jelent az, hogy a Jáva rendszerben egy szemétgyűjtő működik? Mik ennek a következményei?
- Mi a statikus változó?
- Elérhető-e a statikus változó nem statikus metódusból?
- Milyen referenciával lehet elérni a statikus változót?
- Mi a statikus metódus?
- Elérhet-e példányváltozót statikus metódus?

Igaz vagy hamis? Indokolja!

- Az objektumot létrehozásakor inicializálni kell.
- Az objektum kezdeti állapotát a konstruktor állítja be.
- Az osztálymetódus elvileg elérheti a példányváltozót.
- A példánymetódus elvileg elérheti az osztályváltozót.
- A *this* a megszólított objektum referenciája.
- A konstruktor visszatérési értéke *boolean*.
- A konstruktor neveként ajánlatos az osztály nevét adni, de ez nem kötelező.
- Egy statikus metódus meghívhatja ugyanazon osztály egy nem statikus metódusát a *this* kulcsszó segítségével.
- Az osztály konstruktorából meghívható az osztály egy másik, túlterhelt konstruktora, annak nevére való hivatkozással.
- Egy osztálynak minden esetben van paraméter nélküli konstruktora.
- Ha az osztálynak nincs explicit konstruktora, akkor a rendszer megad egy alapértelmezés szerinti, paraméter nélkülit.
- A konstruktor lehet *final*.
- A konstruktor blokkja lehet üres.

- Az osztályinicializáló blokk beállítja az objektum kezdeti értékeit.
- Az inicializálók közül először futnak le az osztályinicializálók, és csak azután kerülnek végrehajtásra a példányinicializálók.
- Egy objektum létrehozható saját osztályából de csak osztálymetódusból.

A következő osztály esetén melyik a helyes konstruktor definíció?

```
public class Test {
    ....
}
```

- *public void Test() {...}*
- *public Test() {...}*
- *public static Test() {...}*
- *public static void Test() {...}*

A következő metódus esetén milyen típusú kifejezést írjunk a *return* után?

```
public void add(int a) {...}
```

- *void*
- *int*
- semmit

11.11. Gyakorló feladatok

Készítsen *Börtön* osztályt, amely a török szultán börtöne ajtajainak nyitott állapotát képes tárolni!

A konstruktor paraméterként a börtön méretét kapja meg (pl. 100). Hozzon létre egy ekkora méretű alkalmas tömböt, és gondoskodjon a megfelelő kezdőértékről (az ajtók kezdetben zárva vannak).

A *kulcsFordít* metódus paraméterként kapja, hogy hányas számú cella kulcsát kell átfordítani (ha nyitva volt, akkor bezárja, és fordítva). Nem megfelelő index esetén magyar nyelvű üzenetet tartalmazó kivételt dobjon (nem itt kell a hibaüzenetet a képernyőre írni).

Oldja meg, hogy nyitott cellák száma jelenjen meg a konzolon, ha a *System.out.println* paramétereként egy *Börtön* objektumot adunk meg (tehát írja felül az öröklött *toString* metódust).

Készítsen *main* metódust, amely az eredeti játékos feladat szerint először minden cella, majd minden második, minden harmadik stb., végül a századik cella kulcsát fordítja át, majd kiírja a szabaduló foglyok számát.

Készítsen Anagramma osztályt, amely sztringek betűinek keverésére használható.

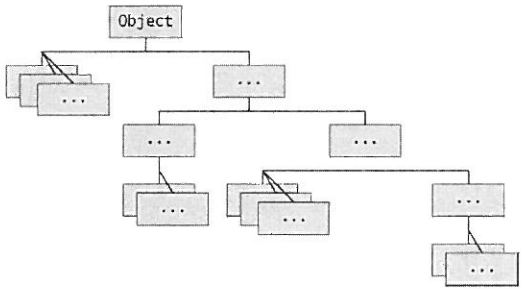
A statikus *fordít* metódus paraméterként kapjon egy *String*-et, visszaad egy másik *String* objektumot, amely az eredeti szöveg első és utolsó betűjét megcserélve tartalmazza (a többi változatlan).

A szintén statikus *kever* metódus tényleges keverést hajtson végre a véletlenszám-generátor (lásd *java.util.Random.nextInt()* metódus) segítségével. (Tipp: pl. 50-szer generáljunk két véletlen indexet, és cseréljük meg a két indexnél levő karaktert. Még jobb, ha nem fix, hanem a *String* hosszától függő ismétlést hajt végre.)

A *main* metódus olvasson be a billentyűzetről szavakat, és írja ki azok *fordít* és *kever* metódussal kapott módosításait.

12.Öröklődés

A *java.lang* csomagban definiált *Object* osztály meghatározza és megvalósítja azokat a metódusokat, amelyek minden osztály számára szükségesek. A következő ábrán látható, hogy sok osztály ered az *Object*-ből, majd sok további osztály származik az előbbi osztályokból, és így tovább, létrehozva ezzel az osztályok hierarchiáját.



A hierarchia csúcsán álló *Object* az osztályok legáltalánosabbja. A hierarchia alján található osztályok sokkal specializáltabb viselkedést eredményeznek. Egy leszármazott osztály valamely osztályból származik. A *superclass* kifejezés (továbbiakban szülőosztály vagy ősosztály) egy osztály közvetlen ősére/elődjére, vagy annak bármely felmenő osztályára utal. Minden osztálynak csak és kizárólag egyetlen közvetlen szülőosztálya van.

Egy leszármazott osztály a változóit és metódusait a szülőosztályától örökli. A leszármazott osztály számára azonban lehet, hogy nem elérhető egy öröklött változó vagy függvény. Például, egy leszármazott osztály számára nem érhető el egy *private* tag, ami a felsőbb osztálytól öröklődött. Mondhatnánk persze, hogy akkor az a tag egyáltalán nem is öröklődött. De igenis öröklődött. Akkor válik ez fontossá, amikor egy olyan belső osztályt használunk, aminek van hozzáférése a mellékelt osztályok *private* tagjaihoz. Ne feledjük, hogy a konstruktorok nem metódusok, tehát az leszármazott osztályok nem örökölhetik azokat.

12.1. Metódusok felülírása és elrejtése

Ha egy leszármazott osztálybeli metódus, melynek ugyanaz a szignatúrája és visszatérési értéke, mint a szülőosztály metódusának, akkor a leszármazott osztály felülírja a szülőosztály metódusát. (Megjegyzendő, hogy egy metódus szignatúrája a nevéből, valamint paramétereinek számából és típusából áll.)

Egy leszármazott osztály felülíró képessége lehetővé teszi, hogy egy osztály örököljön egy olyan szülőosztálytól, melynek viselkedése elég közeli, majd szükség szerint változtasson a viselkedésen. Például az *Object* osztály tartalmaz egy *toString* nevű metódust, amelynek a visszaadja az objektumpéldány szöveges reprezentációját. Minden osztály megörökli ezt a metódust. Az *Object* metódusának végrehajtása általában nem túl hasznos a leszármazott osztályok számára, ezért a metódus felülírása célszerű, hogy jobb információt nyújthasson az objektum saját magáról. Ez különösen hasznos például nyomkövetés esetén. A következő kód egy példa a *toString* felülírására:

```
public class MyClass {
    private int anInt = 4;
    public String toString() {
        return "Instance of MyClass. anInt = " + anInt;
    }
}
```

A felülíró metódusának neve, valamint paramétereinek száma és típusa, valamint visszatérési értéke megegyezik azzal a metódussal, amelyet felülír. (Valójában a leszármazott osztálybeli metódus visszatérési típusa lehet a szülőosztály visszatérő típusának leszármazottja is a Java 5 óta.)

A felülíró metódusnak lehet az őstől eltérő *throws* záradéka, ha nem ad meg olyan típusokat, melyek nincsenek a felülírt metódus záradékában előírva. Másrészt, a felülíró metódus láthatósága lehet bővebb, mint a felülírt metódusé, de szűkebb nem. Például a szülő osztály *protected* metódusa a leszármazott osztályban publikussá (*public*) tehető, de privátté (*private*) nem.

Megjegyzés: Érdemes átgondolni e szabályok hátterét. Egy leszármazott osztály objektuma bárhol használható, ahol egy ősosztálybeli objektum is. Éppen ezért a leszármazott semelyik tagjának láthatósága nem szűkülhet, hiszen akkor az ilyen használat lehetetlen lenne. Ugyanígy egy felülírt metódus által dobott újfajta kivétel kezelése nem lenne biztosított.

Egy leszármazott osztály nem tudja felülírni az olyan metódusokat, melyek az ősosztályban végleges (*final*) minősítésű (a definíció szerint a végleges metódusok nem felülírhatók). Ha mégis megpróbálunk felülírni egy végleges metódust, a fordító hibaüzenetet küld.

Egy leszármazott osztálynak felül kell írnia azon metódusokat, melyek a felsőbb osztályban absztraktnak (*abstract*) nyilvánítottak, vagy maga a leszármazott osztály is absztrakt kell, hogy legyen. Emlékezzünk vissza, hogy a Java programnyelv megengedi a metódusok túlterhelését, ha a metódus paramétereinek a számát vagy típusát megváltoztatjuk. Egy ősosztályban is megengedhető a metódusok túlterhelése. Alábbiakban nézzünk egy példát a *toString* metódus túlterhelésére:

```
public class MyClass {
    private int anInt = 4;

    public String toString() {
        return "Instance of MyClass. anInt = " + anInt;
    }

    public String toString(String prefix) {
        return prefix + ": " + toString();
    }
}
```

Amint azt a példa illusztrálja, túlterhelhetünk egy ősosztálybeli metódust, hogy további funkciókkal is szolgálhasson. Amikor egy olyan metódus írunk, mely azonos nevű a felsőbb osztálybeli metódussal, le kell ellenőrizni a paramétereket és a kivétellistát (*throws* záradék), hogy biztosak lehessünk afelől, hogy a felülírás olyan lett, amilyennek akartuk.

Ha egy leszármazott osztály egy osztálymetódust ugyanazzal az aláírással definiál, mint a felsőbb osztálybeli metódus, akkor a leszármazott osztály metódusa elrejti (másként fogalmazva elfedi) a szülőosztálybelit. Nagy jelentősége van az elrejtés és a felülírás megkülönböztetésének. Nézzük meg egy példán keresztül, hogy miért! E példa két osztályt tartalmaz. Az első az *Animal*, melyben van egy példánymetódus és egy osztálymetódus:

```
public class Animal {
    public static void hide() {
        System.out.println("The hide method in Animal.");
    }
    public void override() {
        System.out.println("The override method in Animal.");
    }
}
```