Nagy Gusztáv

Java programozás

1.3. verzió

2007. február

Jogi nyilatkozat

Nevezd meg! - Ne add el! 2.5 Magyarország

A következőket teheted a művel:

- szabadon másolhatod, terjesztheted, bemutathatod és előadhatod a művet
- származékos műveket (feldolgozásokat) hozhatsz létre

Az alábbi feltételekkel:

Jelöld meg! A szerző vagy a jogosult által meghatározott módon kell megjelölni a művet (pl. a szerző és a cím feltüntetésével).

Ne add el! Ezt a művet nem használhatod fel kereskedelmi célokra.

A szerzői jogok tulajdonosának írásos engedélyével bármelyik fenti feltételtől eltérhetsz.

A fentiek nem befolyásolják a szabad felhasználáshoz fűződő, illetve az egyéb jogokat.

Ez a *Legal Code* (Jogi változat, vagyis a teljes licenc) szövegének közérthető nyelven megfogalmazott kivonata.

Ez a kivonat a http://creativecommons.org/licenses/by-nc/2.5/hu/ oldalon is olvasható. A teljes licensz a http://creativecommons.org/licenses/by-nc/2.5/hu/legalcode oldalon érhető el.

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....9
1.1.Az első csésze kávé....9
1.1.1Az első alkalmazás létrehozása....9
1.2.Bevezetés a Java technológiába....11
1.2.1A Java programozási nyelv....11
1.2.2Java platform....12
1.3.Mire jó a Java technológia?....12
1.4.Még egyszer a HelloWorld programról....13
1.4.1Megjegyzések a Java nyelvben....13
1.4.2Osztálydefiníció....14
1.4.3A main metódus....14
1.4.4Osztályok és objektumok használata....15
1.5.Ellenőrző kérdések....16
2.Objektumorientált paradigma....17
2.1.Az objektum....17
2.2.Az üzenet....18
2.3.Az osztály....19
2.4.Az öröklődés....21
2.5.Publikus interfész....23
2.6.Ellenőrző kérdések....23
2.7.Gyakorló feladat....24
3.Változók....25
3.1.Adattípusok....26
3.2.Változó nevek....27
3.3.Érvényességi tartomány....28
3.4.Változók inicializálása....28
3.5.Végleges változók....29
3.6.Ellenőrző kérdések....29
4.Operátorok....32
4.1.Aritmetikai operátorok....32
4.1.1Implicit konverzió....34
4.2.Relációs operátorok....35
4.3.Logikai operátorok....36
4.3.1Rövidzár kiértékelés....37
4.4.Bitléptető és bitenkénti logikai operátorok....38
4.4.1Bitmanipulációk a gyakorlatban....39
4.5.Értékadó operátorok....40
4.6.Egyéb operátorok....41
4.7.Ellenőrző kérdések....41
5.Kifejezések, utasítások, blokkok....43
5.1.Kifejezések....43
5.2.Utasítások....45
5.3.Blokkok....46
5.4.Összefoglalás....46
5.5.Ellenőrző kérdések....46
5.6.Gyakorló feladatok....47
6.Vezérlési szerkezetek....48
6.1.A while és a do-while ciklusok....48

6.2.A for ciklus....49
6.3.Az if-else szerkezet....51
6.4.A switch-case szerkezet....53
6.4.1A switch utasítás és a felsorolt típus....54
6.5.Vezérlésátadó utasítások....55
6.6.Ellenőrző kérdések....59
6.7.Gyakorló feladatok....60
7.Objektumok használata....62
7.1.Objektumok létrehozása....62
7.2.Hivatkozás egy objektum tagjaira....66
7.3.Metódushívás....66
7.4.Nem használt objektumok eltávolítása....67
7.5.Takarítás....67
7.6.Ellenőrző kérdések....67
7.7.Gyakorló feladat....68
8.Karakterek és sztringek....69
8.1.A Character osztály....69
8.2.String, StringBuffer és StringBuilder osztály....70
8.3.Sztringek darabolása....78
8.4.Ellenőrző kérdések....78
8.5.Gyakorló feladatok....80
9.Számok....81
9.1.A csomagoló osztályok néhány használata....81
9.2.Szövegből számmá konvertálás....83
9.3.Számból szöveggé konvertálás....83
9.4.Számok formázott konvertálása....84
9.5.Aritmetika....87
9.6.Ellenőrző kérdések....90
10.Tömbök....92
10.1.Tömbök létrehozása és használata....92
10.2.Objektum tömbök....94
10.3.Tömbök tömbjei....95
10.4.Tömbök másolása....96
10.5.Ellenőrző kérdések....97
10.6.Gyakorló feladatok....98
11. Osztályok létrehozása....100
11.1.Osztályok deklarálása....100
11.2.Tagváltozók deklarálása....101
11.3.Metódusok deklarálása....102
11.4.Konstruktorok....103
11.5.Információátadás metódus vagy konstruktor számára....104
11.6.A metódusok visszatérési értéke....107
11.7.A this kulcsszó használata....108
11.8.Egy osztály tagjai elérhetőségének felügyelete....109
11.9.A példányok és az osztály tagok....114
11.9.1A példányok és az osztály tagok inicializálása....115
11.10.Ellenőrző kérdések....116
11.11.Gyakorló feladatok....118
12.Öröklődés....120
12.1.Metódusok felülírása és elrejtése....120
12.2.Tagváltozók elrejtése....122
12.3.A super használata....122

12.4.Az Object osztály metódusai.....124
12.5.Végleges osztályok és metódusok.....126
12.6.Ellenőrző kérdések.....127
12.7.Gyakorló feladatok.....128
13.Beágyazott osztályok.....129
13.1.Belső osztályok.....130
13.2.Néhány további érdekesség.....131
13.3.Ellenőrző kérdések.....132
14.Felsorolás típus.....133
14.1.Ellenőrző kérdések.....134
15.Általános programozás.....135
15.1.Általános típus definiálása és használata.....135
15.2.Kapcsolatok az általános típusok között.....136
15.3.Helyettesítő típus.....137
15.4.Általános metódusok definiálása és használata.....138
15.5.Általános típusok használata az öröklésben.....138
15.6.Ellenőrző kérdések.....140
16.Interfészek.....141
16.1.Interfész definiálása.....141
16.2.Interfészek implementálása.....142
16.3.Az interface használata típusként.....143
16.4.Ellenőrző kérdések.....143
17.Csomagok.....145
17.1.Csomag létrehozása.....145
17.2.Egy csomag elnevezése.....146
17.3.Csomag tagok használata.....146
17.4.Forrás és osztály fájlok menedzselése.....148
17.5.Ellenőrző kérdések.....150
18.Kivételkezelés.....152
18.1.Kivételek elkapása vagy továbbengedése.....153
18.2.Kivételek dobása.....159
18.2.1A throw használata.....160
18.3.Eldobható osztályok és leszármazottai.....160
18.4.Láncolt kivételek.....161
18.5.Saját kivétel osztályok létrehozása.....162
18.6.Ellenőrző kérdések.....163
19.Programszálak kezelése.....165
19.1.A Timer és a TimerTask osztály.....166
19.1.1Időzített szálak leállítása.....167
19.1.2Ismételt futtatás.....167
19.2.Szálak példányosítása.....168
19.2.1Thread leszármazott és a run felülírása.....168
19.2.2Runnable interfész példányosítása.....170
19.3.Programszál életciklusa.....172
19.3.1Programszál létrehozása.....172
19.3.2Programszál elindítása.....172
19.3.3Programszál nem futtatható állapotba állítása.....173
19.3.4Programszál leállítása.....174
19.3.5Programszál státusz tesztelése.....175
19.3.6A processzor használatának feladása.....175
19.4.Ellenőrző kérdések.....176
20.Fájlkezelés.....178

20.1.Adatfolyamok....178
20.1.1Fájl adatfolyamok használata....182
20.1.2Szűrő adatfolyamok....183
20.2.Objektum szerializáció....186
20.2.1Objektumok szerializálása....186
20.2.2Objektum szerializáció a gyakorlatban....187
20.2.3Az Externalizable interfész implementálása....188
20.2.4Az érzékeny információ megvédése....188
20.3.Közvetlen elérésű állományok....189
20.3.1A közvetlen elérésű állományok használata....190
20.4.További osztályok és interfészek....190
20.5.Ellenőrző kérdések....191
21.Gyűjtemények....193
21.1.A gyűjtemény keretrendszer....193
21.1.1A Gyűjtemény keretrendszer használatának előnyei....193
21.2.Interfészek....194
21.2.1A gyűjtemény interfészek....195
21.2.2A Collection interfész....196
21.2.3A Set interfész....198
21.2.4A List interfész....201
21.2.5Map interfész....206
21.2.6Objektumok rendezése....208
21.3.Implementációk....212
21.3.1Általános célú Set implementációk....213
21.3.2Általános célú List implementációk....214
21.3.3Általános célú Map implementációk....214
21.4.Algoritmusok....214
21.4.1Rendezés....214
21.4.2Keverés....216
21.4.3Keresés....216
21.5.Ellenőrző kérdések....216
22.Hálózatkezelés....218
22.1.Hálózati alapok....218
22.1.1TCP....218
22.1.2UDP....218
22.1.3A portokról általánosságban....219
22.1.4Hálózati osztályok a JDK-ban....220
22.2.URL-ek kezelése....220
22.2.1URL objektum létrehozása....221
22.2.2URL elemzés....223
22.2.3Közvetlen olvasás URL-ből....224
22.2.4Csatlakozás egy URL-hez....224
22.3.Socketek kezelése....226
22.3.1Mi az a socket?....226
22.3.2Olvasás és írás a socket-ről....227
23.JDBC adatbázis-kezelés....230
23.1.Adatbázis beállítása....230
23.2.Táblák használata....232
23.2.1Tábla létrehozása....232
23.2.2JDBC Statement létrehozása....233
23.2.3SQL parancs végrehajtása....234
23.2.4Lekérdezések eredményének feldolgozása....235

24.Kódolási konvenciók.....237
24.1.Fájlok szervezése.....237
24.2.Behúzás.....238
24.3.Megjegyzések.....240
24.3.1Implementációs megjegyzések.....240
24.3.2Dokumentációs megjegyzések.....241
24.4.Deklarációk.....242
24.4.1A deklaráció helye.....243
24.4.2Osztály és interfész deklaráció.....243
24.5.Utasítások.....244
24.6.Elválasztók.....246
24.7.Elnevezési konvenciók.....246
25.Tervezési minták.....248
25.1.Létrehozási minták.....249
25.1.1Egyke (Singleton).....249
25.1.2Gyártófüggvény (Factory Method) minta.....250
25.2.Szerkezeti minták.....252
25.3.Viselkedési minták.....252
26.Java fejlesztőeszközök.....253
26.1.JCreator.....253
26.2.Netbeans.....253
26.2.1Alapvető használat.....254
26.3.Eclipse.....256
26.3.1Alap tulajdonságok.....256
26.3.2Az Eclipse beszerzése és üzembe helyezése.....257

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

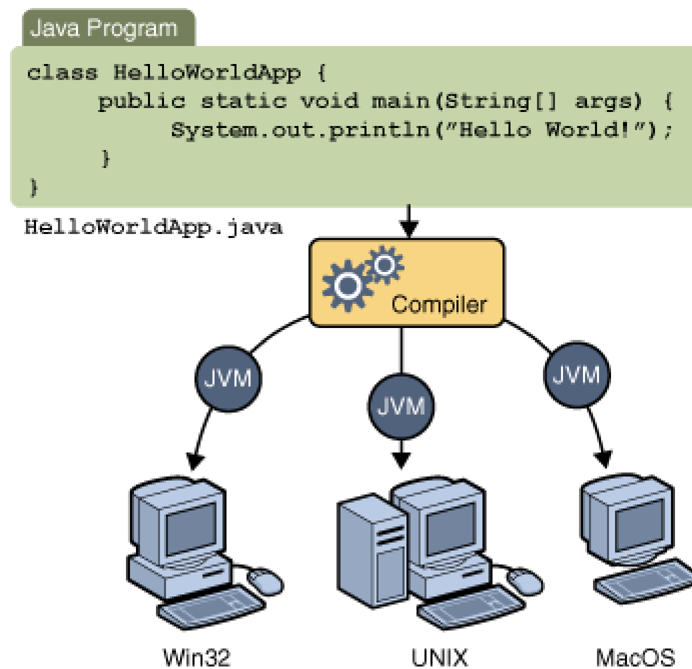
Bármilyen editor megfelel a jegyzettömbtől (Notepad.exe) az összetett programozói editorokig.

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.

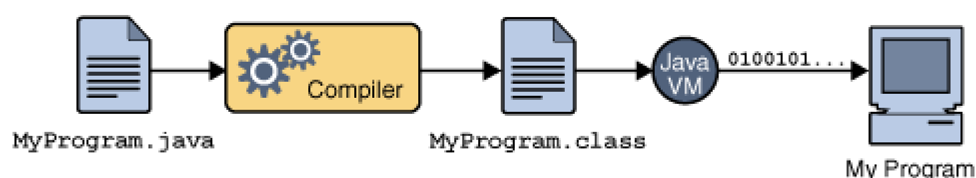
1.2.1 A Java programozási nyelv

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 platform-független 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.

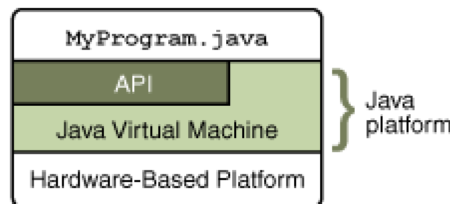
1.2.2 Java platform

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.

az én biciklim :
sebesség = 18km/h sebességfokozat = 5

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 én biciklim :
sebesség = 18km/h sebességfokozat = 5
sebességváltás fékezés

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.

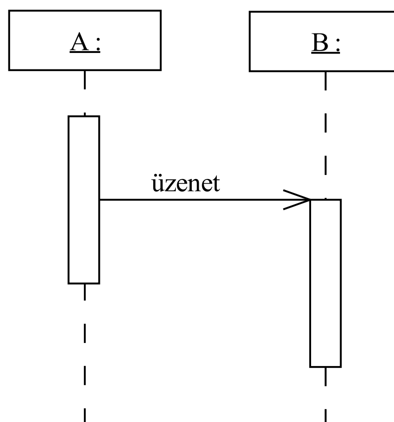
Az egységbezárás tiszta alkalmazása esetén két egyszerű, de nagyon hasznos előnye lesz a szoftverfejlesztőnek:

- **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.

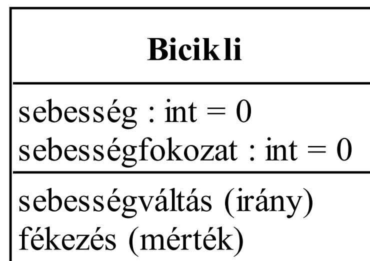
Definíció: **Osztályozásnak** nevezzük azt a folyamatot, amelynek során a hasonló objektumokat közös csoportokba, más néven osztályokba soroljuk.

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();
     }
}
```

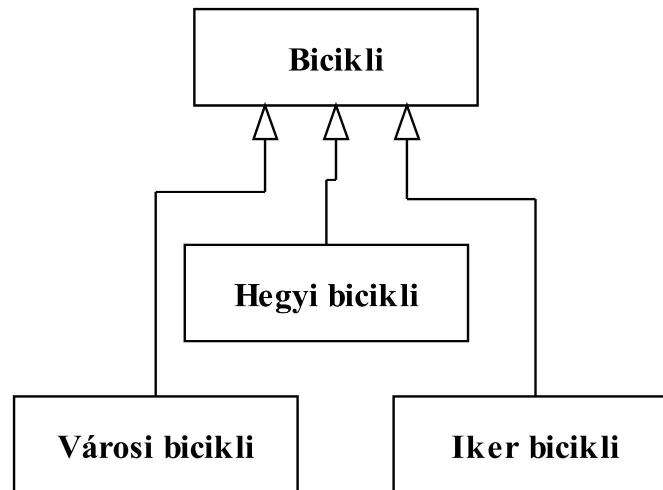
A példányváltozók mellett az osztályok definiálhatnak osztályváltozókat is. Az **osztályváltozók** az összes objektumpéldány számára megosztott információkat tartalmaznak. Például képzeljük el, hogy az összes kerékpár ugyanannyi sebességfokozattal rendelkezik. Ebben az esetben felesleges példányváltozót 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.

variableName | value |

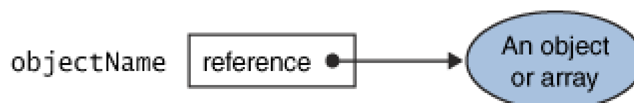
A következő táblázat az összes primitív típust tartalmazza. A példaprogramunk minden típusból deklarál egyet.

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:

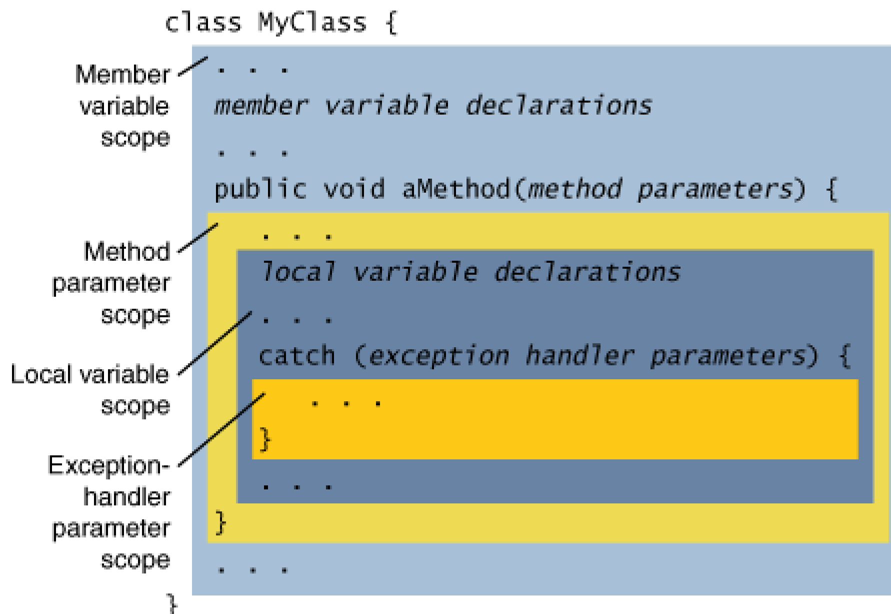
- Valódi azonosító legyen, tetszőlegesen hosszú Unicode karaktersorozat, de az első karakter csak betű lehet.
- Nem lehet foglalt szó, logikai literál (*true* vagy *false*) vagy *null*.
- Egyedinek kell lenni az érvényességi tartományban, viszont más tartománybeli változóval megegyezhet.

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:

<table>
<tr><th>Operátor</th><th>Alkalmazás</th><th>Leírás</th></tr>
<tr><td>&&</td><td>op1 && op2</td><td>Logikai és: true-t ad vissza, ha op1 és op2 egyaránt true; op2 feltételes kiértékelésű</td></tr>
<tr><td>||</td><td>op1 || op2</td><td>Logikai vagy: true-t ad vissza, ha op1 vagy op2 true; op2 feltételes kiértékelésű</td></tr>
<tr><td>!</td><td>!op</td><td>Logikai nem: true-t ad vissza, ha op false</td></tr>
<tr><td>&</td><td>op1 & op2</td><td>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</td></tr>
<tr><td>|</td><td>op1 | op2</td><td>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</td></tr>
<tr><td>^</td><td>op1 ^ op2</td><td>Bitenkénti nem: true-t ad vissza, ha op1 és op2 különböző — vagy egyik, vagy másik, de nem egyszerre mindkét operandus true</td></tr>
</table>

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 viszszaadott 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:

<table>
<tr><th>Operátor</th><th>Alkalmazás</th><th>Leírás</th></tr>
<tr><td>&</td><td>op1 & op2</td><td>Bitenkénti és, ha mindkét operandus szám; feltételes és ha mindkét operandus logikai</td></tr>
<tr><td>|</td><td>op1 | op2</td><td>Bitenkénti vagy, ha mindkét operandus szám; feltételes vagy, ha mindkét operandus logikai</td></tr>
<tr><td>^</td><td>op1 ^ op2</td><td>Bitenkénti kizáró vagy (xor)</td></tr>
<tr><td>~</td><td>~op2</td><td>Bitenkénti negáció</td></tr>
</table>

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):

originOne [•]

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

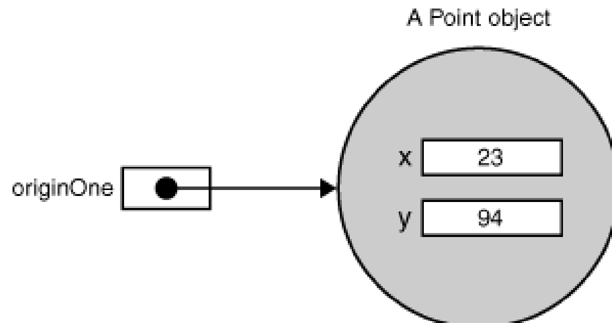
A *Point* osztály kódja:

```
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);
    }
```

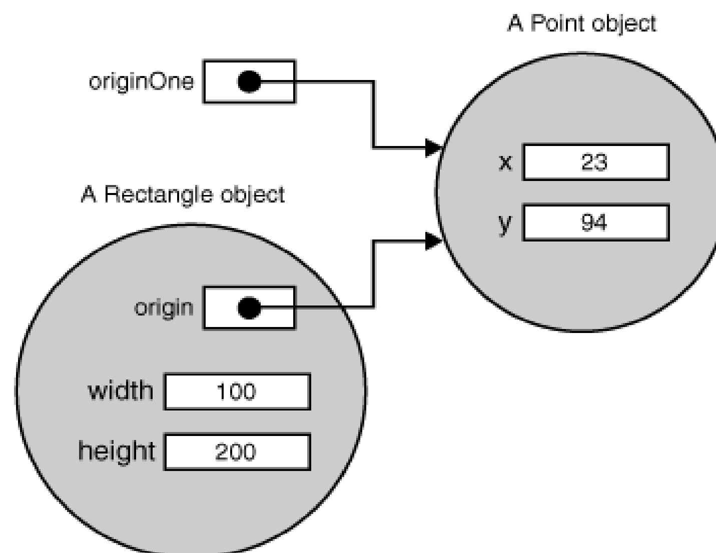
```
    public Rectangle(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }
    public void move(int x, int y) {
        origin.x = x;
        origin.y = y;
    }
    public int area() {
        return width * height;
    }
}
```

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 S*tring* 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.

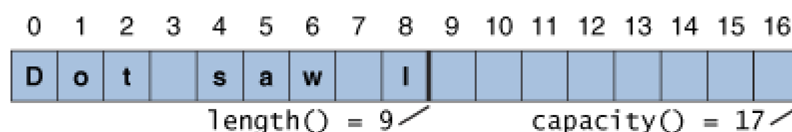
Megjegyzés: Ha egy *StringBuilder* vagy *StringBuffer* objektum méret növelése során a szabad kapacitása elfogy, akkor egy új, kétszer akkora memóriaterület kerül lefoglalásra, ahová a régi tartalom átmásolásra kerül. Ebből is látszik, hogy ha tudjuk, érdemes pontosan (vagy legalább becsülten) megadni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

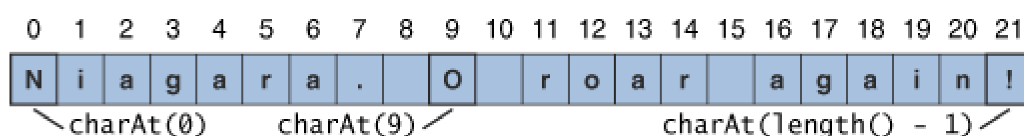
Stringek karaktereinek olvasása

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.

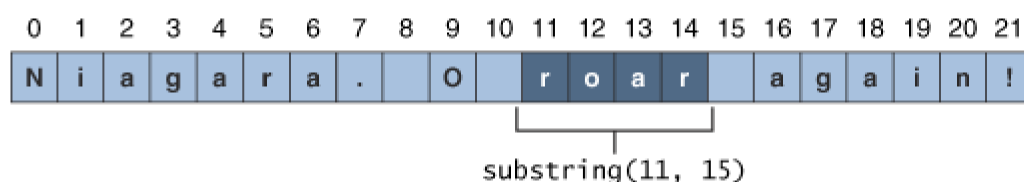
Az ábra is mutatja, hogy hogyan lehet kiszámítani egy *String*-ben az utolsó karakter indexét. Ki kell vonni a *length*() függvény visszatérési értékéből 1-et.

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 S*tring* 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 *String-Buffer*-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 *String-Buffer*-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

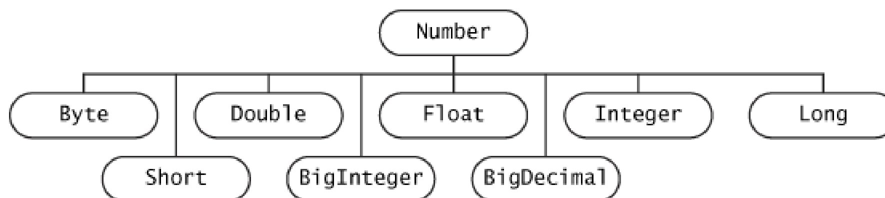
Írjon programot, amely az „Indul a görög aludni” *String* tartalmát

- 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 egyenlőség-vizsgáló 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 ≤ 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 ≤ szám < 10.0.

1-et hozzáadva az intervallum ismét megváltozik:

1.0 ≤ 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?

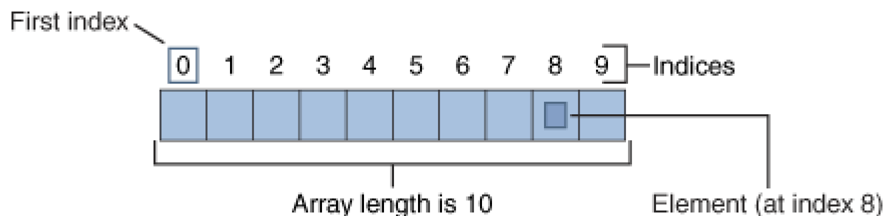
Mit ír ki a következő kódrészlet?

```
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.

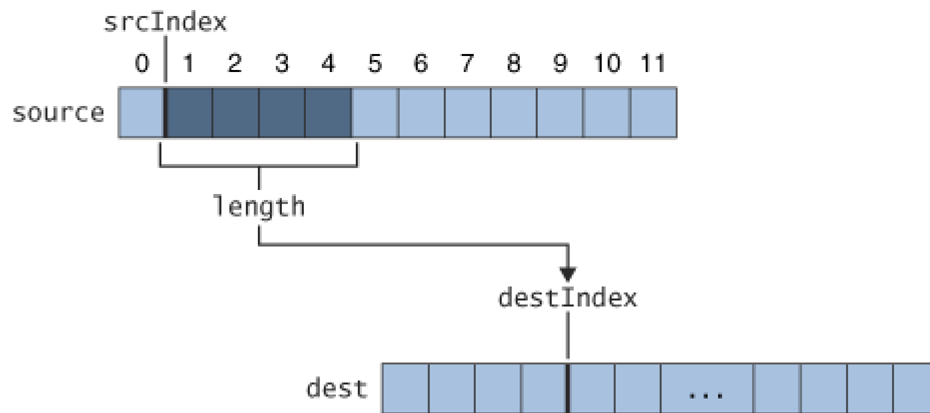
10.4. Tömbök másolása

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:



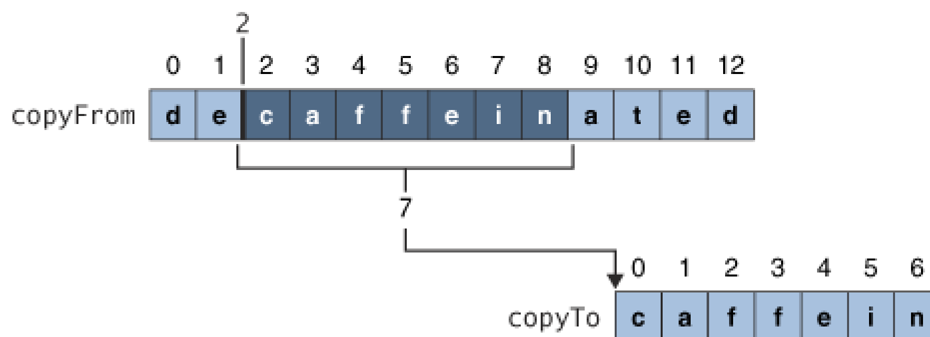
A következő *ArrayCopyDemo* program használja az *arraycopy* metódust, ami az elemeket a *copyFrom* tömbből a *copyTo* tömbbe másolja.

```
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.

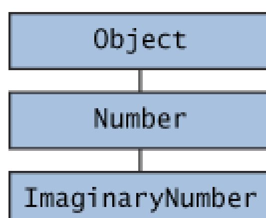
Megjegyzés: A fenti kód helyett általában tömörebb írásmódot szokás alkalmazni. Ennek ellenére ebben a jegyzetben a jobb érthetőség kedvéért időnként a hosszabb formát alkalmazzuk. A következő kód az előzővel teljesen ekvivalens:

```
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:

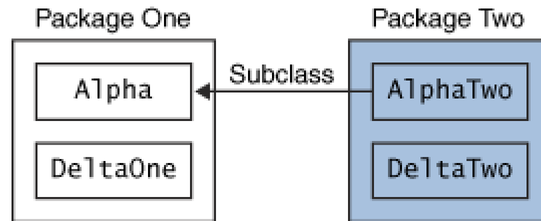
	osztály	csomag	leszármazott	összes
private	I	N	N	N
nincs	I	I	N	N
protected	I	I	I	N
public	I	I	I	I

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.

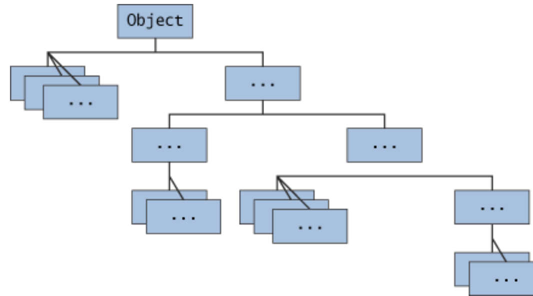
A statikus *fordít* metódus paraméterként kapjon egy *String*-et, visszaad egy másik *String* objektumot, amely az eredeti szöveg első és utolsó betűjét megcserélve tartalmazza (a többi változatlan).

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 priváttá (*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.");
    }
}
```

A második osztály neve *Cat*, ez az *Animal*-nak egy leszármazott osztálya:

```
public class Cat extends Animal {
    public static void hide() {
        System.out.println("The hide method in Cat.");
    }
    public void override() {
        System.out.println("The override method in Cat.");
    }

    public static void main(String[] args) {
        Cat myCat = new Cat();
        Animal myAnimal = (Animal)myCat;
        myAnimal.hide();
        myAnimal.override();
    }
}
```

A *Cat* osztály felülírja az *override* metódust az *Animal*-ban, és elrejti a *hide* osztálymetódust az *Animal*-ban. Ebben az osztályban a *main* metódus létrehoz egy *Cat* példányt, beteszi az *Animal* típusú hivatkozás alá is, majd előhívja mind az elrejtett, mind a felülírt metódust. A program eredménye a következő:

```
The hide method in Animal.
The override method in Cat.
```

A szülőosztályból hívjuk meg a rejtett metódust, a leszármazott osztályból pedig a felülírtat. Osztálymetódushoz a futtatórendszer azt a metódust hívja meg, mely a hivatkozás szerkesztési idejű típusában van definiálva, amellyel a metódust elnevezték. A példánkban az *myAnimal* szerkesztési idejű típusa az *Animal*. Ekképpen a futtatórendszer az *Animal*-ban definiált rejtett metódust hívja meg. A példánymetódusnál a futtatórendszer a hivatkozás futásidejű típusában meghatározott metódust hívja meg. A példában az *myAnimal* futásidejű típusa a *Cat*. Ekképpen a futtatórendszer a *Cat*-ban definiált felülíró metódust hívja meg.

Egy példánymetódus nem tud felülírni egy osztálymetódust, és egy osztálymetódus nem tud elrejteni egy példánymetódust. Mindkét esetben fordítási hibát kapunk.

12.2. Tagváltozók elrejtése

Egy osztály változója, ha ugyanazt a nevet viseli, mint a felsőbb osztály egy változója, akkor elrejti a felsőbb osztály változóját, még akkor is, ha különböző a típusuk. Az leszármazott osztályokon belül a felsőbb osztálybeli változóra nem utalhatunk egyszerűen a nevével. Ehelyett a tagváltozót el tudjuk érni az ősosztályon keresztül, amiről majd a következő fejezet fog szólni. Általánosságban véve nem célszerű a tagváltozók elrejtése.

12.3. A *super* használata

Ha egy metódus felülírja az ősosztálya metódusainak egyikét, akkor a *super* használatával segítségül hívható a felülírt metódus. A *super* arra is használható, hogy egy rejtett tag variánsra utaljunk. Ez a szülőosztály:

```
public class Superclass {
    public boolean aVariable;
```

```
    public void aMethod() {
        aVariable = true;
    }
}
```

Most következzen a *Subclass* nevű leszármazott osztály, mely felülírja *aMethod*-ot és *aVariable*-t:

```
public class Subclass extends Superclass {
    public boolean aVariable; //hides aVariable in Superclass
    public void aMethod() { //overrides aMethod in Superclass
        aVariable = false;
        super.aMethod();
        System.out.println(aVariable);
        System.out.println(super.aVariable);
    }
}
```

A leszármazott osztályon belül az *aVariable* név a *SubClass*-ban deklaráltra utalt, amely a szülőosztályban deklaráltat elrejti. Hasonlóképpen, az *aMethod* név a *SubClass*-ban deklaráltra utalt, amely felsőbb osztályban deklaráltat felülírja. Tehát ha egy a szülőosztályból örökölt *aVariable*-ra és *aMethod*-ra szeretnénk utalni, a leszármazott osztálynak egy minősített nevet kell használnia, használva a *super*-t, mint azt láttuk. A *Subclass aMethod* metódusa a következőket írja ki:

```
false
true
```

Használhatjuk a *super*-t a konstruktoron belül is az ősosztály konstruktora meghívására. A következő kódpélda bemutatja a *Thread* osztály egy részét – az osztály lényegében többszálú programfutást tesz lehetővé –, amely végrehajt egy animációt. Az *AnimationThread* osztály konstruktora beállít néhány kezdeti értékeket, ilyenek például a keretsebesség és a képek száma, majd a végén letölti a képeket:

```
class AnimationThread extends Thread {
    int framesPerSecond;
    int numImages;
    Image[] images;

    AnimationThread(int fps, int num) {
        super("AnimationThread");
        this.framesPerSecond = fps;
        this.numImages = num;
        this.images = new Image[numImages];

        for (int i = 0; i <= numImages; i++) {
            ...
            // Load all the images.
            ...
        }
    }
    ...
}
```

A félkövérrel szedett sor a közvetlen szülőosztály konstruktorának explicit meghívása, melyet a *Thread* nyújt. Ez a *Thread* konstruktor átvesz egy *String*-et, és így nevezi a *Thread*-et. Ha a leszármazott osztály konstruktorában van explicit *super* konstruktorhívás, akkor annak az elsőnek kell lennie. Ha egy konstruktor nem hív meg explicit módon egy szülőosztálybeli konstruktort, akkor a Java futtatórendszer automatikusan (implicit)

a szülőosztály paraméter nélküli konstruktorát hívja meg, még mielőtt a konstruktoron belül bármi utasítást végrehajtana.

Megjegyzés: ha a szülő osztályban nem áll rendelkezésre paraméter nélküli konstruktor, akkor fordítási hibát kapunk. Ilyen esetben kötelesek vagyunk explicit paraméteres konstruktorhívást alkalmazni.

12.4. Az *Object* osztály metódusai

Az *Object* osztály minden osztály közös őse, az osztályhierarchia tetején áll. Minden osztály közvetlen vagy közvetett módon utódja az *Object* osztálynak, így minden osztály rendelkezik az *Object* osztály metódusaival. Ez az osztály definiálja azt az alapvető működést, mely minden objektumnál rendelkezésre áll.

Az *Object* osztály által nyújtott legfontosabb metódusok a következők:

- *clone*
- *equals* és *hashCode*
- *finalize*
- *toString*
- *getClass*

Ezeket sorra tárgyaljuk a következőkben.

A *clone* metódus

A *clone* metódust akkor használjuk, ha létre szeretnénk hozni egy objektumot, egy már meglévő objektumból (másolatot készíteni róla). Az adott osztállyal megegyező típusú új példányt hoz létre:

```
aCloneableObject.clone();
```

A metódus a *CloneNotSupportedException* kivételt dobja, ha a klónozás nem támogatott az osztály számára. A klónozás akkor támogatott, ha az osztály implementálja a *Cloneable* interfészt. Habár az *Object* tartalmazza a *Clone* metódust, nincsen megvalósítva az interfész. Ha az objektum, ahol a *clone*-ra hivatkoztunk, nem implementálja a *cloneable* interfészt, egy eredetivel azonos típusú és értékű objektum jön létre. Legegyszerűbb azonban, ha az osztály deklarációban létrehozunk egy *implements Cloneable* sort.

Bizonyos osztályoknál a helyes működés feltétele a *clone* felüldefiniálása. Tekintsünk egy *Stack* osztályt, mely tartalmaz egy tagváltozót, mely az *Object-ek* tömbjére hivatkozik. Ha a *Stack* az *Object* osztály *clone* metódusára épül, akkor az eredeti és a másolt *Stack* ugyanazokat az elemeket fogja tartalmazni, mivel az adattag tömb, és másoláskor csak referencia másolás fog történni.

A *Stack* osztálynak olyan *clone* implementációra van szüksége, amely lemásolja a *Stack* objektum adattagjait, ezzel biztosítva a megfelelő tartalom szétválasztást:

```
public class Stack implements Cloneable {
    private Object[] items;
    private int top;
    ...
```

```
    protected Stack clone() {
        try {
            Stack s = (Stack)super.clone(); //clone the stack
            s.items = (Object)items.clone(); //clone the array
            return s; // return the clone
        } catch (CloneNotSupportedException e) {
           //This shouldn't happen because Stack is Cloneable.
            throw new InternalError();
        }
    }
}
```

Az implementáció viszonylag egyszerű. Először a *clone* metódus *Object* implementációja hívódik meg a *super.clone* segítségével, mely létrehoz és inicializál egy *Stack* objektumot. Ilyenkor mindkét objektum ugyanazokat az objektumokat tartalmazza. Ezután a metódus lemásolja az objektumokat, és a metódus *Stack*-el tér vissza.

Megjegyzés: A *clone* metódus nem a *new*-t használja a másolat létrehozásánál és nem hív *konstruktorokat*, helyette a *super.clone*-t használja, mely létrehozza az objektumot a megfelelő típussal, és engedélyezi a másolást, minek eredményeképpen a kívánt másolatot kapjuk.

Érdemes még azt is megfigyelni, hogy az adattag másolását sem „kézzel” végezte a metódus, hanem a tömb objektum *clone* metódusával. Ez a metódus egy másik azonos méretű tömböt hoz létre, aminek elemeiről is másolat készül. (A tömbben tárolt tagokról már nem fog másolat készülni, de ez nem is célja egy verem másolásnak.)

Az *equals* és *hashCode* metódusok

Az *equals* metódus két objektumot hasonlít össze és dönti el, hogy egyenlők-e vagy sem (ha egyenlők, *true*-val tér vissza). Ha önmagával hasonlítunk egy objektumot, *true*-t ad vissza.

A következő programrészlet összehasonlít két *Integer*-t:

```
Integer one = new Integer(1);
Integer anotherOne = new Integer(1);
if (one.equals(anotherOne)) {
    System.out.println("objects are equal");
}
```

A program kimenete:

```
objects are equal
```

Egyenlők, mivel az értékük megegyezik. Ha két objektum egyenlő az *equals* metódus szerint, akkor a *hashCode* metódus által szolgáltatott értékeknek is meg kell egyezniük. (Figyelem, fordítva ez nem feltétlenül igaz!)

Ha az *equals* működése nem megfelelő az osztályunk számára, akkor felül kell írnunk az osztályunkban.

A *hashCode* metódus állítja elő az objektumok hash kódját, ami például akkor lehet szükséges, ha az objektumot hashtáblában tároljuk. Hash kódként (a metódus visszatérési értékeként) mindig egy *int* típusú számot kapunk.

Helyes hash függvény írása egyszerű, azonban hatékony függvény írása nehéz lehet, komolyabb munkát igényel. Ez a téma azonban már nem fér bele a jegyzetünkbe.

A *finalize* metódus

Az *Object* osztály ugyancsak tartalmazza a *finalize* metódust. A szemétgyűjtő meghívja, ha már nincs egyetlen hivatkozás sem az objektumra. A *finalize* metódus automatikusan meghívódik, melyet a legtöbb osztály használ, ezért nem is kell külön meghívni.

A *finalize* metódussal legtöbbször nem kell törődnünk, az őstől örökölt metódus többnyire megfelelően működik.

A *toString* metódus

Az objektumot *String*-ként ábrázolja. Hasznos minden új osztály definíciója során felülírni, hogy a megfelelő értékeket reprezentálhassa. Használhatjuk a *toString*-et a *System.out.println*-nel együtt az objektumok szöveges megjelenítésére, pl.:

```
System.out.println(new Double(Math.PI).toString());
```

A futás eredménye:

```
3,14159
```

Nagyon hasznos ez a metódus akkor, ha a program tesztelési fázisában bizonyos objektumok tartalmát ellenőrizni szeretnénk. Ilyenkor csak ki kell írni a kérdéses objektumot, például a konzolra:

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

A *getClass* metódus

Visszaadja a futásidejű osztályát az objektumnak. Az *Object* osztály nem engedi meg a *getClass* metódus felüldefiniálását (*final*).

A következő metódus az objektum osztálynevét jeleníti meg:

```
void PrintClassName(Object obj) {
    System.out.println("The Object's class is "
                       + obj.getClass().getName());
}
```

A következő példa létrehoz az *obj* típusával megegyező másik objektum példányt:

```
Object createNewInstanceOf(Object obj) {
    return obj.getClass().newInstance();
}
```

Ha tudjuk az osztály nevét, kaphatunk egy *Class* objektumot az osztálynévből. A következő két sor egyaránt ugyanazon végeredményt produkálja (a második változat hatékonyabb):

```
String.class
Class.forName("String")
```

12.5. Végleges osztályok és metódusok

Végleges osztályok

A *final* kulcsszó segítségével deklarált változók értékét az inicializálás után nem lehet megváltoztatni, a leszármazott osztály nem módosíthatja, befolyásolhatja az eredeti működését. Fontos szempont a rendszer biztonságának növelése és az objektum orientált tervezés szempontjából.

Biztonság: Az egyik módszer, amit a hackerek használnak rendszerek feltörésénél, egy származtatott osztály létrehozása egy osztályból, majd helyettesítése az eredetivel. A származtatott osztály a metódushívás szempontjából úgy néz ki, mint az eredeti, de a viselkedése teljesen más is lehet, ami hibás működést eredményezhet. Ennek elkerülése érdekében deklarálhatjuk osztályunkat véglegessé, mely megakadályozza a származtatott osztályok létrehozását. A *String* osztály is végleges. Ez az osztály nélkülözhetetlen a Java Platform működéséhez. Ez biztosítja, hogy minden *String* a megfelelő módon működjön.

Ha megpróbáljuk lefordíttatni egy *final* osztály leszármazott osztályát, hibaüzenetet fogunk kapni.

Tervezés: Az objektumorientált tervezésnél érdemes megállapítani, hogy mely osztályokat szeretnénk véglegessé tenni, és tegyük is az adott osztályokat véglegessé a *final* módosító segítségével:

```
final class ChessAlgorithm {
    ...
}
```

Minden leszármaztatási próbálkozás hibás lesz.

Végleges metódusok

A *final* kulcsszót használjuk a deklarációban, ha azt akarjuk elérni, hogy ne lehessen a metódust származtatott osztályban felüldefiniálni. Az *Object* metódusai közül van, amelyik *final* típusú, és van, amelyik nem.

A következő példában a *ChessAlgorithm* osztályban a *nextMove* metódus tesszük véglegessé:

```
class ChessAlgorithm {
    ...
    final void nextMove(ChessPiece pieceMoved,
                        BoardLocation newLocation) {
        ...
    }
    ...
}
```

12.6. Ellenőrző kérdések

- Mit jelent az, hogy egyik osztály leszármazottja a másiknak?
- Lehet-e egy osztályreferenciát a szülőosztály felé konvertálni?
- Lehet-e egy osztályreferenciát a leszármazott osztály felé konvertálni?
- Lehet-e Javában különböző típusú értékek között értékadás? Ha igen, mikor?
- Ha létrehozunk egy példányt, és egy szülőosztály típusa szerinti referenciával hivatkozunk rá, a szülőosztály vagy a leszármazott osztály szerinti metódus hívódik-e meg?
- Mit jelent az osztályok újrafelhasználhatósága? Hogyan valósul meg Javában?
- Az osztály mely tagjait örökli a leszármazott osztály?
- Mikor lehet egy metódust a leszármazottban elfedni (elrejteni)?

- Hogyan lehet hivatkozni a leszármazott osztályban az ős elrejtett adattagjára?
- Mire használható a *super* kulcsszó?
- Milyen esetben szükséges az ősosztály konstruktorát explicit meghívni?

Igaz vagy hamis? Indokolja!

- Egy Java fordítási egységben pontosan egy osztály szerepel.
- Bármely *.class* kiterjesztésű állományt lehet közvetlenül futtatni.
- Lehet-e eltérés az ősben definiált metódus és a leszármazottban felülírt változat láthatóságában?
- Lehet olyan metódus, amelyet egy leszármazottban nem lehet felülírni?
- Végleges osztálynak kell-e végleges metódust tartalmazni?
- A végleges metódust tartalmazó osztály maga is végleges?

Melyik egy publikus, absztrakt metódus helyes deklarációja?

- *public abstract void add();*
- *public abstract void add() {}*
- *public virtual add();*

A leszármazott osztály konstruktorában hova kell írni a szülőosztály konstruktorának hívását?

- akárhova
- a konstruktor első sorába
- a konstruktor utolsó sorába
- nem kell meghívni

12.7. Gyakorló feladatok

Készítsen absztrakt *Jármű* osztályt, amely más speciálisabb jármű osztályok (pl. Autó) közös őse lehet!

A konstruktornak lehessen megadni, hogy alkalmas-e szárazföldi, vízi és légi közlekedésre (külön-külön, tehát 3 logikai paraméterrel). Tárolja el ezeket az adatokat, hogy később lekérdezhetők legyenek.

Készítsen *szárazföldi*, *vízi* és *légi* metódusokat a konstruktorban beállított értékek lekérdezésére.

Készítsen *Autó* osztályt, amely az őséhez képest tudja tárolni az utasok számát is, és a konstruktorán keresztül ezt be lehessen állítani. Készítsen ehhez is lekérdező metódust.

13.Beágyazott osztályok

Megadhatunk egy osztályt egy másik osztály tagjaként. Egy ilyen osztályt beágyazott osztálynak hívunk, és a következőképpen néz ki:

```
class EnclosingClass {
    ...
    class ANestedClass {
        ...
    }
}
```

A beágyazott osztályokat arra használjuk, hogy kifejezzük és érvényesítsük két osztály között a kapcsolatot. Megadhatunk egy osztályt egy másik osztályon belül, hogyha a beágyazott osztálynak a magába foglaló osztályon belüli környezetben van értelme. Pl. a szövegkurzornak csak a szövegkomponensen belüli környezetben van értelme.

A beágyazó osztály tagjaként a beágyazott osztály kiváltságos helyzetben van. Korlátlan hozzáféréssel rendelkezik a beágyazó osztályok tagjaihoz még akkor is, hogy ha azok privátként vannak deklarálva. Azonban ez a speciális kiváltság nem mindig speciális. A hozzáférést biztosító tagok korlátozzák a hozzáféréseket az olyan osztálytagokhoz, amelyek a beágyazó osztályon kívül esnek. A beágyazott osztály a beágyazó osztályon belül található, ebből kifolyólag hozzáférhet a beágyazó osztály tagjaihoz.

Mint ahogyan más tagokat is, a beágyazott osztályokat is statikusként, avagy nem statikusként lehet deklarálni, ezért ezeket pontosan így is hívják: **statikus beágyazott osztály**. A nem statikus beágyazott osztályokat **belső osztály**oknak hívjuk.

```
class EnclosingClass {
    ...
    static class StaticNestedClass {
        ...
    }
    class InnerClass {
        ...
    }
}
```

Ahogy a statikus metódusok és változók esetén, amelyeket mi osztálymetódusoknak és változóknak hívunk, a statikus beágyazott osztályt a beágyazó osztályával kapcsoljuk össze. Ahogy az osztálymetódusok, a statikus beágyazott osztályok sem hivatkozhatnak közvetlenül olyan példányváltozókra vagy metódusokra, amely az ő beágyazó osztályában van megadva. A példánymetódusok és változók esetén egy belső osztály az ő beágyazó osztályának a példányával kapcsolódik össze, és közvetlen hozzáférése van annak az objektumnak a példányváltozóihoz és metódusaihoz. Mivel egy belső osztályt egy példánnyal társítanak, ezért önmaga nem definiálhat akármilyen statikus tagot.

Hogy a továbbiakban könnyebb legyen megkülönböztetni a beágyazott osztály és a belső osztály fogalmát, érdemes ezekre a következőféleképpen tekinteni: a beágyazott osztály két osztály közötti szintaktikus kapcsolatot fejez ki, azaz szintaktikailag az egyik osztályban levő kód megtalálható a másikon belül. Ezzel ellentétben a belső osztály olyan objektumok közötti kapcsolatot fejez ki, amelyek két osztálynak a példányai. Tekintsük a következő osztályokat:

```
class EnclosingClass {
    ...
    class InnerClass {
        ...
    }
}
```

Az e két osztály közötti kapcsolatnál nem az az érdekes, hogy az *InnerClass* szintaktikusan definiálva van az *EnclosingClass*-on belül, hanem az, hogy az *InnerClass*-nak a példánya csak az *EnclosingClass*-nak a példáján belül létezhet, és közvetlen hozzáférése van a példányváltozók és a beágyazó osztály példánymetódusaihoz.

A beágyazott osztályok mindkét fajtájával találkozhatunk a Java API-n belül, és kell is használni őket. Azonban a legtöbb beágyazott osztály, amelyeket használunk, valószínűleg belső osztály lesz.

13.1. Belső osztályok

Ahhoz, hogy megértsük, mi a belső osztály és mire jó, tekintsük át újra a *Stack* osztályt. Tegyük fel, hogy ehhez az osztályhoz hozzá szeretnénk adni egy olyan tulajdonságot, amely lehetővé teszi egy másik osztály számára, hogy kilistázza a veremben lévő elemeket vagy tagokat *java.util.Iterator* interfész használatával. Ez az interfész három metódus deklarációt tartalmaz:

```
public boolean hasNext();
public Object next();
public void remove();
```

Az *Iterator* definiálja azt az interfészt, ami végigmegy egyszer az elemeken egy megadott sorrend szerint, amelyet a következőképpen adunk meg:

```
while (hasNext()) {
    next();
}
```

A *Stack* osztály önmaga nem tudja végrehajtani az *Iterator* interfészt, mert az *Iterator* interfész API-ja bizonyos korlátokat szab: két különböző objektum nem listázhatja kis egyszerre a veremben lévő elemeket. Ugyanis nincs lehetőség arra, hogy megtudjuk, ki hívja meg a következő metódust, és a listázást nem lehet újraindítani, mert az *Iterator* interfésznek nincsen olyan metódusa, amely támogatná ezt. Így a kilistázást csak egyszer lehet végrehajtani, mert az *Iterator* interfésznek nincs arra metódusa, hogy az elejéhez visszamenjen a bejárás. Ehelyett egy belső osztály van, amely elvégzi a verem számára ezt a munkát. A belső osztály hozzáférhet a verem elemeihez, és már csak azért is hozzá kell, hogy tudjon férni, mert a veremnek a publikus interfésze csak LIFO hozzáférést támogat. Itt lép be a képbe a belső osztály. Itt látható egy verem implementáció, ami definiál egy belső osztályt, amelyet *StackIterator*-nak hívunk, és kilistázza a verem elemeit.

```
public class Stack {
    private Object[] items;

    public Iterator iterator() {
        return new StackIterator();
    }
```

```
    class StackIterator implements Iterator {
        int currentItem = items.size() - 1;
        public boolean hasNext() {
            ...
        }
        public Object next() {
            ...
        }
        public void remove() {
            ...
        }
    }
}
```

Figyeljük meg, hogy a *StackIterator* osztály közvetlenül hivatkozik a veremben levő elemek példányainak változóira. A belső osztályokat elsősorban arra használjuk, hogy implementáljuk a segítő osztályokat, úgy, mint itt, ebben a példában is láthatjuk. Ha GUI eseményeket akarunk használni, akkor ismernünk kell a belső osztályok használatának a szabályait, mert az eseménykezelés mechanizmusa elég erőteljesen használja ezeket.

Deklarálhatunk egy belső osztályt anélkül, hogy nevet adnánk neki. Az ilyen osztályokat névtelen osztályoknak hívjuk. Itt látható még egy változata a *Stack* osztálynak, ebben az esetben névtelen osztályt használunk az *Iterator* számára.

```
public class Stack {
    private Object[] items;

    public Iterator iterator() {
        return new Iterator() {
            int currentItem = items.size() - 1;

            public boolean hasNext() {
                ...
            }

            public Object next() {
                ...
            }

            public void remove() {
                ...
            }
        }
    }
}
```

A névtelen osztályok nehézkessé tehetik a kód olvasását, ezért csak olyan osztályokhoz érdemes használnunk, amelyek nagyon kicsik (nem több mint egy-két metódus), és amelyek használata eléggé egyértelmű, mint például az eseménykezelő osztály.

13.2. Néhány további érdekesség

Más osztályokhoz hasonlóan a beágyazott osztályokat is lehet absztraktnak vagy véglegesnek deklarálni. Ennek a két módosítónak a jelentése a beágyazott osztály esetén is ugyanaz, mint más osztályoknál.

Ugyanígy használhatjuk a hozzáférés módosítókat – mint például a *private*, *public* és *protected* –, hogy korlátozzuk a beágyazott osztályokhoz való hozzáférést, mint ahogy minden más osztálytag esetén tehetjük azt. Nem csak a névtelen, hanem bármilyen be-

ágyazott osztály deklarálható bármilyen kódblokkon belül. A beágyazott osztály, amely egy metóduson vagy egy másik kódblokkon belül van deklarálva, hozzáférhet bármilyen hatókörön belüli végleges vagy lokális változóhoz.

13.3. Ellenőrző kérdések

A következő kódban melyik nem kerülhet az XX helyére?

```
public class MyClass1 {
    public static void main(String argv[]){ }
    XX class MyInner {}
}
```

- *public*
- *private*
- *static*
- *friend*

14.Felsorolás típus

A felsorolás típus egy olyan típus, melynek megengedett értékei fix konstansokból állnak. Javában a felsorolás típust *enum* szóval definiáljuk. Pl. a hét napjait így definiálhatjuk:

```
enum Days {
  VASARNAP, HETFO, KEDD, SZERDA, CSUTORTOK, PENTEK, SZOMBAT};
}
```

Vegyük észre, hogy konvencionálisan a nevek a felsorolás típusnál nagy betűkkel írandók, így a kód olvasásánál könnyen felismerhetők a konstansok.

Bármikor használhatunk felsorolás típust, ha szükségünk van kötött konstans értékekre. Ez magában foglalja a természetes felsorolás típusokat, mint a Naprendszer bolygói, a hét napjai, kártyapakli lapjainak neve/értéke és minden olyan esetet, ahol az összes lehetséges étéket tudjuk a fordításkor.

A Java programozási nyelv felsorolás típusa sokkal hatékonyabb, mint más nyelvekben szereplő megfelelői, melyek csak nevesített egész számok. Az *enum* deklaráció egy osztályt definiál, úgynevezett *enum* típust, melynek legfontosabb jellemzői a következők:

- Beszédesebbek az egyszerű literáloknál
- Típusbiztosak
- Saját névterük van
- Érdemes *switch-case* szerkezetben felsorolási típus alapján szervezni
- Van egy statikus *values* metódusuk, mely egy tömböt ad vissza, melyben a típus értékei szerepelnek deklarálási sorrendben. Ez a módszer pl. *for-each* ciklussal nagyon hasznos.
- Tartalmazhat metódusokat, adattagokat, implementálhat interfészeket stb.
- Minden *Object* metódust implementálnak. Összehasonlíthatók és szerializálhatók.

A következő példában a *Planet* egy felsorolás típus, mely a Naprendszer bolygóit jeleníti meg. A bolygónak van egy konstans tömeg és sugár paramétere. Minden konstanst tömeggel és sugárral deklarálnak, melyet átadnak a konstruktornak a létrehozáskor. Vegyük észre, hogy a felsorolás típus konstruktora értelemszerűen privát. Ha megpróbálunk egy publikus konstruktort létrehozni a felsorolás típusnak, akkor a fordító hibaüzenetet fog visszaadni.

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7),
    PLUTO   (1.27e+22,  1.137e6);
```

```
    private final double mass;   //in kilograms
    private final double radius; //in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    public double mass()   { return mass; }
    public double radius() { return radius; }
    public static final double G = 6.67300E-11;
    public double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    public double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
}
```

A bolygónak az értékei mellett metódusai is vannak, melyeken keresztül kinyerhető a felszíni gravitáció és egy tárgy súlya minden bolygón. A következő példaprogram egy ember Földön mért súlya alapján kiszámolja, majd kiírja ugyanennek az embernek a súlyát minden bolygón.

```
public static void main(String[] args) {
    double earthWeight = Double.parseDouble(args[0]);
    double mass = earthWeight/EARTH.surfaceGravity();
    for (Planet p : Planet.values()) {
        System.out.printf("Your weight on %s is %f%n",
                          p, p.surfaceWeight(mass));
    }
}
```

A kimenet:

```
$ java Planet 175
Your weight on MERCURY is 66.107583
Your weight on VENUS is 158.374842
Your weight on EARTH is 175.000000
Your weight on MARS is 66.279007
Your weight on JUPITER is 442.847567
Your weight on SATURN is 186.552719
Your weight on URANUS is 158.397260
Your weight on NEPTUNE is 199.207413
Your weight on PLUTO is 11.703031
```

Csak egyetlen megkötése van a felsorolás típusnak: habár a felsorolás típusok osztályok, nem definiálható hierarchia számukra. Más szavakkal: nem lehet leszármazottja a felsorolási típusnak.

Végezetül a *java.util* csomag tartalmaz két speciális *Set* és *Map* implementációt, mely támogatja a felsorolás típusokat, az *EnumSet*-et és az *EnumMap*-et.

14.1. Ellenőrző kérdések

- Mikor érdemes felsorolás típust alkalmazni?
- Milyen korlátozások vannak a felsorolás típusra más osztályokhoz képest?

15.Általános programozás

Az általános (generikus) programozás típusparaméterek segítségével teszi lehetővé, hogy osztályokat, interfészeket hozhassunk létre úgy, hogy bizonyos paraméterek típusait a példányosításkor dönthessük el.

15.1. Általános típus definiálása és használata

Ahhoz, hogy általános típust definiáljunk, a típusdefiníciónak tartalmaznia kell egy vagy több típus paramétert a típus neve után. A típus paraméterek vesszővel elválasztott listái ’<’ és ’>’ között szerepelnek. Konvencionálisan a típus paraméterek nagybetűsek. A típus paraméterek aztán megjelennek a típus metódusaiban, vagy a metódus paraméterlistájában, vagy visszatérési érték típusként. A gyakorlatban a legtöbb fejlesztőnek nincs szüksége új általános típusok definiálására, de szükséges megjegyezni a szintaxisát és a használatát az általános típusnak.

Nézzük újra át a *Stack* osztályt, melyet az osztályok létrehozásánál mutattunk be. A generikus verzió (*Stack2<T>*), egy gyűjteményt használ (*ArrayList<T>*), hogy a veremben tárolja az érékeket.

```
public class Stack2<T> {
    private ArrayList<T> items;
    ...
    public void push(T item) {...}
    public T pop() {...}
    public boolean isEmpty() {...}
}
```

Vegyük észre, hogy a *T* típus paraméter bevezetésre került az osztálynév után, és ezek után feltűnik, mint a *push* metódus paraméter típusa, és a *pop* metódus visszatérési típusa.

A gyűjtemények gyakran használatosak arra, hogy bemutassuk az általános típus használatát, mivel nagyon jellemzőek az interfészekben és osztályokban. A valóságban a gyűjtemények voltak a fő motivációs erő az általános típusok bevezetésénél a Java nyelvben, mivel elérhetővé teszik a fordítás idejű ellenőrzését a gyűjteményeken végzett műveletek típusbiztonságának. Amikor specifikáljuk egy gyűjteményben tárolt objektum típusát:

- a fordító tud ellenőrizni bármilyen műveletet, ami egy objektumot ad a gyűjteményhez
- ismert az objektum típusa, mely a gyűjteményből lett kinyerve, ezért nincs szükség arra, hogy *cast*-oljuk (átalakítsuk) típussá. Ugyanakkor nincs lehetőség arra sem, hogy átalakítsunk egy rossz típussá, és ekkor megtapasztaljunk egy futás idejű *ClassCastException* kivételt.

Ahogy az előbbiekben írtuk, ha egy általános típust használunk, helyettesítjük a paraméter egy aktuális típus paraméterét, nagyjából ugyanezen módszerrel helyettesítünk egy metódushoz a paramétereihez tartozó aktuális értékeket. Egy aktuális típus paraméternek referencia típusnak kell lenni, nem lehet primitív. Például, itt látható az, hogy hogyan lehet létrehozni *Stack2*-t *String* típus paraméterrel; és ezek után *push*-olni és *pop*-olni a *„hi”* *String*-et.

```
Stack2<String> s = new Stack2<String>();
s.push("hi");
String greeting = s.pop(); //no cast required here
```

Egy nem generikus verem kód így nézne ki:

```
Stack s = new Stack();
s.push("hi");
String greeting = (String)s.pop(); //cast required here
```

Vegyük észre, hogy amikor az általános típust használjuk, akkor a fordító egy olyan technikával fordítja le az általános típust, amit típus törlésnek hívnak.

Gyakorlatilag a fordító törli az összes olyan információt, mely a típus paraméterrel, vagy a típus paraméterekkel kapcsolatos. Például a *Stack2<string>* típust lefordítja *Stack2*-re, melyet nyers típusnak neveznek. Abból következtethetünk a típus törlésre, hogy a típus paraméter nem érhető el futásidőben ahhoz, hogy használjuk típuskényszerítésben, vagy mint az *instanceof* eljárás paramétere.

15.2. Kapcsolatok az általános típusok között

Valószínűleg azt várjuk, hogy a *Stack2<Object>* a szülő típusa a *Stack2<String>*-nek, mert az *Object* szülője a *String*-nek. A valóságban nem létezik ilyen kapcsolat az általános típusok példányai között. A szülő-gyermek kapcsolat hiánya az általános típus példányai között nehézkessé teheti a többalakú (polimorf) eljárások megírását.

Tegyük fel, hogy szeretnénk megírni egy eljárást, ami kiírja egy gyűjteményben tárolt objektumokat az objektumok típusától függetlenül a konzolra.

```
public void printAll(Collection<Object> c) {
    for (Object o : c) {
        System.out.println(o);
    }
}
```

Választhatnánk, hogy létrehozunk egy *String* listát és ezt a metódust használjuk az összes *String* kiíratására:

```
List<String> list = new ArrayList<String>();
...
printall(list);   //error
```

Ha ezt használjuk, akkor észre fogjuk venni, hogy az utolsó sor fordítási hibát ad. Mivel az *ArrayList<String>* nem leszármazott típusa a *Collection<Object>*-nek, ezért nem adható át, mint paraméter a kiíratási eljárásnak annak ellenére, hogy a két típus ugyanannak az általános típusnak a leszármazottai, rokon öröklött típus paraméterekkel. Más részről az öröklés miatt kapcsolatban álló általános típusok ugyanazzal a típus paraméterrel kompatibilisek:

```
public void printAll(Collection<Object> c) {
    for (Object o : c) {
        System.out.println(o);
    }
}

List<Object> list = new ArrayList<Object>();
...
printall(list);   //this works
```

A *List<Object>* kompatibilis a *Collection<Object>*-el, mert a két típus példánya egy általános szülőtípusnak és annak leszármazott típusának, és a példányok ugyanahhoz a típus paraméterhez tartoznak, konkrétan az *Object*-hez.

15.3. Helyettesítő típus

Ahogy körbejárjuk a *printAll* metódus első változata által felvetett kérdést, leszögezhetjük, hogy a *printAll* paramétere egy gyűjtemény, melynek elemi típusa lehet bármi, amit *Collection<?>* formában leírhatunk:

```
public void printAll(Collection<?> c) {
    for (Object o : c) {
        System.out.println(o);
    }
}
```

A *?*-es típus határozatlan típusként ismert. A gyűjteményből bármikor kiolvasható az objektum, mert a visszatérési érték mindig garantáltan *Object*. Azonban nem adható objektum a gyűjteményhez, mert a *?* ismeretlen típust jelöl, és nem lehet egyértelműen tudni, hogy a hozzá adni kívánt objektum leszármazott típusa-e az ismeretlen típusnak. Az egyetlen kivétel a *null*, amely eleme minden típusnak.

Korlátozhatjuk (vagy kényszeríthetjük) is a helyettesítő típust valamely típus segítségével. A korlátozott helyettesítő típus akkor hasznos, mikor csak részben ismerjük a paramétereket. Például tegyük fel, hogy van egy osztály hierarchiánk geometriai alakzatokból (*Shape*), és ennek leszármazott típusaiból (*Circle*, *Rectangle*, és így tovább). A rajzoló program, ami ezekre az objektumokra hivatkozik, meghív egy *drawAll* nevű metódust, hogy egy gyűjteményt rajzoljon ezekből az alakzatokból:

```
public void drawAll(Collection<Shapes> shapes) {
    for (Shape s: shapes) {
        s.draw();
    }
}
```

Mivel láthattuk, hogy nem megengedett a *Shape* típus valamely leszármazott típusának (például a *Circle*) alkalmazása, ezért ez a metódus csak korlátozottan használható: például nem hívható meg a *Collection<Circle>* esetén. Hogy lehetővé tegyük a *Shape* típus valamely leszármazott típusának típus paraméterként való alkalmazását, kiterjeszthetjük az alakzatok gyűjtemény típus paraméterét helyettesítő típusként. De mivel tudjuk, hogy a típus paraméter valamilyen alakzat lesz, a helyettesítést korlátozhatjuk a *Shape* típusra a következőképpen:

```
void drawAll(Collection<? extends Shapes> shapes) { ... }
```

Ez lehetővé teszi a *drawAll* számára, hogy elfogadjon bármilyen, a *Shape* típus leszármazott típusából álló gyűjteményt.

Összefoglalva, a helyettesítő típus felső korlátozással specializálható a *<? extends Type>* módon, ezáltal megfelelővé téve az adott *Type* minden leszármazott típusához. A helyettesítő típust alulról is korlátozhatjuk. Egy alulról korlátozott helyettesítő a *<? super Type>* módon írható le, és a *Type* minden ősosztályára használható. Megfigyelhetjük, hogy továbbra sem lehetséges ismeretlen típusú gyűjteményhez objektumot hozzáadni, és ez nem lehetséges korlátozott, ismeretlen típusú gyűjtemény esetén sem.

15.4. Általános metódusok definiálása és használata

Nem csak a típusokat, a metódusokat is paraméterezhetjük. A statikus és nem statikus metódusoknak épp úgy, ahogy a konstruktoroknak, lehet típus paraméterük.

A metódusok típus paraméter deklarálásának szintaxisa megegyezik az osztályoknál használt szintaxissal. A típusparaméter listát a kerek zárójelek közé kell helyezni, még a metódus visszatérési értékének típusa elé. Például a következő gyűjtemény osztály metódus feltölt egy *<? super T>* típusú listát *T* típusú objektumokkal:

```
static <T> void fill(List<? super T> list, T obj)
```

Az általános metódusok lehetővé teszik a típus paraméterek használatát, hogy egyértelművé tehesse a típusok kapcsolatát egy vagy több paraméter esetén, egy metódus, vagy annak visszatérési értéke számára (vagy mindkettőnek). Az általános metódusok típus paraméterei általában osztálytól függetlenek vagy interfész-szintű típus paraméterek. A *Collections* osztály algoritmusainak definíciója sok lehetőséget kínál az általános metódusok használatára.

Az egyetlen különbség az általános típusok és általános metódusok között az, hogy az általános metódusokat hagyományos metódusként hívjuk meg. A típus paramétereket a hívás függvényében állapítjuk meg, ahogy a *fill* metódus alábbi hívásánál is:

```
public static void main(String[] args) {
    List<String> list = new ArrayList<String>(10);
    for (int i = 0; i < 10; i++) {
        list.add("");
    }
    String filler = args[0];
    Collections.fill(list, filler);
    ...
}
```

15.5. Általános típusok használata az öröklésben

A fejezetben már korábban tárgyalt *Cat* osztálynak tekintsük a *getLitter* nevű metódusát, melynek visszatérési értéke egy *Cat* objektumokból álló gyűjtemény:

```
public Collection getLitter(int size) {
    ArrayList litter = new ArrayList(size);
    for (int i = 0; i < size; i++) {
        litter.add(i, new Cat());
        return litter;
    }
}
```

Megfigyelhető, hogy a *Collection* objektum határozatlan típusú: a gyűjteményben lévő objektumok típusa nem meghatározott. Azonos helyzet állt fenn minden további metódusnál is, melyek objektumot adnak vissza, még mielőtt az általános típusok elérhetővé váltak volna a Java 5.0-ás verzióban.

Tegyük fel, hogy írunk egy programot *Cats* néven, mely ezt a visszatérési értéket adja át egy gyűjteménynek, melynek típusa kifejezetten *Cat* kell, hogy legyen:

```
public static void main(String[] args) {
Collection<Cat> litter = myCat.getLitter(3);
for (Cat c : litter) {
    System.out.println(c.getColor());
}
```

Mikor lefordítja a *Cats.java* állományt, a következő figyelmeztetést kapja:

```
A Cats.java ellenőrizetlen vagy nem biztonságos műveleteket
használ. Fordítsa le újra a -Xlint:unchecked kapcsolóval a
részletek megtekintéséhez.
```

Az *Xlint:checked* kapcsoló használata információgyűjtéshez:

```
% javac -Xlint:unchecked Cats.java
Cats.java:5: warning: [unchecked] unchecked conversion
found   : java.util.Collection
required: java.util.Collection<Cat>
                Collection<Cat> litter = myCat.getLitter(3);
                                                     ^
```

Összegezve, ha a *Cat*-et olyan fordítóval fordítjuk újra, mely támogatja a típus paramétereket, a következő figyelmeztetést kapjuk:

```
% javac -Xlint:unchecked Cat.java
Cat.java:19: warning: [unchecked] unchecked call to
add(int,E) as a member of the raw type java.util.ArrayList
                                 litter.add(i, new Cat());
                                           ^
```

Habár a kód hibátlan, ez a figyelmeztetés mutatja, hogy a fordító nem tudja biztosítani a művelet pontosságát, mikor speciális típusú gyűjteményeket használunk. Amikor „ellenőrizetlen” figyelmeztetést kap, ellenőriznie kell, hogy a művelet, mely a figyelmeztetést generálta, valóban megfelelő-e.

Végül tekintsünk át egy listát a teljes Stack2 osztályról:

```
public class Stack2<T> implements Cloneable {
    private ArrayList<T> items;
    private int top=0;

    public Stack2() {
        items = new ArrayList<T>();
    }
    public void push(T item) {
        items.add(item);
        top++;
    }

    public T pop() {
        if (items.size() == 0)
        throw new EmptyStackException();
        T obj = items.get(--top);
        return obj;
    }

    public boolean isEmpty() {
        if (items.size() == 0)
            return true;
        else
            return false;
    }
```

```
    protected Stack2<T> clone() {
        try {
            Stack2<T> s = (Stack2<T>)super.clone();
            s.items =  (ArrayList<T>)items.clone();
            return s; // Return the clone
        } catch (CloneNotSupportedException e) {
            throw new InternalError();
        }
    }
}
```

Megfigyelhetjük, hogy a *clone* metódus a *clone* metódusokat az ősosztályából és a tartalmazott listájából hívja meg. A *clone* metódusok öröklődő metódusok, mert még az általános típusok elérhetősége előtt definiálva lettek.

Mikor lefordítja a *Stack2.java* állományt, a következő figyelmeztetést kapja:

```
% javac -Xlint:unchecked Stack2.java
Stack2.java:32: warning: [unchecked] unchecked cast
found   : java.lang.Object
required: Stack2<T>
         Stack2<T> s = (Stack2<T>)super.clone();
                                             ^
Stack2.java:33: warning: [unchecked] unchecked cast
found   : java.lang.Object
required: java.util.ArrayList<T>
         s.items =  (ArrayList<T>)items.clone();
                                             ^
2 warnings
```

Ez a figyelmeztetés mutatja, hogy a fordító nem képes biztosítani az eljárás kifogástalan működését. Más szóval, mivel a *clone* metódust úgy definiálták, hogy egy *Object* osztálybeli objektumot adjon visszatérési értéknek, ezért a fordító nem tudja biztosítani, hogy a gyűjtemény visszatérési értéke *Stack2<T>* legyen. Azonban a *clone* metódus feletti megállapodás szerint a művelet engedélyezett, habár „ellenőrizetlen” figyelmeztetést okoz.

Rengeteg apró dologra kell figyelni az általános típusok használatánál az öröklésben.

15.6. Ellenőrző kérdések

- Mikor érdemes típus paramétert alkalmazni?
- Mi az előnye a típus paraméter alkalmazásának?

16.Interfészek

Az interfész olyan viselkedéseket definiál, amelyet az osztályhierarchia tetszőleges osztályával megvalósíthatunk. Egy interfész metódusok halmazát definiálja, de nem valósítja meg azokat. Egy konkrét osztály megvalósítja az interfészt, ha az összes metódusát megvalósítja.

Definíció: Az interfész implementáció nélküli metódusok névvel ellátott halmaza.

Mivel az interfész a megvalósítás nélküli, vagyis absztrakt metódusok listája, alig különbözik az absztrakt osztálytól. A különbségek:

- Az interfész egyetlen metódust sem implementálhat, az absztrakt osztály igen.
- Az osztály megvalósíthat több interfészt, de csak egy ősosztálya lehet.
- Az interfész nem része az osztályhierarchiának. Egymástól "független" osztályok is megvalósíthatják ugyanazt az interfészt.

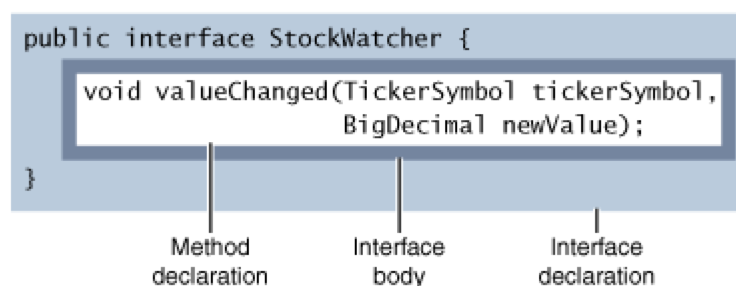
Ebben a fejezetben egy olyan példaprogramot fogjuk használni, ami a Tervezési minták (*Design Patterns*) között szokás emlegetni megfigyelő (*Observer*) néven. Ez az osztály megengedi más osztályoknak, hogy regisztrálják magukat bizonyos adatai változásának figyelésére. A *StockApplet* osztály fog megvalósítani egy olyan metódust, amivel regisztrálni tud a változás figyeléséhez:

```
public class StockMonitor {
     public void watchStock(StockWatcher watcher,
                            TickerSymbol tickerSymbol,
                            BigDecimal delta) {
     ...
     }
}
```

Az első paraméter egy *StockWatcher* objektum. A *StockWatcher* annak az interfésznek a neve, amelyet hamarosan látni fogunk. Az interfész egyetlen *valueChanged* nevű metódust definiál. Egy objektum akkor tudja magát megfigyelőként regisztrálni, ha az osztálya megvalósítja ezt az interfészt. Amikor a *StockMonitor* osztály érzékeli a változást, meghívja a figyelő *valueChanged* metódusát.

16.1. Interfész definiálása

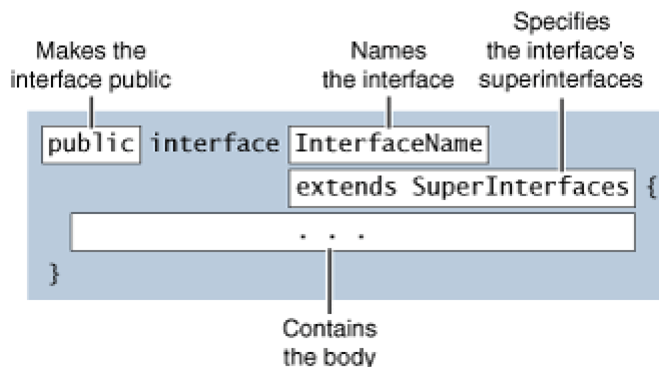
A következő ábra bemutatja az interfész definíció két összetevőjét: az interfész deklarációt és a törzset.



A *StockWatcher* interfész deklarálja, de nem implementálja a *valueChanged* metódust. Az interfészt megvalósító osztályok fogják a metódust implementálni.

Interfész deklaráció

A következő ábra az interfész deklaráció minden részét bemutatja:



Az interfész deklarációban két elem kötelező: az *interface* kulcsszó és az interfész neve. Ez után szerepelhetnek a szülőinterfészek.

Az interfész törzs

Az interfész törzs metódus deklarációkat tartalmaz ';'-el lezárva. Minden deklarált metódus értelemszerűen publikus és absztrakt (bár maguk a kulcsszavak nem írhatók ki).

Az interfész tartalmazhat konstans deklarációt is. Minden konstans értelemszerűen publikus, statikus és végleges. (A kulcsszavak itt sem írhatók ki.) Nem használhatók a *transient*, *volatile*, *synchronized*, *private* és *protected* módosítók sem.

16.2. Interfészek implementálása

Egy interfész viselkedési formákat definiál. Az interfészt megvalósító osztály deklarációjában szerepel az *implements* záradék. Az osztály akár egy vagy több interfészt is megvalósíthat. (Az interfészek között megengedett a többszörös öröklődés.)

Konvenció: az *implements* záradék az *extends* záradékot követi, ha mindkettő van.

A következő példa applet implementálja a *StockWatcher* interfészt.

```
public class StockApplet extends Applet
                         implements StockWatcher {

    public void valueChanged(TickerSymbol tickerSymbol,
                             BigDecimal newValue) {
        switch (tickerSymbol) {
            case SUNW:
                ...
                break;
            case ORCL:
                ...
                break;
            case CSCO:
                ...
                break;
            default:
                // handle unknown stocks
                ...
                break;
        }
    }
}
```

Amikor egy osztály megvalósít egy interfészt, akkor alapvetően aláír egy szerződést. Az osztálynak implementálni kell az interfészben és szülőinterfészeiben deklarált összes metódust, vagy az osztályt absztraktként kell deklarálni. (Az ilyen absztrakt osztályoknak csak akkor lehet nem absztrakt egy leszármazottja, ha az ezen az öröklési szinten meg nem valósított metódusokat már maradéktalanul megvalósítja.)

A *StockApplet* megvalósítja a *StockWatcher* interfészt, azaz szolgáltatásként nyújtja a *valueChanged* metódust.

16.3. Az *interface* használata típusként

Amikor új interfészt definiálunk, referenciát definiálunk egy új adattípushoz. Az interfész nevét akárhol lehet használni, ahol egyéb adattípus is előfordulhat.

```
public class StockMonitor {
    public void watchStock(StockWatcher watcher,
                           TickerSymbol tickerSymbol,
                           BigDecimal delta) {
    ...
    }
}
```

16.4. Ellenőrző kérdések

Igaz vagy hamis? Indokolja!

- Az interfész csak metódust tartalmazhat, változót nem.
- Minden osztály legfeljebb egy interfészt implementálhat.
- Az interfész összes változója implicit *static*.
- Az interfésznek nincs konstruktora.

- Az interfész összes metódusa implicit *public*.
- Ha egy osztály megvalósít egy interfészt, azt a *uses* kulcsszóval kell jelezni.
- Az interfész összes metódusa implicit *abstract*.
- Az interfész összes változója implicit *public*.
- Az interfészt megvalósító osztálynak minden, az interfészben deklarált metódust implementálni kell.
- Létre lehet hozni interfész-referenciát.
- Minden interfésznek van őse.
- Az interfész összes változója implicit *final*.
- Minden interfészt legalább egy osztályban meg kell valósítani.
- Az interfész metódusainak nincs törzse.
- Az interfész és az absztrakt osztály között nincs lényeges különbség.

17. Csomagok

A típusok könnyebb megtalálásához és használatához, névütközések elkerüléséhez és az elérés szabályozásához a programozók egybe csomagolhatják az összetartozó típusaikat csomagokká.

Definíció: A csomag összetartozó típusok gyűjteménye.

A Java platform típusai funkciók szerint különböző csomagokba vannak szervezve: az alapvető osztályok a *java.lang* csomagban, az I/O osztályok a *java.io*-ban, és így tovább. Ezen kívül a saját típusainkat is tehetjük csomagokba.

A következő osztályokat megvizsgálva látszik, hogy közös csomagba érdemes őket sorolni, mivel grafikus objektumok csoportjába tartoznak a körök, téglalapok, vonalak és pontok. Ha írunk egy *Draggable* interfészt, az azt megvalósító osztályok lehetővé teszik a vonszolást is.

```
//in the Graphic.java file
public abstract class Graphic {
    . . .
}
//in the Circle.java file
public class Circle extends Graphic implements Draggable {
    . . .
}
//in the Rectangle.java file
public class Rectangle extends Graphic implements Draggable {
    . . .
}
//in the Draggable.java file
public interface Draggable {
    . . .
}
```

A következő okok miatt érdemes az osztályokat és interfészeket közös csomagba helyezni:

- Más programozók számára is látszik, hogy kapcsolódó típusokról van szó.
- Más programozók is láthatják, hol kell keresni a grafikához kapcsolódó osztályokat.
- A típusaink nevei nem kerülnek összeütközésbe más csomagok neveivel, mert a csomagok önálló névtereket hoznak létre.
- A típusaink korlátlanul láthatják egymást, de egyéb típusok csak korlátozottan férhetnek a típusokhoz.

17.1. Csomag létrehozása

A csomag létrehozásához mindössze bele kell tenni egy típust (osztály, interfész). A csomag megnevezést a forrásállomány elején, a típusdefiníciók előtt kell megtenni.

```
package graphics;

public class Circle extends Graphic implements Draggable {
    . . .
}
```

Ez után a *graphics* csomag összes forráskódja elején ugyanezt a csomagmegjelölést kell alkalmaznunk, hogy közös csomagba kerüljenek:

```
package graphics;

public class Rectangle extends Graphic implements Draggable {
    . . .
}
```

Ha nem alkalmazunk csomagmegjelölést, akkor az osztály(ok) egy úgynevezett alapértelmezett név nélküli csomagba kerülnek.

17.2. Egy csomag elnevezése

Az egész világon írnak a programozók a Java programnyelvhez osztályokat, interfészeket, kulcsszavakat és megjegyzéseket, és valószínűleg két programozó ugyanazt a nevet két különböző feladatú osztálynál fogja használni. Valójában, az előző példa esetén, amikor definiálunk egy *Rectangle* osztályt, akkor a *Rectangle* osztály már benne van a *java.awt* csomagban. A fordító mégis engedélyezi két osztálynak ugyanazt a nevet. Miért? Mert azok különböző csomagokban vannak, és mindegyik osztálynak a teljes neve magába foglalja a csomag nevét. Tehát a *graphics* csomagban levő *Rectangle* osztály teljes neve *graphics.Rectangle*, és a *java.awt* csomagban levő *Rectangle* osztály teljes neve *java.awt.Rectangle*. Ez rendszerint csak akkor működik jól, hogyha két egymástól független programozó nem ugyanazt a nevet adja a csomagoknak. Mivel hárítható el ez a probléma? Megállapodással.

Megállapodás: Fordított *domain* (tartomány) nevet használnak a csomagok nevének, ilyen módon: *com.company.package*. Névütközés előfordulhat egyetlen cégen belül is, amit a cégnek le kell kezelni egy belső megállapodással. Lehet, hogy emiatt tartalmazni fog tartomány vagy projekt neveket a társaság neve után, mint például *com.company.region.package*.

17.3. Csomag tagok használata

Csak a publikus csomag tagok érhetőek el a csomagon kívül. Ahhoz hogy használjunk egy publikus csomag tagot a csomagján kívülről, a következők valamelyikét kell tennünk (vagy akár többet):

- A teljes (vagy más néven minősített) nevén keresztül kell hivatkoznunk rá
- Importáljuk a csomag tagot
- Importáljuk a tag teljes csomagját

Mindegyiket különböző szituációkban alkalmazhatjuk, amelyeket a következő részekben tisztázunk.

Név szerinti hivatkozás egy csomag tagra

Eddig a példákban a típusokra az egyszerű nevükön keresztül hivatkoztunk, mint például *Rectangle*. Akkor használhatjuk egy csomag tagjának az egyszerű nevét, ha az osztály ugyanabban a csomagban van, mint a tag, vagy ha a tag importálva van.

Ha megpróbálunk használni egy tagot egy másik csomagból, és a csomagot nem importáltuk, akkor a tag teljes nevét használnunk kell. A *Rectangle* osztály teljes neve:

```
graphics.Rectangle
```

A következőképpen használhatjuk a minősített nevet, hogy létrehozzuk a *graphics.Rectangle* osztály egy példányát:

```
graphics.Rectangle myRect = new graphics.Rectangle();
```

Ha minősített neveket használunk, valószínűleg bosszantó lesz, hogy újra és újra be kell gépelni a *graphics.Rectangle*-t. Ezen kívül rendezetlen és nehezen olvasható programot kapunk. Ilyen esetekben inkább importáljuk a tagot.

Egy csomag tag importálása

Ahhoz hogy importálhassuk a megadott tagot az aktuális fájlban, a fájl elején ki kell adni az *import* utasítást az osztály vagy interfész definiálása előtt, de a *package* utasítás után, ha van olyan. Úgy tudjuk importálni a *Circle* osztályt a *graphics* csomagból, hogy:

```
import graphics.Circle;
```

Most már tudunk hivatkozni a *Circle* osztályra az egyszerű nevével:

```
Circle myCircle = new Circle();
```

Ez a szemlélet akkor működik jól, ha csak néhány tagot használunk a *graphics* csomagból. De ha sok típusát használjuk egy csomagnak, akkor inkább importáljuk az egész csomagot.

Egy teljes csomag importálása

Ahhoz hogy egy csomagnak az összes típusát importáljuk, az *import* utasítást a csillag (*) helyettesítő karakterrel kell használnunk:

```
import graphics.*;
```

Most már hivatkozhatunk bármelyik osztályra vagy interfészre a *graphics* csomagból az egyszerű rövid nevével:

```
Circle myCircle = new Circle();
Rectangle myRectangle = new Rectangle();
```

Az *import* utasításban a csillag karakterrel az összes osztályát megadjuk a csomagnak. Nem használhatunk olyan megfeleltetést, amely egy csomagban egy osztály részhalmazára utal. Például helytelen megfeleltetés az, hogy a *graphics* csomagból az '*A*' betűvel kezdődő összes osztály importáljuk:

```
import graphics.A*;
```

Ez az utasítás fordítási hibát generál. Az *import* utasítással, egy csomagnak egyetlen tagját vagy egész csomagot importálhatunk.

Ugyanígy nem megengedett, hogy egyszerre több csomagot importáljunk, pl.:

```
import graphics.*.*;
```

Megjegyzés: Ha kevesebb tagot akarunk importálni, akkor megengedett, hogy csak egy osztályt, és annak a belső osztályait importáljuk. Például, ha a *graphics.Rectangle* osztálynak a belső osztályait akarjuk használni, mint *Rectangle.DoubleWide* és *Rectangle.Square*, a következő képen importálhatjuk:

```
import graphics.Rectangle.*;
```

Könnyítésül a Java fordító automatikusan importál három teljes csomagot:

- A névtelen csomagot (ha nem hozunk létre csomagot, vagyis nem használjuk a *package* utasítást)
- Az alapértelmezés szerinti aktuális csomagot (ha létrehozunk csomagot)
- A *java.lang* csomagot

Megjegyzés: A csomagok nem hierarchikusak. Ha például importáljuk a *java.util.**-ot, nem hivatkozhatunk a *Pattern* osztályra. Minden esetben úgy kell hivatkoznunk, hogy *java.util.regex.Pattern,* vagy ha importáljuk a *java.util.regex.**-ot, akkor csak egyszerűen *Pattern*.

Névütközés megszüntetése

Ha egy tag ugyanazon a néven megtalálható egy másik csomagban, és mindkét csomag importálva van, akkor a tagra teljes nevével kell hivatkoznunk. Például a *Rectangle* osztály definiálva van a *graphics* csomagban, és a *java.awt* csomagban is. Ha a *graphics* és *java.awt* is importálva van, akkor a következő nem egyértelmű:

```
Rectangle rect;
```

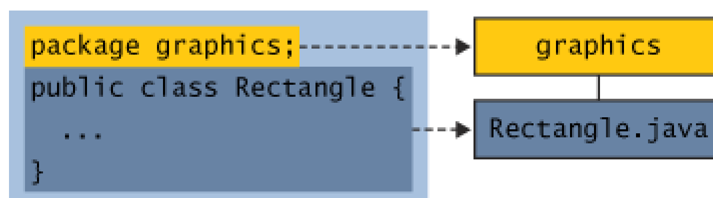
Az ilyen esetekben a minősített nevet kell használnunk, hogy pontosan meg tudjuk adni, melyik *Rectangle* osztályt akarjuk használni:

```
graphics.Rectangle rect;
```

17.4. Forrás és osztály fájlok menedzselése

A Java környezet sok megvalósításánál számít a hierarchikus fájlrendszer a forrás és osztály állományok menedzseléséhez, bár maga a Java nem igényli ezt. A stratégia a következő.

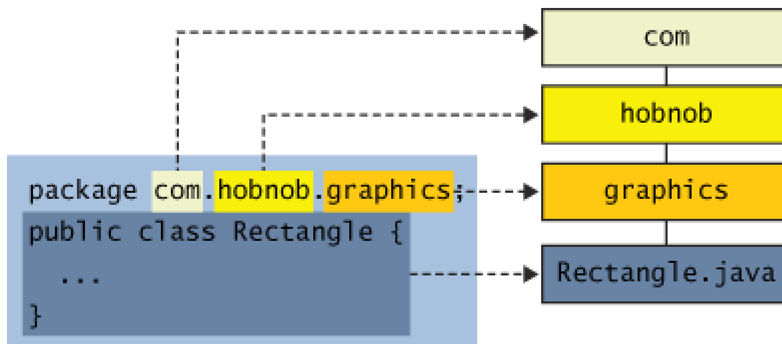
A forráskód egy osztályt, interfészt vagy felsorolt típust tartalmaz, amelynek neve a típus egyszerű neve és a kiterjesztése: *.java*. A forrásfájlt egy könyvtárba tesszük, amelynek a neve kifejezi a csomag nevét, amelyikhez a típus tartozik. Például, a forráskódban lévő *Rectangle* osztály adja a fájl nevét, *Rectangle.java*, és a fájl egy *graphics* nevű könyvtárban lesz. A *graphics* könyvtár bárhol lehet a fájl rendszeren. A lenti ábra mutatja, hogyan működik.



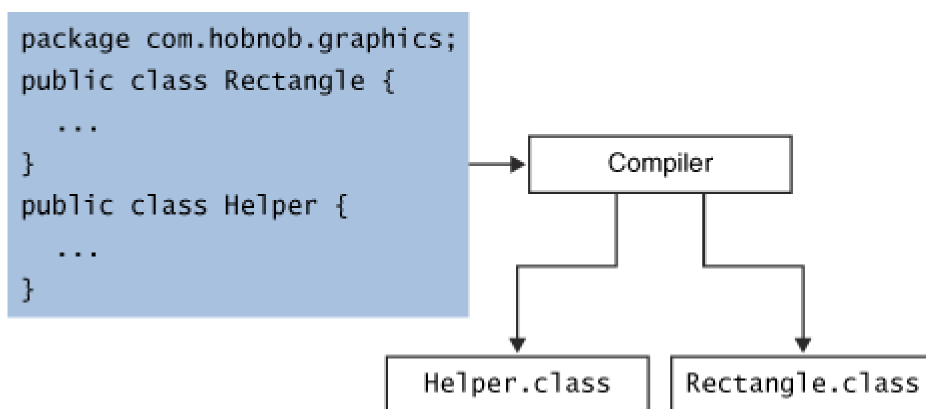
A csomag tag teljes neve és a fájl elérési útja hasonló:

Osztály név	*graphics.Rectangle*
Fájl elérési út	*graphics\Rectangle.java*

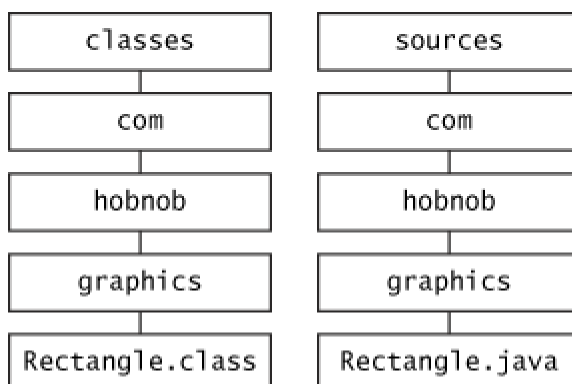
Emlékezzünk vissza, hogy a megállapodás értelmében a cégek fordított internet domén nevet használnak a csomagok elnevezésére. A példában a cég domén neve *hobnob.com*, ezért mindegyik csomagnak a nevét *com.hobnob* fogja megelőzni. A csomag névnek mindegyik összetevője megegyezik egy-egy alkönyvtárral. Így ha Hobnob-nak van egy *graphics* csomagja, amely tartalmazza a *Rectangle.java* forrásfájlt, akkor ezek alkönyvtárak sorozatában tárolódnak:



Mikor lefordítjuk a forrásállományt, a fordító létrehoz különböző kimeneti állományt minden egyes definiált osztálynak és interfésznek. A kimeneti állománynak az alap neve az osztály vagy interfész neve és a kiterjesztése: *.class*, ahogyan a következő ábra is mutatja:



A *.class* fájlnak ugyanúgy, mint egy *.java* fájlnak is kell a csomagokat kifejező könyvtárszerkezet. Azonban nem lehet ugyanabban a könyvtárban, mint a forrás. Külön-külön könyvtárba kell rendezni a forrást és az osztályt:



Ha ez alapján csináljuk, oda tudjuk adni az osztály könyvtárakat más programozóknak anélkül, hogy megmutatnánk a forrásunkat. Ekkor használhatjuk a Java fordítónak a –d opcióját az osztály fájlok megadásához, ehhez hasonlóan:

```
javac -d classes sources\com\hobnob\graphics\Rectangle.java
```

A forrás és osztály állományok kezeléséhez erre a módszerre van szükségünk, így a fordító és a Java virtuális gép (JVM) megtalálja az összes típust, amit a programunk használ. Mikor a fordító összeakad egy új osztállyal a program fordításkor, meg fogja találni az osztályt, úgy hogy feloldja a neveket, típusvizsgálatot hajt végre és így tovább. Hasonlóan, amikor a JVM összeakad egy új osztállyal program futáskor, meg fogja találni az osztályhoz tartozó metódusokat. A fordító és a JVM az osztályokat a *class path*–ban listázott könyvtárakban és JAR fájlokban keresi.

Definíció: A *class path*, könyvtárak vagy JAR fájlok egy rendezett listája, amelyben *class* fájlokat kereshetünk. Fizikailag ez egy CLASSPATH nevű környezeti változót jelent.

Mindegyik könyvtár listázva van a *class path*-ban egy legfelső szinten lévő könyvtárban, amelyben csomag könyvtárak láthatók. A legfelső szintű könyvtártól a fordító és a JVM összeállítja az elérési útnak a maradékát, az alaphoz a csomagot és az osztály névnél az osztályt. A fordító és a JVM is összeállítja az elérési út nevet a *.class* fájlhoz a csomag teljes nevével.

Alapértelmezésként a fordító és a JVM az aktuális könyvtárban és a JAR fájlban keres, amely magába foglalja a Java környezeti *class* fájlokat. Más szóval, az aktuális könyvtár és a Java környezeti *class* fájlok automatikusan benne vannak a *class path*-ban. Legtöbbször az osztályokat megtalálják ezen a két helyen. Így nem kell aggódnunk az osztályaink elérési útja miatt. Néhány esetben azonban lehet, hogy meg kell adni osztályaink elérési útját.

17.5. Ellenőrző kérdések

- Mi a hasonlóság az alkönyvtárak és a csomagok között?
- Hogyan kell egy Java osztályt egy adott csomagba tenni?
- Mit jelent a *package* direktíva és mi a paramétere?
- Hogyan lehet egy csomagban levő osztályt elérni?
- Mit egyszerűsít az import direktíva?
- Hogyan leli fel a virtuális gép a *class* fájlokat?
- Mi a *CLASSPATH* környezeti változó?

Igaz vagy hamis? Indokolja!

- Különböző csomagokban lehetnek azonos nevű típusok.
- Csomagot létrehozni a *create package* utasítással lehet.
- Egy csomag típusait a *uses* kulcsszó segítségével lehet elérhetővé tenni.
- Ha egy forrásállományban nem nevezünk meg tartalmazó csomagot, akkor a forrásállományban szereplő osztályok semelyik csomagnak nem lesznek részei.
- Ha két csomagban van azonos nevű típus, akkor nem importálhatjuk egyszerre mindkettőt.
- A *package* utasítás csak a forrásállomány legelső utasítása lehet.
- A csomag kizárólag logikai szintű csoportosítás.

- Egy csomag csak publikus osztályokat tartalmazhat.
- Egy fordítási egységben csak egy osztályt lehet deklarálni.
- Elnevezési konvenció szerint a csomag neve nagybetűvel kezdődik, a többi kicsi.
- Egy fordítási egység kötelezően tartalmaz *package* deklarációt.
- Egy fordítási egység kötelezően tartalmaz *import* deklarációt.
- Ha az osztály deklarációjánál nem adunk meg láthatóságot, akkor arra a csomag más osztályaiból lehet hivatkozni.

Melyik a helyes sorrend egy forrásállomány esetén?

- *package, import, class*
- *class, import, package*
- *import, package, class*
- *package, class, import*

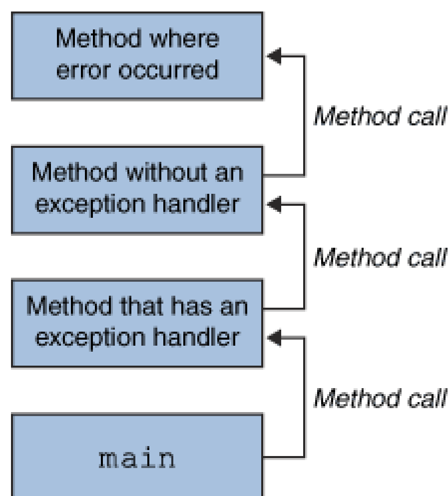
18.Kivételkezelés

A *kivétel* a „kivételes esemény” kifejezés rövidítése.

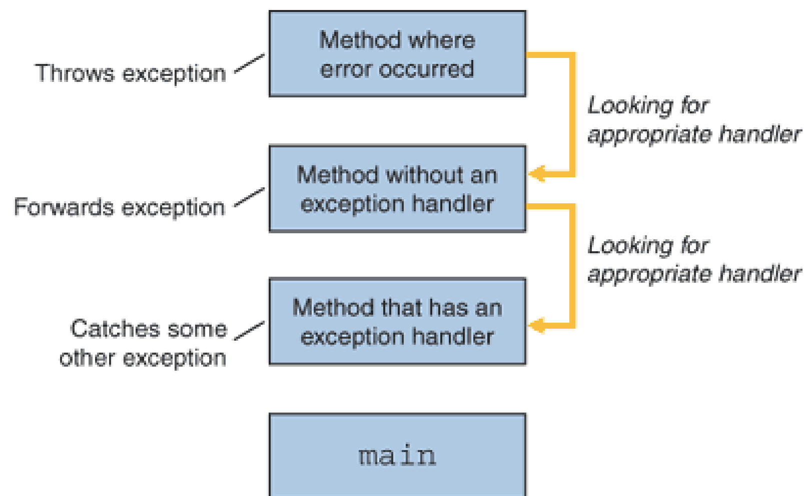
Definíció: A kivétel egy olyan esemény, amely a program végrehajtásakor keletkezik, megszakítva az utasítások végrehajtásának normális folyamatát.

Ha egy metódusban hiba keletkezik, a metódus egy objektumot hoz létre, melyet átad a futtatási környezetnek. Az objektum – melyet kivétel objektumnak neveznek – tartalmazza az információt a hibáról, annak típusáról és a program állapotáról, amikor a hiba létrejött. Kivétel objektum létrehozását és futtatási rendszer által történő kezelését kivételdobásnak hívják.

Miután egy metódus eldob egy kivételt, a futtató környezet megpróbál a kezelésére találni valamit. A lehetséges dolgok, melyek a kivételt kezelik a meghívott metódusok rendezett listája abban a metódusban, ahol a hiba keletkezett. A metódusok listáját hívási veremnek nevezzük.



A futtató rendszer átkutatja a hívási vermet olyan metódus után, mely tartalmaz kivétel kezelésére alkalmas kódblokkot. Ezt a blokkot kivételkezelőnek nevezzük. A keresés abban a metódusban kezdődik, ahol a hiba generálódott, majd a hívási verem metódusainak fordított sorrendjében folytatódik. Mikor egy megfelelő kezelőt talál, a futtató rendszer, átadja a kivételt a kezelőnek. Egy kivételkezelő megfelelő, ha az eldobott kivétel objektum típusa megegyezik azzal a típussal, melyet a kezelő kezelni tud. A kivételkezelő kiválasztását úgy is nevezik, hogy elkapni a kivételt. Ha a futtatókörnyezet a metódusok átkutatása után sem talál megfelelő kivételkezelőt, mint ahogy a következő ábra mutatja, a futtató rendszer (és ez által a program) leáll.



Hibák kezelésére használt kivételeknek van néhány előnye a hagyományos hibakezelési technikákkal szemben. (Később visszatérünk a témára.)

18.1. Kivételek elkapása vagy továbbengedése

A Java futtató rendszer megköveteli, hogy a metódus elkapja, vagy felsorolja az összes ellenőrzött kivételt, melyet a metódus eldobhat (másként fogalmazva a kivételt továbbengedi). Először is tekintsünk át néhány kifejezést.

Elkapás (catch)

A metódus elkaphat egy kivételt, ha rendelkezik ilyen típusú kivételek kezelőjével.

Továbbengedés (throws záradék)

A metódus a deklarációja *throws* záradékában írja le, hogy milyen kivételeket dobhat.

Ellenőrzött kivételek

Kétfajta kivétel létezik: futási időben keletkezett kivétel és nem futási időben keletkezett kivétel. Futási idejű kivétel a Java futtatórendszerében keletkezik: aritmetikus kivételek (például nullával való osztás), referenciával kapcsolatos kivételek (mint például egy objektum tagjaihoz való hozzáférés *null* hivatkozással) és indexeléssel kapcsolatos kivételek (mint például egy tömb elemeihez való hozzáférés olyan indexszel, mely túl nagy vagy túl kicsi). Egy metódusnak nem kötelező előírnia futási idejű kivételeket, de ajánlott.

Nem futási időben keletkezett kivételek olyan kivételek, melyek a Java futási rendszeren kívül keletkeznek. Például: kivételek, melyek I/O során keletkeznek. A fordító biztosítja, hogy a nem futási időben keletkezett kivételeket elkapják, vagy továbbengedjék; ezért ezeket ellenőrzött kivételeknek is nevezzük.

Sok programozó inkább futási időben keletkezett kivételeket használ az ellenőrzött kivételekkel szemben, hogy ne keljen elkapniuk vagy továbbengedniük őket. Ez általában nem ajánlott.

Kivételek, melyeket a metódus eldobhat

- Minden a metódus által közvetlenül eldobott kivétel
- Minden közvetetten eldobott kivétel másik metódus hívásával, mely kivételt dob.

Kivételek elkapása és kezelése

Ez a fejezet bemutatja, hogyan kell használni a kivételkezelő három komponensét – a *try*, *catch*, és *finally* blokkokat – egy kivételkezelő megírásához. A fejezet utolsó része végigmegy egy példán, bemutatva, mi történik különböző esetekben.

A következő példa a *ListOfNumbers* osztályt definiálja és implementálja. A *ListOfNumbers* konstruktor egy *Vector* objektumot hoz létre, mely tíz *Integer* elemet tartalmaz a nullától kilencig terjedő index értékekhez. A *ListOfNumbers* osztály egy *writeList* metódust is definiál, mely a számok listáját kiírja egy szövegfájlba, melynek neve *OutFile.txt*. A példa program a *java.io* csomagban definiált kimeneti osztályokat használja, melyekről részletesebben később lesz szó.

```
import java.io.*;
import java.util.Vector;

public class ListOfNumbers {
    private Vector vector;
    private static final int SIZE = 10;

    public ListOfNumbers () {
        vector = new Vector(SIZE);
        for (int i = 0; i < SIZE; i++) {
            vector.addElement(new Integer(i));
        }
    }

    public void writeList() {
        PrintWriter out = new PrintWriter(
            new FileWriter("OutFile.txt"));

        for (int i = 0; i < SIZE; i++) {
            out.println("Value at: " + i + " = " +
                vector.elementAt(i));
        }

        out.close();
    }
}
```

Az első vastag betűs sor egy konstruktor hívása. A konstruktor egy kimeneti folyamot inicializál. Ha a fájlt nem lehet megnyitni, a konstruktor *IOException*-t dob. A második vastag betűs sor a *Vector* osztály *elementAt* metódusát hívja meg, mely egy *ArrayIndexOutOfBoundsException*-t dob, ha az paramétereinek értéke túl kicsi (kisebb, mint nulla), vagy túl nagy (nagyobb, mint a változók száma, melyeket a *Vector* tartalmaz). Ha megpróbáljuk lefordítani a *ListOfNumbers* osztályt, a fordító egy hibaüzenetet ír ki a *FileWriter* konstruktor által dobott kivételről, de nem fog hibaüzenetet megjeleníteni az *elementAt* által dobott hibáról. Ennek oka, hogy a konstruktor által dobott kivétel, *IOException*, ellenőrzött kivétel, míg a másik, *ArrayIndexOutOfBundsException*, egy futási időben keletkezett kivétel. A Java programozási nyelv csak az ellenőrzött kivételek kezelését követeli meg, tehát a felhasználó csak egy hibaüzenetet fog kapni.

Most, hogy megismerkedünk a *ListOfNumbers* osztállyal és az abban eldobott kivételekkel, készen állunk egy kivételkezelő megírására, mely ezeket a hibákat elkapja, és kezelni tudja.

A *try* blokk

Egy kivételkezelő elkészítésének első lépése, hogy elhatároljuk a kódot, ami hibát dobhat a *try* blokkban. A *try* blokk általában így néz ki:

```
try {
    code
}
catch and finally blocks ...
```

A példakódban található szegmens tartalmaz egy vagy több olyan sort, amely kivételt dobhat. (A *catch* és a *finally* blokkokról részletes magyarázatot a következő részben találhatunk.)

Ahhoz, hogy egy kivételkezelőt készítsünk a *ListOfNumbers* osztály *writeList* metódusához, a *writeList* metódus kivétel-dobó részeit el kell határolnunk a *try* blokkal. Ezt többféleképpen tehetjük meg. A programkód azon sorait, melyekről feltételezzük, hogy kivételt dobhatnak, *try* blokkba tesszük, és mindegyiknél gondoskodunk a kivételek kezeléséről. Vagy pedig betehetjük az összes *writeList* kódot egy egyszerű *try* blokkba, és ehhez hozzákapcsolhatunk többféle kivételkezelőt. A következőkben láthatjuk, hogy az egész metódusra használjuk a *try* blokkot, mivel a szóban forgó kód nagyon rövid:

```
...
private Vector vector;
private static final int SIZE = 10;
...
PrintWriter out = null;

try {
    System.out.println("Entered try statement");
    out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++) {
        out.println("Value at: " + i + " = " +
            vector.elementAt(i));
    }
} /// catch and finally statements ...
```

Amennyiben a kivétel bekövetkezik a *try* blokkon belül, a kivétel lekezelésre kerül a kivételkezelő által. Ahhoz, hogy a kivételkezelést hozzá tudjuk kapcsolni a *try* blokkhoz, utólag használnunk kell *catch* blokkot is. A következő rész megmutatja, hogyan is kell ezt használnunk.

Megjegyzés: A kivételkezeléssel ismerkedők esetén gyakori hiba, hogy csak a kivételt eldobó függvényhívást teszik a *try* blokkba. Érdemes ezen a példán átgondolni, hogy ha nem sikerülne az állomány megnyitása, akkor semmi értelme nem lenne a *for* ciklus lefutásának. A fenti megoldásnál ez nem is fog bekövetkezni, hiszen a kivétel létrejöttekor a vezérlés a teljes *try* blokkból kilép.

A *catch* blokk

A *try* blokkhoz hozzáilleszthetjük a kivételkezelést, amennyiben egy vagy több *catch* blokkot használunk közvetlenül a *try* blokk után. Semmilyen programkód nem lehet a *try* blokk vége és az első *catch* blokk között!

```
try {
    ...
} catch (ExceptionType name) {
    ...
} catch (ExceptionType name) {
    ...
} ...
```

Minden *catch* blokk egy kivételkezelő, és azt a típusú kivételt kezeli, amilyet a paraméter tartalmaz. A paraméter típusa (*ExceptionType*) deklarálja a kivétel típusát, amit a kezelő lekezel.

A *catch* blokk tartalmazza azt a programkódot, amely végrehajtásra kerül, amikor a kivételkezelőt meghívjuk. A futtatórendszer meghívja azt a kivételkezelőt, amelyik esetén az *ExceptionType* megfelel a dobott kivétel típusának.

Itt látható két kivételkezelő a *writeList* metódushoz – az ellenőrzött kivételek két típusához, melyeket a *try* blokk dobhat:

```
try {
    ...
} catch (FileNotFoundException e) {
  System.err.println("FileNotFoundException: "+e.getMessage());
  throw new SampleException(e);
} catch (IOException e) {
  System.err.println("Caught IOException: "+e.getMessage());
}
```

Mindkét kezelő egy hibaüzenetet ír ki. A második kezelő semmi mást nem hajt végre. Bármilyen *IOException* (I/O kivétel) elkapása esetén (amit az első kezelő nem kapott el), megengedi a programnak, hogy folytassa a futtatást.

Az első kezelő a kiírandó szövegek összeillesztése során egy saját definiált kivételt dob. Ebben a példában a *FileNotFoundException* kivétel létrejöttekor egy saját definiált kivételt hoz létre, aminek *SampleException* a neve, és ezt dobja a program.

A kivételkezelők többet is tudnak, mint hogy hibaüzeneteket írnak ki, vagy pedig megállítják a program futását. Hibajavításra is képesek, amint a felhasználó hoz egy rossz döntést, vagy pedig a hibák túlnőnek a legmagasabb szintű kezelőn láncolt kivételeket használva.

A *finally* blokk

A kivételkezelő beállításának utolsó lépése, hogy rendet tegyünk magunk után, mielőtt átadjuk az irányítást a program különböző részeinek. Ezt a takarító programkódot a *finally* blokkba kell beírnunk. A *finally* blokk tetszőlegesen használható. Olyan mechanizmust nyújt, ami kiküszöböli azokat a figyelmetlenségeket, amik a *try* blokkban történtek. A *finally* blokkot például arra használhatjuk, hogy bezárjuk a fájlokat, amelyekre se a hiba nélküli futás, se a hiba dobása esetén nem szükségesek már.

A *writeList* metódus *try* blokkja (amivel eddig dolgoztunk) megnyitja a *PrintWriter*-t. A *writeList* metódusból való kilépés előtt programnak be kellene zárnia ezt a folyamatot. Ez felvet egy némiképp komplikált problémát, mivel *writeList* metódus *try* blokkja a következő három lehetőség közül csak egyféleképpen tud kilépni:

- A *new FileWriter* hibát jelez és *IOException*-t dob.
- A *vector.elementAt(i)* hibát jelez és *ArrayIndexOutOfBoundsException*-t dob.
- Minden sikerül és a *try* blokk zökkenőmentesen kilép.

A futtatórendszer mindig végrehajtja azokat az utasításokat, amelyek a *finally* blokkban vannak, függetlenül attól, hogy történt-e kivétel dobása, vagy nem. Így ez a legmegfelelőbb hely, hogy kitakarítsunk magunk után.

A következő *finally* blokk a *writeList* metódust takarítja ki, és bezárja a *PrintWriter*-t.

```
finally {
    if (out != null) {
        System.out.println("Closing PrintWriter");
        out.close();
    } else {
        System.out.println("PrintWriter not open");
    }
}
```

A *writeList* példában, a *finally* blokkba való beavatkozás nélkül tudunk gondoskodni a kitakarításról. Például, a *PrintWriter* bezárását végző programkódot a *try* blokk végéhez tudjuk illeszteni, továbbá az *ArrayIndexOutOfBoundsException* kivétel-kezelőhöz:

```
try {
    ...
    out.close();
} catch (FileNotFoundException e) {
    out.close();
    System.err.println(
            "Caught: FileNotFoundException: "+e.getMessage());
    throw new RuntimeException(e);
} catch (IOException e) {
    System.err.println("Caught IOException: " +
                       e.getMessage());
}
```

Azonban ez mégis lemásolja a kódot, így amennyiben később módosítjuk a programkódot, nehéz lesz olvasni benne, és a hibák kiterjedhetnek. Például, ha a *try* blokkhoz egy olyan programkóddal bővítjük, ami egy új típusú kivételt dobhat, emlékeznünk kell arra, hogy bezárjuk a *PrintWriter*-t az új kivétel-kezelőben.

Az összeillesztés

Az előző részekben volt arról szó, hogyan építsünk fel a *ListOfNumbers* osztályban *try*, *catch* és *finally* blokkokat a *writeList* metódus számára. A következőkben a forráskódot nézzük meg, vizsgálva a végeredményt.

Mindent egybevéve, a *writeList* metódus a következőképp néz ki:

```
public void writeList() {
    PrintWriter out = null;
```

```
        try {
            System.out.println("Entering try statement");
            out =
                new PrintWriter(new FileWriter("OutFile.txt"));
            for (int i = 0; i < SIZE; i++)
                out.println("Value at: " + i + " = "
                  + vector.elementAt(i));
        } catch (ArrayIndexOutOfBoundsException e) {
            System.err.println("Caught " +
                        "ArrayIndexOutOfBoundsException: " +
                          e.getMessage());
        } catch (IOException e) {
            System.err.println("Caught IOException: " +
                             e.getMessage());
        } finally {
            if (out != null) {
                System.out.println("Closing PrintWriter");
                out.close();
            } else {
                System.out.println("PrintWriter not open");
            }
        }
}
```

Ahogy azt előzőekben említettük, a metódusban a *try* szerkezetnek három lehetséges kimenete lehet:

- A kódunk hibás és kivételt dob. Ez lehet egy *IOException* kivétel, amit a *FileWriter* okoz vagy egy *ArrayIndexOutOfBoundsException* kivétel egy rosszul megadott érték esetén a *for* ciklusban, esetleg egy *RuntimeException* kivétel bármilyen futási hiba esetén.
- Minden rendben, a várt kimenetet kapjuk.

Nézzük meg, mi történik a *writeList* metódusban ezen esetekben:

Kivétel történik

Számos oka lehet annak, hogy a *FileWriter* hibás működést fog eredményezni.

Okozhatja a konstruktor (ha nem tudjuk létrehozni a fájlt, vagy nem tudunk beleírni).

Amikor a *FileWriter* az *IOException* kivételt dobja, a program azonnal leállítja a *try* blokk végrehajtását. Ezek után a rendszer a metódus elejétől indul, és meghívja a megfelelő kivételkezelőt. Habár a *FileWriter* konstruktornak nincs megfelelő kivételkezelője, a rendszer megnézi a *writeList* metódust.

A *writeList* metódusnak két kivételkezelője van: egyik az *IOException*, másik, pedig az *ArrayIndexOutOfBoundsException*.

A rendszer megnézi a *writeList* kivételkezelőit, de csak a *try* blokk után. Az első kivételkezelő nem foglalkozik a konkrét kivétel típussal, ezért a rendszer automatikusan a második kivételkezelőt használja (*IOException*). Ez be tudja azonosítani a típusát, ennek köszönhetően a rendszer már megtalálja a megfelelő kivételkezelőt, így a *catch* blokk végrehajtódik.

Miután a kivételkezelő lefutott, a rendszer a *finally* blokkra ugrik, melynek futása elindul a kivételeket figyelmen kívül hagyva. Miután a *finally* blokk lefutott a rendszer normális ütemben fut tovább.

Az *IOException* által dobott kimenet:

```
Entering try statement
Caught IOException: OutFile.txt
PrintWriter not open
```

A *try* blokk normális működése

Minden, ami a *try* blokkon belül van, sikeresen lefut és megfelelő kimenetet ad eredményként. Nem kapunk kivételt és a rendszer továbblép a *finally* blokkra. A sikeresség eredményeképpen a *PrintWriter* megnyílik, majd amikor a rendszer eléri a *finally* blokkot, bezárja. Miután a *finally* blokk lefutott, a rendszer normális ütemben fut tovább.

A kivételek nélküli kimenet:

```
Entering try statement
Closing PrintWriter
```

Metódusok által dobott kivételek

Az előzőekben láthattuk, hogyan írhatunk kivételkezelőt a *ListOfNumbers* osztályban a *writeList* metódusnak. Olykor ez megfelelő a működést illetően, de vannak esetek, amikor jobb a veremkezelőt használni. Például, ha a *ListOfNumbers* osztályt egy másik osztály csomagjaként hozunk létre, ebben az esetben jobban tesszük, ha más módon kezeljük a kivételt.

Bizonyos esetekben szükség lehet az eredeti *writeList* metódus módosítására, melynek hatására már a kívánt működést kaphatjuk. Az eredeti *writeList* metódus, ami nem fordul le:

```
public void writeList() {
    PrintWriter out =
                new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++) {
        out.println("Value at: " + i + " = " +
                    victor.elementAt(i));
    }
    out.close();
}
```

A *writerList* metódus deklarációjában a *throws* kulcsszó hozzáadásával két kivételt adunk meg. A *throws* kulcsszót egy vesszővel elválasztott lista követ a kivételosztályokkal.

A *throws* a metódus neve és a paraméter lista után áll, utána következik a kapcsos zárójelben a definíció:

```
public void writeList() throws IOException,
                               ArrayIndexOutOfBoundsException {
```

Az *ArrayIndexOutOfBoundsException* futásidejű kivétel, nem kötelező lekezelni, így elegendő az alábbi forma használata:

```
public void writeList() throws IOException {
```

18.2. Kivételek dobása

Bármelyik forráskód tud kivételt dobni, legyen az saját kódunk, egy csomagból származó, más által írt kód, bármi is annak a kimenete, a *throws* minden esetben jelen van.

A Java platform számos osztályt nyújt a kivételek kezelésére, melyek egytől egyig leszármazottjai a *Throwable* osztálynak. Ezek az osztályok lehetővé teszik a programok számára, hogy megkülönböztessék az eltérő típusú kivételeket, melyek a program futása közben keletkeznek.

Saját magunk is létrehozhatunk speciális osztályokat a leendő problémák reprezentációjával. Fejlesztőként saját magunknak kell létrehoznunk az osztályt és elvégezni a szükséges beállításokat.

18.2.1 A *throw* használata

A kivétel dobásánál minden metódus a *throw* kulcsszót használja, melynek a következő formai követelménynek kell, hogy megfeleljen:

```
throw someThrowableObject;
```

Az alábbi *pop* metódus egy osztály által létrehozott közös veremobjektumból jön létre. A metódus eltávolítja a verem felső rekeszét és az objektummal tér vissza:

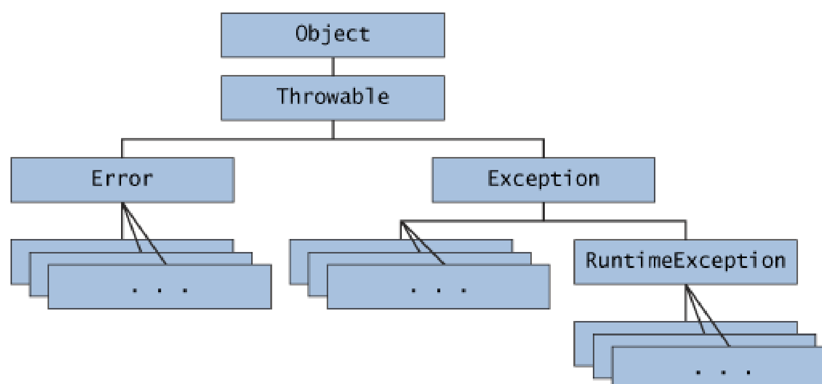
```
public Object pop() throws EmptyStackException {
    Object obj;

    if (size == 0) {
        throw new EmptyStackException();
    }

    obj = objectAt(SIZE - 1);
    setObjectAt(SIZE - 1, null);
    size--;
    return obj;
}
```

A *pop* metódus megnézi, van-e valami a veremben. Ha a verem üres (mérete 0), akkor a *pop* által létrejön egy új *EmptyStackException* objektumot (a *java.util* tagja) és eldobja azt. Ezek az objektumok a *java.lang.Throwable* osztály leszármazottjai kell, hogy legyenek, a fejezet későbbi része a kivétel osztályok létrehozásával foglalkozik.

A *pop* metódus deklarációja magában foglalja a *throws* kulcsszót. Az *EmptyStackException* kivételt a *pop* metódus nem kapja el, így a metódusnak a *throws* kulcsszót kell, hogy használja a deklarációhoz.

18.3. Eldobható osztályok és leszármazottai

Az ábra a *Throwable* osztály és annak legjelentősebb származtatott osztályait (osztályhierarchiáját) reprezentálja. Amint láthatjuk, a *Throwable* osztálynak kettő direkt módon származtatott utódja van: *Error* és *Exception*.



Error osztály

Ha a művelet valamilyen okból kifolyólag nem hajtható végre, sikertelen a végrehajtás, a Java virtuális gép *Error*-t fog dobni. Egyszerű programok nem kapnak el, illetve nem dobnak *Error*-t.

Exception osztály

A legtöbb program elkap és dob objektumokat, melyek az *Exception* osztályból származnak. A legtöbb általunk megírt program elkap és dob kivételeket (jelzi a hibát).

Az *Exception* osztálynak számos leszármazottja definiált a Java Platformban, ezek a leszármazottak jelzik a különböző típusú kivételeket. Például *IllegalAccessException* jelzi, ha egy különös metódus nem található, a *NegativeArraySizeException* pedig, ha egy tömb méretét helytelenül adtunk meg (negatív szám).

A *RuntimeException* kivételek futás közben generálódnak a Java Virtuális gép által. Ennek egyik példája a *NullPointerException*, mely akkor jön létre, ha egy metódus megpróbál hozzáférni *null* hivatkozással valamely objektumhoz.

18.4. Láncolt kivételek

A kivételeknek sok esetben egy másik kivétel az oka. Ilyen esetben célszerű az eredeti (okozó) kivételt és az új kivételt együtt kezelni. Erre a problémára jelent megoldást a Java 1.4 verzió óta rendelkezésre álló kivételláncolás. Ennek lényege, hogy egy kivétel létrehozásakor becsomagoljuk az okozó kivételt az új kivétel objektumba.

Ennek érdekében két új metódus és két új konstruktor lett hozzáadva a *Throwable* típushoz.

A következő új metódusok és konstruktorok támogatják a láncolt kivételeket a *Throwable* osztályban:

```
Throwable getCause()
Throwable initCause(Throwable)
Throwable(String, Throwable)
Throwable(Throwable)
```

Az *initCause* és a *Throwable* konstruktorok *Throwable* paramétere egy kivétel, amik okozták az aktuális kivételt. A *getCause* visszaad egy kivételt, ami okozta az aktuális kivételt, az *initCause* pedig visszaadja az aktuális kivételt.

A következő példa megmutatja, hogyan használjuk a láncolt kivételeket:

```
try {
    ...
} catch (IOException e) {
    throw new SampleException("Other IOException", e);
}
```

Ebben a példában, amikor az *IOException*-t kap el a program, akkor egy új *SampleException* kivétel jön létre az eredeti okkal egybekötve, és a kivételek lánca feldobódik egy következő, magasabb szintű kivételkezelőhöz.

Hívási verem információk kinyerése

Most feltételezzünk, hogy egy magasabb szintű kivételkezelő akarja a kimenetre írni a hívási verem tartalmát egyedi formátumban.

Definíció: a hívási verem információt szolgáltat az aktuális szál futási történetéről, az osztályok és a metódusok neveiről, amik akkor hívódnak meg, amikor a kivétel bekövetkezett. A hívási verem egy nagyon hasznos hibakereső eszköz.

A következő kód megmutatja, hogyan kell a *getStackTrace* metódust használni a kivétel objektumon:

```
catch (Exception cause) {
    StackTraceElement elements[] = cause.getStackTrace();
    for (int i = 0; n = elements.length; i < n; i++) {
        System.err.println(elements[i].getFileName() + ":"
                + elements[i].getLineNumber() + ">> "
                + elements[i].getMethodName() + "()");
    }
}
```

API naplózás

A következő kódrészlet *catch* blokkjában naplózásra látunk példát. Bármennyire célszerűnek tűnik is az előző példa megközelítése, érdemes inkább a naplózás mechanizmusát alkalmazni *a java.util.logging* segítségével:

```
try {
    Handler handler = new FileHandler("OutFile.log");
    Logger.getLogger("").addHandler(handler);
} catch (IOException e) {
    Logger logger = Logger.getLogger("package.name");
    StackTraceElement elements[] = e.getStackTrace();
    for (int i = 0; n = elements.length; i < n; i++) {
        logger.log(Level.WARNING, elements[i].getMethodName());
    }
}
```

18.5. Saját kivétel osztályok létrehozása

Amikor szükségessé válik egy kivétel dobása, az egyik lehetőség egy valaki által már megírtat használni – a Java Platform rengeteg kivétel osztállyal van ellátva – vagy magunk is tudunk írni a céljainknak megfelelőt.

Saját kivétel osztályt kell írnunk, ha a következő kérdések bármelyikére igennel válaszolhatunk. Különben valószínűleg egy valaki más által megírtat kell használnunk.

- Olyan típusú kivétel osztályra van-e szükség, ami nem áll rendelkezésre a Java Platformban?
- Fog segíteni a felhasználóknak, ha képesek lesznek megkülönböztetni a mi kivételeinket a mások által megírt kivételosztályoktól?
- A kódunk több helyen fog-e kivételt dobni?
- Ha valaki más kivételeit használjuk, a felhasználó hozzá fognak-e tudni férni azokhoz a kivételekhez?
  Egy hasonló kérdés: a csomagunknak függetlennek és önállónak kell-e lennie?

Tegyük fel, hogy írtunk egy láncolt lista osztályt, amit szeretnénk megosztani nyílt forrású programként. A láncolt lista osztályunk többek között a következő metódusokat támogatja:

objectAt(int n)

Visszaadja a lista n-dik pozíciójában levő objektumot. Kivételt dob, ha a paraméter kisebb, mint 0, vagy nagyobb, mint az aktuális objektumok száma a listában.

firstObject()

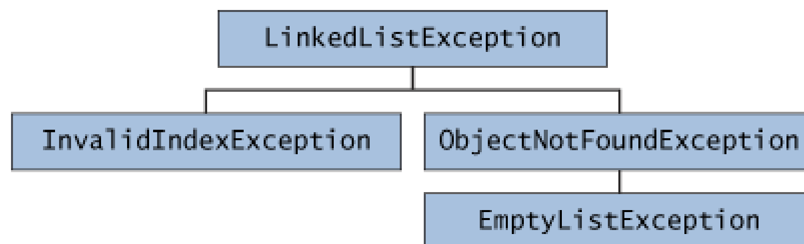
Visszaadja az első objektumot a listában. Kivételt dob, ha a lista nem tartalmaz objektumot.

indexOf(Object o)

Átnézi a listát egy speciális objektumért és visszaadja a pozícióját a listában. Kivételt dob, ha az objektum nincs a listában.

A láncolt lista osztály képes különböző kivételeket dobni, ugyanakkor lehetőséget ad arra is, hogy a felhasználó program el tudja kapni a láncolt lista által dobott összes kivételt egy kivételkezelővel. Ha a láncolt listánkat egy csomagban tervezzük megosztani, minden kapcsolódó kódnak egy közös csomagban kell lennie.

A következő ábra illusztrál egy lehetséges osztályhierarchiát a láncolt listák által dobható kivételekről:



Szülőosztály választása

Az *Exception* valamennyi leszármazott osztályát lehet a *LinkedListException* szülőosztályának választani. Azonban könnyen látható, hogy ezek a leszármazott osztályok nem odaillők, mert vagy túl specializáltak, vagy nem függnek össze a *LinkedListException*-al. Ezért a *LinkedListException* szülő osztályának az *Exception*-nek kell lennie.

A legtöbb applet és alkalmazás, amit írunk, *Exception* objektumokat dob.

Figyelem: az olvasható kód érdekében egy jó gyakorlat, hogy hozzáfűzzük az *Exception* szót minden osztály nevéhez, amelyik közvetlenül vagy közvetve öröklődik az *Exception* osztályból.

18.6. Ellenőrző kérdések

- Mi a kivétel?
- Hogyan lehet kivételeket létrehozni?
- Kell-e deklarálni a metódusban létrehozott kivételeket, és ha igen, hogyan?
- Mire szolgál a *try-catch*?
- Hány *catch* ága lehet egy *try-catch* szerkezetnek?
- Hány *finally* ága lehet egy *try-catch-finally* szerkezetnek?
- Mi a teendő, ha metódusunk olyan másik metódust hív meg, ami kivételt generálhat?
- Mik a nem ellenőrzött kivételek, és hogyan lehet elkapni őket?

Korrekt a következő kód?

```
try {

} finally {

}
```

Milyen típusú kivételeket fog elkapni a következő kezelő?

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

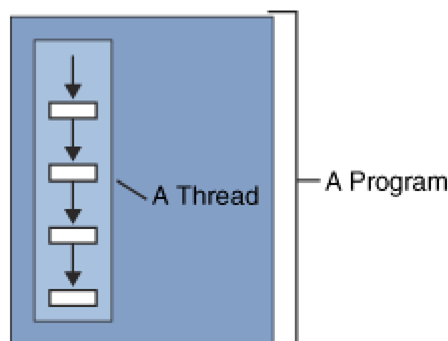
Igaz vagy hamis? Indokolja!

- A metódusban létrejövő kivételt kötelesek vagyunk a metódusban elkapni.
- kivételt dobni a *throws* utasítással lehet.

19.Programszálak kezelése

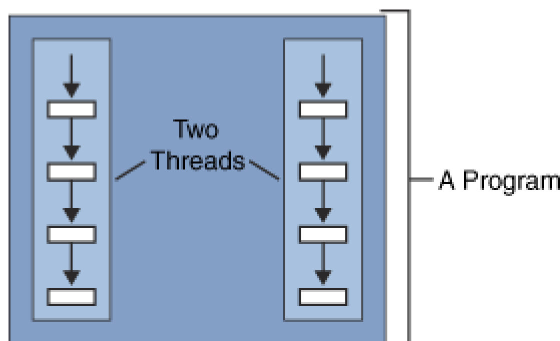
A programok jelentős része soros végrehajtású. Van kezdetük, egy végrehajtandó rész és egy befejezés. A program csak bizonyos időszakaszokban fut, hiszen a processzor már programokat is futtat.

A szál (*thread*) hasonló a soros programhoz. A szálnak is van kezdete, egy végrehajtandó része és egy befejezése. A szál is csak bizonyos időszakaszokban fut. De a szál maga nem egy program, a szálat nem lehet önállóan futtatni. Inkább úgy tekinthetjük, hogy a program részeként fut. A következő ábra megmutatja a kapcsolatot.



Definíció: A szál egy egyedülálló irányító folyamat a programon belül.

Persze nem arra használjuk a szálakat, hogy csak egyet alkalmazunk, inkább több szálat futtatva egy időben különböző feladatokat elvégezve egyetlen programban. A következő kép ezt illusztrálja.



A böngésző programunk is egy példa a többszálú alkalmazásra. Egy általános böngészőben egy oldalon belül egyszerre történik egy kép, applet letöltése, mozgó animáció vagy hang lejátszása, az oldal nyomtatása a háttérben, miközben egy új oldalt tölt be vagy éppen három rendező algoritmus versenyét tekintheti meg.

Néha a szálat könnyűsúlyú processznek nevezik, mert egy teljes programon belül fut, annak lefoglalt erőforrásait és a futtató környezetét használja.

Mint irányító folyamat, a szálnak is kell saját erőforrásokkal rendelkeznie. Rendelkezni kell egy végrehajtó veremmel és a programszámlálóval. Ebben a környezetbe fut a szál kódja. Néhol ezt a környezetet a szál fogalom szinonimájának nevezik.

A szál programozás nem egyszerű. Ha mégis szálakat kell használni, akkor érdemes a magas-szintű szál API-kat használni. Példaképpen, ha a programban egy feladatot többször kell elvégezni, érdemes a *java.util.Timer* osztályt alkalmazni. A *Timer* osztály időzítéses feladatoknál is hasznos lehet. Erre hamarosan látunk példát.

19.1. A *Timer* és a *TimerTask* osztály

Ebben a részben az időzítők használatát tárgyaljuk. A *Timer* osztály a *java.util* csomag része, a *TimerTask* osztály példányai ütemezését végzi. *Reminder.java* egy példa arra, hogy használjuk az időzítőket a késleltetett futtatáshoz:

```
import java.util.Timer;
import java.util.TimerTask;

public class Reminder {
    Timer timer;

    public Reminder(int seconds) {
        timer = new Timer();
        timer.schedule(new RemindTask(), seconds*1000);
    }

    class RemindTask extends TimerTask {
        public void run() {
            System.out.println("Lejárt az idő!");
            timer.cancel(); //A timer szál megszüntetése
        }
    }

    public static void main(String args[]) {
        new Reminder(5);
        System.out.println("Munka ütemezve.");
    }
}
```

Indulás után ezt látjuk:

```
Munka ütemezve.
```

Öt másodper múlva ezt látjuk:

```
Lejárt az idő!
```

A program egy példa arra, hogyan kell példányosítani és ütemezni a szálakat:

- *TimerTask* leszármazott osztály példányosítása. A *run* metódus tartalmazza a futtatás során végrehajtandó kódot. A példába ezt az leszármazott osztály *RemindTask*-ként neveztük el.
- Létrehozzuk a *Timer* osztály egy példányát.
- Létrehozunk egy időzítő objektumot (*new RemindTask()*).
- Beütemezzük az időzítőt. Ez a példa a *schedule* metódust használja, amelynek paraméterei az időzítő és késleltetés ezredmásodpercben (5000).

Egy másik lehetőség az ütemezésre, hogy az indítási időpontot adjuk meg. Példaképpen a következő kód 23:01-re ütemezi a végrehajtást:

```
Calendar calendar = Calendar.getInstance();
calendar.set(Calendar.HOUR_OF_DAY, 23);
calendar.set(Calendar.MINUTE, 1);
calendar.set(Calendar.SECOND, 0);
Date time = calendar.getTime();

timer = new Timer();
timer.schedule(new RemindTask(), time);
```

19.1.1 Időzített szálak leállítása

Alapesetben a program addig fut, amíg az időzítő szál is fut. Négy leállítási módszer közül választhatunk:

- A *cancel* metódust hívjuk meg. Ezt a program bármelyik pontján megtehetjük, mint a példa *run* metódusában.
- Az időzítő szálát démon szállá kell tenni a következőképpen: *new Timer(true)*. Ha a programban csak démon szálak maradnak, akkor a program kilép.
- Ha befejeződtek az ütemezések, el kell távolítani a *Timer* objektumhoz tartozó mutatókat. Ezzel a szál is megszűnik.
- A *System.exit* metódust hívjuk meg, amely leállítja a programot (benne a szálakat is).

A *Reminder* példa az első módszert használja, a *cancel* metódust hívja meg a *run* metódusból. A démon szálként való létrehozás a példánkban nem működne, mert a programnak futni kell a szál lefutása után is.

19.1.2 Ismételt futtatás

Ebben a példába másodpercenként ismétli a kód futását:

```
public class AnnoyingBeep {
    Toolkit toolkit;
    Timer timer;

    public AnnoyingBeep() {
        toolkit = Toolkit.getDefaultToolkit();
        timer = new Timer();
        timer.schedule(new RemindTask(),
                       0,        //kezdeti késleltetés
                       1*1000);  //ismétlési ráta
    }

    class RemindTask extends TimerTask {
        int numWarningBeeps = 3;
        public void run() {
            if (numWarningBeeps > 0) {
                toolkit.beep();
                System.out.println("Síp!");
                numWarningBeeps--;
            } else {
                toolkit.beep();
                System.out.println("Idő lejárt!");
            //timer.cancel(); // Nem szükséges
                              // mert van System.exit is.
                System.exit(0);   // Leállítja AWT szálat
                              // (és minden mást)
            }
        }
    }
    ...
}
```

Futtatás közben ez lesz a kimenet:

```
Munka ütemezve.
Síp!
Síp!
Síp!
Idő lejárt!
```

Az *AnnoyingBeep* program három paraméteres *schedule* metódust használ, hogy meghatározza a taszk másodpercenkénti indítását. A *Timer* metódus változatai:

- *schedule(TimerTask task, long késleltetés, long gyakoriság)*
- *schedule(TimerTask task, Date idő, long gyakoriság)*
- *scheduleAtFixedRate(TimerTask task, long késleltetés, long gyakoriság)*
- *scheduleAtFixedRate(TimerTask task, Date kezdetiIdő, long gyakoriság)*

A *schedule* metódust akkor használjuk, ha a többszörösen futtatott taszk ismétlési ideje számít, a *scheduleAtFixedRate* metódust, ha az ismétlések időben kötődnek egy pontos időhöz. A példában is a *schedule* metódust alkalmaztuk, amitől 1 másodperces intervallumokban sípol a gép. Ha valamelyik sípszó késik, akkor a következők is késlekednek. Ha úgy döntünk, hogy a program 3 másodperc múlva kilép az első sípszó után – ami azt is eredményezheti, hogy a két sípszó kisebb időközzel szólal meg, ha késlekedés lép fel – akkor a *scheduleAtFixedRate* metódust használjuk.

19.2. Szálak példányosítása

Ha a fenti megközelítés nem alkalmas a feladat ellátására, akkor saját szál példányosítása lehet a megoldás. Ez a fejezet elmagyarázza, hogyan lehet a szál *run* metódusát egyénivé alakítani.

A *run* metódusban van, amit a szál ténylegesen csinál, ennek a kódja határozza meg a futást. A szál *run* metódussal mindent meg lehet tenni, amit a Java programnyelvben le lehet írni. Pl.: prímszámokkal való számolás, rendezés, animáció megjelenítés.

A *Thread* osztály példányosításával egy új szál jön létre, ami alapból nem csinál semmit, de lehetőség van a leszármazott osztályban tartalommal megtölteni.

19.2.1 *Thread* leszármazott és a *run* felülírása

Az első lépés a szálak testreszabásánál, hogy *Thread* osztály leszármazottját létrehozzuk, és az üres *run* metódusát felülírjuk, hogy csináljon is valamit. Nézzük meg a *SimpleThread* osztályt, amely ezt teszi:

```
public class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
```

```
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((long)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("KÉSZ! " + getName());
    }
}
```

A *SimpleThread* osztály első metódusa egy konstruktor, amely paraméterként egy *String*-et fogad. A konstruktor meghívja az ősosztály konstruktorát, ami azért érdekes számunkra, mert az beállítja a szál nevét. Ezt a nevet használjuk a későbbi programban a szál felhasználói azonosítására. A nevet később a *getName* metódussal tudjuk lekérdezni.

A *SimpleThread* osztály következő metódusa a *run*. Itt történik a *Thread* érdemi működése. Ebben az esetben egy tizes *for* ciklust tartalmaz. Minden ciklusban kiírja az iteráció számát, a *Thread* nevét, és utána altatásra kerül véletlenszerűen 1 másodperces határon belül. Ha végzett, a *KÉSZ!* feliratot írja ki a szál nevével együtt. Ennyi a *SimpleThread* osztály. Használjuk fel ezt a *TwoThreadsTest*-ben.

A *TwoThreadsTest* osztály *main* metódusába két *SimpleThread* szálat hozunk létre: Jamaica és Fuji. (Ha valaki nem tudná eldönteni hova menjen nyaralni, akkor használja ezt a programot.)

```
public class TwoThreadsTest {
    public static void main (String[] args) {
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start();
    }
}
```

A *main* metódus azonnal mindkét szálat létrehozza a start metódus meghívásával, amely a *run* metódust hívja meg. Fordítás és futtatás után kibontakozik az úti cél. A kimenet hasonló kell, hogy legyen:

```
0 Jamaica
0 Fiji
1 Fiji
1 Jamaica
2 Jamaica
2 Fiji
3 Fiji
3 Jamaica
4 Jamaica
4 Fiji
5 Jamaica
5 Fiji
6 Fiji
6 Jamaica
7 Jamaica
7 Fiji
8 Fiji
9 Fiji
8 Jamaica
DONE! Fiji——Look out, Fiji, here I come!
9 Jamaica
DONE! Jamaica
```

Figyeljük meg, hogy a szálak kimenete mennyire független egymástól. Ennek az az oka, hogy a *SimpleThread* szálak egyidejűleg futnak. Mindkét *run* metódus fut, és mindkettőnek van saját kimenete ugyanakkor. Ha a ciklus befejeződik, a szálak leállnak és meghalnak.

19.2.2 *Runnable* interfész példányosítása

A következő *Clock* applet a jelenlegi időt mutatja, és azt másodpercenként frissíti. A frissítés állandó, mivel az óra kijelzése egy saját szálon fut.

Az applet más módszert használ, mint a *SimpleThread*. Itt ugyanis a szülőosztály csak az *Applet* lehet, de így az egyszeres öröklődés miatt a *Thread* már nem lehet szülő. Ezért a példa *Thread* leszármazott létrehozása helyett egy *Runnable* interfészt implementál, ezért a *run* metódust ebben definiálja. A *Clock* létrehoz egy szálat, amelynek a frissítés a célja.

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

public class Clock extends java.applet.Applet implements
Runnable {
    private volatile Thread clockThread = null;
    DateFormat formatter;     // Formats the date displayed
    String lastdate;          // String to hold date displayed
    Date currentDate;         // Used to get date to display
    Color numberColor;        // Color of numbers
    Font clockFaceFont;
    Locale locale;

    public void init() {
        setBackground(Color.white);
        numberColor = Color.red;
        locale = Locale.getDefault();
        formatter =
            DateFormat.getDateTimeInstance(DateFormat.FULL,
            DateFormat.MEDIUM, locale);
        currentDate = new Date();
        lastdate = formatter.format(currentDate);
        clockFaceFont = new Font("Sans-Serif",
                                 Font.PLAIN, 14);
        resize(275,25);
   }

    public void start() {
        if (clockThread == null) {
            clockThread = new Thread(this, "Clock");
            clockThread.start();
        }
    }
```

```
    public void run() {
        Thread myThread = Thread.currentThread();
        while (clockThread == myThread) {
            repaint();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e){ }
        }
    }
    public void paint(Graphics g) {
        String today;
        currentDate = new Date();
        formatter =
             DateFormat.getDateTimeInstance(DateFormat.FULL,
             DateFormat.MEDIUM, locale);
        today = formatter.format(currentDate);
        g.setFont(clockFaceFont);
        // Erase and redraw
        g.setColor(getBackground());
        g.drawString(lastdate, 0, 12);
        g.setColor(numberColor);
        g.drawString(today, 0, 12);
        lastdate = today;
        currentDate=null;
    }
    public void stop() {
        clockThread = null;
    }
}
```

A *Clock* applet *run* metódusa addig nem lép ki a ciklusból, amíg a böngésző le nem állítja. Minden ciklusban az óra kimenete újrarajzolódik. A *paint* metódus lekérdezi az időt, formázza és kijelzi azt.

Kell-e a *Runnable* interfész?

Két módszert ismertettünk a *run* metódus előállítására.

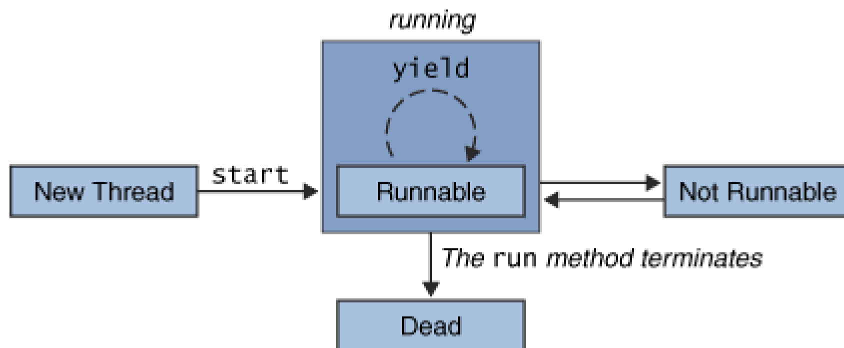
- Leszármazott *Thread* osztály létrehozása és a *run* metódus felülírása.
- Egy olyan osztály létrehozása, amelyik megvalósítja a *Runnable* interfészt és így a *run* metódust is. Ebben az esetbe a *Runnable* objektum szolgáltatja a *run* metódust a szálnak. Ezt feljebb láthatjuk.

Megjegyzés: A *Thread* is megvalósítja a *Runnable* interfészt.

Ahhoz, hogy egy böngészőben futhasson a *Clock* osztálynak az *Applet* osztály leszármazottjának kell lennie. Továbbá a *Clock* applet folyamatos kijelzése miatt egy szálra van szüksége, hogy ne a processzt vegye át, amiben fut. (Néhány böngésző figyelhet arra, hogy az appletek saját szálat kapjanak, elkerülendő, hogy a hibás appletek elvegyék a böngésző szálát. De erre nem építhetünk egy applet írásánál. Az általunk írt appletnek tudnia kell a saját szál létrehozását, ha olyan munkát végez.) De mivel a Java nyelv nem engedélyezi a több osztályokon át való öröklődést, a *Clock* nem lehet egyszerre a *Thread* és az *Applet* osztály leszármazottja. Így a *Clock* osztálynak egy *Runnable* interfészen keresztül kell a szálak viselkedését felvennie.

19.3. Programszál életciklusa

A következő ábra bemutatja a szál különböző állapotait, amibe élete során tud kerülni, és illusztrálja, hogy melyik metódusok meghívása okozza az átmenetet egy másik fázisba. Az ábra nem egy teljes állapotdiagram, inkább csak egy áttekintés a szál életének fontosabb és egyszerűbb oldalairól. A továbbiakban is a *Clock* példát használjuk, hogy bemutassuk a szál életciklusát különböző szakaszaiban.



19.3.1 Programszál létrehozása

Az alkalmazás, melyben egy szál fut, meghívja a szál *start* metódusát, amikor a felhasználó a programot elindítja. A *Clock* osztály létrehozza a *clockThread* szálat a *start* metódusában, ahogy a lenti programkód bemutatja:

```
public void start() {
    if (clockThread == null) {
        clockThread = new Thread(this, "Clock");
        clockThread.start();
    }
}
```

Miután a kiemelt utasítás végrehajtódik, a *clockThread* az „új szál” (*New Thread*) állapotban van. Egy szál ebben az állapotában csupán egy üres szál, semmiféle rendszererőforrás nincs hozzárendelve. Ebben az állapotban csak elindítható a szál. A *start* metóduson kívül bármilyen metódus meghívása értelmetlen, és *IllegalThreadStateException* kivételt eredményez. (Alapértelmezetten a rendszer mindig ezt a hibaüzenetet dobja, ha egy olyan metódus kerül meghívásra, amelyik a szál pillanatnyi állapotában nem engedélyezett.)

Érdemes megjegyezni, hogy a *Clock* típusú példány az első paraméter a szál konstruktorban. Ez a paraméter meg kell, hogy valósítsa a *Runnable* interfészt, csak így lehet a konstruktor paramétere. A *Clock* szál a *run* metódust a *Runnable* interfésztől örökli. A konstruktor második paramétere csupán a szál neve.

19.3.2 Programszál elindítása

Most gondoljuk át, mit eredményez a programkódban a *start* metódus:

```
public void start() {
    if (clockThread == null) {
        clockThread = new Thread(this, "Clock");
        clockThread.start();
    }
}
```

A *start* metódus létrehozza a szükséges rendszererőforrásokat a szál futtatásához, előkészíti a szálat a futáshoz, és meghívja a szál *start* metódusát. A *clockThread run* metódusa a *Clock* osztályban van definiálva.

Miután a *start* metódus visszatér, a szál fut. Mégis, ennél azért kicsit komplexebb a dolog. Ahogy az előző ábra is mutatja, egy elindított szál a futtatható állapotba kerül. Sok számítógép csak egy processzorral rendelkezik, így lehetetlen, hogy minden futási állapotban lévő szál egyszerre fusson. Ezért a Java futtatórendszernek implementálnia kell egy ütemezési sémát, ami elosztja a processzor erőforrásait az összes futó szál között. Szóval egy bizonyos időpontban, egy futó szál valószínűleg vár, hogy ő kerüljön sorra a CPU ütemezésében.

Még egyszer a *Clock run* metódusa:

```
public void run() {
   Thread myThread = Thread.currentThread();
   while (clockThread == myThread) {
        repaint();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            // The VM doesn't want us to sleep anymore,
            // so get back to work
        }
    }
}
```

A *Clock run* metódusa addig lépked a ciklusban, amíg a *clockThread == myThread* feltétel nem lesz igaz.

A cikluson belül a szál újra kirajzolja az appletet, majd utasítja a szálat, hogy kerüljön alvó állapotba 1 másodpercre. A *repaint* metódus végső soron meghívja a *paint* metódust, ami az aktuális frissítéseket végzi a program megjelenítési felületén. A *Clock paint* metódusa eltárolja az aktuális rendszeridőt, formázza, majd megjeleníti:

```
public void paint(Graphics g) {
    //get the time and convert it to a date
    Calendar cal = Calendar.getInstance();
    Date date = cal.getTime();
    //format it and display it
    DateFormat dateFormatter = DateFormat.getTimeInstance();
    g.drawString(dateFormatter.format(date), 5, 10);
}
```

19.3.3 Programszál nem futtatható állapotba állítása

Egy szál akkor válik nem futtathatóvá, ha a lenti események közül bármelyik is bekövetkezik:

- Meghívásra kerül a *sleep* metódusa.
- A szál meghívja a *wait* metódust, hogy az várjon egy bizonyos feltétel teljesülésére.
- A szál blokkolt egy I/O művelet miatt.

A *clockThread* a *Clock* szálon belül akkor kerül nem futtatható állapotba, mikor a futás metódus meghívja a *sleep*-et a jelenlegi szálban:

```
public void run() {
    Thread myThread = Thread.currentThread();
    while (clockThread == myThread) {
        repaint();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
           // A VM nem akarja, hogy tovább aludjunk,
           // szóval irány vissza dolgozni ☺
        }
    }
}
```

Az alatt a másodperc alatt, amíg a *clockThread* alvó állapotban van, a szál nem fut, még akkor sem fog, ha a processzor erőforrásai elérhetővé válnak. Miután letelik az egy másodperc, a szál újra futtathatóvá válik; ha a processzor erőforrásai elérhetővé válnak, a szál újra futni kezd.

Minden egyes nem futtatható állapotba kerüléskor, egy specifikus és megkülönböztethető *exit* metódus téríti vissza a szálat a futtatható állapotba. Egy *exit* hívás után csak a megfelelő feltételek teljesülése esetén fog ismét futni. Például: miután egy szál alvó állapotba lett helyezve, a meghatározott ezredmásodpercek lejárta után újból futtatható állapotba kerül. A következő felsorolás leírja a nem futtatható állapot kilépési feltételeit:

- Ha a szál alvó állapotba lett helyezve, a meghatározott időnek le kell telnie.
- Ha egy szál egy feltételre vár, akkor egy másik objektumnak kell értesítenie a várakozó szálat a feltétel teljesüléséről a *notify* vagy *notifyAll* metódus meghívásával.
- Ha egy szál blokkolva volt egy I/O művelet miatt, az I/O műveletnek be kell fejeződnie.

19.3.4 Programszál leállítása

Habár a *Thread* osztályban rendelkezésre áll a *stop* metódus, ezen metódus alkalmazása nem javallott, mert nem biztonságos, pl. adatvesztés léphet fel. Egy szálnak inkább saját magának kell befejezni a futását úgy, hogy a *run* metódusa természetesen áll le. Például a *while* ciklus a *run* metódusban egy véges ciklus 100-szor fut le, majd kilép:

```
public void run() {
    int i = 0;
    while (i < 100) {
        i++;
        System.out.println("i = " + i);
    }
}
```

A szál a *run* metódusban természetesen „hal meg” mikor a ciklus befejeződik és a *run* metódus kilép.

Nézzük meg, hogyan hajtja végre a *Clock* szál a saját halálát. Nézzük meg újra a *Clock run* metódusát:

```
public void run() {
    Thread myThread = Thread.currentThread();
    while (clockThread == myThread) {
        repaint();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            //the VM doesn't want us to sleep anymore,
            //so get back to work
        }
    }
}
```

A kilépési feltétel ebben a *run* metódusban ugyanaz a kilépési feltétele, mint a *while* ciklusnak, mert a ciklus után nincs kód:

```
while (clockThread == myThread) {
```

E feltétel jelzi, hogy a ciklus kilép, ha a jelenleg futásban lévő szál nem egyezik a *clockThread*-el. Mikor lesz ez a helyzet? Akkor, amikor a felhasználó elhagyja az oldalt, az alkalmazás, melyben a szál fut, meghívja a szál *stop* metódusát. Ez a metódus ezután beállítja a *clockThread*-et *null* értékre, így utasítva a fő ciklust, hogy szakítsa meg a futás metódust:

```
public void stop() {    // applets' stop method
    clockThread = null;
}
```

19.3.5 Programszál státusz tesztelése

Az 5.0-ás verzióban került bevezetésre a *Thread.getState* metódus. Mikor ez kerül meghívásra, az alábbi *Thread.State* értékek valamelyike tér vissza:

- *NEW*
- *RUNNABLE*
- *BLOCKED*
- *WAITING*
- *TIMED_WAITING*
- *TERMINATED*

A *Thread* osztályhoz tartozó API tartalmazza az *isAlive* metódust is. Az *isAlive* metódus igaz visszatérési értéket generál, ha a szálat elindították, de még nem lett leállítva. Ha az *isAlive* visszatérési értéke hamis, akkor tudhatjuk, hogy a szál vagy új szál (*NEW*), vagy halott állapotban van (*TERMINATED*). Ha a visszatérési érték igaz, akkor viszont a szál vagy futtatható (*RUNNABLE*), vagy nem futtatható állapotban van.

Az 5.0-ás verziót megelőzően nem lehetett különbséget tenni az új szál vagy a halott szálak között. Ugyancsak nem lehetett megkülönböztetni a futtatható, vagy nem futtatható állapotban lévő száltól.

19.3.6 A processzor használatának feladása

Ahogyan el tudjuk képzelni, CPU-igényes kódok negatív hatással vannak más szálakra, amelyek azonos programban futnak. Vagyis próbáljunk jól működő szálakat írni, ame-

lyek bizonyos időközönként önkéntesen felhagynak a processzor használatával, ezzel megadva a lehetőséget minden szálnak a futásra.

Egy szál önkéntesen abbahagyhatja a CPU használatát a *yield* metódus meghívásával.

19.4. Ellenőrző kérdések

Melyik az az interfész, amelyet megvalósítva több szálú programfutás érhető el?

- *Runnable*
- *Run*
- *Threadable*
- *Thread*
- *Executable*

Mi annak a metódusnak a neve, amellyel a külön programszál futását kezdeményezni tudjuk?

- *init*
- *start*
- *run*
- *resume*
- *sleep*

Mi annak a metódusnak a neve, amellyel a programszál futását le tudjuk állítani?

(Minden helyes választ jelöljön meg!)

- *sleep*
- *stop*
- *yield*
- *wait*
- *notify*
- *notifyAll*
- *synchronized*

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

```
public class Background extends Thread{
    public static void main(String argv[]){
        Background b = new Background();
        b.run();
    }
    public void start(){
        for (int i = 0; i <10; i++){
            System.out.println("Value of i = " + i);
        }
    }
}
```

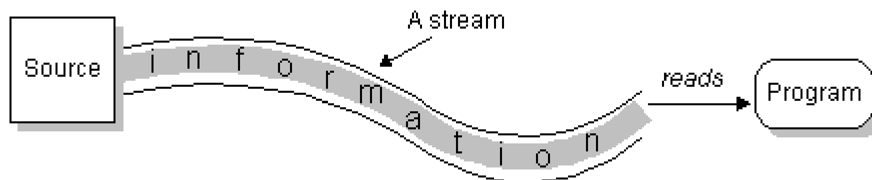
- Fordítási hiba, mert a *run* metódus nincs definiálva a *Background* osztályban
- Futási hiba, mert a *run* metódus nincs definiálva a *Background* osztályban
- A program lefut, és 0-tól 9-ig ír ki számokat
- A program lefut, de nincs kimenete

20.Fájlkezelés

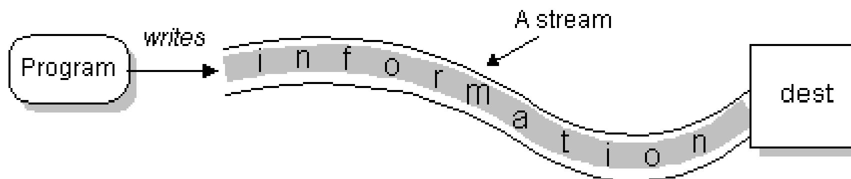
A legtöbb programnak szüksége van arra, hogy külső forrásból olvasson adatokat, vagy külső célba írja a működése (rész)eredményeit. A forrás és a cél sokféle lehet: konzol, háttértár, másik program, vagy egyéb speciális eszközök. Az adatok is különbözők lehetnek: számok, szövegek, vagy akár tetszőleges objektumok. Ez a fejezet bemutatja, hogy hogyan lehet Java nyelven kezelni az adatállományainkat.

20.1. Adatfolyamok

Információ kinyeréséhez a program megnyit egy adatfolyamot az információforráshoz (fájl, memória, socket), és sorban kiolvassa az információt, amint az ábrán látszik:



Hasonlóképpen, egy program külső célba is küldheti az információkat adatfolyam megnyitásával, és információkat írhat ki sorban, mint itt:



Nem számít, mekkora adat jön be, vagy megy ki, és nem számít a típusa, az algoritmus sorban olvassa és írja, a következőköz hasonlóan:

Olvasás

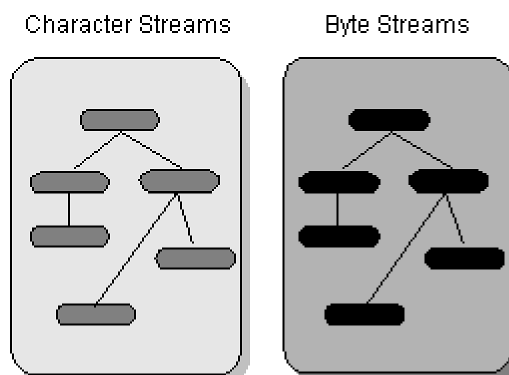
```
adatfolyam megnyitása
amíg van újabb információ
    olvasás
adatfolyam bezárása
```

Írás

```
adatfolyam megnyitása
amíg van újabb információ
    írás
adatfolyam bezárása
```

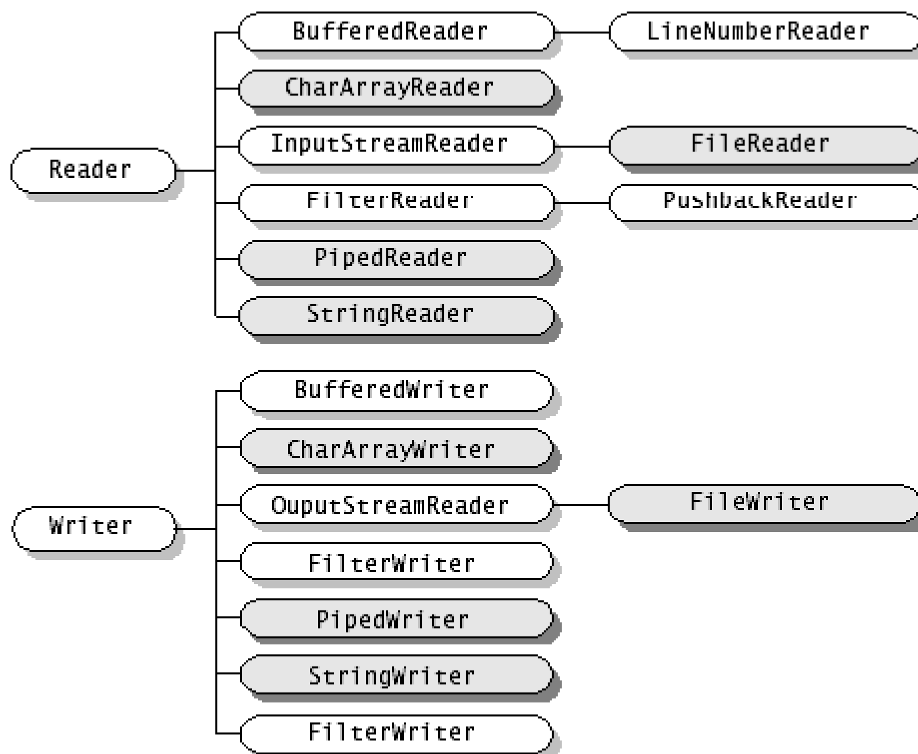
A *java.io* csomag tartalmazza azon osztályokat, melyek lehetővé teszik az írást és olvasást. Az osztályok használatához importálni kell a *java.io* csomagot.

Az adatfolyam osztályok két hierarchiára oszthatók az adattípusuk alapján (karakter vagy bájt):



Karakter folyamok

Reader és *Writer* a *java.io* absztrakt ősosztályai. 16 bites karakterekkel való műveleteket teszik lehetővé. A *Reader* és *Writer* leszármazott osztályai speciális műveleteket végeznek, és két külön kategóriára oszthatók: az adat olvasás és írás (szürkével jelezve) és feldolgozások végrehajtása (fehérek). Az utóbbiakat szűrő folyamoknak is nevezzük. A *Reader* és *Writer* osztályok osztályhierarchiáját mutatja az ábra.

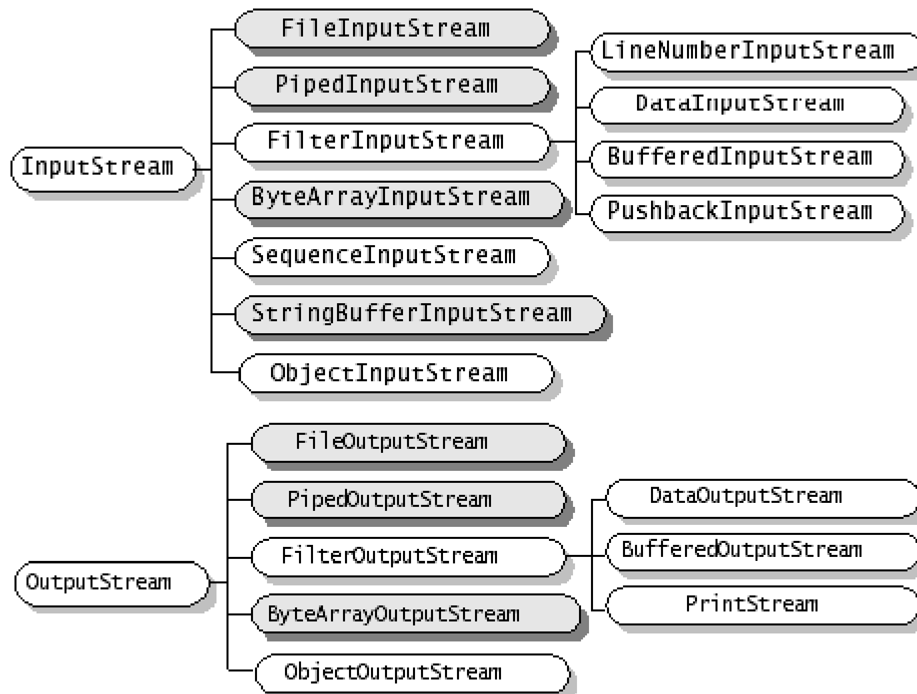


A legtöbb programnak szüksége van a *Reader* és *Writer* osztályokra szöveges információk olvasásához és írásához. Ezek bármilyen karaktert képes lekezelni Unicode karakterként.

Bináris folyamok

A 8-bites bájt írásához és olvasásához a programnak használnia kell a bináris folyamokat, melyek az *InputStream* és *OutputStream* leszármazottai. Ezek a folyamok jellemzően bináris adatok olvasására és írására alkalmasak, mint egy kép vagy hang. A bájt folyam két osztálya, *ObjectInputStream* és *ObjectOutputStream* használható objektumok soros folyamon keresztüli kezeléshez.

Az *InputStream* és *OutputStream* osztályok a *Reader* és *Writer* osztályokhoz hasonlóam két kategóriára oszthatók, a következő osztályhierarchia alapján: adatkezelő folyamok (szürke) és feldolgozó vagy más néven szűrő folyamok (fehér).



Az I/O szülőosztályok

Reader és *InputStream* azonos API-kat határoznak meg, de különböző adattípusra. A *Reader* metódusai karakterek és karaktertömbök olvasására alkalmasak:

```
int read()
int read(char cbuf[])
int read(char cbuf[], int offset, int length)
```

Az *InputStream* hasonló metódusai bináris adatok és tömbök olvasására alkalmasak:

```
int read()
int read(byte cbuf[])
int read(byte cbuf[], int offset, int length)
```

A *Reader* és *InputStream* is megvalósítja az adatfolyamon belüli állapot jelölését, ugrálást, és az aktuális pozíció visszaállítását.

A *Writer* és *OutputStream* hasonló szolgáltatásokkal rendelkeznek. A *Writer* karakterek, és karaktertömbök írását valósítja meg:

```
int write(int c)
int write(char cbuf[])
int write(char cbuf[], int offset, int length)
```

Az *OutputStream* azonos metódusú de bináris adatra:

```
int write(int c)
int write(byte cbuf[])
int write(byte cbuf[], int offset, int length)
```

Mind a négy osztály esetén igaz, hogy automatikusan megnyílik az adatfolyam, ha létre van hozva az objektum. Minden adatfolyam bezárható a *close* meghívásával. A szemét-

gyűjtő is bezárja, ha már nincs rá hivatkozás, de célszerű inkább direkt módon meghívni a *close* metódust.

Megjegyezés: A program „kötelessége”, hogy minden erőforrást csak a ténylegesen szükséges ideig foglaljon le. Éppen ezért az állomány megnyitását olyan későn kell megtenni, amikor csak lehet, és ugyanilyen hamar be is kell zárni.

Az adatfolyam osztályok használata

A táblázat a *java.io* adatfolyamokat és feladatuk leírását tartalmazza.

I/O típus	Osztályok	Leírás
Memória	*CharArrayReader* *CharArrayWriter* *ByteArrayInputStream* *ByteArrayOutputStream*	Az adatfolyamok kiolvasásra és a memóriába írására használható. Az adatfolyamok létrehozhatók egy meglévő tömbbel, és az olvasást és írást használva kiolvas és a tömbbe ír.
	StringReader *StringWriter*	A *StringReader* karaktereket olvas a memóriából. A *StringWriter* egy *String*-be ír. A *StringWriter* kiírandó karaktereket gyűjt egy *StringBuffer*-be, ami *String*-be konvertál.
	StringBufferInputStream	*StringBufferInputStream* hasonló a *StringReader*-hez, kivéve, hogy egy *StringBuffer*-ből karaktereket olvas.
Cső	*PipedReader* *PipedWriter* *PipedInputStream* *PipedOutputStream*	A bemenet és kimenet között valósít meg egy összekötő csövet. A csövek a kimenettől a másik bemenetig tartó folyamot valósítanak meg.
Fájl	*FileReader* *FileWriter* *FileInputStream* *FileOutputStream*	Együttesen fájl folyamoknak hívják, melyekkel fájlból olvashatunk, vagy fájlba írhatunk.
Lánc	*SequenceInputStream*	Több bemenet adatfolyamainak összefűzése egy bemeneti adatfolyamba.
Objektum sorosítása	*ObjectInputStream ObjectOutputStream*	Objektumok soros szervezésű tárolásához és visszaállításához.
Adat konverzió	*DataInputStream* *DataOutputStream*	Primitív adattípusok írása és olvasása gépfüggetlen formátumban.
Számláló	*LineNumberReader* *LineNumberInputStream*	Nyomon követhető a sorok száma olvasáskor.
Előre olvasás	*PushbackReader*	A bemeneti adatfolyamok egy vissza-

	PushbackInputStream	tehető pufferbe kerülnek. Az adatok olvasásakor néha hasznos a következő néhány bájt vagy karakter ismerete a következő lépés eldöntéséhez.
Kinyomtatás	*PrintWriter* *PrintStream*	Nyomtatási metódusokat tartalmaz. Az adatfolyamok tartalmának kiíratására használható.
Pufferelés	*BufferedReader* *BufferedWriter* *BufferedInputStream BufferedOutputStream*	Olvasáskor és íráskor az adatok pufferelődnek, ezáltal csökkentve a forrás adat eléréseinek számát. A pufferelt adatfolyamok hatékonyabbak, mint a nem pufferelt adatfolyamok, és gyakran használhatók más adatfolyamokkal is.
Szűrés	*FilterReader* *FilterWriter* *FilterInputStream* *FilterOutputStream*	Ezek az absztrakt osztályok a szűrő adatfolyamok vázát adják meg, milyen szűrés használható íráskor vagy olvasáskor.
Bájtok és karakterek közti konverzió	*InputStreamReader* *OutputStreamWriter*	Az olvasó és író páros hidat képez a bájt és a karakterfolyamok között. Az *InputStreamReader* bájtokat olvas egy *InputStream*-ből és karakterbe konvertálja, az alapértelmezett karakterkódolást vagy egy megnevezett karakterkódolást használva. Az *OutputStreamWriter* karaktereket konvertál bájtba, az alapértelmezett karakterkódolást vagy egy megnevezett karakterkódolást használva és a bájtokat egy *OutputStream*-be írja.

20.1.1 Fájl adatfolyamok használata

A fájl adatfolyamok talán a legkönnyebben megérthető adatfolyamok. A fájl adatfolyamok (*FileReader*, *FileWriter*, *FileInputStream*, és *FileOutputStream*) a natív fájlrendszer fájljait olvassák, vagy írják. Létrehozhatunk fájl adatfolyamot fájlnév *String*-ből, *File* objektummal, vagy *FileDescriptor* objektummal.

A következő *Copy* program *FileReader* és *FileWriter* segítségével az *input.txt* tartalmát átírja (átmásolja) az *ouput.txt* fájlba:

```
import java.io.*;

public class Copy {
    public static void main(String[] args) throws IOException{
        File inputFile = new File("input.txt");
        File outputFile = new File("output.txt");
```

```
        FileReader in = new FileReader(inputFile);
        FileWriter out = new FileWriter(outputFile);
        int c;

        while ((c = in.read()) != -1)
           out.write(c);

        in.close();
        out.close();
    }
}
```

A program nagyon egyszerű. A *FileReader* megnyitja a *farrago.txt* fájlt, a *FileWriter* pedig az *outagain.txt* fájlt. A program az *in* objektummal beolvas minden karaktert a bemeneten a bemeneti fájlból, és az *out* objektummal kiírja a karaktereket. Ha a bemenet elfogyott (-1), a program bezárja az olvasót és az írót is.

Megjegyzés: Érdemes megfigyelni, a *read* és *write* metódusok nem *char*, hanem *int* típussal dolgoznak. Ennek mindössze az az oka, hogy egy speciális, a *char* megengedett értéktartományán kívüli visszatérési értéket (a -1-et) is kell nyújtani abban az esetben, ha nincs több olvasható karakter, így a *char* nem lenne alkalmazható.

A *Copy* program ezt a kódot használja a *FileReader* létrehozásához:

```
File inputFile = new File("input.txt");
FileReader in = new FileReader(inputFile);
```

A kód létrehoz egy *File* objektumot, ami a fájl logikai leírására szolgál. A *File* segédosztályt a *java.io* tartalmazza. A *Copy* program ezt az objektumot használja a *FileReader* felépítéséhez. Azonban a program az *inputFile*-t akkor is használhatja, ha információt szeretnénk szerezni a fájl egészéről (pl. olvasható-e, mekkora a mérete stb).

A program futása után ugyanannak kell szerepelnie az *input.txt* és az *output.txt* fájlban is.

Ne feledjük, hogy a *FileReader* és a *FileWriter* 16-bites karaktert olvas és ír. Azonban, a legtöbb natív fájlrendszer 8-biten alapszik Az adatfolyamok kódolják a karaktereket, az alapértelmezett karakterkódolási séma alapján. A karakterkódolás megkapható a *System.getProperty("file.encoding")* használatával. Új alapértelmezett kódolás megadásához létre kell hozni egy *OutputStreamWriter* vagy egy *FileOutputStream* objektumot, és megadni a kódolást.

20.1.2 Szűrő adatfolyamok

A *java.io* csomag tartalmazza az absztrakt osztályok speciális beállításainak lehetőségét, mellyel definiálhatunk, és részlegesen implementálhatunk szűrőket, olvasás, illetve írás céljából. A használatos filter adatfolyamok a *FilterInputStream*, *FilterOutputStream*. Egy filter adatfolyam egy másik alap-adatfolyamra épül, melybe a *write* metódus fog adatokat menteni, de csak a szűrés után. Vannak adatfolyamok, melyek speciális feladatokat látnak el, mint például a konvertálás vagy számlálás folyamata.

A legtöbb filter adatfolyamot a *java.io* csomag által szolgáltatott származtatott osztályok nyújtják, ezek a következők:

- *DataInputStream* és *DataOutputStream*
- *BufferedInputStream* és *BufferedOutputStream*
- *LineNumberInputStream*
- *PushbackInputStream*

- *PrintStream*

A *java.io* csomag csak egy származtatott osztályát tartalmazza a *FilterReader*-nek, ez nem más, mint a *PushbackReader*. A következő fejezet bemutatja egy példán szemléltetve, hogyan használjuk a szűrő adatfolyamokat a *DataInputStream* és a *DataOutStream* használatával. Ugyancsak tárgyalja, hogyan hozhatunk létre származtatott osztályokat a *FilterInputStream* és *FilterOutputStream*-ből, melyek segítségével saját szűrő adatfolyamunkat valósíthatjuk meg.

Szűrő adatfolyamok használata

A szűrő objektumokat csatolni kell egy másik I/O adatfolyamhoz, példaként csatolhatunk szűrő adatfolyamot a standard input adatfolyamra:

```
BufferedReader d = new BufferedReader(new
    DataInputStream(System.in));
String input;

while ((input = d.readLine()) != null) {
    ... //do something interesting here
}
```

Megjegyzés: A *readLine* metódus nem használható a *DataInputStream*-ben, ezért hoztuk létre a *BufferedReader* típusú *d* szűrő objektumot.

A következőkben egy példán szemléltetjük a *DataInputStream* és *DataOutputStream* használatát. Ezek olyan szűrők melyek képesek írni/olvasni elemi adattípusokat.

Előfordul, hogy a feldolgozás független a kezelt adat formátumától, de néha szorosan összefügg az illető adatokkal vagy annak formátumaival, úgymint adatok írása, olvasása sorokból vagy cellákból.

DataInputStream és *DataOutputStream* használata

Ez a fejezet a *DataInutStream* és *DataOutputStream* osztályok használatát mutatja be egy *DataIODemo* példaprogramon keresztül, mely olvas és ír táblázatba (cellákba) foglalt adatokat. A cellák tartalmazzák az eladási árat, a rendelt áru mennyiségét, és a leírást. Elméletileg az adat a következőképpen néz ki, habár a gép ezt bináris adatként kezeli:

```
19.99   12      Java T-shirt
9.99    8       Java Mug
```

DataOutputStream-et, mint ahogyan más kimeneti adatfolyamokat, csatolni kell egy másik OutputStream-hez. Ebben az esetben a *FileOutputStream*-hez csatoltuk. Példa:

```
DataOutputStream out = new DataOutputStream(
                        new FileOutputStream("invoice1.txt"));
```

A *DataIODemo* a *DataOutputStream* speciális *write* metódusait használja írásra, amennyiben az megfelelő típusú.

```
for (int i = 0; i < prices.length; i ++) {
    out.writeDouble(prices[i]);
    out.writeChar('\t');
    out.writeInt(units[i]);
    out.writeChar('\t');
    out.writeChars(descs[i]);
    out.writeChar('\n');
}
out.close();
```

A *DataIODemo* program beolvassa az adatokat a *DataInputStream* speciális metódusai segítségével:

```
try {
    while (true) {
        price = in.readDouble();
        in.readChar();       //throws out the tab
        unit = in.readInt();
        in.readChar();       //throws out the tab
        char chr;
        desc = new StringBuffer(20);
        char lineSep =
            System.getProperty("line.separator").charAt(0);

        while ((chr = in.readChar() != lineSep) {
            desc.append(chr);
        }

        System.out.println("You've ordered " + unit
            +  " units of "
                               + desc + " at $" + price);
        total = total + unit * price;
    }
} catch (EOFException e) { }
System.out.println("For a TOTAL of: $" + total);
in.close();
```

Mihelyst beolvasta az adatokat, a *DataIODemo* megjeleníti az összegzést, a rendelt és az összes mennyiséget, majd bezárul.

Megjegyzés: a *DataIODemo* által használt ciklus a *DataInputStream*-ből olvassa az adatokat:

```
while ((input = in.read()) != null) {
    . . .
}
```

A *read* metódus *null* értékkel tér vissza, mely a fájl végét jelenti.

Számos alkalommal azonban ezt nem tehetjük meg, a szabálytalanul megadott értékek esetén hibajelzést kapunk. Tegyük fel, hogy -1 értéket szeretnénk ugyanerre a célra felhasználni. Ezt nem tehetjük meg, ugyanis a -1 nem szabályos érték, kiolvasása a *readDouble* vagy *readInt* használatával történik. Tehát *DataInputStream*-ek esetén nem kapunk *EOFException* kivételt.

A program futása a következő kimenetet eredményezi:

```
You've ordered 12 units of Java T-shirt at $19.99
You've ordered 8 units of Java Mug at $9.99
You've ordered 13 units of Duke Juggling Dolls at $15.99
You've ordered 29 units of Java Pin at $3.99
You've ordered 50 units of Java Key Chain at $4.99
For a TOTAL of: $892.8800000000001
```

20.2. Objektum szerializáció

A *java.io* két adatfolyama -az *ObjectInputStream* és az *ObjectOutputStream*- abban más az átlagos bájtfolyamoktól, hogy objektumokat tudnak írni és olvasni.

Az objektumok írásának az alap feltétele, hogy egy olyan szerializált formára hozzuk az objektumot, ami elegendő adatot tartalmaz a későbbi visszaalakításhoz. Ezért hívjuk az objektumok írását és olvasását objektum szerializációnak.

Objektumokat a következő két módon lehet szerializálni:

- Távoli eljáráshívás (*Remote Method Invocation*, RMI)
  Kliens és szerver közötti objektumátvitel esetén
- Könnyűsúlyú perzisztencia (*Lightweight persistence*)
  Egy objektum archiválása későbbi használatra

20.2.1 Objektumok szerializálása

Az *ObjectOutputStream* írása

A következő kódrészlet megkapja a pontos időt a *Date* objektum konstruktorának a meghívásával, és aztán szerializálja az objektumot:

```
FileOutputStream out = new FileOutputStream("theTime");
ObjectOutputStream s = new ObjectOutputStream(out);
s.writeObject("Today");
s.writeObject(new Date());
s.flush();
```

Az *ObjectOutputStream* konstruktorát egy másik adatfolyamra kell meghívni. Az előző példában az *ObjectOutputStream* konstruktora egy *FileOutputStream*-re hívódik, így egy 'theTime' nevű fájlba szerializálja az objektumot. Ezek után egy *String* („*Today*”) és egy *Date* objektum íródik az adatfolyamra az *ObjectOutputStream writeObject* nevű metódusával.

A *writeObject* metódus szerializálja az adott objektumot, rekurzívan átmenti a más objektumokra való hivatkozásait, így megtartja az objektumok közötti kapcsolatot.

Az *ObjectOutputStream* implementálja a *DataOutput* interfészt, ami primitív adattípusok írására való metódusokat tartalmaz (*writeInt*, *writeFloat*, *writeUTF*...). Ezeket a metódusokat használjuk primitív adattípusok *ObjectOutputStream*-re való írásához.

A *writeObject* metódus *NotSerializableException*-t dob, ha a megadott objektum nem szerializálható. Egy objektum csak akkor szerializálható, ha implementálja a *Serializable* interfészt.

Az *ObjectInputStream* olvasása

A következő példaprogram kiolvassa a *theTime* fájlba előzőleg kiírt *String* és *Date* objektumokat:

```
FileInputStream in = new FileInputStream("theTime");
ObjectInputStream s = new ObjectInputStream(in);
String today = (String)s.readObject();
Date date = (Date)s.readObject();
```

Az *ObjectOutputStream*-hez hasonlóan az *ObjectInputStream* konstruktorát is egy másik adatfolyamra kell hívni. Mivel egy fájlba írtuk az objektumokat, itt is egy *FileInputStream*-re kell hívni az *ObjectInputStream*-et. Az olvasást az *ObjectInputStream readObject* metódusa végzi. Az objektumokat mindenképpen abban a sorrendben kell kiolvasni, ahogy az adatfolyamra lettek írva.

A *readObject* metódus deszerializálja a soron következő objektumot. Rekurzívan vizsgálja a hivatkozásokat, hogy az összes hivatkozott objektumot is deszerializálja, így megtartva az objektumok közötti kapcsolatot.

Az *ObjectInputStream* adatfolyam implementálja a *DataInput* interfészt, ami primitív adattípusokat olvasó metódusokat tartalmaz. A *DataInput* metódusai a *DataOutput* metódusainak olvasó megfelelői (*readInt*, *readFloat*, *readUTF*...).

20.2.2 Objektum szerializáció a gyakorlatban

Egy objektum csak akkor szerializálható, ha az osztálya implementálja a *Serializable* interfészt. Ezt könnyen megtehetjük:

```
public class MySerializableClass implements Serializable {
    ...
}
```

A *Serializable* egy üres interfész (vagyis nincs egyetlen tagja sem), csak a szerializálható osztályok elkülönítésére való. Az osztály példányainak a szerializálását az *ObjectOutputStream* osztály *defaultWriteObject* nevű metódusa végzi. A deszerializáció az *ObjectInputStream* osztály *defaultReadObject* metódusával végezhető el. A legtöbb objektumhoz ez a módszer elég is. Néhány esetben viszont lassú lehet, és az osztálynak is szüksége lehet a szerializáció feletti határozottabb vezérlésre.

A szerializáció testreszabása

Az objektum szerializáció testreszabásához két metódus felülírására van szükségünk: a *writeObject* metódus (írásra) és a *readObject* (olvasásra vagy adatfrissítésre).

A *writeObject* deklarálása:

```
private void writeObject(ObjectOutputStream s) throws
IOException {
    s.defaultWriteObject();
    // testreszabott szerializáló kód
}
```

A *readObject* metódust deklarálása:

```
private void readObject(ObjectInputStream s) throws IOException
{
    s.defaultReadObject();
    //  testreszabott deszerializáló kód
    ...
    // objektumokat frissítő kód (ha szükséges)
}
```

Mindkét metódust pontosan a bemutatott módon kell deklarálni.

A *writeObject* és a *readObject* metódusok csak az adott osztály szerializálásáért felelősek, a szülő osztályok esetleges szerializálása automatikus. Ha egy osztálynak a szülőosztályaival teljesen összhangban kell lennie a szerializációhoz, akkor az *Externalizable* interfészt kell implementálni.

20.2.3 Az *Externalizable* interfész implementálása

Hogy egy osztály teljes mértékben kontrollálni tudja a szerializációs folyamatot, implementálnia kell az *Externalizable* interfészt. *Externalizable* objektumoknál csak az objektum egy 'reprezentációja' kerül az adatfolyamra. Az objektum maga felelős a tartalom írásáért, olvasásáért és a szülőobjektumokkal való együttműködésért.

Az *Externalizable* interfész definíciója:

```
package java.io;

public interface Externalizable extends Serializable {
    public void writeExternal(ObjectOutput out)
        throws IOException;
    public void readExternal(ObjectInput in)
        throws IOException, java.lang.ClassNotFoundException;
}
```

Egy *Externalizable* osztálynál a következőket kell szem előtt tartani:

- implementálnia kell a *java.io.Externalizable* interfészt.
- implementálnia kell a *writeExternal* metódust az objektum állapotának az elmentéséhez.
- implementálnia kell a *readExternal* metódust a *writeExternal* által az adatfolyamra írt tartalom visszaállításához.
- külső adatformátum írásáért és olvasásáért kizárólag a *writeExternal* és a *readExternal* metódusok felelősek.

A *writeExternal* és a *readExternal* metódusok publikusak, így lehetőséget nyújtanak arra, hogy egy kliens az objektum metódusai nélkül is tudja olvasni/írni az objektum adatait, ezért csak kevésbé biztonság-igényes esetekben alkalmazhatóak.

20.2.4 Az érzékeny információ megvédése

Ha egy osztályunk elérést biztosít bizonyos forrásokhoz, gondoskodnunk kell a kényes információk és függvények védelméről. A deszerializáció alatt az objektum *private* állapota is helyreáll. Például egy fájlleíró eljárás hozzáférést biztosít egy operációs rendszerforráshoz. Ha ez egy adatfolyamon keresztül áll helyre, a sebezhetősége miatt visszaélésekre ad lehetőséget. Ezért a valós időben történő szerializáció esetében nem szabad feltétlenül megbízni az objektum-reprezentációk valódiságában. Két módon oldhatjuk meg a problémát: vagy nem szabad az objektum érzékeny tulajdonságait az adatfolyamról visszaállítani, vagy ellenőrizni kell azokat.

A védelem legegyszerűbb módja az érzékeny adatok *private transient*-ként való feltüntetése. A *transistent* és a *static* adattagok nem kerülnek szerializálásra, így nem is jelennek meg az adatfolyamon. Mivel a *private* adattagok olvasását és írását nem lehet az osztályon kívülről helyettesíteni, az osztály *transient* tagjai így biztonságban vannak.

A különösen érzékeny osztályokat egyáltalán nem szabad szerializálni. Ez esetben egyszerűen nem szabad implementálni sem a *Serializable*, sem az *Externalizable* interfészt.

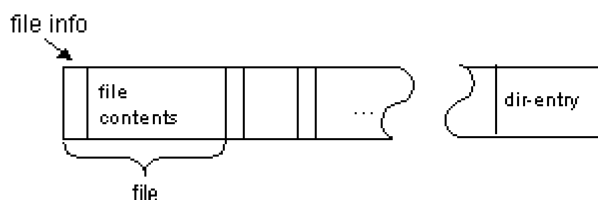
Néhány osztálynál különösen előnyös az írás és az olvasás arra való használata, hogy a deszerializálás közben kezelni és ellenőrizni tudjuk az érzékeny adatot. A *writeObject* és a *readObject* metódusok itt csak a helyes adatot menthetik, vagy állíthatják helyre. Ha a

hozzáférés tiltására van szükség, egy *NotSerializableException* dobása elég, hogy megtagadjuk a hozzáférést.

20.3. Közvetlen elérésű állományok

Az ki és bemenő adatfolyamok eddig a leckéig soros hozzáférésűek voltak, melyek írása és olvasása az elejétől a végéig sorban történik. Bár az ilyen adatfolyamok hihetetlenül hasznosak, de ezek csak a soros hozzáférésű médiumok (papír és mágnesszalag) következtében jöttek létre. Másrészről a közvetlen hozzáférésű fájlok, nem-soros, hanem véletlenszerű (közvetlen) hozzáférést biztosítanak a fájl tartalmához.

Tehát miért van szükség közvetlen elérésű fájlokra? Gondoljunk csak a tömörített ZIP formátumra. A ZIP állományok fájlokat tartalmaznak, tömörítve a helytakarékosság miatt. A ZIP állományok tartalmaznak egy az állomány végén elhelyezett *dir* bejegyzést, mely megmutatja, hogy a különböző fájlok hol találhatók a ZIP belsejében.



Feltéve, hogy egy meghatározott fájlt akarunk kicsomagolni a ZIP-ből, soros hozzáférés esetén a következőket kell tennünk:

- Megnyitjuk a ZIP fájlt
- Végigkeressük a ZIP fájlt a keresett állományért
- Kicsomagoljuk az állományt
- Bezárjuk a ZIP-et

Ezt az algoritmust használva, átlagban legalább a ZIP állomány felét végig kell olvasni a keresett állományért. Kitömöríthetjük ugyanezt a fájlt a ZIP-ből sokkal hatékonyabban, ha a közvetlen elérésű fájl *seek*, azaz keresési funkcióját használjuk a következő lépések szerint:

- Megnyitjuk a ZIP fájlt
- Megkeressük a *dir* bejegyzést és azon belül a kitömörítendő fájl helyét
- Megkeressük (visszafelé) a fájl pozícióját a ZIP-en belül
- Kicsomagoljuk az állományt
- Bezárjuk a ZIP-et

Ez a módszer sokkal hatékonyabb, mert csak a *dir* bejegyzést és kicsomagolandó fájlt kell beolvasni. A *java.io* csomagban található *RandomAccessFile* osztály implementálja közvetlen elérésű fájlokat.

20.3.1 A közvetlen elérésű állományok használata

A *RandomAccessFile* osztály mind írásra, mind olvasásra alkalmas, ellentétben a *java.io* csomag által tartalmazott ki-, és bemeneti adatfolyamokkal. Egy *RandomAccessFile* objektumot a szerint hozunk létre különböző paraméterekkel, hogy írni vagy olvasni akarunk vele. A *RandomAccessFile* nincs kapcsolatban a *java.io* csomag ki-, és bemeneti adatfolyamaival, és ez az, amiért nem az *InputStream* és *OutputStream* leszármazottja. Ennek az a hátránya, hogy a *RandomAccessFile*-ra nem alkalmazhatók azok a szűrők, amelyek az adatfolyamokra igen. Habár a *RandomAccessFile* implementálja a *DataInput* és *DataOutput* interfészt, tehát ha készítünk egy olyan filtert, mely vagy az *DataInputtal* vagy a *DataOutputtal* együttműködik, ez működni fog néhány soros elérésű állománnyal valamint az összes közvetlen elérésűvel.

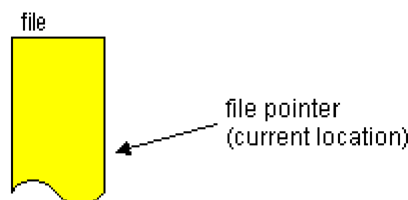
Ha meg akarjuk nyitni olvasásra az *input.txt* állományt, akkor a következő kóddal tehetjük meg:

```
new RandomAccessFile("input.txt", "r");
```

Ha olvasni és írni is akarjuk:

```
new RandomAccessFile("input.txt", "rw");
```

Ezek után a *read* és *write* metódusok a szokásos módon állnak rendelkezésre.



Fontos további szolgáltatás, hogy ezen kívül az aktuális állomány pozíció közvetlenül is manipulálható a következő metódusokkal:

int skipBytes(int)	előre mozgatja az aktuális pozíciót
void seek(long)	a megadott pozícióra mozgatja az aktuális pozíciót
long getFilePointer()	lekérdezi a pillanatnyi pozíciót

20.4. További osztályok és interfészek

Az előzőekben bemutatott osztályokon és interfészeken kívül *a java.io* csomag többek között a következő osztályokat és interfészeket tartalmazza:

File

Egy fájlt képvisel a natív fájlrendszerben. Létrehozhatunk a *File* objektumot egy állományhoz a natív fájlrendszerben, és adatokat kérdezhetünk le az állományról. Például a teljes elérési útját.

FileDescriptor

Egy fájlkezelőt valósít meg nyitott állományok és portok esetén. Nem sokszor használt.

StreamTokenizer

Adatfolyam tartalmát tördeli részekre (tokenekre). A tokenek a legkisebb egységek (szavak, szimbólumok), melyeket a szövegfelismerő algoritmusok képesek felismerni. A *StreamTokenizer* objektumok bármely szöveges fájl elemzésére használhatók. Például használható a forrásállományoktól a változónevek, operátorok, illetve HTML fájlokból HTML tagok kigyűjtésére. (A *StringTokenizer*-hez hasonló feladatot lát el.)

FilenameFilter

A *File* osztály *list* metódusa használja, hogy mely állományok vannak egy listázandó könyvtárban. A *FilenameFilter* használható az egyszerű kifejezés stílusú állomány kereső minták implementálására, mint például **.doc*, vagy akár sokkal bonyolultabb szűrésekre is.

Találhatóak még egyéb input osztályok a *java.util.zip* csomagban:

CheckedInputStream és *CheckedOutputStream*

Egy ki- és bementi adatfolyam pár, mely egy ellenőrző összeget számít az adatok írásakor és olvasásakor.

DeflaterOutputStream és *InflaterInputStream*

Betömöríti és kitömöríti az adatokat íráskor és olvasáskor.

GZIPInputStream és *GZIPOutputStream*

GZIP formában tömörített adatokat ír és olvas.

ZipInputStream és *ZipOutputStream*

ZIP formában tömörített adatokat ír és olvas.

20.5. Ellenőrző kérdések

- Mi az író karaktercsatornák absztrakt őse?
- Mi az olvasó karaktercsatornák absztrakt őse?
- Mi az író bájtcsatornák absztrakt őse?
- Mi az olvasó bájtcsatornák absztrakt őse?
- Melyik metódus használható az összes olvasó csatorna esetén?
- Melyik metódus használható az összes író csatorna esetén?
- Mondjon példát memória-csatornára!
- Írjon példát a *File* osztály szolgáltatásaira!
- Mik a szűrő folyamok?

Igaz vagy hamis? Indokolja!

- A *RandomAccessFile* egy szűrő folyam.

- A véletlen elérésű állomány mutatóját lehet állítani.
- A véletlen elérésű állomány segítségével egy fájlnak bármelyik bájtja módosítható.
- Az objektumfolyamra írható primitív adat is.
- Az objektumfolyamra írt objektumnak implementálnia kell a *Serializable* interfészt.
- A pufferező folyamhoz mindig csatolni kell egy másik folyamatot.

A következők közül melyik hoz létre *InputStreamReader* objektumot?

(Minden helyes választ jelöljön meg!)

- *new InputStreamReader(new FileInputStream("data"));*
- *new InputStreamReader(new FileReader("data"));*
- *new InputStreamReader(new BufferedReader("data"));*
- *new InputStreamReader("data");*
- *new InputStreamReader(System.in);*

A következők közül melyik korrekt?

- *RandomAccessFile("data", "r");*
- *RandomAccessFile("r", "data");*
- *RandomAccessFile("data", "read");*
- *RandomAccessFile("read", "data");*

Melyikre nem képes a *File* osztály?

- Aktuális könyvtár beállítása
- Szülő könyvtár nevének előállítása
- Állomány törlése
- Állomány(ok) keresése

21.Gyűjtemények

A gyűjtemények (vagy más néven tárolók, konténerek, kollekciók) olyan típuskonstrukciós eszközök, melynek célja egy vagy több típusba tartozó objektumok memóriában történő összefoglaló jellegű tárolása, manipulálása és lekérdezése.

21.1. A gyűjtemény keretrendszer

A gyűjtemény keretrendszer (*Java Collections Framework,* JCF) egy egységes architektúra, ami a gyűjtemények használatára és manipulálására szolgál. A gyűjtemény keretrendszer tartalmazza:

- **interfészek**: absztrakt adattípusok reprezentációja. Az interfészek lehetővé teszik a gyűjtemények szolgáltatásainak (publikus interfészének) megvalósítás-független ábrázolását.
- **implementációk**: a gyűjtemény interfészek konkrét implementációi. Főként ezek az újrafelhasználható adatstruktúrák.
- **algoritusok**: azok a metódusok, amelyek hasznos műveleteket valósítanak meg, mint például a keresés vagy a rendezés egy objektumon, ami implementálható különböző gyűjtemény interfészeken. Ezeket az algoritmusokat többalakúnak hívjuk: azonos metódus használható különböző implementációk esetén is. Elsősorban az algoritmusok újrafelhasználható funkcionalitása.

A JCF-hez hasonló legismertebb megoldás a C++ nyelvű szabványos minta-könyvtár (*Standard Template Library*, STL).

21.1.1 A Gyűjtemény keretrendszer használatának előnyei

A Java Gyűjtemény keretrendszer az alábbi előnyökkel rendelkezik:

Csökkenti a fejlesztési időt

Mivel kész adatstruktúrák és algoritmusok állnak rendelkezésünkre, a Gyűjtemény keretrendszer lehetővé teszi, hogy a program fontosabb részével foglalkozzunk, ahelyett, hogy alacsonyszintű programozással kelljen foglalkoznunk.

Növeli a programozás sebességét és minőségét

A JCF gyors és jó minőségű algoritmus-, és adatstruktúra-implementációkkal rendelkezik. A különböző implementációk minden egyes interfésznél felcserélhetők, így a programokat könnyen össze lehet hangolni a gyűjtemény implementációkkal. Több időt tud arra fordítani, hogy növelje a programok minőségét és sebességét, mivel megszabadítja attól a rabszolgamunkától, hogy saját adatstruktúrákat kelljen írnia.

Megengedi az együttműködést a nem kapcsolódó API-k között

Ha két különböző függvénykönyvtár nem illeszthető egymáshoz közvetlenül, akkor lehet akár a keretrendszer a közös nevező az illesztés megteremtése érdekében.

Csökkenti az új API-k használatának és tanulásának nehézségét

Régen minden egyes API-nál egy kis segéd API-t készítettek arra, hogy manipulálja az egyes gyűjteményeket. Kevés összefüggés volt az erre a célra készült gyűjtemények segéd API-jai között, így külön-külön meg kell tanulni azokat, így könnyen hibát ejthetünk ezek használatával. Az általános gyűjtemény interfészek megjelenésétől ez a probléma már a múlté.

Megkönnyíti az új API-k tervezését

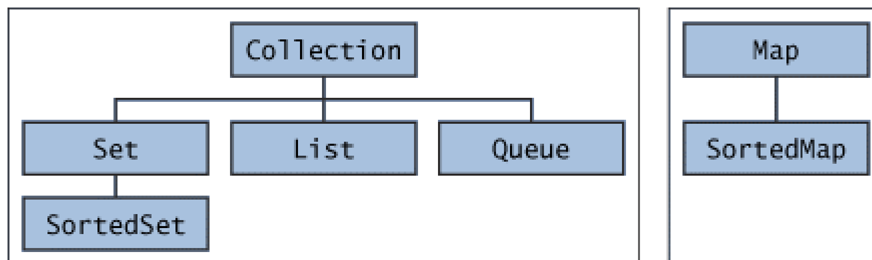
A tervezőknek és a kivitelezőknek nem kell újra kidolgozni az API-t, valahányszor készítenek egy gyűjteményekre alapozott API-t, hanem az általános gyűjtemény interfészt használhatják.

Elősegíti a szoftver újrafelhasználhatóságát

Az gyűjtemény interfészekkel összhangba hozott új adatstruktúrák természetesen újra felhasználhatók. Hasonlóan, új algoritmus fejlesztésekor könnyen hozzákapcsolható lesz az összes létező eddigi megvalósításhoz.

21.2. Interfészek

A gyűjtemény interfészek különböző típusú gyűjtemények leírását teszik lehetővé, amint az alábbi ábra mutatja. Ezek az interfészek lehetővé teszik a különböző gyűjtemény-reprezentációk egységes kezelését. A gyűjtemény interfészek a Gyűjtemény keretrendszer alapjainak tekinthetők.



Ahogy az ábrán látható, a gyűjtemény interfészek hierarchiát alkotnak: a *Set* egy specializált fajtája a *Collection*-nek, a *SortedSet* specializált fajtája a *Set*-nek, és így tovább. A hierarchia két egymástól független fából áll: a *Map* nem *Collection* leszármazott.

Vegyük figyelembe, hogy a gyűjtemény interfészek (általános, generikus) típus paraméterekkel dolgoznak. Például a *Collection* interfész deklarációja:

```
public interface Collection<E> ...
```

Az <*E*> szintaxis azt jelenti, hogy az interfész általános (generikus) típussal működik. Amikor deklarálunk egy *Collection*-t, meg tudjuk határozni, hogy milyen típusú objektumot tartalmaz a gyűjtemény. A típus paraméter a fordítóprogram számára lehetővé teszi, hogy csak az adott típusú objektumot engedje belerakni a gyűjteménybe, így csökkenti a futásidejű hibák számát.

Mikor megértjük, hogyan használjuk ezeket az interfészeket, akkor megértettük a Gyűjtemény keretrendszer lényegét. Ez a fejezet be fogja mutatni az általános irányelveket, hogy hogyan lehet hatékonyan használni ezeket az interfészeket. Továbbá irányt ad, hogy mikor melyik interfészt használjuk. Minden egyes interfészhez tanulni fogunk programozási trükköket, hogy segítsen megoldani a legtöbb későbbi problémát.

Hogy megőrizze a gyűjtemény interfészek kezelhetőségét, a Java platform nem különíti el az interfészeket a tartalom módosíthatósága szempontjából. (Bizonyos esetekben a gyűjtemények lehetnek állandók, fix méretűek vagy éppen csak bővíthetők.) A módosítási lehetőségek az egyes interfészekben szabadon választhatók: az adott implementáció választhatja azt is, hogy nem támogatja az összes műveletet. Ha egy nem támogatott művelet kerül meghívásra, az adott gyűjtemény implementáció az *UnsupportedOperationException* kivételt dobja. A Java platform minden általános célú implementációja támogatja az összes opcionális műveletet is.

21.2.1 A gyűjtemény interfészek

Collection

A gyűjtemény hierarchia gyökere. A *Collection* objektumok csoportját reprezentálja, amiket elemeknek hívjuk. A gyűjtemény interfész a legkisebb közös nevező a gyűjtemény implementációk között, és akkor érdemes ezt választani, ha a lehető legnagyobb rugalmasságra van szükség. A gyűjtemények néhány típusa engedélyezi az elemek duplikálását, a többi pedig nem. A gyűjtemények néhány típusa rendezett, a többi nem rendezett. A Java platform nem biztosít közvetlen implementációt ehhez az interfészhez, ehelyett a sokkal specifikusabb al-interfészeket implementálja.

Set

Ez a gyűjtemény nem tartalmazhat duplikált elemeket. Ezt az interfészt halmazok tárolására szokták használni, mint például a futó processzek halmaza.

List

A *List* rendezettséget biztosító gyűjtemény, néha szekvenciának is hívják. A listák tartalmazhatnak duplikált elemeket. A lista felhasználója egész index segítségével hozzá tud férni az elemekhez (hasonlóan a tömbök indexeléséhez), valamint lehetőséget kap a lista egyszerű bejárásához.

Queue

Tipikusan ez a gyűjtemény szokta tárolni a feldolgozásra váró elemeket. A *Collection* alapműveletein felül lehetőséget ad további beszúrási, kivételi, és megtekintési műveletekre.

Legtöbbször, de nem szükségképpen az elemeket FIFO (*first-in-first-out*) elv szerint rendezi. A kivételek között van a prioritási sor, amelyik rendezi az elemeket valamilyen szempont alapján. Bármilyen rendezést használunk, a sor elején van az az elem, amit legelőször el szeretnénk távolítani a *remove* vagy a *poll* metódus meghívásával. A FIFO sor minden új elemet a sor végére illeszt be. Más fajta sorok használhatnak más elhelyezési szabályokat.

Megjegyzés: A *Queue* gyűjteményekkel terjedelmi okokból a továbbiakban nem foglalkozunk.

Map

A kulcsokat értékekké képezi le. A leképezések nem tartalmazhatnak duplikált kulcsokat. Minden egyes kulcs legfeljebb egy értéket tud leképezni. Másként fogalmazva a *Map* kulcs-érték párokat tárol.

Az utolsó két gyűjtemény interfész csupán rendezett verziója a *Set*-nek és a *Map*-nek:

SortedSet

Az elemek növekvő sorrendben tárolását teszi lehetővé. Számos kiegészítő művelettel biztosítja, hogy kihasználhassuk a rendezettséget. Ezt az interfészt szokták használni olyan rendezett halmazként, mint például egy szólista vagy tagsági lista.

SortedMap

A leképezések kulcs szerinti növekvő sorrendbe tárolását teszi lehetővé. A rendezett leképezést használjuk a rendezett gyűjtemények kulcs/érték párjainál. Ilyen például a szótár vagy a telefonkönyv.

21.2.2 A *Collection* interfész

A *Collection* az objektumok (elemek) csoportját tárolja.

```
public interface Collection<E> extends Iterable<E> {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);         // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();

    // Bulk Operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); // Optional
    boolean removeAll(Collection<?> c);        // Optional
    boolean retainAll(Collection<?> c);        // Optional
    void clear(); // Optional

    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Az interfésznek van metódusa, ami megmondja hány elem van a gyűjteményben (*size*, *isEmpty*), leellenőrzi, hogy az adott objektum benne van-e a gyűjteményben (*contains*), hogy hozzáadjon elemet gyűjteményhez vagy elvegye belőle (*add*, *remove*), bejárót (más néven iterátort) szolgáltat a bejáráshoz (*iterator*).

Az *add* metódus *boolean* visszatérési értéke azt jelzi, hogy történt-e a gyűjteményben változás. Ugyanis olyan implementáció esetén, ahol az elemek többszörös tárolása nem megengedett, nem történik változás, ha az elemet már eleve tartalmazta a gyűjtemény. Hasonlóan, a *remove* metódus esetén csak akkor változik a gyűjtemény állapota, ha benne volt az adott elem, és így sikeres volt a kivétel.

A gyűjtemények bejárása

Két módja van hogy bejárjuk a gyűjteményeket. Az egyik a *for-each* ciklussal, a másik az iterátorok használatával.

For-each ciklus

A *for-each* ciklus megengedi, hogy bejárjuk a gyűjteményt vagy tömböt a *for* ciklus használatával. A következő kód a *for-each* ciklust használja, és minden elemet külön sorban jelenít meg:

```
for (Object o : collection)
    System.out.println(o);
```

Iterátorok

Az iterátor egy olyan objektum, ami megengedi, hogy bejárjuk a gyűjteményt és eltávolítsuk az elemeket a gyűjteményből, ha akarjuk.

Az *Iterator* interfész:

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); // Optional
}
```

A *hasNext* metódus visszatérési értéke *true*, ha az iterációnak van még be nem járt eleme. A *remove* metódus eltávolítja a gyűjteményből az utolsó elemet, amit a *next* hívására kaptunk. A *remove* metódus kivételt dob, ha még nem volt kiolvasott elem.

Az *Iterator.remove* az egyetlen biztonságos út, hogy módosítsuk a gyűjteményt az iteráció folyamán. A viselkedés meghatározatlan, ha a gyűjtemény más egyéb úton módosítva lett, amíg az iteráció folyamatban volt.

Iterátor használata célszerű a *for-each* ciklus helyett a következő esetekben:

- Törölni szeretnénk a bejárás közben.
- Ki szeretnénk cserélni az elemeket.
- Egyszerre többféle bejárásra is szükség van.

A következő metódus mutatja, hogyan használjuk az iterátort szűrőnek egy tetszőleges gyűjteményben (ezzel áttekintjük, hogy a gyűjtemény hogyan távolítja el a specifikus elemeket):

```
static void filter(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); )
        if (!cond(i.next()))
            i.remove();
}
```

Ez az egyszerű kis kódrészlet sokoldalú, ami azt jelenti, hogy működik bármely gyűjteményben, kivétel nélkül minden implementációban. Ez a példa bemutatja, milyen könnyű sokoldalú algoritmust írni a Gyűjtemény keretrendszer segítségével.

A *Collection* interfész tömeges metódusai

A tömeges metódusok nem egy elemmel, hanem egész gyűjteménnyel hajtanak végre műveleteket. Implementálni lehet ezeket a metódusokat az alap metódusokat használva, bár a legtöbb esetben az ilyen implementációk kevésbé hatékonyak.

A tömeges metódusok

- *containsAll*
  true a visszatérési értéke, ha a cél gyűjtemény tartalmaz minden elemet a meghatározott gyűjteményben (részhalmaz)
- *addAll*
  Minden elemet hozzáad a meghatározott gyűjteményhez (unió)
- *removeAll*
  Eltávolítja a célgyűjteményből az összes olyan elemet, amit a meghatározott gyűjtemény tartalmazott (különbség)
- *retainAll*
  Eltávolítja a célgyűjteményből az összes olyan elemet, amit a meghatározott gyűjtemény nem tartalmazott. Azaz, csak azokat tartja meg a célgyűjteményben, amiket a meghatározott gyűjtemény is tartalmazott. (metszet)
- *clear*
  Eltávolítja az összes elemet a gyűjteményből.

Az *addAll*, *removeAll*, és a *retainAll* metódus mindig *true*-val tér vissza, ha a célgyűjtemény módosítva volt a művelet végrehajtása alatt.

Egy egyszerű példa tömeges metódusok erejéről: a következő kifejezés eltávolítja az *e* elem minden példányát a *c* gyűjteményből:

```
c.removeAll(Collections.singleton(e));
```

Különösen praktikus, ha le akarjuk törölni az összes *null* elemet a gyűjteményből:

```
c.removeAll(Collections.singleton(null));
```

A kifejezés használja a *Collections.singleton* metódust, amelyik egy statikus gyártó metódus, ami visszaad egy olyan gyűjteményt, amely csak egy meghatározott elemet tartalmaz.

A *Collection* interfész tömb metódusai

A *toArray* metódusok olyanok, mint egy híd a gyűjtemények és a régebbi API-k között, amik tömböket várnak a bementen. A tömb metódusok lehetővé teszik, hogy a gyűjtemény tartalmát tömbként is elérhetővé tegyék. A paraméterek nélküli változat készíti el egy új *Object* tömböt. A bonyolultabb formája lehetővé teszi, hogy a hívó paraméterként egy tömböt adjon, és az eredmény is ilyen típusú tömb lesz.

Például, feltételezzük, hogy *c* egy gyűjtemény. Ekkor a következő metódus létrehoz egy tömböt, amelynek métere megegyezik a gyűjtemény méretével, elemei pedig a gyűjtemény elemei:

```
Object[] a = c.toArray();
```

Tegyük fel, hogy *c* ismert és csak sztringet tartalmaz (*c* típusa *Collection<String>*). Ekkor használható a következő változat is:

```
String[] a = c.toArray(new String[0]);
```

21.2.3 A *Set* interfész

A *Set* egy speciális *Collection*, amely nem tartalmaz duplikált elemeket. A *Set* nem tartalmaz új metódust a *Collection*-tól örököltekhez képest. A *Set* a tartalmazott objektumok

equals és a *hashcode* metódusát használja arra, hogy a többszörös tárolást kiszűrje. Két *Set* objektum akkor egyenlő, ha ugyanazon elemeket tartalmazzák. A *Set* interfész:

```
public interface Set<E> extends Collection<E> {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);         // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();

    // Bulk Operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); // Optional
    boolean removeAll(Collection<?> c);        // Optional
    boolean retainAll(Collection<?> c);        // Optional
    void clear();                              // Optional

    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

A Java platform három általános *Set* implementációt tartalmaz: a *HashSet*, *TreeSet*, és *LinkedHashSet*. *HashSet* esetén az elemeket egy hash táblában tárolja, ez a legjobb választás, ha az elemek bejárása nem fontos. A *TreeSet* az elemeket egy piros-fekete fában tárolja, ami az elemek bejárását hatékonyan meg tudja valósítani, de egy elem elérése, beszúrása lassabb, mint a *HashSet* esetén. A *LinkedHashSet* ötvözi a láncolt listákat a hash táblával.

Nézzünk egy egyszerű példát. Feltételezzük, hogy van egy *c* nevű *Collection*-ünk, és szeretnénk még egy *Collection*-t létrehozni, amely ugyanazokat az elemeket tartalmazza, de minden elemet csak egyszeresen.

```
Collection<Type> noDups = new HashSet<Type>(c);
```

Létrejön egy *Set,* amely nem tartalmaz duplikált elemeket, de a *c* összes eleme benne van. Az alapértelmezett konverziós konstruktort használjuk a tényleges létrehozásra.

A *Set* interfész alapvető műveletei

A *size* metódus visszatér az elemek számával. Az *add* metódus a *Set*-hez adja a megadott elemet, ha nem tudja, akkor *false* értékkel tér vissza. A *remove* metódus eltávolítja a megadott elemet a listából, ha ez sikerült, akkor *false* értékkel tér vissza. Az *iterator* metódus visszaad egy *Iterator* objektumot.

A következő példa a parancssori paraméterként kapott szavakat eltárolja az *s* objektumban, kiírja a duplikált szavakat, végül az eltérő szavak számát, és a szavakat:

```
import java.util.*;
public class FindDups {
    public static void main(String args[]) {
        Set<String> s = new HashSet<String>();
        for (String a : args)
            if (!s.add(a))
                System.out.println("Duplicate: " + a);
        System.out.println(s.size()+" distinct words: "+s);
    }
}
```

Futtassuk a programot:

```
java FindDups i came i saw i left
```

A kimenet a következő lesz:

```
Duplicate: i
Duplicate: i
4 distinct words: [i, left, saw, came]
```

Megjegyezzük, hogy a referencia típusát szokás interfész típussal (*Set*) leírni, míg az objektum típusa egy implementációs osztály lesz (*HashSet*). Ez egy nagyon ajánlott programozási gyakorlat, amivel könnyebb rugalmas kódot készíteni. Ha később esetleg abc sorrendben akarjuk a szavakat kiírni, csupán le kell cserélni *HashSet*-et *TreeSet*-re. Ez az egyszerű, egysoros változtatás ilyen eredményt produkál:

```
Duplicate word: i
Duplicate word: i
4 distinct words: [came, i, left, saw]
```

A *Set* interfész tömeges metódusai

A tömeges metódusok különösen jó szolgálatot tesznek a *Set* esetén, hiszen így egyszerűen megvalósíthatók az algebrából ismert halmazműveletek:

- *s1.containsAll(s2):*
  Igaz logikai értékkel tér vissza, ha s2 részhalmaza *s1*-nek (*s2* részhalmaza *s1*-nek, ha *s1* tartalmazza az *s2* összes elemét).
- *s1.addAll(s2):*
  Az *s1*-be *s1* és *s2* uniója kerül.
- *s1.retainAll(s2)*:
  Az *s1*-be *s1* és *s2* metszete kerül.
- *s1.removeAll(s2)*:
  Az *s1*-be *s1* és *s2* különbsége (*s1**s2*) kerül.

Ha azt szeretnénk, hogy az unió, metszet vagy különbség képzés során egyik halmaz se módosuljon (ahogy az matematikailag is várható), akkor a következő módon kell alkalmazni a metódusokat:

```
Set<Type> union = new HashSet<Type>(s1);
union.addAll(s2);

Set<Type> intersection = new HashSet<Type>(s1);
intersection.retainAll(s2);

Set<Type> difference = new HashSet<Type>(s1);
difference.removeAll(s2);
```

Nézzük meg újra a *FindDups* programot. Azt akarjuk tudni, hogy melyik szavak fordulnak elő egyszer a vitatott listában, és mely szavak fordulnak elő többször, de nem akarunk duplán kiírt szavakat látni. Ezt úgy tudjuk elérni, hogy két halmazt hozunk létre: az egyik minden elemet tartalmaz, a másik csak a duplikáltakat. Nézzük a programot:

```
import java.util.*;
public class FindDups2 {
    public static void main(String args[]) {
        Set<String> uniques = new HashSet<String>();
        Set<String> dups = new HashSet<String>();
```

```
        for (String a : args)
            if (!uniques.add(a))
                dups.add(a);

        // Destructive set-difference
        uniques.removeAll(dups);
        System.out.println("Unique words:    " + uniques);
        System.out.println("Duplicate words: " + dups);
    }
}
```

Ha lefuttatjuk az előző parancssori paraméterlistával (*i came i saw i left*), akkor a következő eredményt kapjuk:

```
Unique words:    [left, saw, came]
Duplicate words: [i]
```

A halmazelméleti szimmetrikus differencia a következőképpen állítható elő:

```
Set<Type> symmetricDiff = new HashSet<Type>(s1);
symmetricDiff.addAll(s2);
Set<Type> tmp = new HashSet<Type>(s1);
tmp.retainAll(s2));
symmetricDiff.removeAll(tmp);
```

A *symmetricDiff*-ben először a két halmaz unióját állítjuk elő az *addAll* metódussal, majd a *temp*-ben előállított metszetet kivonjuk belőle a *removeAll* metódussal.

A *Set* interfész tömb metódusai

Ezek a metódusok semmiben nem térnek el a *Collection*-nál leírtaktól.

21.2.4 A *List* interfész

A *List* egy rendezett *Collection*. A *List* duplikáltan is tartalmazhat elemeket. A *Collection*-ból örökölt metódusokon kívül a *List* interfész a következő lehetőségeket tartalmazza:

- Pozíció szerinti elérés: Az elemek a listában betöltött helyük alapján is elérhetők.
- Keresés: A megadott elemet kikeresi a listából, és visszaadja a pozícióját.
- Bejárás: Az *Iterator* alapú szolgáltatások bővültek.
- Részlista: Lehetővé tesz részlista műveleteket.

A *List* interfész:

```
public interface List<E> extends Collection<E> {
    // Positional Access
    E get(int index);
    E set(int index, E element);    // Optional
    boolean add(E element);         // Optional
    void add(int index, E element); // Optional
    E remove(int index);            // Optional
    abstract boolean addAll(int index,
        Collection<? extends E> c);  //Optional

    int indexOf(Object o);
    int lastIndexOf(Object o);
```

```
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    List<E> subList(int from, int to);
}
```

A Java platform két alapvető *List* megvalósítást tartalmaz: *Arraylist* és *LinkedList*.

Collection metódusok

A metódusok öröklődnek a *Collection*-ból, és mindent az elvárásainknak megfelelően használhatunk.

A *remove* metódus mindig az első előforduló elemet törli a listából. Az *add* és *addAll* metódusnál az elem a lista végére kerül. Például a *list1* végére másolja a *list2* elemeit.

```
list1.addAll(list2);
```

A következő kód elkészíti a *list3* nevű listát, amelynek eredménye ugyanaz, mint az előzőnek, de nem rontja el *list1*-et:

```
List<Type> list3 = new ArrayList<Type>(list1);
list3.addAll(list2);
```

Két *List* objektum akkor egyenlő, ha ugyanazok az elemek ugyanolyan sorrendben fordulnak elő benne.

A pozíció szerinti elérés és keresés metódusai

A *set* és *remove* metódusok egy adott elemhez pozícionálnak és felülírják, vagy eltávolítják azt. A keresési metódusok (*indexOf* és *lastIndexOf*) viselkedése tökéletesen megegyezik *String* osztálynál megismertekkel.

Az *addAll* metódus az adott *Collection* elemeit elhelyezi az adott pozíciótól kezdődően. Az elemeket sorrendben helyezi el egy *Iterator* segítségével.

A következő metódus megcseréli a két értéket a listában:

```
static <E> void swap(List<E> a, int i, int j) {
    E tmp = a.get(i);
    a.set(i, a.get(j));
    a.set(j, tmp);
}
```

A következő metódus arra használja a *swap*-et, hogy véletlenszerűen keverje az elemeket:

```
static void shuffle(List<?> list, Random rnd) {
    for (int i = list.size(); i > 1; i--)
        swap(list, i - 1, rnd.nextInt(i));
}
```

Az algoritmus találomra felcseréli az adott *list* elemeit Ez egy egyszerű módszer a keverésre. Másrészt az összes keverés valószínűsége megegyezik, feltéve, hogy a forrás is véletlenszerű, valamint gyors (csal *list.size()-1* csere történt). A következő program bemutat egy példa felhasználást:

```
import java.util.*;
public class Shuffle {
    public static void main(String args[]) {
        List<String> list = new ArrayList<String>();
        for (String a : args)
            list.add(a);
        Collections.shuffle(list, new Random());
        System.out.println(list);
    }
}
```

A programot még rövidebbé és gyorsabbá tudjuk tenni. Az *Array* osztálynak van egy *asList* nevű statikus metódusa, ami egy tömb elemeit *List*-ként is elérhetővé teszi. Ez a metódus nem másolja át a tömb elemeit, csupán referenciákkal dolgozik, és ugyanazok az elemek a listából és a tömbből is elérhetők.

A visszaadott *List* objektum nem tartalmazza az opcionális *add* és *remove* metódusokat, hiszen a tömb mérete nem változhat. Nézzük meg a következő kódot:

```
import java.util.*;
public class Shuffle {
    public static void main(String args[]) {
        List<String> list = Arrays.asList(args);
        Collections.shuffle(list);
        System.out.println(list);
    }
}
```

Iterátorok

A *List* interfész esetén természetesen használható az őstől örökölt módszer a bejárásra, de van egy ennél több lehetőséget nyújtó megoldás is: a *ListIerator*. Ez az interfész megengedi a lista kétirányú bejárását és az ehhez kapcsolódó további műveleteket. A *ListIterator* interfész:

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous ();
    int nextIndex();
    int previousIndex();
    void remove(); // Optional
    void set(E o); // Optional
    void add(E o); // Optional
}
```

Három metódusát (*hasNext*, *next* és *remove*) az *Iterator*-tól örökölte, a *hasPrevious* és *previous* a visszafelé történő bejárást támogatja.

A következő példa visszafelé járja be a listát:

```
for (ListIterator<Type> i = list.listIterator(list.size());
     i.hasPrevious(); ) {
    Type t = i.previous();
    ...
}
```

A *List* interfész kétféle paraméterezéssel tud *ListIterator*-t gyártani. Paraméter nélkül a lista elejére pozícionál, míg az *int* paramétert váró verzió esetén a paraméter alapján pozícionál *ListIterator* tér vissza a *listIterator* metódus.

Az index vagy kurzor pozíció a következő alapján értendő: *n* hosszúságú lista esetén *n+1* érvényes értéke van az indexnek, 0-tól n-ig minden egész. Az index mindig két elem között van a következő ábra szerint:

Nézzük meg az *indexOf* működését:

```
public int indexOf(E o) {
    for (ListIterator<E> i = listIterator(); i.hasNext(); )
        if (o==null ? i.next()==null : o.equals(i.next()))
            return i.previousIndex();
    return -1; // Object not found
}
```

Ha az *indexOf* metódus visszatérési értéke *i.previosIndex()*, akkor megtalálta a keresett objektumot.

Az *Iterator* interfész szolgáltatása a *remove* metódus, mely a gyűjteményből eltávolítja az adott elemet. A *ListIterator* interfész további két metódust nyújt, amely módosítja a listát: *set* és *add*. A *set* metódus felülírja az aktuális elemet. A következő metódus használja *set* metódust az adott elem összes előfordulásának cserélése során:

```
public static <E> void replace(List<E> s, E val, E newVal) {
   for (ListIterator<E> i = s.listIterator(); i.hasNext(); )
      if (val==null ? i.next()==null : val.equals(i.next()))
         i.set(newVal);
}
```

Egy kis trükk van ebben a példában a *val* és az *i.next()* egyenlőség vizsgálatában. Ha a *val* értéke *null*, akkor a *NullPointerException* kivétel nem váltódik ki, hiszen ha *val* értéke *null* lenne, akkor a feltételes operátor másik ága értékelődik ki.

Az *add* metódussal új elemeket adhatunk a *List*-hez az aktuális kurzorpozíció elé. A következő példa bemutatja, hogy hogyan lehet a *val* minden előfordulását kicserélni a *newVals* listával:

```
public static <E> void replace(List<E> s, E val,
                               List<E> newVals) {
  for (ListIterator<E> i = s.listIterator(); i.hasNext(); ){
    if (val==null ? i.next()==null : val.equals(i.next())) {
      i.remove();
      for (E e : newVals)
        i.add(e);
    }
  }
}
```

Részlista művelet

A *subList(int fromIndex, int toIndex)* részlista metódus visszaadja a lista egy szeletét, a paraméterként megadott indexek figyelembevételével: *fromIndex* része, de *toIndex* nem

része a visszaadott szeletnek. (Emlékeztetőül: a *String* osztály *substring* metódusa is így működik.) Ezért gyakori példa a következő:

```
for (int i = fromIndex; i < toIndex; i++) {
    ...
}
```

A visszaadott lista kapcsolatban marad az eredeti listával. Így bármilyen módosítás történik a részlistán, az hatással lesz az eredetire is, sőt ez fordítva is igaz. Ezért nincs szükség további részlista metódusokra. Nézzünk erre egy olyan példát, amikor a lista egy részét akarjuk törölni:

```
list.subList(fromIndex, toIndex).clear();
```

A következő példák csak egy részlistában keresnek:

```
int i = list.subList(fromIndex, toIndex).indexOf(o);
int j = list.subList(fromIndex, toIndex).lastIndexOf(o);
```

Arra érdemes figyelni, hogy itt a visszaadott indexek a talált elemek részlistabeli indexeit jelentik, nem az eredeti *list*-belieket.

A következő metódus is a *subList*-et használja, hogy könnyítse a munkát. A feladata az, hogy a lista egy adott méretű végét levágja, és egyben vissza is adja azt. Másként fogalmazva egy index mentén történő szétvágásról van szó:

```
public static <E> List<E> dealHand(List<E> deck, int n) {
    int deckSize = deck.size();
    List<E> handView = deck.subList(deckSize - n, deckSize);
    List<E> hand = new ArrayList<E>(handView);
    handView.clear();
    return hand;
}
```

A következő program az előző *dealHand* metódust használja a *Collection.suhffle*-val együtt, hogy 52 lapból osszon le. A program 2 parancssori paramétert használ: a kezek számát és az egy kézbe osztandó lapok számát:

```
import java.util.*;

class Deal {
    public static void main(String[] args) {
        int numHands = Integer.parseInt(args[0]);
        int cardsPerHand = Integer.parseInt(args[1]);

        // Make a normal 52-card deck
        String[] suit = new String[]
            {"spades", "hearts", "diamonds", "clubs"};
        String[] rank = new String[]
            {"ace","2","3","4","5","6","7","8",
             "9","10","jack","queen","king"};
        List<String> deck = new ArrayList<String>();
        for (int i = 0; i <suit.length; i++)
            for (int j = 0; j <rank.length; j++)
                deck.add(rank[j] + " of " + suit[i]);

        Collections.shuffle(deck);

        for (int i=0; i<numHands; i++)
            System.out.println(dealHand(deck, cardsPerHand));
    }
}
```

A futó program ezeket az adatokat jeleníti meg:

```
% java Deal 4 5

[8 of hearts, jack of spades, 3 of spades, 4 of spades,
    king of diamonds]
[4 of diamonds, ace of clubs, 6 of clubs, jack of hearts,
    queen of hearts]
[7 of spades, 5 of spades, 2 of diamonds, queen of diamonds,
    9 of clubs]
[8 of spades, 6 of diamonds, ace of spades, 3 of hearts,
    ace of hearts]
```

List algoritmusok

A *Collections* osztály nagyon hatékonyan használható algoritmusokat nyújt listák kezelésére. A következő lista csak felsorolja a fontosabbakat:

- *sort*: Rendezi a listát.
- *shuffle*: Véletlenszerűen felcserél elemeket a listában. (Permutál.)
- *reverse*: Megfordítja az elemek sorrendjét a listában.
- *rotate*: Megforgatja az elemeket.
- *swap*: Felcserél két meghatározott pozícióban levő elemet.
- *replaceAll*: Az összes előforduló elemet kicseréli egy másikra.
- *fill*: Felülírja az összes elemet egy meghatározott elemre.
- *copy*: Átmásolja a forráslistát egy céllistába.
- *binarySearch*: Egy elemeket keres a bináris keresési algoritmust használva.
- *indexOfSubList*: Visszatér az első olyan indexszel, amelynél kezdődő részlista egyenlő a másik listával.
- *lastIndexOfSubList*: Visszatér az utolsó olyan indexszel, amelynél kezdődő részlista egyenlő a másik listával.

21.2.5 *Map* interfész

A *Map* olyan tároló, ami kulcs-érték párokat tartalmaz. A *Map* nem tud tárolni duplikált kulcsokat, egy kulcshoz csak egy érték rendelhető. A *Map* interfész:

```
public interface Map {
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    void putAll(Map<? extends K,? extends V> t);
    void clear();
```

```
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

A Java platform három különböző *Map* implementációt tartalmaz: *HashMap*, *TreeMap*, *LinkedHashMap*.

A *Map* interfész főbb metódusai

A metódusok (*put*, *get*, *containsKey*, *containsValue*, *size*, és *isEmpty*) hasonlóképpen működnek, mint a *Collection*-nál. A következő program kilistázza a szövegben szereplő szavakat és azok gyakoriságát.

```
import java.util.*;
public class Freq {
    public static void main(String args[]) {
        Map<String, Integer>     m =
            new HashMap<String, Integer>();

        // Initialize frequency table from command line
        for (String a : args) {
            Integer freq = m.get(a);
            m.put(a, (freq == null ? 1 : freq + 1));
        }

        System.out.println(m.size() + " distinct words:");
        System.out.println(m);
    }
}
```

A program a következőképpen futtatható:

```
java Freq if it is to be it is up to me to delegate
```

A program kimenete:

```
8 distinct words:
{to=3, delegate=1, be=1, it=2, up=1, if=1, me=1, is=2}
```

Ha betűrendi sorrendbe akarjuk rendezni, akkor a *HashMap*-et *TreeMap*-re kell cserélni, és a következőket írja ki a képernyőre:

```
8 distinct words:
{be=1, delegate=1, if=1, is=2, it=2, me=1, to=3, up=1}
```

Ha a felbukkanás sorrendjére vagyunk kíváncsiak, akkor ez a *LinkedHashMap*-el tegyük, és a kimenet a következő lesz:

```
8 distinct words:
{if=1, it=2, is=2, to=3, be=1, up=1, me=1, delegate=1}
```

Két *Map* objektum könnyedén összehasonlítható. Akkor egyenlőek, ha a kulcs-érték párjaik megegyeznek. (A sorrend nem számít.))

A *Map* konstruktora egy új *Map* objektumot hoz létre egy már meglévő *Map* objektum tartalmával.

```
Map<K, V> copy = new HashMap<K, V>(m);
```

A *Map* interfész tömeges műveletei

A *clear* metódus törli a *Map* teljes tartalmát. A *putAll* metódussal átmásolható egy másik objektum tartalma. A következő példában a paraméterként kapott két *Map* unióját állítjuk elő:

```
static <K, V> Map<K, V> new AttributeMap(
        Map<K, V>defaults, Map<K, V> overrides) {
    Map<K, V> result = new HashMap<K, V>(defaults);
    result.putAll(overrides);
    return result;
}
```

Collection nézetek

A *Map* tartalmát háromféle szempont alapján nézhetjük vissza:

- *keySet*: kulcsok halmaza (*Set*)
- *values*: értékek gyűjteménye (*Collection*)
- *entrySet*: kulcs-érték párok halmaza (*Set*)

A következő példa a kulcsok halmazának kiíratása:

```
for (KeyType key : m.keySet())
    System.out.println(key);
```

Ugyanez iterátorral:

```
for (Iterator<Type> i=m.keySet().iterator(); i.hasNext(); )
    if (i.next().isBogus())
        i.remove();
```

Kulcs-érték párok kiírása:

```
for (Map.Entry<KeyType, ValType> e : m.entrySet())
    System.out.println(e.getKey() + ": " + e.getValue());
```

21.2.6 Objektumok rendezése

Rendezzünk egy tetszőleges tartalmú *l* listát:

```
Collections.sort(l);
```

Ha a lista *String*-eket tartalmaz, akkor azt (Unicode-beli) abc sorrendbe rendezhetjük. Ha pedig Dátum (*Date*) tagokat tartalmaz, akkor időrendi sorrendbe. Hogyan történik ez? A *String* és a *Date* is megvalósítják a *Comparable* interfészt. Ez az interfész tartalmazza a legáltalánosabb rendezési eljárásokat, amely megengedi az osztálynak, hogy automatikusan rendezve legyen valamely szempont szerint. A következő táblázat összefoglalja a legfontosabb osztályokat, amelyek az interfészt megvalósítják.

Osztály	**Alapértelmezett sorrend**
Byte	előjeles szám
Character	előjelnélküli szám
Long	előjeles szám
Integer	előjeles szám
Short	előjeles szám
Double	előjeles szám
Float	előjeles szám
BigInteger	előjeles szám
BigDecimal	előjeles szám
Boolean	*false* < *true*
File	rendszerfüggő abc szerinti a teljes név alapján
String	abc szerinti
Date	időrendi
CollationKey	abc szerinti a helyi jellemzők alapján

Ha a listánk olyan objektumokat tartalmaz, amelyek nem valósítják meg a *Comparable* interfészt, akkor a *Collections.sort(list)* hívás *ClassCastException* kivételt fog dobni.

Saját összehasonlítható osztály létrehozása

A *Comparable* interfész egyetlen metódust tartalmaz:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

A *comprateTo* metódus összehasonlítja az objektumot az átvett objektummal, és visszatérési értéke negatív egész, nulla, vagy pozitív egész, ha az átvevő objektum kisebb, egyenlő vagy nagyobb, mint az átvett objektum. Ha a metódus nem tudja összehasonlítani a két objektumot, akkor *ClassCastException*-t dob.

A következő osztály emberek nevét tartalmazza összehasonlító eszközökkel kiegészítve:

```
import java.util.*;
public final class Name implements Comparable<Name> {
    private final String firstName, lastName;
```

```
    public Name(String firstName, String lastName) {
        if (firstName == null || lastName == null)
            throw new NullPointerException();
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String firstName() { return firstName; }
    public String lastName()  { return lastName;  }
    public boolean equals(Object o) {
        if (!(o instanceof Name))
            return false;
        Name n = (Name) o;
        return n.firstName.equals(firstName) &&
            n.lastName.equals(lastName);
    }

    public int hashCode() {
        return 31*firstName.hashCode() + lastName.hashCode();
    }

    public String toString() {
        return firstName + " " + lastName;
    }

    public int compareTo(Name n) {
        int lastCmp = lastName.compareTo(n.lastName);
        return (lastCmp != 0 ? lastCmp :
                firstName.compareTo(n.firstName));
    }
}
```

Ebben a rövid példában az osztály némileg korlátozott: nem támogatja a középső nevet, kötelező megadni a vezeték- és keresztnevet, és nem veszi figyelembe a nyelvi sajátosságokat. Azonban így is illusztrál néhány fontos pontot:

- A konstruktor ellenőrzi, hogy az paraméterei *null* értékűek-e. Ez minden létrejövő *Name* objektum számára biztosítja, hogy jól formázott legyen, így semelyik más metódus nem dob *NullPointerException*-t.
- A *hashCode* metódus újradefiniált. (Azonos objektumoknak azonos hash kódjuknak kell lennie)
- Az *equals* metódus *false* értékkel tér vissza, ha a paraméterként kapott másik objektum *null*, vagy helytelen típus. Ilyen esetben a *compareTo* metódus futásidejű kivételt dob.
- A *toString* metódus újradefiniálásával a *Name* olvasható formában jeleníthető meg. Ez mindig jó ötlet, különösen olyan objektumok esetén, amiket gyűjteménybe helyezünk. A különböző típusú gyűjtemények *toString* metódusai jól olvasható formában jelenítik meg a gyűjtemények tartalmát.
- A *comparateTo* metódus összehasonlítja az objektumok legfontosabb részét (*lastName*). (Mint ahogy itt is, máskor is gyakran tudjuk használni a részek típusa szerinti alapértelmezett rendezést.) A *lastName* adattag *String* típusú, így az alapértelmezett rendezés pontosan megfelelő lesz. Ha az összehasonlítási eredmények nullától különbözőek, amely egyenlőséget jelent, kész vagyunk: csak vissza kell adni az eredményt. Ha a legfontosabb részek egyenlők, összehasonlítjuk a következőket (itt *firstName*). Ha ez alapján sem dönthető el a sorrend, továbblépünk.

A következő példa félépíti a nevek listáját:

```
import java.util.*;
public static void main(String[] args) {
        Name nameArray[] = {
            new Name("John", "Lennon"),
            new Name("Karl", "Marx"),
            new Name("Groucho", "Marx"),
            new Name("Oscar", "Grouch")
        };
        List<Name> names = Arrays.asList(nameArray);
        Collections.sort(names);
        System.out.println(names);
    }
}
```

A program futása után a következőt írja ki:

```
[Oscar Grouch, John Lennon, Groucho Marx, Karl Marx]
```

A *Comparator*-ok

Hogyan tudjuk az objektumokat az alapértelmezett rendezéstől eltérő más sorrendbe rendezni? Ilyen esetben jön jól a *Comparator* interfész, amely egyetlen egyszerű metódust tartalmaz:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

A *compare* metódus összehasonlítja a két paramétert, visszatérési értéke negatív egész, nulla, vagy pozitív egész, ha az első paraméter kisebb, egyelő vagy nagyobb, mint a második. Ha valamelyik paraméter helytelen típusú a *Comparator* számára, a *compare* metódus *ClassCastException*-t dob.

Tegyük fel, hogy van egy *Employee* nevű osztályunk:

```
public class Employee implements Comparable<Employee> {
    public Name name()     { ... }
    public int number()    { ... }
    public Date hireDate() { ... }
        ...
}
```

Ha az *Employee* objektumok alapértelmezett rendezése a *Name* tag szerinti, akkor nem egyszerű a legmagasabb rangú dolgozókat kikeresni a listából. A következő program elkészíti a listát:

```
import java.util.*;
class EmpSort {
    static final Comparator<Employee> SENIORITY_ORDER =
                                 new Comparator<Employee>() {
        public int compare(Employee e1, Employee e2) {
            return e2.hireDate().compareTo(e1.hireDate());
        }
    };
    // Employee Database
    static final Collection<Employee> employees = ... ;
```

```
    public static void main(String[] args) {
        List<Employee>e = new ArrayList<Employee>(employees);
        Collections.sort(e, SENIORITY_ORDER);
        System.out.println(e);
    }
}
```

21.3. Implementációk

Az implementációk az előző fejezetben leírt interfészeket valósítják meg (implementálják). Az alábbiakban az implementációk fajtáit láthatjuk.

- **Általános célú** implementációk: a leggyakrabban használt implementációk, mindennapos használatra tervezték
- **Speciális célú** implementációk: arra tervezték, hogy speciális helyzetekben esetében használjuk. Nem szabványos teljesítmény karakterisztikák megjelenítésére, használati és viselkedési korlátozásokra.
- **Konkurens** implementációk: erős konkurencia támogatására tervezték, többnyire az egyszálú teljesítmény rovására. Ezek az implementációk a *java.util.concurrent* csomag részét képzik.
- **Csomagoló** implementációk: a többi implementációval összekombinálva használhatók (gyakran az általános célúval), hogy extra funkcionalitást biztosítsanak, vagy bizonyos funkciókat korlátozzanak
- **Kényelmi** implementációk: mini implementációk, tipikusan a gyártó metódusokon keresztül érhetők el. Kényelmes és hatékony alternatívát biztosítanak az általános célú implementációk számára speciális esetekben.
- **Absztrakt** implementációk: vázlatos implementációk, amik megkönnyítik az egyéni implementációk létrehozását.

Az általános-célú implementációkat az alábbi táblázat foglalja össze:

	Hash tábla	**Átméretezhető tömb**	**Fa**	**Láncolt lista**	**Hasító tábla + Láncolt lista**
Set	*HashSet*		*TreeSet*		*LinkedHashSet*
List		*ArrayList*		*LinkedList*	
Map	*HashMap*		*TreeMap*		*LinkedHashMap*

A táblázatban látható hogy a *Set*, a *List* és a *Map* interfészek többfajta általános célú implementációja használható. A *SortedSet* és a *SortedMad* interfészeknek nincs saját sora a táblázatban, ezeknek az interfészeknek egy implementációja van (*TreeSet* és *TreeMap*).

Az általános célú implementációk mind biztosítják az összes opcionális műveletet, amit az interfészek tartalmaznak. Mindegyik megengedi a *null* elemet mind a kulcsok, mind az értékek esetén. Egyik sem szinkronizált. Mindegyik szerializálható (*Serializable*), és biztosít egy publikus *clone* metódust.

A korábbival ellentétes elképzelést jelent az a tény, hogy ezek az implementációk nem szinkronizáltak. A régebbi *Vector* és *Hashtable* gyűjtemények szinkronizáltak. A jelenle-

gi megközelítés azért alakult ki, mert a gyűjteményeket sűrűn használják olyan helyeken, ahol a szinkronizálhatóság nem jelent előnyt. Ilyenek például az egyszálú használat, csak olvasási használat, és ha egy olyan nagyobb adat objektum részét képzik, amely megcsinálja a saját szinkronizációját. Általános API tervezési tapasztalat az, hogy ne kötelezzük a felhasználót olyan szolgáltatás használatára, amit nem sokszor használ. Ráadásul a szükségtelen szinkronizáció adott körülmények között akár holtpontot is eredményezhet.

Általános szabály, hogy programíráskor az interfészeken, és nem az implementációkon kell gondolkodni. Ez az oka annak, hogy ebben a fejezetben nincsenek példa programok. Legtöbb esetben az implementáció megválasztása csak a teljesítményt befolyásolja. Ahogy azt a korábbiakban említettük, egy előnyben részesített programozói stílus az, ha válasszunk egy implementációt a *Collection* létrehozásakor, és rögtön hozzá rendeljük az új *Collection*-t egy olyan változóhoz, ami adott interfész típusú (vagy egy olyan eljárásnak adjuk át a *Collection*-t, ami egy adott interfész típusú paramétert vár). Így a program nem függ majd egy adott implementáció esetén az ahhoz hozzáadott új metódusoktól, ezáltal a programozó bármikor szabadon változtathatja az implementációt, mikor jobb teljesítményt szeretne elérni, vagy a működési részleteket módosítani.

21.3.1 Általános célú *Set* implementációk

Három általános célú *Set* implementáció létezik: *HashSet*, *TreeSet*, és *LinkedHashSet*. Általában egyszerű eldönteni azt, hogy melyiket kell használni. A *HashSet* sokkal gyorsabb, mint a *TreeSet* (konstans végrehajtási idő szemben a logaritmikus idővel a legtöbb művelet esetén), de nem biztosít rendezhetőséget. Ha a *SortedSet* interfész műveleteire, vagy egy érték szerint rendezett iterációra van szükség, akkor a *TreeSet* használata javasolt, egyéb esetben a *HashSet*.

A *LinkedHashSet* bizonyos értelemben egy átmenetet képez a *HashSet* és a *TreeSet* között. Megvalósítását tekintve ez egy olyan hash-tábla, aminek elemei egy láncolt lista segítségével vannak összekötve, így rendezett-beszúrásos iterációt biztosít (a legritkábban használt elemet szúrja be a legsűrűbben használtba), és majdnem olyan gyorsan fut, mint a *HashSet*. A *LinkedHashSet* implementációja megkíméli a használóját az olyan nem specifikált, általában kaotikus rendezéstől, mint a *HashSet* esetében, valamint a *TreeSet* esetében felmerülő többlet költségektől is.

Figyelemre méltó, hogy *HashSet* használata esetén az iterációk száma lineárisan arányos a benne tárolt bejegyzések számával, valamint a beszúrások számával (a kapacitással). Emiatt egy túl magas kezdeti kapacitás választása esetén feleslegesen pazaroljuk az időt és a memóriát. Másrészt a túl alacsony kezdeti kapacitás választás esetén szintén időt vesztünk azzal, hogy az adatstruktúrát mindig át kell másolnunk, mikor a kapacitást növeljük. Ha nem specifikálunk kezdeti kapacitást, akkor az alapbeállítás lép érvénybe, ami 16. A kapacitást mindig felfele kerekítik a legközelebbi kettő hatványra. A kezdeti kapacitást a konstruktor *int* paramétereként adhatjuk meg. A következő kódsor egy olyan *HashSet* példány létrehozását mutatja be, ahol a kezdeti kapacitás 64.

```
Set<String> s= new HashSet<String>(64);
```

A *LinkedHashSet*-nek hasonló beállítási paraméterei vannak, mint a *HashSet*-nek, de az iteráció idejét nem befolyásolja a kapacitás. A *TreeSet* nem rendelkezik beállítási paraméterekkel.

21.3.2 Általános célú *List* implementációk

Két általános célú *List* implementáció létezik: *ArrayList* és *LinkedList*. Leggyakrabban valószínűleg az *ArrayList*–et fogjuk használni. Konstans elérési időt biztosít a lista elemeinek pozícionált elérésekor, egyszerű és gyors. Nem szükséges csomópont objektumot foglalni minden lista elemhez, és képes kihasználni a *System.arraycopy* nyújtotta előnyöket, amikor egyszerre több listaelemet kell mozgatunk.

Abban az esetben, ha gyakran kell elemeket hozzáadni egy lista elejéhez, vagy iterálni a listán, hogy valamelyik belső elemét kitöröljük, akkor érdemes *LinkedList*-t használni. Ezek a műveletek konstans idő alatt végrehajtódnak egy *LinkedList*-ben, míg egy *ArrayList*-ben lineáris idejűek, de ezért cserébe nagy árat kell fizetni a teljesítmény terén. A pozícionált elérés lineáris idejű *LinkedList* esetén, míg ugyanez konstans *ArryList*-nél, továbbá ez a konstans-faktor a *LinkedList* esetében sokkal rosszabb.

Az *ArrayList*-nek egy állítható paramétere a kezdeti kapacitása. Ez azt határozza meg, hogy hány eleme legyen az *ArryaList*-nek, mielőtt növelni kéne az elemek számát. A *LinkedList*-nek nincs állítható paramétere.

21.3.3 Általános célú *Map* implementációk

A három általános célú *Map* Implementáció a *HashMap*, a *TreeMap* és a *LinkedHashMap*. Ha szükség van összegyűjtött információra, a *TreeMap*-et használjuk, ha a legjobb sebesség kell, és nem szükségesek az iterációk, használhatjuk a *HashMap*-et, ha az utóbbihoz hasonló sebesség kell és iteráció is, akkor használjunk *LinkedHashMap*-et. Hasonlóképp, a *Set* implementáció blokkban minden ugyanúgy működik, mint a *Map* implementációban.

21.4. Algoritmusok

Ebben a fejezetben a Java platform által kínált újra felhasználható, sokalakú algoritmusok egy részével foglalkozunk. Mindegyikük a *Collections* osztály statikus metódusa. Az algoritmusok széles skálája a *List*-eknél működik. De egy pár tetszőleges kollekciók esetében is működik.

21.4.1 Rendezés

A *sort* algoritmus újrarendezi a listát. A művelet két formája adott. Az egyszerűbb forma veszi a listát, és rendezi azt az elemek természetes (alapértelmezett) sorrendjében.

Itt egy egyszerű program, ami kiírja betűrendben a parancssori paramétereket:

```
import java.util.*;
public class Sort {
    public static void main(String args[]) {
        List<String> l = Arrays.asList(args);
        Collections.sort(l);
        System.out.println(l);
    }
}
```

Futtassuk a programot:

```
java Sort i walk the line
```

A következő kimenetet kapjuk:

```
[i, line, the, walk]
```

A program megmutatta, hogy ezt az algoritmust nagyon könnyű használni.

Tegyük fel, hogy vannak anagrammákat tartalmazó szólistáink. Szeretnénk kiíratni azokat a listák hossza szerinti csökkenő sorrendben, de csak az adott méretnél (pl. 8) nem kevesebb elemű listákat. A következő példa megmutatja, hogy hogyan érhetjük el ezt a *sort* metódus második formájának segítségével.

Maguk a szólisták egy *m* objektumban vannak tárolva. Erről leválogatjuk a megfelelő hosszúságúakat. Ezután a kód rendezi a listát, összehasonlítást végez, és végrehajtja a fordított méret szerinti rendezést. Végül a kód ismét végigfut a listán és kiírja az elemeket (a szavak csoportját):

```
List<List<String>> winners = new ArrayList<List<String>>();
for (List<String> l : m.values())
    if (l.size() >= minGroupSize)
        winners.add(l);

Collections.sort(winners, new Comparator<List<String>>() {
    public int compare(List<String> o1, List<String> o2) {
        return o2.size() - o1.size();
    }
});

// Szócsoportok kiírása
for (List<String> l : winners)
    System.out.println(l.size() + ": " + l);
```

Futtassuk a programot, a minimum anagramma csoport méretével (8) együtt a következő kimenetet kapjuk:

```
12: [apers, apres, asper, pares, parse, pears, prase, presa,
rapes, reaps, spare, spear]
11: [alerts, alters, artels, estral, laster, ratels, salter,
slater, staler, stelar, talers]
10: [least, setal, slate, stale, steal, stela, taels, tales,
teals, tesla]
9: [estrin, inerts, insert, inters, niters, nitres, sinter,
triens, trines]
9: [capers, crapes, escarp, pacers, parsec, recaps, scrape,
secpar, spacer]
9: [palest, palets, pastel, petals, plates, pleats, septal,
staple, tepals]
9: [anestri, antsier, nastier, ratines, retains, retinas,
retsina, stainer, stearin]
8: [lapse, leaps, pales, peals, pleas, salep, sepal, spale]
8: [aspers, parses, passer, prases, repass, spares, sparse,
spears]
8: [enters, nester, renest, rentes, resent, tenser, ternes,
treens]
8: [arles, earls, lares, laser, lears, rales, reals, seral]
8: [earings, erasing, gainers, reagins, regains, reginas,
searing, seringa]
8: [peris, piers, pries, prise, ripes, speir, spier, spire]
8: [ates, east, eats, etas, sate, seat, seta, teas]
8: [carets, cartes, caster, caters, crates, reacts, recast,
traces]
```

21.4.2 Keverés

A keverés algoritmusa bizonyos értelemben az ellenkezőjét teszi, mint amit a rendezés tesz. Lerombol bármilyen rendezést, ami esetleg található a listában. Ez az jelenti, hogy ez az algoritmus újrarendezi a listát úgy, hogy a véletlenszerűséget használja az összes azonos valószínűséggel bekövetkező permutációhoz, feltételezve a ténylegesesen rendszertelen forrást. Például tudjuk használni a keverést egy pakli kártya megkeveréséhez a kártya objektumainak listájában. Ezenkívül hasznos tesztesetek, helyzetek generálásához is.

21.4.3 Keresés

A bináris keresés algoritmus (*binarySearch* metódus) elemeket keres a rendezett listában. Ennek az algoritmusnak két formája van. Az első vesz egy listát és egy elemet, amit keres (egy kereső kulcsot). Ez a forma feltételezi, hogy a lista elemei rendezve vannak a természetes rendezés szerint a tárolóban. A második forma nem az alapértelmezett rendezést alkalmazza, hanem még egy *Comparator* paramétert is vár.

A visszatérési érték azonos mindkét formában. Ha a lista tartalmazza keresett elemet, visszatér az indexével. Ha nem, a visszatérési érték negatív, és egyben utal arra is, hogy hol kellene lennie az elemnek, ha szerepelne a listában.

A következő példa keresi egy adott elem meglétét, és beilleszti a helyére, ha még nem szerepel:

```
int pos = Collections.binarySearch(list, key);
    if (pos < 0)
       l.add(-pos-1);
```

21.5. Ellenőrző kérdések

- Mik a gyűjtemények?
- Milyen részei vannak a gyűjtemény keretrendszernek? Mire szolgálnak?
- Írjon példát olyan gyűjtemény interfészre, amely engedélyezi a többszörös tárolást!
- Írjon példát olyan gyűjtemény interfészre, amely nem engedélyezi a többszörös tárolást!
- Mi a *Set* interfész specialitása más gyűjteményekhez képest?
- Mi a *List* interfész specialitása más gyűjteményekhez képest?
- Mi a *Map* interfész specialitása más gyűjteményekhez képest?
- Mi az iterátor? Mire használhatjuk? Hogyan jön létre?

Igaz vagy hamis? Indokolja!

- A *Set* elemei indexelhetők.
- Egy *Iterator* objektum segítségével a kollekció elemei többször is bejárhatók.
- A *Comparator* segítségével egy *TreeSet* rendezettsége kívülről is megadható.
- A *Set* implementációi minden esetben rendezettek.

- A *TreeSet* osztály implementálja a *Set* interfészt.
- Minden kollekció implementálja az *Iterator* interfészt.
- A kollekcióban lehet primitív adatokat is tárolni.

Melyik interfészt érdemes alkalmazni?

Olyan tároló objektumra van szükségünk, amelyikbe egyedi elemeket akarunk tárolni. A sorrend nem lényeges, de a többszörös tárolás semmiképpen sem megengedett.

- *Set*
- *List*
- *Map*

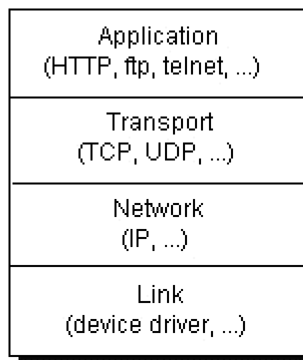
Mit tesz egy *Set* objektum annak érdekében, hogy ne legyenek duplikált elemei?

- Az *add* metódus kivételt dob, ha duplikált elemet akarunk beszúrni.
- Az *add* metódus *false* értéket ad vissza, ha duplikált elemet akarunk beszúrni.
- A duplikált értékeket a fordító kiszűri.

22.Hálózatkezelés

22.1. Hálózati alapok

Az Internetre kötött számítógépek vagy a TCP (*Transmission Controll Protocol*) vagy az UDP (*User Datagram Protocol*) protokollt használják, amint azt az alábbi ábrán is látni lehet:



Ha egy Java programot írunk, ami használja a hálózatot, a felhasználói réteget programozzuk. Általában nincs szükség a TCP vagy az UDP protokollok közvetlen használatára, ehelyett használhatjuk az osztályokat a *java.net* csomagból. Ezek az osztályok rendszerfüggetlen hálózati kommunikációt biztosítanak. Ha szeretnénk tudni, hogy a programunk pontosan melyik osztályokat kell, hogy használja, meg kell érteni az alapvető különbségeket a TCP és az UDP protokollok között.

22.1.1 TCP

Ha két program megbízhatóan akar egymással kommunikálni, létrehoznak egy kapcsolatot, és ezen keresztül küldik az adatokat, hasonlóan a telefon működéséhez. A TCP – akár egy telefontársaság – garantálja, hogy az adat, amit küldünk, helyes sorrendben (!) megérkezik a vevőhöz. Ha ez nem sikerül, hibajelzést küld vissza.

A TCP egy megbízható pont-pont csatornát ad azoknak az alkalmazásoknak, amelyek megbízható kapcsolatot kívánnak meg. Ilyen alkalmazások például: *Hyper Text Transfer Protokoll* (HTTP), *File Transfer Protokoll* (FTP) és a *Telnet*. A hálózaton átküldött és fogadott adatok sorrendje elengedhetetlen az alkalmazások helyes működéséhez. Mikor a HTTP-t egy URL-ből való olvasáshoz használjuk, az adatokat olyan sorrendben kell kiolvasnunk, ahogy azt küldték, különben könnyen lehetne az eredmény egy összekuszált HTML fájl, hibás ZIP fájl, vagy valami más érvénytelen információ.

Definíció: a **TCP** (*Transmission Controll Protocol*) egy kapcsolat alapú protokoll, ami az adatok megbízható folyamát adja két számítógép között.

22.1.2 UDP

Az UDP nem ad megbízható kapcsolatot egy hálózaton levő két számítógép között, mivel ez nem kapcsolat alapú, mint a TCP, inkább az adatok független csomagjait küldi az alkalmazások között. Ezeket a csomagokat adatcsomagoknak is hívjuk. A adatcsomagok küldése leginkább a levelek postán át való küldéséhez hasonlít: a szállítás nem garantált, a sorrend nem fontos, és mindegyik üzenet különbözik a másiktól.

Definíció: az **UDP** (*User Datagram Protocol*) egy olyan protokoll, ami az adatok olyan független csomagjait továbbítja egyik számítógépről a másikra, amiket adatcsomagoknak hívunk, és nincs garancia a megérkezésükre. Az UDP nem kapcsolat alapú, mit a TCP.

Számos alkalmazásnál a megbízhatóság garantálása kritikus, hogy az információ eljusson a hálózat egyik végéről a másikra. Mindemellett a kommunikáció egyéb formái nem kívánnak meg olyan szigorú normákat.

Nézzünk például egy óraszervert, ami a pontos időt küldi el a klienseknek, ha azok igénylik. Ha a kliens hiányol egy csomagot, nem szükséges újraküldeni azt, mert az idő pontatlan lesz, ha második próbálkozásra kapja meg. Mikor a kliens két kérést irányít a szerver felé, és a csomagokat rossz sorrendben kapja, a kliens észreveszi ezt, és újabb kérést küld. A megbízható TCP protokoll ebben az esetben nem szükséges, mivel az a teljesítmény rovására mehet, és esetleg csak akadályozza a szolgáltatást.

A *ping* parancs egy másik kitűnő példa olyan szolgáltatásra, ami nem igényel megbízható csatornát. A parancs teszteli egy hálózaton levő két számítógép közötti kapcsolatot. A parancsnak tudnia kell, hogy a csomag megérkezett-e, hogy megállapítsa, működik-e a kapcsolat. Egy megbízható csatornán ez a szolgáltatás se működik megfelelően.

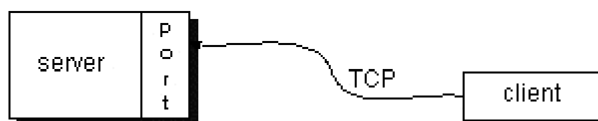
Megjegyzés: Sok tűzfal és router úgy van konfigurálva, hogy ne engedje át az UDP csomagokat. Ha problémánk van egy, a tűzfalon kívül levő szolgáltatáshoz való kapcsolódással, vagy a kliens nem tud hozzád csatlakozni, engedélyezzük az UDP kapcsolatokat.

22.1.3 A portokról általánosságban

Ha nagyvonalakban akarunk beszélni, azt mondjuk, a számítógép egy egyszerű fizikai kapcsolaton keresztül kapcsolódik a hálózatra. Minden adat ezen a kapcsolaton keresztül érkezik, bár az adatok a számítógép különböző programjait használják. Honnan tudja a számítógép, hogy melyik alkalmazásnak melyik adatot kell továbbítani? A portok használata által.

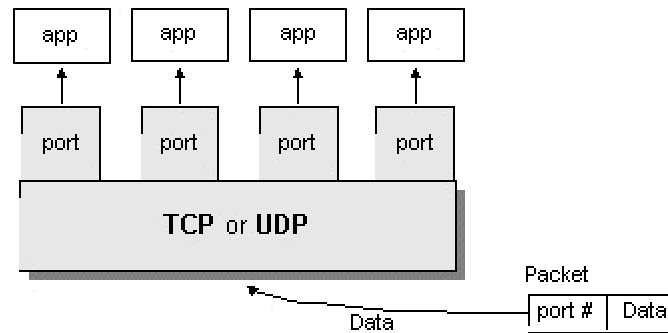
Az adat, amit az interneten keresztül küldenek, el van látva címzési információval, ami azonosítja a célszámítógépet és a portját. A számítógép a 32 bites IP címmel van azonosítva, melyet arra használunk, hogy az adat a megfelelő számítógépre érkezzen meg. A portot egy 16 bites számmal azonosítjuk, amit a TCP vagy UDP arra használ, hogy az adat a megfelelő programhoz jusson.

A kapcsolat alapú kommunikációnál, mint amilyen a TCP, a szerver program leköt egy foglalatot egy jellemző portnak. Ennek eredményeképpen a szerver megkap minden adatot, ami ezen a porton keresztül érkezik. A kliens ezután kommunikálhat a szerverrel a megadott porton keresztül, amit az alábbi ábra illusztrál:



Definíció: a TCP és az UDP protokollok portokat használnak, hogy a bejövő adatokat a számítógép megfelelő programjai felé irányítsák.

Az adatcsomag alapú kommunikációnál, mit amilyen az UDP, az adatcsomag csomagok tartalmazzák a célállomás portszámát, és az UDP irányítja a megfelelő helyre a csomagot.



A portok a 0 - 65535 intervallumba kell, hogy essenek, mivel 16 bites számként vannak ábrázolva. A 0 és 1023 közötti portok fent vannak tartva olyan ismert szolgáltatásoknak, mit például a HTTP vagy az FTP vagy más rendszerszolgáltatás. Ezeket a portokat jól ismert portoknak hívjuk. A saját programjainknak nem szabad lekötni őket.

22.1.4 Hálózati osztályok a JDK-ban

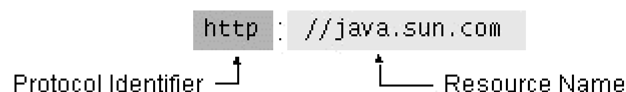
A java programok használhatják a *java.net* osztályain keresztül a TCP vagy UDP protokollokat, hogy kommunikáljanak az Interneten. Az *URL*, *URLConnection*, *Socket* és *ServerSocket* osztályok mindegyike a TCP-n keresztül kommunikál, míg a *DatagramPacket*, *DatagramSocket* és *MulticastSocket* osztályok az UDP protokollt használják.

22.2. URL-ek kezelése

Ha már szörfözőtt a weben, kétségtelenül hallott már az URL-ről, és használta is a HTML oldalakhoz való hozzáférésekhez. A legegyszerűbb arra gondolni, hogy az URL egy fájlra mutat a weben, azonban ez nem teljesen helyes megállapítás. Az URL más erőforrásokra is tud mutatni, például adatbázis lekérdezésekre és parancs kimenetekre.

Definíció: az **URL** (*Uniform Resource Locator*) az interneten található erőforrásokra tud hivatkozni.

A következő ábra egy olyan URL-t mutat, ami a Java weboldalára mutat, amit a Sun Microsystems üzemeltet:



Ahogy látszik is, az URL két fő komponensből tevődik össze:

- Protokollazonosító (*Protocol identifier*)
- Erőforrás neve (*Resource name*)

Jegyezzük meg, hogy a protokoll azonosító és az erőforrás neve kettősponttal és két perjellel van elválasztva. A protokoll azonosító azt a protokollt jelzi, ami eléri a kívánt erőforrást. A fenti példa a *Hypertext Transfer Protocol*-t (HTTP) használja, amit weboldalak eléréséhez használunk. A HTTP csak egy a sok különböző protokoll közül. Használjuk még a *File Transfer Protocol*-t (FTP), a *Gopher*, *File*, *News* protokollokat.

Az erőforrás neve tartalmazza az erőforrás pontos címét. A formája függ a használt protokolltól, de a legtöbb protokoll esetében az erőforrás nevében benne van az alábbi komponensek közül legalább egy.

Hosztnév	A gép neve, amelyen az erőforrás van
Fájlnév	Az elérési út neve a hoszton
Portszám	Az a portszám, amelyikre csatlakozni akarunk
Hivatkozás (horgony)	Ha a fájlon belül akarunk elérni valamit

A legtöbb protokollnál a hosztnév és a fájlnév kötelező, míg a portszám és a hivatkozás csak opcionális. Például az erőforrás név egy HTTP típusú URL-ben megad egy szervert a hálózaton (hosztnév) és a dokumentum elérési útját a gépen (fájlnév), esetleg megad egy portszámot és egy hivatkozást (horgonyt). A Java weboldal URL-ben a *java.sun.com* a hosztnév.

22.2.1 *URL* objektum létrehozása

URL objektumot legegyszerűbben egy olyan sztringből tudunk előállítani, ami ember számára olvasható formában tartalmazza az URL címet. Ez általában olyan alak, hogy más is felhasználhatja az URL-t. Például a Gamelan oldal URL-e, ami Java forrásokat használ, a következőképpen néz ki:

```
http://www.gamelan.com/
```

A Java programban a következőt kell használni, hogy URL objektumot hozzunk létre:

```
URL gamelan = new URL("http://www.gamelan.com/");
```

A fenti URL objektum egy abszolút URL. Az abszolút URL minden információt tartalmaz, ami szükséges lehet, hogy elérjük a forrást. Ezen kívűl létre lehet hozni objektumokat relatív URL címmel is.

Relatív URL létrehozása

A relatív URL-eket gyakran a HTML fájlokon belül használjuk. Például tegyük fel, hogy írtunk egy HTML fájlt, aminek a neve *JoesHomePage.html*. Az oldalon belül vannak linkek más oldalakra, pl. *PicturesOfme.html* és *MyKids.html*, amik ugyanazon a gépen és ugyanabban a könyvtárban vannak, mint a *JoesHomePage.html* fájl. A linkek meghatározhatóak fájlnévként, amit a következő programrészlet mutat be:

```
<a href="PicturesOfMe.html">Pictures of Me</a>
<a href="MyKids.html">Pictures of My Kids</a>
```

Ezek relatív URL címek, ahol az URL arra a szerverre hivatkozik, ahol a *JoesHomePage* oldal is található.

A Java programokban létrehozhatunk URL objektumokat relatív URL-ből is. Például tegyük fel, hogy ismerünk két URL-t a Gamelan oldalról.

```
http://www.gamelan.com/pages/Gamelan.game.html
http://www.gamelan.com/pages/Gamelan.net.html
```

Hozzunk létre *URL* objektumot ezekből az oldalakból, hivatkozva a közös könyvtárukra (*http://www.gamelan.com/pages/*), mit itt:

```
URL gamelan = new URL("http://www.gamelan.com/pages/");
URL gamelanGames = new URL(gamelan, "Gamelan.game.html");
URL gamelanNetwork = new URL(gamelan, "Gamelan.net.html");
```

Ez a kódrészlet az *URL* azon konstruktorát használja, amivel létrehozhatunk egy *URL* objektumot egy másik alapján. A konstruktor általános alakja a következő:

```
URL(URL baseURL, String relativeURL)
```

Az első paraméter egy *URL* objektum, ami az új *URL* bázisát jelzi. A második paraméter egy sztring, ami specifikálja a bázisra hivatkozó forrás nevét. Ha a *baseURL* nincs megadva, a konstruktor abszolút URL-ként kezeli a *relativeURL* sztringet. Viszont, ha a *relativeURL* sztring egy abszolút URL, a konstruktor figyelmen kívül hagyja a *baseURL*-t.

Ez a konstruktor akkor is hasznos lehet, ha olyan URL objektumot akarunk létrehozni, amiben van horgony (*anchor*). Tegyük fel, hogy a *Gamelan.network.html* fájl tartalmaz egy horgonyt, amit *BOTTOM*-nak hívnak, és a fájl aljára mutat. A következő kód a relatív URL konstruktort használja az objektum létrehozására:

```
URL gamelanNetworkBottom = new URL(gamelanNetwork, "#BOTTOM");
```

Más URL konstruktorok

Az *URL* osztály tartalmaz két másik konstruktort is az *URL* objektumok létrehozásához. Ezek a konstruktorok akkor hasznosak, ha olyan URL-ekkel dolgozunk, mit például a HTTP, ami tartalmaz hosztnevet, fájlnevet, portszámot és hivatkozást (horgonyt) is. Ez a két konstruktor akkor is hasznos lehet, ha nincs egy olyan sztringünk, ami tartalmazza a komplett URL-t, de ismerjük néhány összetevőjét.

Például feltételezzük, hogy egy hálózat-tallózó panelt fejlesztünk éppen, ami hasonlóan működik, mint egy fájltallózó, vagyis a felhasználó választhat protokollt, hosztnevet, portot és fájlnevet. Létrehozhatunk URL-t a panel komponenseiből. Az első konstruktor az URL-t a protokollból, a hosztnévből, és a fájlnévből állítja elő. Az alábbi példa a *Gamelan.net.html* fájlból (ami a gamelan oldalon található meg) csinál URL-t.

```
new URL("http", "www.gamelan.com", "/pages/Gamelan.net.html");
```

Ez ekvivalens a következővel:

```
new URL("http://www.gamelan.com/pages/Gamelan.net.html");
```

Az első paraméter a protokoll, a második a hosztnév és az utolsó a fájl elérési útja. A fájlnévnek perjellel kell kezdődnie. Ez jelzi, hogy a fájl gyökere a hosztnévben van megadva. A következő konstruktor egy portszámmal több paramétert tartalmaz az előzőhöz képest:

```
URL gamelan = new URL("http", "www.gamelan.com", 80,
                      "pages/Gamelan.network.html");
```

Ez egy URL objektumot csinál a következő URL-nek:

```
http://www.gamelan.com:80/pages/Gamelan.network.html
```

Ha a fenti konstruktorok egyikével hozunk létre URL objektumot, a *toString* vagy *toExternalForm* paranccsal létrehozhatunk egy sztringet, amely tartalmazza a teljes URL címet.

MalformedURLException

A konstruktorok mindegyike *MalformedURLException* kivételt ad vissza, ha valamelyik paraméter null, vagy ismeretlen a protokoll. Ezt úgy tudjuk lekezelni, ha a konstruktort *try/catch* blokkba rakjuk, mit a következő kódban:

```
try {
    URL myURL = new URL(. . .)
} catch (MalformedURLException e) {
    . . .
        // Kivételkezelő kód
    . . .
}
```

Megjegyzés: Az URL objektumok egyszer írhatóak. Miután létrehoztunk egyet, nem lehet megváltoztatni az attribútumait (protokoll, hosztnév, fájlnév vagy portszám). Érdemes azt is megemlíteni, hogy az *URL* objektumok – hasonlóan a korábbi *File* objektumokhoz – a létrejöttükkor nem ellenőrzik az URL elérhetőségét, érvényességét stb., mivel az URL-nek egyenlőre csak egy logikai reprezentációja.

22.2.2 URL elemzés

Az URL osztály több metódussal szolgál, amellyel kérdéseket tehetünk fel az *URL* objektumoknak. Egy *URL*-től megkaphatja a protokollt, a hoszt nevét, a port számát és a fájlnevet az alábbi metódusokat használva:

- *getProtocol*: Visszaadja az URL protokoll azonosító komponensét.
- *getHost*: Visszaadja az URL hoszt név komponensét.
- *getPort*: Visszaadja az URL port szám komponensét. A *getPort* metódus egy egészet ad vissza, ez a port szám. Ha a port nincs beállítva, akkor -1-et ad.
- *getFile*: Visszaadja az URL fájlnév komponensét.
- *getRef*: Visszaadja az URL hivatkozás komponensét.

Megjegyzés: Ne felejtse el, hogy egyes URL címek nem tartalmazzák ezeket a komponenseket. Az *URL* osztály azért nyújtja ezeket a metódusokat, mert a HTTP URL-ek és a leggyakrabban használt URL-ek tartalmazni szokták. Az *URL* osztály némiképp HTTP-centrikus.

Ezeket a *getXXX* metódusokat arra használhatja, hogy információt kapjon a URL-ről, tekintet nélkül a konstruktorra, amit az URL objektum elkészítésére használt.

Az *URL* osztály a metódusokkal együtt megszabadít az URL-ek állandó újraelemzésétől. Ha adott egy tetszés szerinti URL *String*, csak készítenünk kell egy *URL* objektumot, és meghívni valamelyik metódusát a szükséges információkért. Ez a kis példaprogram készít egy *URL*-t egy sztringből, és ezután az *URL* objektum metódusait használva elemzi az URL-t:

```
import java.net.*;
import java.io.*;

public class ParseURL {
    public static void main(String[] args) throws Exception {
        URL aURL = new URL("http://java.sun.com:80/docs/books/"
                           + "tutorial/index.html#DOWNLOADING");
        System.out.println("protocol = " + aURL.getProtocol());
        System.out.println("host = " + aURL.getHost());
        System.out.println("filename = " + aURL.getFile());
        System.out.println("port = " + aURL.getPort());
        System.out.println("ref = " + aURL.getRef());
    }
}
```

Ez a program kimenete:

```
protocol = http
host = java.sun.com
filename = /docs/books/tutorial/index.html
port = 80
ref = DOWNLOADING
```

22.2.3 Közvetlen olvasás *URL*-ből

Miután sikeresen készítettünk egy *URL*-t, az *URL openStream* metódusát meghívva kap egy adatfolyamot, amiből kiolvashatja az *URL* tartalmát. Az *openStream* metódus visszatérési értéke egy *java.io.InputStream* objektum, így olyan könnyű az *URL*-ből olvasni, mint egy tetszőleges állományból.

A következő kis program az *openStream* használatával kap egy bemenő folyamot a http://www.yahoo.com/ URL-re. Ezután nyit egy *BufferedReader*-t, ahonnan az URL tartalmát olvassa. Minden beolvasott sort átmásol a szabványos kimenetre:

```
import java.net.*;
import java.io.*;

public class URLReader {
    public static void main(String[] args) throws Exception {
    URL yahoo = new URL("http://www.yahoo.com/");
    BufferedReader in = new BufferedReader(
                new InputStreamReader(
                yahoo.openStream()));

    String inputLine;

    while ((inputLine = in.readLine()) != null)
        System.out.println(inputLine);

    in.close();
    }
}
```

Amikor futtatjuk a programot, láthatjuk a parancs ablakában görgetve a HTML elemeket és a szöveges tartalmat a *http://www.yahoo.com/* címen található HTML fájlból. Esetleg a program leállhat kivétellel. Ha a utóbbi esemény bekövetkezik, a programunkban be kell állítanunk a proxyt, hogy a program megtalálhassa a Yahoo szervert.

22.2.4 Csatlakozás egy URL-hez

Miután sikeresen készítettünk egy *URL* objektumot, meghívhatjuk az *URL* objektum *openConnection* metódusát, hogy csatlakoztassuk. Amikor kapcsolódunk egy URL-hez, a hálózaton keresztül inicializálunk egy kommunikációs kapcsolatot a Java programja és az URL között. Például, nyithatunk egy kapcsolatot a Yahoo oldalra az alábbi kóddal:

```
try {
    URL yahoo = new URL("http://www.yahoo.com/");
    URLConnection yahooConnection = yahoo.openConnection();
} catch (MalformedURLException e) { // new URL() failed
    . . .
} catch (IOException e) {           // openConnection() failed
    . . .
}
```

Ha lehetséges, az *openConnection* metódus készít egy új *URLConnection*-t, inicializálja, csatlakozik az URL-hez, és visszatér az *URLConnection* objektummal. Ha valami nem

megfelelő – például, a Yahoo szerver nem üzemel –, akkor az *openConnection* metódus egy *IOException*-t dob.

Miután sikeresen csatlakoztunk az URL-hez, írhatunk a csatlakozáshoz vagy olvashatunk róla az *URLConnection* objektum használatával.

Ha sikeresen használtuk az *openConnection*-t, hogy kommunikációt kezdeményezzen egy URL-lel, akkor van egy *URLConnection* objektumhivatkozásunk. Az *URLConnection* egy HTTP-centrikus osztály; ennek sok metódusa csak akkor hasznos, ha HTTP URL-ekkel dolgozik. Azonban a legtöbb URL protokoll megengedi, hogy olvasson és írjon a kapcsolatra.

Olvasás egy *URLConnection*-ról

Az alábbi program azonos funkciót hajt végre, mint az *URLReader* program. De ahelyett, hogy egy bemenő folyamot kapna közvetlenül az URL-ből, ez a program határozottan nyit egy kapcsolatot egy URL-hez, és egy bemenő folyamot kap a kapcsolatból. Ekkor, mint az *URLReader*, ez a program készít egy *BufferedReader*-t a bemenő folyamon, és onnan olvas. A félkövér szakaszok eltérőek az előző és a mostani példákban.

```
import java.net.*;
import java.io.*;


public class URLConnectionReader {
    public static void main(String[] args) throws Exception {
        URL yahoo = new URL("http://www.yahoo.com/");
        URLConnection yc = yahoo.openConnection();
        BufferedReader in = new BufferedReader(
                                new InputStreamReader(
                                yc.getInputStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
        in.close();
    }
}
```

A program kimenete megegyezik az előző kimenettel. Használhatja mind a két módszer az URL-ből olvasásra. Azonban az *URLConnection*-ból olvasás sokkal használhatóbb lehet, mert az *URLConnection* objektumot használhatja más feladatokra is (például az URL-be írás) egy időben.

Még egyszer, ha a program leáll, vagy hibaüzenetet lát, be kell állítani a proxyt, hogy a program megtalálhassa a Yahoo szervert.

URLConnection-ba írás

Sok HTML lap tartalmaz űrlapokat – szöveg mezőket és más GUI objektumokat –, hogy beküldje a kitöltött adatokat a szerverre. Miután begépelte a szükséges információkat és elküldi a kérést egy gomb lenyomásával, a böngészőnk adatokat ír az URL-re a hálózaton keresztül. A szerver fogadja az adatokat, feldolgozza és visszaküld egy választ, általában egy új HTML oldal formájában.

Sok szkript használja a POST metódust a kliens adatainak továbbítására. A szerver-oldali szkriptek a POST metódust a bemenetük olvasására használják.

Egy Java program szintén kommunikálhat a szerver oldalon futó szkriptekkel. Egyszerűen lehet írni egy *URL*-be, így küldve adatokat a szerverre. Ezt a következő lépésekkel tudjuk megtenni:

- Készítünk egy *URL*-t.
- Nyitunk egy kapcsolatot az *URL*-hez.
- Beállítjuk a kimenő képességet az *URLConnection*-on.
- Kapunk egy kimenő folyamot. Ez a kimenő folyam összekapcsolt a szkript bemenő folyamával a szerveren.
- Írunk a kimenő folyamba.
- Bezárjuk a kimenő folyamot.

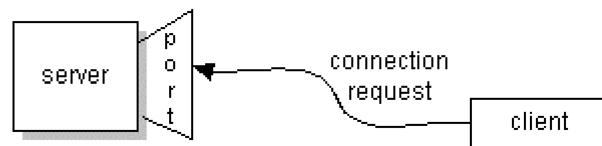
22.3. Socketek kezelése

A korábbi példák a kliens és szerver közötti magas szintű kommunikáció lehetőségeit mutatták be. Gyakran van olyan szituáció, amikor a magas szintű megközelítés nem megfelelő. Ekkor alacsonyabb szintű, úgynevezett socket alapú kommunikációra van szükség.

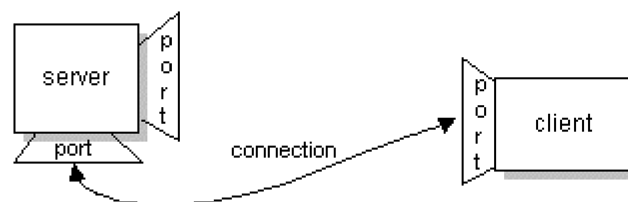
22.3.1 Mi az a socket?

Normális esetben a szerver egy speciális számítógépen fut és van egy socket-je, ami egy speciális port számra van kötve. A szerver várakozik, figyeli a socket-et hogy van-e felkérés egy klienstől a kapcsolódásra.

A kliens oldalon: A kliens ismeri annak a számítógépnek a hosztnevét, amelyiken a szerver fut, és annak a portnak a számát, amelyiken keresztül a szerver kapcsolódva van. A kapcsolatra való felkéréshez a kliensnek érintkeznie kell a szerverrel a szerver gépén és port-ján.



Ha minden jól megy, a szerver elfogadja a kapcsolatot. Az elfogadás során, a szerver egy új socket porthoz kapcsolódik. Kell egy új socket (és természetesen egy különböző port szám is), ezért folytatódik a figyelés az eredeti socket-en a kapcsolatkérésre, amíg a kliensnek kapcsolódnia kell.



A kliens oldalon, ha a kapcsolat elfogadásra került, a socket sikeresen létrejött, és a kliens használni tudja a socket-et a szerverrel való kommunikálásra.

Megjegyzés: a kliens oldali socket nincs összekötve azzal a port számmal, amit a szerverrel való kapcsolattartásra használ. A kliens a helyi kijelölt port számmal azonosítható azon a gépen, amin fut.

A kliens és a szerver kapcsolatot tud létrehozni a socket-jeik által írásra, vagy olvasásra.

Definíció: A *socket* egy végpontja egy kétvégű kommunikációs hálózatnak, amin két program fut. A socket egy port számhoz van kötve, ezért a TCP réteg azonosítani tudja a kérést, amit ahhoz kértek, hogy elküldhessék az adatot.

A *java.net* csomag a tartalmaz egy *Socket* osztályt, ami alkalmas egy kétirányú kapcsolat egyik oldalának vezérlésére egy, a hálózaton lévő másik program felé. Ezen kívül a *java.-net* tartalmazza a *ServerSocket* osztályt, amely figyel és elfogadja a kapcsolatot a kliensektől.

Ha a webhez akarunk kapcsolódni, az *URL* osztály és a kapcsolódó osztályok (*URLConnection*, *URLEncoder*) talán alkalmasabbak, mint a *Socket* osztály. Valójában az *URL* viszonylag magas szintű kapcsolatot teremt a webbel, és használja a socket-eket.

22.3.2 Olvasás és írás a socket-ről

Nézzünk egy egyszerű példát arra, hogy hogyan lehet kiépíteni egy kapcsolatot a szerver programmal a *Socket* osztályt használva, azután megnézzük, hogy hogyan tud a kliens adatot küldeni és fogadni a szervertől a socket-en keresztül.

A példaprogram része az *EchoClient*, ami össze van kötve az echo szerverrel. Az echo szerver elég egyszerűen tud adatot fogadni a klienstől és választ küldeni rá. Az echo szerver egy jól ismert kiszolgáló, amit a kliens a 7-es port-on keresztül tud elérni.

Az *EchoClient* létrehoz egy socket-et azzal, hogy kapcsolódik az echo szerverhez. Ez beolvas a felhasználótól egy sort, elküldi az echo szervernek a socket-en keresztül. A szerver ezen keresztül válaszol a kliensnek. A kliens program olvassa ezt, és kiírja az adatot, amit visszaküldött neki a szerver:

```
import java.io.*;
import java.net.*;

public class EchoClient {
    public static void main(String[] args) throws IOException {
Socket echoSocket = null;
        PrintWriter out = null;
        BufferedReader in = null;

        try {
            echoSocket = new Socket("taranis", 7);
            out = new PrintWriter(echoSocket.getOutputStream(),
                                  true);
            in = new BufferedReader(new InputStreamReader(
                                 echoSocket.getInputStream()));
        } catch (UnknownHostException e) {
            System.err.println(
                "Az alábbi host nem ismert: taranis.");
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for "
                             + "Kapcsolódás ehhez: taranis.");
            System.exit(1);
        }

    BufferedReader stdIn = new BufferedReader(
                            new InputStreamReader(System.in));
    String userInput;
```

```
        while ((userInput = stdIn.readLine()) != null) {
            out.println(userInput);
            System.out.println("echo: " + in.readLine());
        }

        out.close();
        in.close();
        stdIn.close();
        echoSocket.close();
        }
}
```

Az *EchoClient* olvasni és írni is tud a socket-ről azáltal, hogy átküld és átvesz adatot az echo szervertől.

Nézzük a program érdekesebb részeit. A *main* metódus *try* blokkjában szereplő sorok kiépítik a kapcsolatot a socket-en keresztül a kliens és a szerver között, és megnyitják a *PrintWriter*-t és a *BufferedReader*-t a socket-en.

```
echoSocket = new Socket("taranis", 7);
out = new PrintWriter(echoSocket.getOutputStream(), true);
in = new BufferedReader(new InputStreamReader(
                            echoSocket.getInputStream()));
```

Sorrendben az első rész létrehozza az új *Socket* objektumot, és nevet ad neki: *echoSocket*. A *Socket* konstruktor használata igényli annak a gépnek a nevét és a port-jának a számát, amelyikkel a kapcsolatot létre akarjuk hozni. A példa program a *taranis* host nevet használja. Ez a helyi hálózatunk virtuális gépének a neve. Mikor futtatjuk a programunkat, akkor saját szerver gépünk host neve kerüljön ide. A 7-es porton figyel az echo szerver.

A második sor összekapcsolja a *Socket* kimenetét és a *PrintWriter*-t. A harmadik sor hasonló ehhez, összekapcsolja a *Socket* bemenetét és a *BufferedReader*-t. A példa Unicode karaktereket tud írni a socket-en át.

Az *EchoClient*-nek az adatok socket-en való elküldéséhez egyszerűen a *PrintWriter*-hez kell írnia. Ahhoz, hogy megkapja a szerver válaszát, olvasnia kell a *BufferedReader*-ről. A program hátralevő részét ez teszi ki.

A *while* ciklus a program következő érdekes része. A ciklus beolvas egy sort egy bizonyos idő alatt a szabványos bementről, és azonnal elküldi azt a szervernek a *PrintWriter*-be való írással, ami a sockethez kapcsolódik:

```
String userInput;
while ((userInput = stdIn.readLine()) != null) {
    out.println(userInput);
    System.out.println("echo: " + in.readLine());
}
```

A *while ciklus* utolsó sorában beolvas egy sor szöveget a *BufferedReader*-ről, ami a socket-hez kapcsolódik. A *readLine* metódus addig vár, amíg a szerver válaszol az információra az *EchoClient*-nek. Mikor a *readLine* válaszol, az *EchoClient* kiírja a szöveget a szabványos kimenetre.

A *while* ciklus addig tart, amíg a felhasználó be nem gépeli az utolsó karaktert. Vagyis az *EchoClient* olvas a felhasználó bemenetéről, majd elküldi ezt az echo szervernek, kap egy választ a szervertől, és utána kiírja azt, amíg az elér az utolsó karakterig. A *while* ciklus ekkor bezárul, és a program folytatódik, futtatja a kód következő négy sorát:

```
out.close();
in.close();
stdIn.close();
echoSocket.close();
```

A kód e sorai a takarítást végzik. A jó program mindig kitakarít maga után, és ez a program is ilyen. Ezek a részek bezárják az olvasót és az írót, ami a socket-hez kapcsolódik és a szabványos kimenetet és bemenetet is, ami a szerverhez kapcsolódik. A sorrend itt nagyon fontos. Be kell zárnunk minden folyamatot, ami a socket-hez kapcsolódik, mielőtt kikapcsoljuk magát a socket-et.

- Socket megnyitása.
- Kimeneti és bementi folyam megnyitása a socket-en.
- Olvasás és írás a szerver protokollja szerint.
- A folyamatok bezárása.
- Socket bezárása.

23.JDBC adatbázis-kezelés

A *Java Database Connectivity* (JDBC) olyan programozói felületet (API-t) ad a programozó kezébe, amellyel a konkrét adatbázis-kezelőtől függetlenül működő, adatbázis alapú alkalmazások készíthetők.

Kezdő lépések

Első lépésként meg kell győződni, hogy minden helyes van-e beállítva. Ez a következő lépéseket tartalmazza:

- **Telepítsük a meghajtót a számítógépére**
  A meghajtó tartalmazza a telepítésre vonatkozó utasításokat. A speciális DBMS-hez (adatbázis-kezelőhöz) írott JDBC meghajtók telepítése egyszerűen annyiból áll, hogy átmásoljuk a meghajtót a számítógépére; semmilyen különleges beállításra nincs szükség.
  A JDBC-ODBC híd (*JDBC-ODBC Bridge*) telepítése már nem ennyire egyszerű. Akár a Solaris, akár a Windows verzióját használjuk, automatikus megkapjuk a JDBC-ODBC hidat is, amely önmagában nem igényel semmilyen speciális beállítás. Azonban az ODBC igen.
- **Telepítsü a DBMS-t, amennyiben szükséges**
  Amennyiben még nincs a DBMS feltelepítve, kövesse a forgalmazó utasításait a telepítésre vonatkozóan. A legtöbb esetben a DBMS telepítve van és működik az általánosan elfogadott adatbázisok bármelyikével.

23.1. Adatbázis beállítása

Tételezzük fel, hogy a *Coffebreak* nevű adatbázis már létezik. (Egy adatbázis létrehozása nem olyan bonyolult, de különleges jogokat igényel és általában az adatbázis adminisztrátor végzi el). A jegyzet példáiban használt táblák az alapértelmezett adatbázisban lesznek létrehozva. A könnyebb kezelhetőség érdekében szándékosan tartottuk alacsonyan a táblák számosságát és méretét.

Tételezzük fel, hogy a mintaadatbázist egy **Coffee Break** nevű kávéház tulajdonosa használja, ahol szemes kávét árulnak fontban mérve, és főzött kávét mérnek csészébe. Az egyszerűség kedvéért tételezzük fel azt is, hogy a tulajdonosnak csak két táblára van szüksége, az egyik a kávé típusok nyilvántartására, a másik a kávé kiegészítők adatainak tárolására.

Először azt mutatjuk meg, hogyan kell a DBMS-sel létrehozni egy kapcsolatot, majd miután a JDBC küldi el az SQL kódot a DBMS-nek, bemutatunk néhány SQL kódot. Ezután megmutatjuk, milyen egyszerű a JDBC-vel elküldeni ezt az SQL mondatot a DBMS-nek, és feldolgozni a visszakapott eredményt.

Ez a kód a legtöbb ismertebb DBMS-n tesztelve lett, azonban előfordulhat néhány kompatibilitási probléma a régebbi ODBC meghajtók és a JDBC-ODBC Híd használatakor.

Kapcsolat létesítése

Első lépésként a kapcsolatot kell létrehozni azzal a DBMS-sel, amit használni akarunk. Ez két lépést tartalmaz: a meghajtó betöltését és a kapcsolat megteremtését.

Meghajtó betöltése

A használandó meghajtó vagy meghajtók betöltése nagyon egyszerű, és csak egy kódsort tartalmaz. Például a JDBC-ODBC Híd meghajtót a következő kód tölti be:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

A *driver* dokumentációja megadja, hogy melyik osztályt kell használni. Amennyiben az osztály neve *jdbc.DriverXYZ*, a következő kóddal tudjuk betölteni:

```
Class.forName("jdbc.DriverXYZ");
```

Nem szükséges a meghajtót példányosítani és regisztrálni a *DriverManager*-rel, mert a *Class.forName* hívás automatikusan megteszi azt. Amennyiben saját példányt hoznánk létre, egy szükségtelen duplikáció történne, igaz, ez nem okozna hibát.

Amikor a kód betöltődött, készen állunk a kapcsolat létrehozására a DBMS-sel.

Kapcsolat létrehozása

A második lépés a kapcsolat létrejöttéhez, hogy a megfelelő meghajtó hozzá legyen kapcsolva a DBMS-hez. A következő kód bemutatja az általános eljárást:

```
Connection con = DriverManager.getConnection(url,
         "myLogin", "myPassword");
```

Ez a lépés is nagyon egyszerű, a legnehezebb dolog, hogy mit adjunk meg URL-nek. Amennyiben a JDBC-ODBC híd meghajtót használjuk, a JDBC URL így kezdődik: *jdbc:odbc:*. Az URL többi része általában az adatbázis vagy az adatbázis séma neve. Például, ha ODBC-t használunk egy *Fred* nevű ODBC adatforrás eléréséhez, a JDBC URL a következő lesz: *jdbc:odbc:Fred*. A *mylogin* helyére a DBMS-ben használt felhasználói nevet, amíg a *myPassword* helyére a jelszót kell behelyettesíteni. Azaz, ha a DBMS-hez a bejelentkezési név *Fernanda,* és a jelszó *J8*, csak erre a két sorra van szükség, hogy a kapcsolat létrejöjjön:

```
String url = "jdbc:odbc:Fred";
Connection con =
    DriverManager.getConnection(url, "Fernanda", "J8");
```

Amennyiben egy külső cég által fejlesztett JDBC-t használunk, a dokumentáció megmondja, milyen protokollt használjunk, mit kell a *jdbc:* után írnunk a JDBC URL-ben. Például, ha a meghajtó fejlesztője az *acme* nevet regisztrálta, az első és a második része az URL-nek a *jdbc:acme:*. A dokumentáció útmutatót tartalmaz a JDBC URL további részéhez is. A JDBC URL utolsó része tartalmazza az adatbázis azonosítására szolgáló információkat.

Amennyiben valamelyik betöltött meghajtó azt érzékeli, hogy a JDBC URL a *DriverManager.getConnection* metódussal lett betöltve, a meghajtó fogja létrehozni a kapcsolatot a DBMS-sel a JDBC URL-ben megadott módon. A *DriverManager* osztály (hűen a nevéhez) a háttérben lekezeli a kapcsolódás összes részletét. Ha nem írunk saját meghajtót, talán soha nem is fogjuk a *Driver* interfész más metódusát használni, és az egyetlen *DriverManager* metódus, amit igazán szükséges ismernünk, a *DriverManager.getConnection*.

A *DriverManager.getConnection* metódus által visszaadott kapcsolat egy nyitott kapcsolat, amelynek JDBC mondatokat adhatunk meg, amely eljuttatja az SQL mondatot a DBMS-nek. Az előző példában *con* egy nyitott kapcsolat, és a következő példához fogjuk használni.

23.2. Táblák használata

23.2.1 Tábla létrehozása

Először létrehozzuk az egyik táblát a példa adatbázisban. Ez a tábla a *Coffees*, mely tartalmazza az alapvető információkat **a Coffee Break** által eladott kávékról, ideértve a kávé nevét, annak árát, aktuális héten eladott mennyiséget és a napi eladott mennyiséget. A *Coffee* tábla látható itt, melyet később részletesebben leírunk:

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	0	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

A COF_*NAME* mező, amely a kávé nevét tárolja, maximum 32 karakter hosszú *VARCHAR* típusú. Miután minden egyes kávétípust különböző névvel látunk el, a név egyértelműen meghatározza az egyes kávékat, így ez lehet a tábla elsődleges kulcsa. A *SUP_ID* a kávé-kiegészítő egyedi azonosítóját tartalmazza, SQL típusa *INTEGER*. A harmadik mező neve *PRICE*, típusa *FLOAT*, mert tizedes szám tárolására van szükség.

Megjegyzés: A pénz adat általában SQL *DECIMAL* vagy *NUMERIC* típusú mezőben tárolódik, de a DBMS-k közötti különbség miatt, és hogy elkerüljük az inkompatibilitást valamelyik régebbi JDBC-vel, most az általánosabb FLOAT típust használjuk.

A *SALES* mező *INTEGER* típusú adatokat tárol, és megmutatja, hogy mennyi kávé lett eladva az adott héten. Az utolsó mező a *TOTAL*, amely *INTEGER* típusú, és megadja, hogy mennyi kávét adtak el összesen az adott napig.

Az adatbázis SUPPLIERS nevű táblája, amely információt tárol az egyes kávé-kiegészítőkről:

SUP_ID	SUP_NAME	STREET	CITY	STATE	ZIP
101	*Acme, Inc.*	*99 Market Street*	*Groundsvill e*	*CA*	*95199*
49	*Superior Coffee*	*1 Party Place*	*Mendocino*	*CA*	*95460*
150	*The High Ground*	*100 Coffee Lane*	*Meadows*	*CA*	*93966*

A COFFEES és a SUPPLIERS tábla is tartalmazza a SUP_ID mezőt, ezért ez a mező használható a SELECT utasításoknál, hogy a két táblát össze tudjuk kapcsolni. A SUP_ID oszlop az egyedi azonosítója a SUPPLIERS táblának. A COFFEES táblában pedig idegen kulcsként szerepel. (Azért idegen kulcs, mert ez egy másik táblából lett importálva.) Minden egyes SUP_ID csak egyszer szerepel a SUPPLIER táblában, hiszen ez az elsődleges kulcsnál követelmény. A COFEE táblában, ahol idegen kulcsként szerepel, teljesen természetes, hogy többszörösen megjelenik, hiszen egy

kiegészítő eladható több típusú kávéhoz is. A fejezet későbbi részében bemutatunk néhány példát az elsődleges és az idegen kulcs használatára a SELECT lekérdezésekben.

A következő SQL utasítás létrehozza a *COFFEES* nevű táblát. A két külső zárójel tartalmazza az egyes oszlopok nevét és SQL típusát vesszővel elválasztva. A *VARCHAR* típusnál zárójelek között megadható annak maximális hossza. A példakód megadja, hogy a *COF_NAME* oszlop maximum 32 karakter hosszú lehet:

```
CREATE TABLE COFFEES (
  COF_NAME VARCHAR(32),
  SUP_ID INTEGER,
  PRICE FLOAT,
  SALES INTEGER,
  TOTAL INTEGER )
```

Az utasítás végén nem szerepel DBMS záró jel, mert ez az egyes DBMS-eknél különböző. Például az Oracle pontosvesszőt (;) használ a mondatok lezárásra, a Sybase pedig a *go* szót. Ezért a Java kódban nem szükséges megadni, a driver automatikusan a mondat végére teszi a megfelelő jelet.

Másik dolog, ami meg kell említenünk az SQL utasításokról, azok formája. A *CREATE TABLE* mondatban a kulcsszavak nagybetűvel és minden rész külön sorban van írva. Az SQL nem követeli ezt meg, ez a konvenció csak az olvashatóságot teszi könnyebbé. Alapvető az SQL-ben, hogy a kulcsszavakban nem kis-nagybetű érzékeny (case sensitive), így például a *SELECT* szót bármilyen módon le lehetett volna írni. Példának okáért a két verzióval korábbi teljesen megegyezik az SQL illetően:

Az egyes DBMS-k különbözőek lehetnek a névazonosításban. Például néhány DBMS megköveteli, hogy az oszlop és a tábla nevek pontosan megfeleljenek a *CREATE TABLE* utasításban megadottaknak, míg mások nem. A biztonság kedvéért nagybetűt használunk az azonosítóknál, mint például *COFFEES* és *SUPPLIERS*, mert így definiáltuk őket.

Eddig kész vagyunk az SQL utasítás megírásával, amely létrehozza a *COFFEES* nevű táblát. Most tegyük idézőjelek közé (készítsünk *String*–et), és nevezzük el ezt a *String*-et *createTableCoffees*-nek, így ezt a változót fel tudjuk használni a Java kódban.

Amint láttuk, a DBMS nem foglalkozik a sortörésekkel, de a Java programozási nyelvben az a *String* objektum, ami meghaladja az egy sort, nem fog lefordulni. Következésképpen, amikor *String*-et adunk meg, minden egyes sort idézőjelek közé kell tenni, és pluszjelet (+) használni az összefűzéshez:

```
String createTableCoffees = "CREATE TABLE COFFEES " +
    "(COF_NAME VARCHAR(32), SUP_ID INTEGER, PRICE FLOAT, " +
    "SALES INTEGER, TOTAL INTEGER)";
```

Mielőtt futtatunk egy alkalmazást, menjük végig a JDBC alapjain.

23.2.2 JDBC *Statement* létrehozása

A *Statement* objektum küldi el az SQL utasításunkat az adatbázis-kezelőnek. Egyszerűen létrehozunk egy *Statement* objektumot, majd végrehajtjuk az *execute* metódust alkalmazva. A *SELECT* utasítások végrehajtására az *executeQuery* metódus szolgál. Azoknál az utasításoknál, amelyek táblát hoznak létre, vagy módosítanak, az *executeUpdate* metódus használatos.

Egy már élő adatbázis kapcsolatra (*Connection* objektum) épül a *Statement* objektum létrehozására. A következő példa a már meglévő *con Connections* objektumot használja az *stmt Statement* objektum létrehozására:

```
Statement stmt = con.createStatement();
```

Ebben a pillanatban *stmt* létrejött, de még nincs megadva az elküldendő SQL utasítás. Az *stmt execute* metódusában kell ezt megadnunk. Például, a következő kódrészletben mi az *executeUpdate* metódusnak adjuk át a SQL utasítást:

```
stmt.executeUpdate("CREATE TABLE COFFEES " +
    "(COF_NAME VARCHAR(32), SUP_ID INTEGER, PRICE FLOAT, " +
    "SALES INTEGER, TOTAL INTEGER)");
```

Miután létrehoztunk egy *createTableCoffees* nevű *String* típusú változót, megadhatjuk ebben a formában is.

```
stmt.executeUpdate(createTableCoffees);
```

23.2.3 SQL parancs végrehajtása

Az *executeUpdate* metódust használtuk, mert az *createTableCoffees* nevű SQL utasítás egy DDL (*data definiton language*) utasítás. Azok a parancsok, amelyek létrehoznak, módosítanak, törölnek táblát, mind DLL utasítások és az *executeUpdate* metódussal hajthatók végre. Amint az a nevéből kitűnik, az *executeUpdate* metódus rekordok értékeinek módosítására is szolgál. Ez sokkal gyakoribb, mivel rekord módosítása többször is történhet, amíg egy tábla létrehozása csak egyszer.

A leggyakrabban használt SQL utasítást végrehajtó parancs az *executeQuery*. Ez a metódus használandó egy *SELECT* lekérdezés végrehajtásához, ami a leggyakoribb SQL utasítás.

Adatbevitel egy táblába

Megmutattuk, hogyan kell létrehozni a *COFFEES* nevű táblát, megadva a mezők nevét és típusát, de ez csak a tábla szerkezetét állítja be. A tábla még nem tartalmaz egyetlen adatot sem. Soronként visszük be az adatokat a táblába, megadva az egyes mezők értékét.

Megjegyezés: A beillesztendő értékeket ugyanabban a sorrendben kell megadni, mint ahogy azt deklaráltuk a tábla létrehozásakor.

A következő kód egy rekordot illeszt be: *Colombian* a *COF_NAME* oszlopba, *101* a *SUP_ID* oszlopba, *7.99* a *PRICE*, *0* a *SALES* és *0* a *TOTAL* oszlopba. (Amióta a *Coffee Break* működik, a heti eladási mennyiség és az eddigi összmennyiség az összes kávéfajtánál 0-val indul.) Csakúgy mint a *COFFEES* tábla létrehozásánál tettük, egy *Statement* objektumot hozunk létre, majd az *executeUpdate* metódust használjuk.

Mivel az SQL utasítás nem fér el egy sorba, két *String*-et fogunk pluszjellel összefűzni. Érdemes figyelni a szükséges szóközökre a *COFFEES* és a *VALUES* szó között. A szóköznek az aposztrófok között kell lennie vagy a *COFFEES* szó mögött, vagy a *VALUES* szó előtt; a szóköz nélkül az SQL utasítás hibásan *INSERT INTO COFFEESVALUES*...–nek fogja olvasni, és a DBMS a *COFFEESVALUES* nevű táblát fogja keresni. Szintén oda kell figyelni, hogy egyszeres idézőjelet használunk a kávé nevének megadásakor, mert az két dupla aposztróf közé van beágyazva. A legtöbb adatbázis-kezelőnél általános szabály, hogy szabadon választható, melyik aposztrófot használjuk.

```
Statement stmt = con.createStatement();
stmt.executeUpdate(
    "INSERT INTO COFFEES " +
    "VALUES ('Colombian', 101, 7.99, 0, 0)");
```

Az *INSERT* utasítás második sora fogja beilleszteni a rekordot a *COFFEES* táblába. Használjuk fel inkább újra az *stmt* nevű *Statement* objektumot, mint újat példányosítsunk minden végrehajtásnál.

```
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('French_Roast', 49, 8.99, 0, 0)");
```

A további sorokat az alábbiak szerint illesztjük be:

```
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('Espresso', 150, 9.99, 0, 0)");
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('Colombian_Decaf', 101, 8.99, 0, 0)");
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('French_Roast_Decaf', 49, 9.99, 0, 0)");
```

Adatok kinyerése a táblából

Most, hogy a *COFFEES* táblában már adatok is vannak, megírhatjuk a *SELECT* utasítást az adatok kinyeréséhez. A következő kódban a csillag (*) mondja meg, hogy az összes oszlopot akarjuk kiválasztani. Miután nincs *WHERE* feltétellel meghatározva, hogy mely rekordokat adja vissza, a következő SQL utasítás az összes sort vissza fogja adni.

```
SELECT * FROM COFFEES
```

Az eredmény megegyezik azzal, amit az a kimenet ad, amikor az SQL lekérdezést direkt az adatbázis-kezelőnek adjuk meg. Amikor egy Java alkalmazáson keresztül érjük el az adatbázist, vissza kell, hogy nyerjük az adatokat, hogy felhasználhassuk. A következő részben ezt mutatjuk be.

Egy másik példa a SELECT utasításra, amelyik visszaadja a kávék listáját és az árat.

```
SELECT COF_NAME, PRICE FROM COFFEES
```

A lekérdezés eredménye a következő:

A fenti SELECT utasítás visszaadta az összes rekordot, a következő pedig csak azokat, amelyek kevesebbe kerülnek, mint $9.00

```
SELECT COF_NAME, PRICE
FROM COFFEES
WHERE PRICE < 9.00
```

23.2.4 Lekérdezések eredményének feldolgozása

Nézzük meg, hogy az előző SQL parancsot hogyan lehet Javában futtatni, és hogyan lehet az eredményt kinyerni belőle.

A lekérdezést futtató *executeQuery* metódus egy *ResultSet* objektumot ad, amiből az adatokat kinyerhetjük.

```
ResultSet rs = stmt.executeQuery(
    "SELECT COF_NAME, PRICE FROM COFFEES");
```

A *next* metódus

Az eredményhalmaz elérhetővé tevő *rs* objektummal minden rekordon végig fogunk menni, hogy az egyes rekordok adatait meg tudjuk jeleníteni a képernyőn. A *next* metódus az úgynevezett a kurzort mindig a következő sorra mozgatja (ezt aktuális sornak hívjuk). Mivel a kurzor kezdetben még nem mutat egyetlen sorra sem (mintha a táblázat előtt lennénk), az első meghívása után érhető el az eredmény első sora. Egymást követő metódushívásokkal a kurzort a következő rekordokra léptetjük, így járjuk be a teljes halmazt.

A *getXXX* metódusok

A *get* kezdetű metódusok segítségével tudjuk az aktuális rekord egyes mezőit lekérdezni. Például az SQL *VARCHAR* típusú *COF_NAME* mezőjét a *getString*-el olvashatjuk ki. Az SQL *FLOAT* típusú *PRICE* mezője logikusan a *getFloat* metódussal kérdezhető le.

Nézzük a teljes kódot:

```
String query = "SELECT COF_NAME, PRICE FROM COFFEES";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
    String s = rs.getString("COF_NAME");
    float n = rs.getFloat("PRICE");
    System.out.println(s + "   " + n);
}
```

A kimeneten ehhez hasonló fog megjelenni:

```
Colombian   7.99
French_Roast   8.99
Espresso   9.99
Colombian_Decaf   8.99
French_Roast_Decaf   9.99
```

Paraméterként nem csak a mező nevét, hanem oszlopindexét is használhatjuk:

```
String s = rs.getString(1);
float n = rs.getFloat(2);
```

24.Kódolási konvenciók

A kódolási konvenciók a következők miatt szükségesek:

- A szoftver életciklus költségének 80%-a mehet el a karbantartásra
- A szoftvereknek csak kis résztét tartja karban az eredeti fejlesztő
- A kódolási konvenciók javítják az olvashatóságot, segítve a későbbi megértést

A konvenciókhoz mindenkinek alkalmazkodni kell a programozás folyamán. Mindenkinek.

24.1. Fájlok szervezése

Minden java forrásállomány egyetlen publikus osztályt vagy interfészt tartalmaz, de ezen túl tartalmazhat még privát, a forrásállományon kívül kapcsolattal nem rendelkező további osztályokat és interfészeket. A publikus típust javasolt először leírni. A publikus típus neve és a forrásfájl neve meg kell, hogy egyezzen, beleértve a kisbetű-nagybetű különbségét is.

A Java forrásállomány a következő elemeket tartalmazza:

- Kezdő megjegyzés
- Csomag és import kifejezések
- Osztály és interfész deklarációk

Kezdő megjegyzés

Minden forrásállománynak kellene tartalmaznia egy C stílusú megjegyzést a (publikus) osztály nevével, verzió információival, dátummal és szerzői jogokkal:

```
/*
 * Osztálynév
 *
 * Verzió információk
 *
 * Dátum
 *
 * Copyright
 */
```

Javasolt a hagyományos megjegyzés helyett/mellett dokumentációs megjegyzéseket is alkalmazni.

Csomag és import kifejezések

A megjegyzés utáni első nem üres sor a csomag kifejezés kell, hogy legyen. Azután következhetnek az import kifejezések:

```
package java.awt;
import java.awt.peer.CanvasPeer;
```

Megjegyzés: Az egyedi csomagnév első része csupa angol kisbetűből álló, fordított doménnév.

Osztály és interfész deklaráció

A következő táblázat összefoglalja az osztály és interfész deklaráció szabályait:

osztály vagy interfész dokumentációs megjegyzés *(/**...*/)*	
osztály vagy interfész kifejezés	
osztály vagy interfész megvalósítási megjegyzés *(/*...*/)*, ha szükséges	Ez a megjegyzés olyan információkat tartalmaz, amik csak a megvalósítás részleteiről szólnak, az osztály későbbi felhasználása során nem lesz szükség ezekre az információkra. Ilyen módon a dokumentációs megjegyzésbe nem kerül bele.
osztály (statikus) változók	Először a publikus, majd védett, csomag szintű, és végül privát változók szerepeljenek.
példányváltozók	Ugyanolyan láthatósági sorrendben, mint az osztályváltozók
konstruktorok	
metódusok	A metódusok feladat szerint és láthatóság szerint csoportosítva.

24.2. Behúzás

Megjegyzés: Ebben a jegyzetben a nyomtatás speciális tördelése miatt néhol eltértünk a következő szabályoktól.

2 vagy 4 szóköz, vagy a tabulátor karakter használata célszerű.

A sorok hossza ne legyen több 80 karakternél. Bizonyos esetekben érdemes 70 karakterben maximálni a sor hosszát.

Hosszú sorok tördelése

Ha egy kifejezés nem fér el egy sorban, a következők alapján tördeljünk:

- Törjünk vessző (,) után
- Törjünk egy operátor előtt
- Részesítsük előnyben a magasabb szintű művelet előtt való törést
- Igazítsuk az új sor kezdetét az előző sor ugyanolyan szintű kifejezéséhez
- Ha az előzőek alapján zavaros lesz az eredmény, vagy így is túl hosszú a sor, akkor a szokásos behúzás kétszeresét (8 karakter) is alkalmazhatjuk

Nézzünk egy példát illusztrációként:

```
someMethod(longExpression1, longExpression2, longExpression3,
        longExpression4, longExpression5);
var = someMethod1(longExpression1,
                someMethod2(longExpression2,
                        longExpression3));
```

A következő két példa aritmetikai kifejezések töréséről szól. Az első példa elfogadható, a törés alkalmazkodik a logikai szerkezethez, de a második nem javasolt.

```
longName1 = longName2 * (longName3 + longName4 - longName5)
           + 4 * longname6;

longName1 = longName2 * (longName3 + longName4
                       - longName5) + 4 * longname6;
```

A következő két kód metódusdeklaráció bemutatására szolgál. Az első a szokásos esetet mutatja, de a másodiknál ez nem működne. Ezért ott a fix kétszeres behúzást alkalmazzuk:

```
someMethod(int anArg, Object anotherArg, String yetAnotherArg,
           Object andStillAnother) {
    ...
}

private static synchronized horkingLongMethodName(int anArg,
        Object anotherArg, String yetAnotherArg,
        Object andStillAnother) {
    ...
}
```

Az *if* utasítás törése esetén is a dupla behúzás javasolható, az egyszeres behúzás nehezebben felismerhetővé teszi a törzset. Az első tehát nem célszerű, helyette a második vagy harmadik javasolható:

```
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}

if ((condition1 && condition2)
        || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}

if ((condition1 && condition2) || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

Végül három jó példa a feltételes operátor alkalmazására:

```
alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta
                                 : gamma;

alpha = (aLongBooleanExpression)
        ? beta
        : gamma;
```

24.3. Megjegyzések

A Java programok kétféle megjegyzést alkalmaznak: dokumentációs és implementációs megjegyzést. Az utóbbi a C++ nyelvhez hasonló, míg az előbbi a Java specialitása. (Megjegyzés: Ezt a bevált ötletet a PHP is átvette.) A dokumentációs megjegyzés /** és */ közötti karaktersorozat. A *javadoc* segédprogram képes ez alapján automatikusan generálni HTML alapú, teljesen hipertext felépítésű dokumentációt.

A dokumentációs megjegyzés a forráskódot dokumentálja olyan megközelítésben, hogy az alapján a későbbi megértés, az osztály felhasználása maradéktalanul megoldható legyen a kód kézzel történő végignézése nélkül is.

24.3.1 Implementációs megjegyzések

A programok négyféle implementációs megjegyzést tartalmazhatnak: blokk, egysoros, nyomkövetési és sorvégi.

Blokk megjegyzések

A blokk megjegyzések leírást tartalmaznak a fájlhoz, metódushoz, adatszerkezethez vagy algoritmushoz. Fájl esetén a fájl elején, a többi esetben a metódus belsejében szokás elhelyezni, alkalmazkodva a megjegyzéssel ellátott egység behúzásához.

A blokk megjegyzés egy üres sorral kezdődik és fejeződik is be:

```
/*
 * Blokk megjegyzés.
 */
```

Egysoros megjegyzések

A rövid megjegyzéseket egyetlen sorba írva, a bekezdési szintet figyelembe véve lehet elhelyezni. Ha nem lehet egy sorba írni, akkor blokk megjegyzést érdemes alkalmazni. Egy üres sor után érdemes írni a megjegyzést, hogy vizuálisan láthatók legyenek az összefüggések.

```
if (feltétel) {

    /* Eset leírása. */
    ...
}
```

Nyomkövetési megjegyzések

Nagyon rövid megjegyzéseket az adott kódsor végére is lehet írni, megfelelő közzel elválasztva a kódtól. Célszerű az egymás utáni nyomkövetési megjegyzéseket azonos pozícióban kezdeni. A következő példa mutatja a használat módját:

```
if (a % 2 == 0) {
    return TRUE;            /* páros szám esetén kész */
} else {
    return isPrime(a);      /* különben vizsgáljuk */
}
```

Természetesen alkalmazható a // is, de csak következetesen, nem keverve a két módot. Nem érdemes olyan megjegyzéseket alkalmazni, amik a későbbi megértést nem segítik elő jelentősen.

Megjegyzés: A jegyzet példái oktatási célból készültek, sokszor túl részletesen kommentezve. Valódi fejlesztéskor csak a ténylegesen szükséges megjegyzéseket érdemes elhelyezni.

24.3.2 Dokumentációs megjegyzések

A következő példa egy jó áttekintést ad az alapokról:

```
/*
 * @(#)Blah.java        1.82 99/03/18
 *
 * Copyright (c) 1994-1999 Sun Microsystems, Inc.
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 *
 * This software is the confidential and proprietary
 * information of Sun Microsystems, Inc. ("Confidential
 * Information").  You shall not disclose such Confidential
 * Information and shall use it only in
 * accordance with the terms of the license agreement
 * you entered into with Sun.
 */
package java.blah;
import java.blah.blahdy.BlahBlah;
/**
 * Class description goes here.
 *
 * @version     1.82 18 Mar 1999
 * @author     Firstname Lastname
 */
public class Blah extends SomeClass {
    /* A class implementation comment can go here. */
    /** classVar1 documentation comment */
    public static int classVar1;
    /**
     * classVar2 documentation comment that happens to be
     * more than one line long
     */
    private static Object classVar2;
    /** instanceVar1 documentation comment */
    public Object instanceVar1;
    /** instanceVar2 documentation comment */
    protected int instanceVar2;
    /** instanceVar3 documentation comment */
    private Object[] instanceVar3;
```

```
    /**
     * ...constructor Blah documentation comment...
     */
    public Blah() {
        // ...implementation goes here...
    }
    /**
     * ...method doSomething documentation comment...
     */
    public void doSomething() {
        // ...implementation goes here...
    }
    /**
     * ...method doSomethingElse documentation comment...
     * @param someParam description
     */
    public void doSomethingElse(Object someParam) {
        // ...implementation goes here...
    }
}
```

A legfelső szintű osztályok és interfészek a sor elején kezdődnek, míg a tagok behúzva. Az osztályok és interfészek dokumentációs megjegyzésének első sora nem behúzott, a további sorok egy szóközzel bentebb kezdődnek. Minden tag (osztály, interfész, adattag, metódus, konstruktor) dokumentációs megjegyzésének első sora 4 szóközzel, a továbbiak 5 szóközzel bentebb kezdődnek. A további szintek ugyanígy, értelemszerűen.

24.4. Deklarációk

A könnyű kommentezhetőség (és a jobb átláthatóság) érdekében minden sor csak egy deklarációt tartalmaz:

```
int level; // indentation level
int size;  // size of table
```

Kevésbé javasolható:

```
int level, size;
```

Semmiképpen ne kerüljenek egy sorba különböző típusú deklarációk:

```
int foo,  fooarray[]; //WRONG!
```

A kód még könnyebben olvasható lesz, ha a típusok, azonosítók és megjegyzések táblázatszerűen tabuláltak:

```
int         level;           // indentation level
int         size;            // size of table
Object      currentEntry;    // currently selected table entry
```

A lokális változókat lehetőleg már a deklarációnál inicializáljuk. Az egyetlen ésszerű kivétel, ha a kezdőérték bonyolult számításokkal fog előállni.

24.4.1 A deklaráció helye

A deklarációk a blokk legelején legyenek. Bár vannak érvek mellette, mégsem érdemes később, az első használat helyén deklarálni az azonosítókat. Ennek az lenne a hátránya, hogy akadályozná az érvényességi tartományon belüli hordozhatóságot.

```
void myMethod() {
    int int1 = 0;         // beginning of method block

    if (condition) {
        int int2 = 0;     // beginning of "if" block
        ...
    }
}
```

Az egyetlen logikus kivétel, amikor a *for* ciklusban hozzuk létre a ciklusváltozót:

```
for (int i = 0; i < maxLoops; i++) { ... }
```

Érdemes elkerülni, hogy egy lokális deklaráció elfedjen egy magasabb szintű deklarációt:

```
int count;
...
myMethod() {
    if (condition) {
        int count = 0;     // AVOID!
        ...
    }
    ...
}
```

24.4.2 Osztály és interfész deklaráció

Amikor osztályt vagy interfészt deklarálunk, érdemes megfogadni a következőket:

- A metódus neve és a ’(’ között ne legyen elválasztó.
- A blokk kezdő ’{’ az adott sor végén legyen
- A blokkzáró '}' behúzása megegyezik a blokk kezdetének behúzásával. Ha üres a blokk, a két zárójelet ne válassza el köz.

```
class Sample extends Object {
    int ivar1;
    int ivar2;

    Sample(int i, int j) {
        ivar1 = i;
        ivar2 = j;
    }

    int emptyMethod() {}
    ...
}
```

- A metódusokat válasszuk el egy üres sorral.

24.5. Utasítások

Egyszerű utasítások

Soronként csak egy utasítás szerepel:

```
argv++;       // Correct
argc--;       // Correct
argv++; argc--;       // AVOID!
```

Összetett utasítások

Az összetett utasítás '{' és '}' között utasítások sorozatát tartalmazza. A tartalmazott utasítások egy közzel (4 szóközzel) bentebb kezdődnek, mint az összetett utasítás blokk.

Feltételes utasítások

Az *if*, *if-else*, *if else-if else* utasítások:

```
if (condition) {
    statements;
}
```

```
if (condition) {
    statements;
} else {
    statements;
}
```

```
if (condition) {
    statements;
} else if (condition) {
    statements;
} else{
    statements;
}
```

Mindig alkalmazzuk a blokkot jelző { és } karaktereket. E nélkül több a hibalehetőség.

```
if (condition) //AVOID! THIS OMITS THE BRACES {}!
    statement;
```

A *switch* szokásos formája:

```
switch (condition) {
case ABC:
    statements;
    /* falls through */

case DEF:
    statements;
    break;

case XYZ:
    statements;
    break;

default:
    statements;
    break;
}
```

Mindig írjuk ki a *break* utasítást. Ha nem kell kilépni a *case* végén, akkor megjegyzéskén szerepeljen.

Ciklusutasítások

A *for* utasítás formája a következő:

```
for (initialization; condition; update) {
    statements;
}
```

A törzs nélküli utasítás blokk nélkül, ’;’-el írható:

```
for (initialization; condition; update);
```

Amikor használjuk a ’,’ operátort az inicializáló vagy a növekmény részben, akkor a kód nehezebben áttekinthető lesz. Érdemes megfontolni, hogy ilyen esetben nem jobb-e az inicializációt a ciklus elé írni.

A szokásos és az üres törzsű *while* utasítás formája:

```
while (condition) {
    statements;
}
```

```
while (condition);
```

A *do-while* formája:

```
do {
    statements;
} while (condition);
```

try-catch utasítás

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
}
```

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
} finally {
    statements;
}
```

24.6. Elválasztók

Az üres sorok logikailag tagolják, és így olvashatóbbá teszik a kódot.

Két üres sort alkalmazzunk osztályok elválasztására. Egy sor elegendő a következő esetekben:

- Metódusok között
- Változó deklarációk és a blokk első utasítása között
- Blokk vagy egysoros megjegyzés előtt
- A metódus logikai egységei között (jellemzően 3-10 soronként)

Üres szóközöket alkalmazunk a következő esetekben:

- A kulcsszó és a ’(’ között:

```
while (true) {
    ...
}
```

- A paraméterlistában vesszők után
- A’.’ kivételével minden bináris operátor előtt és után, unáris operátorok esetén azonban nem alkalmazzuk:

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

while (d++ = s++) {
    n++;
}
printSize("size is " + foo + "\n");
```

- A *for* utasításban a ‘;’-k után:

```
for (expr1; expr2; expr3)
```

- Explicit konverzió esetén:

```
myMethod((byte) aNum, (Object) x);
myMethod((int) (cp + 5), ((int) (i + 3)) + 1);
```

24.7. Elnevezési konvenciók

Az elnevezési konvenciók segítségével jobban olvasható programkódot kapunk. Már a név formája alapján lehet tudni, hogy konstans, csomagnév, osztály vagy változó.

Azonosító típusa	Szabály	Példák
Csomag	Az egyedi csomagnév mindig csupa kisbetűkből áll. Az internet domén név legfelső szintű eleme (pl. com, hu) után a vállalat neve, majd szükség szerint további alcsomag-nevek következnek ponttal elválasztva. Csak az angol abc betűi használhatók.	*com.sun.eng* *com.apple.quicktime.v2* *edu.cmu.cs.bovik.cheese*
Osztály Interfész	Az osztálynév többnyire főnév, a kezdőbetű és a köztes szókezdő betűk nagybetűk, a többi kisbetű. Az aláhúzás karakter nem alkalmazható. Ne legyen rövidítés, mindig a teljes szavakat tartalmazza. Lehet ékezetes karaktereket is alkalmazni, bár ezt bizonyos fejlesztőkörnyezetek (tehát nem a Java!) nem engedik. Emiatt általában érdemes kerülni az ékezetek használatát.	*class Raster;* *class ImageSprite;* *interface RasterDelegate;* *interface Storing;*
Metódus	A metódus neve többnyire ige, a köztes szavak kezdőbetűi kivételével minden kisbetűs, a legelső betű is.	*run();* *runFast();* *getBackground();*
Változó	A változónevek rövidek, de beszédesek legyenek. Az adott érvényességi tartomány bármelyik pontján egyértelmű legyen a szerepe. Lehetőleg kerüljük az egy karakteres neveket, kivéve a ciklusváltozók szokásos nevei (i, j, stb.). A köztes szavak kezdőbetűi kivételével minden kisbetűs, a legelső betű is.	*int i;* *float myWidth;*
Konstans	A konstans változók neve csupa nagybetűs, a többszavas összetételeket az '_' karakterrel kell elválasztani.	*static final int MIN_WIDTH = 4;* *static final int MAX_WIDTH = 999;* *static final int GET_THE_CPU = 1;*

25.Tervezési minták

Ebben a fejezetben alapos elméleti bevezetést nem tudunk adni, csupán néhány egyszerű példát áttekinteni van lehetőségünk. A téma sokkal alaposabb megértéséhez *Erich Gamma, Ralph Johnson, Richard Helm, John Vlissides: Programtervezési minták* című könyvét érdemes elővenni. (Letenni az OOP iránt érdeklődők úgysem tudják :)

Mi a tervezési minta?

Ha egy feladat újra előkerül a fejlesztés folyamán, akkor valószínűleg a megoldás hasonló lesz a korábbihoz. A tervezési minták olyan objektumközpontú megoldásokat jelentenek, amelyek már bizonyítottak a gyakorlatban. Ezek felhasználása rugalmasabban módosítható és könnyebben, jobban újrahasznosítható alkalmazásokat készíthetünk.

A minták leírására egységes módszereket szokás alkalmazni. Ez lényegét tekintve a következőket tartalmazza:

- Motiváció: mi volt az az eredeti probléma, ami miatt a téma előkerült.
- A résztvevők és a struktúra leírása.
- A használat feltételei: meddig terjed a minta alkalmazhatósága.
- További alkalmazási példák.

E jegyzetben – terjedelmi okokból – csupán egy kevésbé formális, az érdeklődés felkeltésére szolgáló bevezetőt tudunk nyújtani.

Bevezetésként még következzen egy áttekintő táblázat:

<table>
<tr><th></th><th></th><th colspan="3">Cél</th></tr>
<tr><th></th><th></th><th>Létrehozási</th><th>Szerkezeti</th><th>Viselkedési</th></tr>
<tr><th rowspan="2">Hatókör</th><th>Osztály</th><td>Gyártófüggvény</td><td>(Osztály)illesztő</td><td>Értelmező
Sablonfüggvény</td></tr>
<tr><th>Objektum</th><td>Elvont gyár
Építő
Prototípus
Egyke</td><td>(Objektum)illesztő
Híd
Összetétel
Díszítő
Homlokzat
Pehelysúlyú
Helyettes</td><td>Felelősséglánc
Parancs
Bejáró
Közvetítő
Emlékeztető
Megfigyelő
Állapot
Stratégia
Látogató</td></tr>
</table>

25.1. Létrehozási minták

A létrehozási mintáknak az a céljuk, hogy az egyes objektumpéldányok létrehozásakor speciális igényeknek is eleget tudjunk tenni.

Az Egyke (*Singleton*) minta például lehetővé teszi, hogy egy osztályból csak egyetlen példányt lehessen létrehozni.

Az Elvont gyár (*Abstract Factory*), Építő (*Builder*) és Gyártófüggvény (*Factory Method*) minták szintén a példányosítást támogatják, amikor nem akarjuk, vagy nem tudjuk, milyen típusú is legyen az adott példány.

25.1.1 Egyke (*Singleton*)

Az Egyke minta akkor hasznos, ha egy osztályból csak egyetlen példány létrehozását szeretnénk engedélyezni. A módszer lényege, hogy kontrolláljuk a példányosítást egy privát konstruktor létrehozásával. Az osztályból csak egyetlen példányt hozunk létre, azt is csak szükség (az első kérés) esetén. A példány elérése egy statikus gyártó metóduson keresztül történik.

Nézzünk először egy általános példát:

```
class Singleton {
  private static Singleton instance = null;
  private Singleton() { }

  static public Singleton instance() {
    if (instance == null) {
      instance = new Singleton();
    }
    return instance;
  }

  public void finalize() {
    instance = null;
  }
}
```

Példányt létrehozni a következő módon tudunk:

```
Singleton one = Singleton.instance();
```

Ha egy újabb példányt kérünk, akkor is ugyanazt az objektumot kapjuk. Hibás lenne viszont, ha közvetlenül próbálnánk példányosítani:

```
Singleton one = new Singleton(); // hibás
```

Egy kicsit más megoldást láthatunk egy távoli kapcsolat kiépítését megvalósító osztálynál. Itt a kapcsolat mindig szükséges, a duplikálás elkerülése a fő célunk.

Megjegyzés: Érdemes belegondolni, milyen zavarokkal járna, ha a programozó figyelmetlensége miatt egyszerre két vagy több távoli kapcsolatot próbálnánk kiépíteni és használni ugyanazon erőforrás felé.

```
final class RemoteConnection {
  private Connect con;

  private static RemoteConnection rc =
      new RemoteConnection(connection);
```

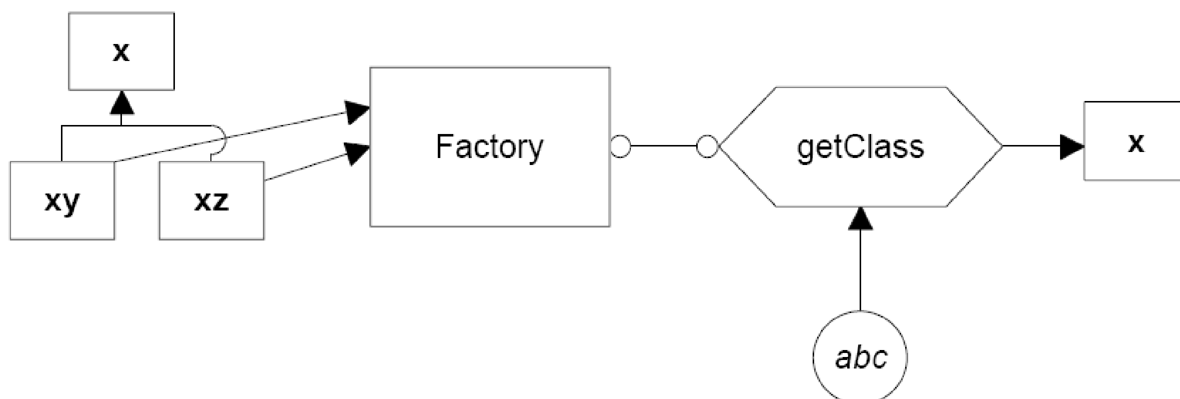
```
  private RemoteConnection(Connect c) {
    con = c;
    ....
  }
  public static RemoteConnection getRemoteConnection() {
    return rc;
  }
  public void setConnection(Connect c) {
    this(c);
  }
}
```

Szerzői megjegyzés: További minták bemutatása tervben van.

25.1.2 Gyártófüggvény (Factory Method) minta

A gyártófüggvény minta az egyik legtöbbet alkalmazott tervezési minta. A minta célja egy objektum létrehozása különböző információk alapján.

A következő ábrán a bemeneti információt az *abc* jelképezi. Ez alapján a gyártófüggvény az *x* ősosztály valamelyik leszármazottját (*xy* vagy *xz*) fogja példányosítani. A *getClass* hívására létrejövő objektum tényleges típusáról többnyire nem is kell tudnia a felhasználónak.



Nézzünk egy konkrét példát. A feladatunk az, hogy a nevek kezelése körüli következő problémát megoldjuk. Az angol nyelvben kétféle módon is megadhatjuk ugyanazt a nevet:

- Jack London
- London, Jack

Ha egy beviteli mezőben a felhasználó megadja a nevét, akkor bármelyiket használja. Nekünk az a feladatunk, hogy a név alapján egy olyan objektumot hozzuk létre, amelyikből bármikor elérhetők a szükséges szolgáltatások.

Először nézzük meg azt az ősosztályt, amelynek a szolgáltatásaira végső soron szükségünk lesz:

```
abstract class Namer {
    protected String last;
    protected String first;
    public String getFirst() {
        return first;
    }
```

```
    public String getLast() {
        return last;
    }
}
```

Nézzük meg a két igen egyszerű leszármazott osztályt is.

Az első leszármazottunk a név megadásánál az első (szóközös) megadást feltételezi.

```
class FirstFirst extends Namer {
    public FirstFirst(String s) {
        int i = s.lastIndexOf(" ");
        if (i > 0) {
            first = s.substring(0, i).trim();
            last =s.substring(i+1).trim();
        } else {
            first = "";
            last = s;
        }
    }
}
```

A másik leszármazott a vessző karaktert keresi elválasztóként:

```
class LastFirst extends Namer {
    public LastFirst(String s) {
        int i = s.indexOf(",");
        if (i > 0) {
            last = s.substring(0, i).trim();
            first = s.substring(i + 1).trim();
        } else {
            last = s;
            first = "";
        }
    }
}
```

A tényleges példányosítást (gyártást) a következő osztály végzi:

```
class NameFactory {
    public Namer getNamer(String entry) {
        int i = entry.indexOf(",");
        if (i>0)
            return new LastFirst(entry);
        else
            return new FirstFirst(entry);
    }
}
```

Elegendő a *getNamer* metódust a nevet tartalmazó *String* paraméterrel meghívni, eredményül pedig egy *Namer* (leszármazott) objektumot kapunk.

Megjegyezés: A példa láttán felmerülhet az a kifogás, hogy elegendő lett volna a *Namer* osztály konstruktorában ezt a kétféle inputot megkülönbözteti. Ennél a példánál tényleg járható lenne ez az út is.

A példa egyszerűsége abban rejlik, hogy a leszármazottak csak a konstruktorunkban térnek el egymástól. Más összetettebb szituáció esetén a leszármazottak érdemi működése is jelentősen eltérhet egymástól.

Legerősebb érvként pedig azt érdemes meggondolni, hogy ha a bemutatott struktúrát kell bővítenünk egy újfajta viselkedéssel, akkor elegendő egy új leszármazott osztály létrehozása és a *getNamer* metódus bővítése, a többi osztályhoz egyáltalán nem kell hozzányúlni. Ez egy nagyobb alkalmazás esetén nagyon erőteljes érv lehet.

25.2. Szerkezeti minták

A szerkezeti minták segítségével előírhatjuk, hogy az egyes osztályokból vagy objektumokból hogyan álljon elő egy komplexebb struktúra.

Az osztály minták célja, hogy olyan öröklési hierarchiát alakítsunk ki, amelyik jól használható programfelületet nyújt. Ezzel szemben az objektum minták az objektumok összeillesztésének célszerű módjait alkalmazzák.

Az Illesztő (*Adapter*) mintára akkor van szükség, ha különböző felületű osztályoknak kell kapcsolatba hozni. Így tulajdonképpen a két félnek nem is kell egymásról konkrétan tudni, elég, ha a köztük lévő illeszti mindkét felet.

A Híd (*Bridge*) minta ezzel szemben nem kényszerhelyzet, hanem tudatos tervezés miatt ad valami más felületet. A szolgáltatás felületét (interfészét) és megvalósítását (implementációját) tudatosan választja szét.

A Homlokzat (*Facade*) mintával egy nagyobb komponensnek egységes felületet (interfészt) tudunk nyújtani.

A Pehelysúlyú (*Flyweight*) minta lehetőséget ad arra, hogy elrejtsünk egy objektumot, és azt csak akkor hozzuk elő, ha arra tényleg szükség lesz.

25.3. Viselkedési minták

A viselkedési minták (a szerkezeti mintákkal szemben) nem az állandó kapcsolatra, hanem az objektumok közötti kommunikációra adnak hatékony megoldást.

A Megfigyelő (*Observer*) minta célja, hogy egy objektum állapotváltozásainak figyelését lehetővé tegye tetszőleges más objektumok számára. A figyelő pozícióba feliratkozással juthatunk, de a leiratkozás is bármikor megejthető.

A Közvetítő (Mediator) minta célja, hogy két – egymással kommunikálni képtelen osztály között közvetítő szerepet töltsön be. Érdekesség, hogy a két közvetve kommunikáló osztálynak semmit sem kell egymásról tudnia.

A Felelősséglánc (*Chain of Responsibility*) minta az objektumok közül megkeresi a felelőst. Egyszerű példaként el lehet képzelni egy böngésző alkalmazás ablakát, ahol a felület tulajdonképpen többszörösen egymásba ágyazott komponensek segítségével épül fel. Egy egérkattintás esetén a főablak-objektumból kiindulva (a vizuális tartalmazás mentén) egyre pontosabban meg tudjuk határozni, hogy a kattintás melyik doboz, melyik bekezdés, melyik űrlap, melyik űrlap-elem stb. területén történt.

A Stratégia (*Strategy*) minta egy algoritmust egy osztályba zár. Így az algoritmus későbbi leváltása csak az ős egy másik leszármazottját fogja igényelni.

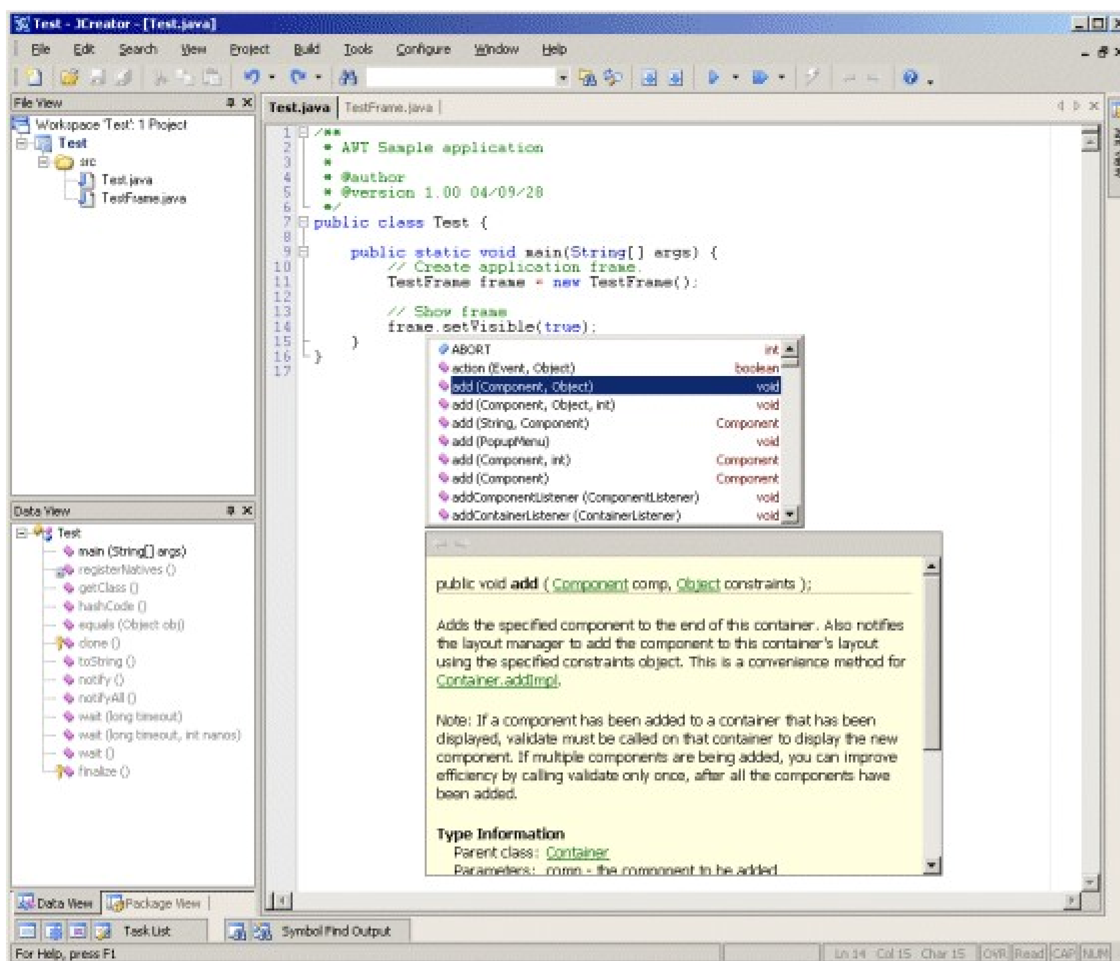
Végül a Bejáró (*Iterator*) minta már ismerős lehet a Java *Iterator* interfésze miatt: feladata a bejárás biztosítása pl. tároló objektumokon.

26.Java fejlesztőeszközök

Bármelyik fejlesztőeszközt is választjuk, telepítsük először a fejlesztőkörnyezetet és a dokumentációt az 1.1 fejezetben leírt módon.

26.1. JCreator

A JCreator Java programozásra alkalmas editor. A program letölthető a *http://www.jcreator.com/download.htm* címről. A *Pro* változat 30 napig működő demó, de letölthető a korlátlan ideig használható (*freeware*), bár némileg kevesebb tudású *Lite Edition* változat is.



26.2. Netbeans

A Netbeans a Sun saját fejlesztőkörnyezete a Java platformokhoz.

A J2SE változattal közös csomagban is letölthetjük, így a telepítés is nagyon egyszerű lesz. (A közös csomag a Sun honlapjáról tölthető le.)

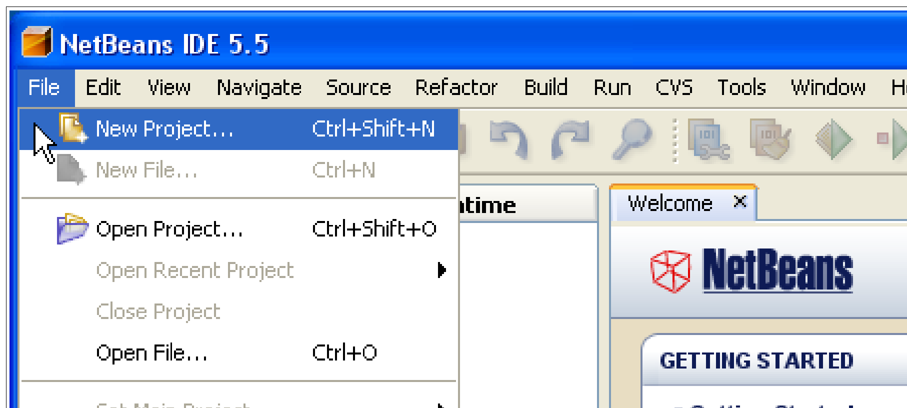
A *http://www.netbeans.org/* címről nem csak az alapcsomagot, hanem különböző összeállításokat és kiegészítőket is letölthetünk. Ha nem a J2SE-vel közös csomagot telepítjük, akkor (a JCreatorhoz hasonlóan) a Netbeanst érdemes később telepíteni.

26.2.1 Alapvető használat

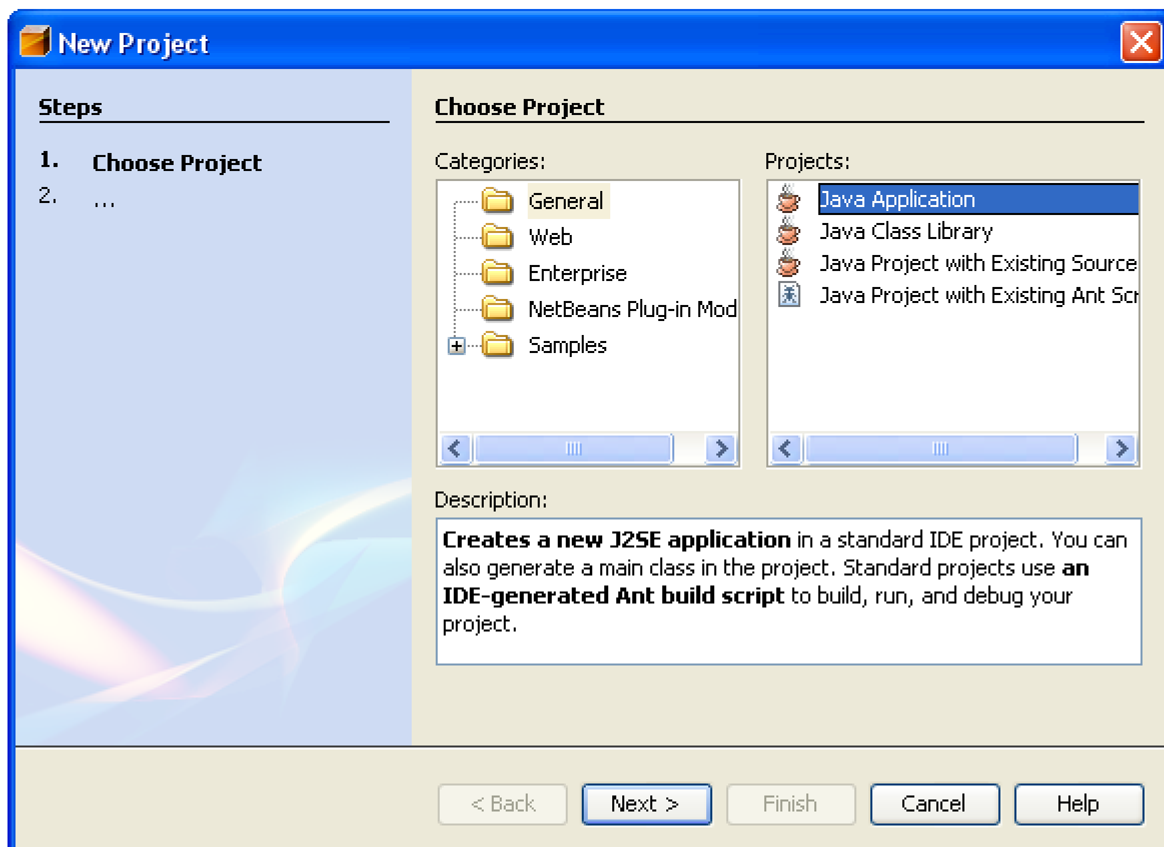
Ebben a részben egy egyszerű alkalmazás készítésének lépéseivel ismerkedünk meg.

Projekt létrehozása

A Netbeans indítása után *File > New Project*:



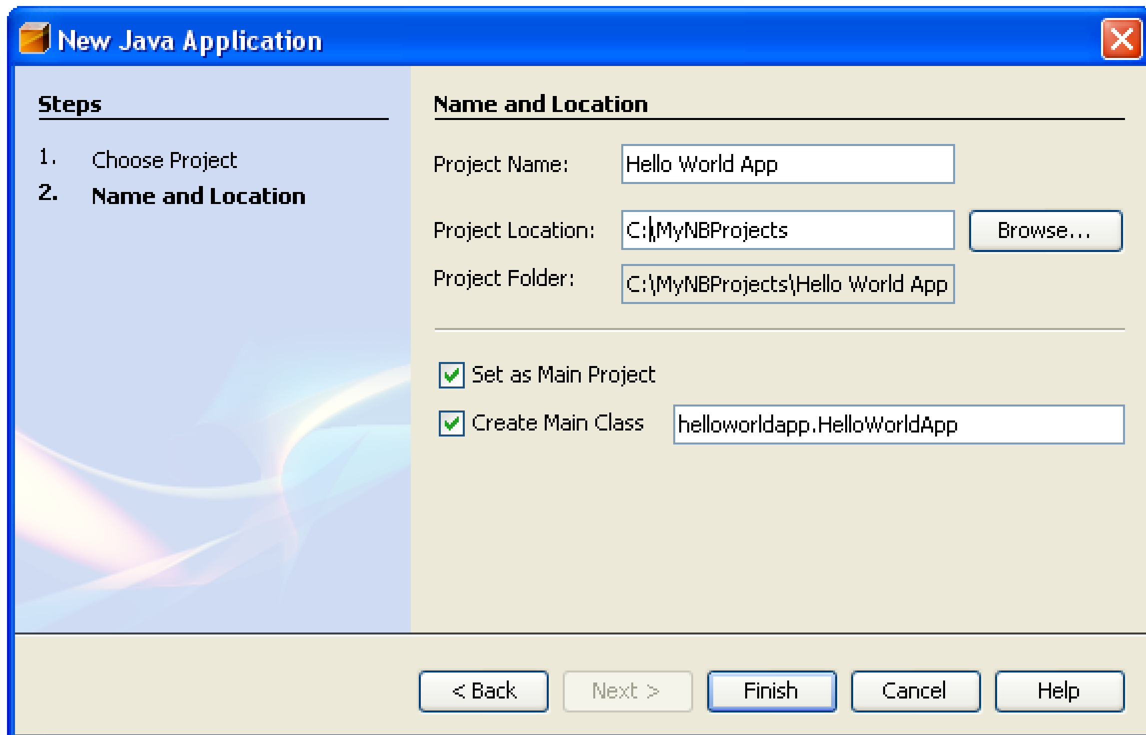
Egy alapvető Java projekthez a *General* csoport *Java Application* eleme alkalmas.



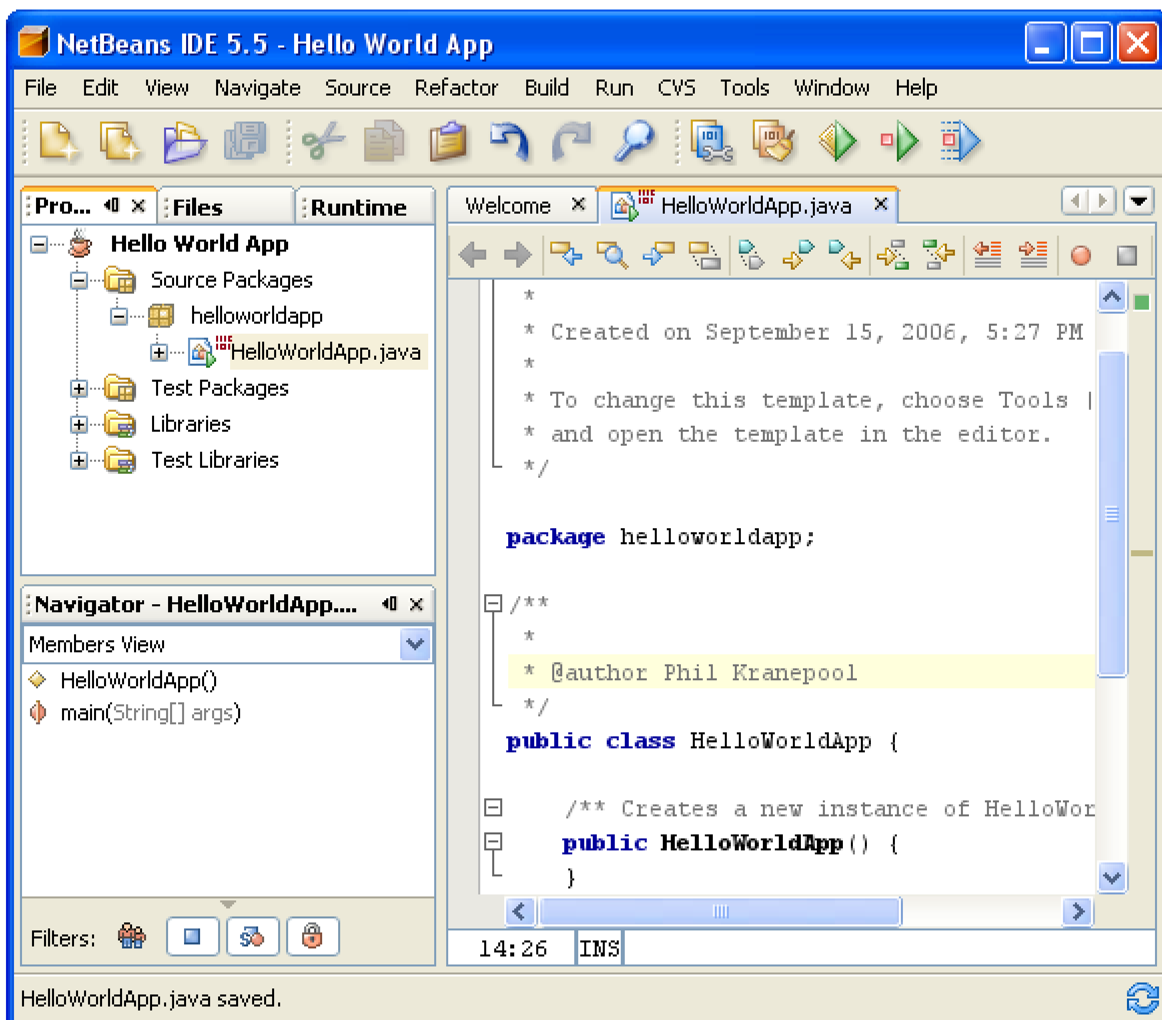
Következő lépésként a projekt nevét és elérési útját kell megadnunk.

(A projekt alkalmas arra, hogy egy több osztályból álló alkalmazás forrásállományit egy egységként kezeljük.)

Beállíthatjuk projektünket elsődlegesnek (Main project), és megadhatjuk a fő osztályunk nevét is:

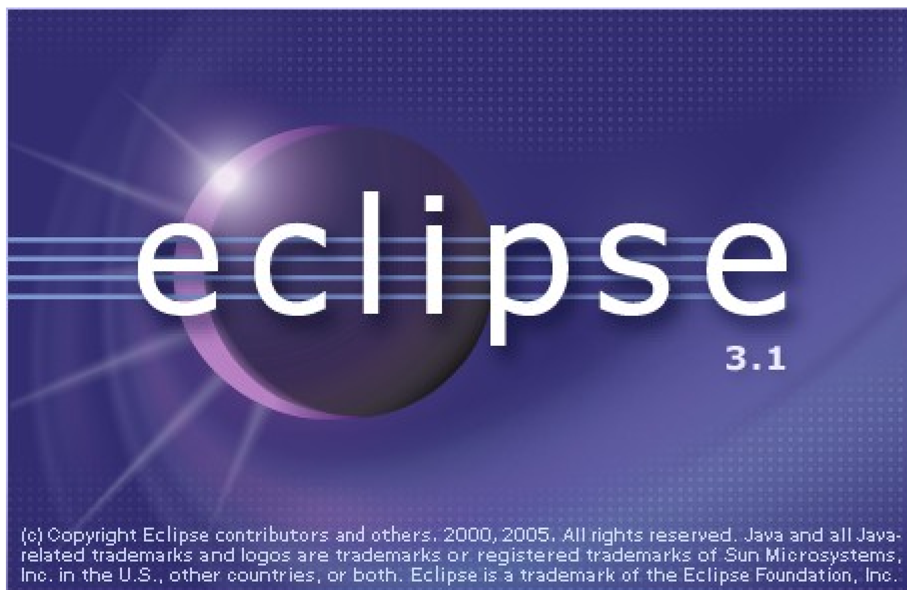


A varázsló futásának eredménye jól látható a következő képen:



26.3. Eclipse

Az Eclipse egy több platformon (Windows mellett Linux, Solaris, AIX, HP-UX, Mac OS-X) rendelkezésre álló, több programozási nyelvet (Java, PHP, C/C++, stb.) támogató és többfajta fejlesztői környezetben (asztali alkalmazásfejlesztés, webfejlesztés, mobil alkalmazásfejlesztés, UML2 szerkesztés, vizuális szerkesztés stb.) alkalmazható nyílt forrású szoftverfejlesztő projekt.



Ebben a fejezetben a Java fejlesztéshez szükséges minimális alapismeretekről lesz szó.

Megjegyzés: Az Eclipse igényli a Java SDK telepítését, tehát az 1. fejezetben ismertetett módon először a fejlesztőkörnyezetet kell letölteni és telepíteni!

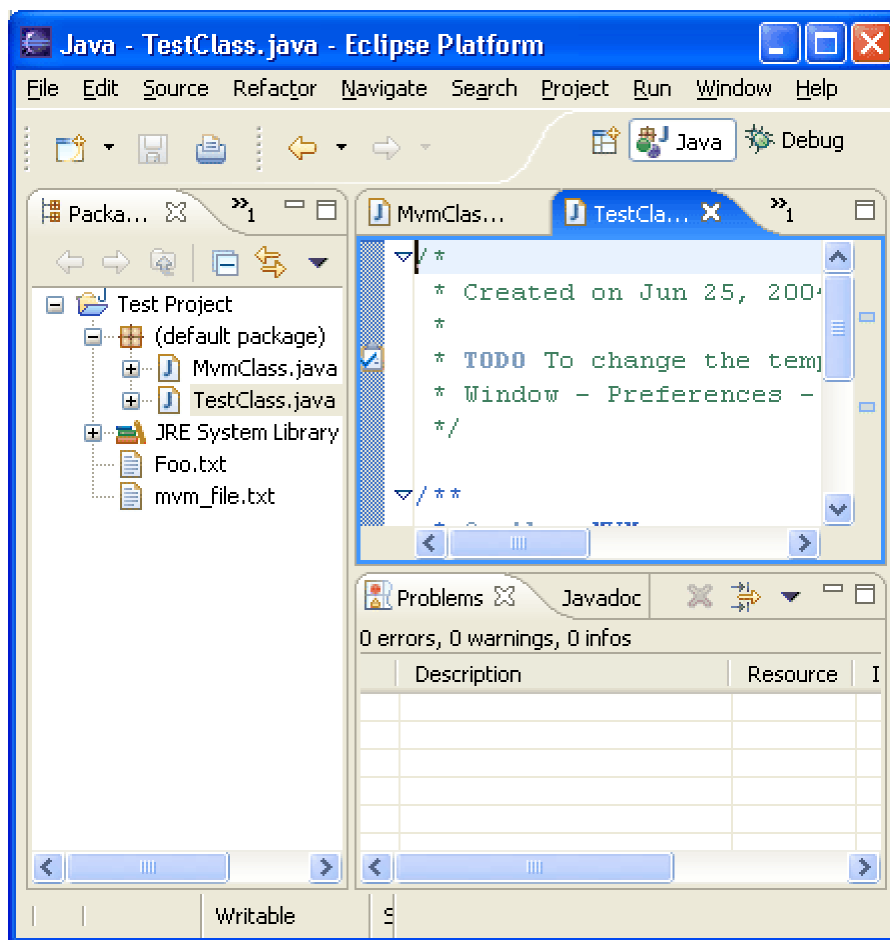
26.3.1 Alap tulajdonságok

Az Eclipse egy teljesen a felhasználó igénye szerint kialakítható felülettel rendelkezik (melyet perspektívának hívnak), egyszerű (fogd és vidd) felületen átpakolhatjuk a megnyitott ablakokat (nézeteket). Több felületet is kialakíthatunk magunknak az éppen aktuális munkától függően (programozás, HTML szerkesztés, stb.), melyek között a rendszer automatikusan is tud váltani, illetve mi is váltogathatunk.

Számos nézet áll rendelkezésünkre alapkiépítésben, illetve ezek listája beépülő modulokkal tovább bővíthető. Ilyen a szerkesztőmező és a projekt fájlok listája (*navigator*) mellett a problémák, tennivalók listája, a kivonat doboz (az aktuális állomány váza: XML szerkezet, vagy a függvények listája, stb.).

A beépített szerkesztő főleg Java és kapcsolódó kódszínezéseket támogat, de ezt a Colorer plugin[2] segítségével kiegészíthetjük: így rengeteg nyelv áll rendelkezésünkre, illetve ha nem találunk egyet, XML leírók segítségével kibővíthetjük a rendszert.

[2] http://colorer.sourceforge.net/



26.3.2 Az Eclipse beszerzése és üzembe helyezése

Java fejlesztéshez mindenekelőtt a *http://www.eclipse.org/downloads/* oldalról le kell tölteni az *Eclipse SDK* nevű termék legfrissebb stabil verzióját. Mivel a letöltött állomány egy ZIP formátumú tömörített állomány, kézzel kell kicsomagolnunk, pl. a *C:\eclipse* könyvtárba.

További telepítésre nincs szükség, csupán az indítás után néhány alapbeállítást (konfigurálást) kell megtenni. Ízlés szerint parancsikont is lehet létrehozni az *Exclipse.exe* számára.

Az első futtatás alkalmával be kell állítanunk a munkaterületünket, vagyis ki kell jelölnünk azt a könyvtárat, ahol a fejlesztéshez kapcsolódó állományainkat tárolni szeretnénk. A későbbiekben itt hozhatunk létre projekteket, amelyek más-más alkönyvtárban jönnek létre.

A konfigurálás első lépéseként a Java fejlesztőkörnyezet kell beállítanunk: *Window* > *Preferences*, majd *Java* > *Installed JREs*. Itt jó esetben a telepített futtatókörnyezetet fel is ajánlja. Szükség esetén lehet módosítani is a beállításokat.

Érdemes még átnézni a *General* > *Wokspace* és az egyéb *General* beállításokat is.