Objektumorientált programozás

Antal Margit

TARTALOMJEGYZÉK

1. FEJEZET

Programozási paradigmák

A programozási paradigma, vagy egyszerűen programozási mód meghatározza, hogy milyen egységekben gondolkodunk. A programozási mód gyakorlatilag az eszközkészletet határozza meg, amelyből felépítjük programjainkat. Egy programozási nyelv többféle programozási paradigmát is megengedhet, illetve a megengedettek közül bizonyosakat támogathat is. Ha egy programozási nyelv támogat egy adott programozási paradigmát, akkor eszközöket is biztosít hozzá. Például a BASIC nyelv első változata nem támogatta a strukturált programozást, mert nem biztosította a strukturált programozáshoz szükséges iterációkat. Azt is elmondhatjuk viszont, hogy a nyelv megengedte strukturált programok készítését, csak ez rendkívül nehézkes volt, bizonyos feladatoknál pedig egyszerűen lehetetlen. Egy másik példa a C programozási nyelv, amely nem támogatja az objektumorientált programozást, mert nem biztosítja a legalapvetőbb nyelvi eszközöket, mint például az osztály, de megengedi, hiszen struktúrák segítségével az osztály szimulálható, csak rendkívül nehézkes.

A programozás történelmében nagyon sok programozási paradigma volt, ezek a programozás fejlődésével párhuzamosan alakultak ki. Időrendi sorrendben a következő programozási paradigmákat említhetjük:

- Strukturált programozás
- Eljárásközpontú programozás
- Moduláris programozás

– Objektumorientált programozás

– Általánosított programozás

A `strukturált programozás` három alapelemből építi fel a programokat: szekvencia, elágazás, iteráció. Böhm és Jacopini tétele alapján minden olyan program, amelynek egy belépési és egy kilépési pontja van, felépíthető e három alapelemből. A strukturált programozásban az iterációknak (ciklusoknak) is egy belépési és egy kilépési pontja van, ennek következtében áttekinthetőbbek a strukturált programok. A strukturált programozás a 70-es évek programozási stílusa.

Az `eljárásközpontú programozás` esetében a hangsúly a kód szervezésére és nem pedig az adatszervezésre esik. Így programjainkat függvényekből és eljárásokból rakjuk össze, kevésbé figyelve az adatok szervezési módjára. Természetesen egy program többféle programozási paradigmát is implementálhat, lehet egyidőben eljárásközpontú és lehet strukturált is.

A harmadik programozási paradigma a `moduláris programozás`. A programozás fejlődése során a programtervezés súlypontja az eljárások felől az adatszervezés irányába tolódott el. Az egymással rokon eljárások az általuk kezelt adatokkal együtt modult alkotnak. A modulokkal már megvalósítható az adatrejtés elve. Ennek szemléltetésére tekintsük a következő karaktereket tartalmazó verem adatstruktúra C nyelvű megvalósítását. Az `adatrejtést` úgy valósíthatjuk meg, hogy elválasztjuk az interfész részt az implementációs résztől, az előbbiben pedig semmit nem adunk meg az adatok szervezésére vonatkozóan.

```
typedef struct stack* STACKPTR;

STACKPTR create( int _size);
void destroy( STACKPTR _ptr);
void push( STACKPTR _ptr, char _elem);
char pop( STACKPTR _ptr);
```

Az fenti interfész rögzíti ugyan a tárolt adatok típusát (az egyszerűség kedvéért), de szervezésükre vonatkozóan nem nyújt semmilyen részletet. Így nyugodtan ábrázolhatjuk a vermet láncolt lista vagy akár tömb segítségével is. Természetesen a `create` függvény sejteti, hogy itt korlátos méretű veremről van szó, amely

ábrázolására a tömb alkalmasabb. Az interfész implementációját az alábbi programrészlet szemlélteti.

```
#include "stack.h"

struct stack{
   char * elements;
   char * top;
   int size;
};

//helyfoglalás és a top illetve a size mező beállítása
STACKPTR create( int _size){ ... }

//lefoglalt hely felszabadítása
void destroy( STACKPTR _ptr){ ... }

//túlcsordulás ellenőrzése és _elem behelyezése
void push( STACKPTR _ptr, char _elem){ ... }

//alulcsordulás ellenőrzése és a legfelső eleme kiemelése
char pop( STACKPTR _ptr){ ... }
```

Az `objektumorientált paradigma` az osztályt tekinti az építkezés alapegységének és a moduláris programozás továbbfejlesztéseként jött létre. Már a moduláris programozás is egy modulba zárja az adatokat az azokon végrehajtható műveletekkel, az osztály biztonságosabbá teszi ezt az összezárást azáltal, hogy láthatósági módosítókat vezet be az osztály tagjaira nézve, így garantáltan csak az osztályban deklarált műveleteken keresztül férhetünk hozzá adatainkhoz. Tehát az adatrejtés mellett szavatolja az `adatok biztonságát` is. Az objektumorientált programozás három fő tulajdonságát már most bevezetjük, részletesen a következő fejezetekben fejtjük ki.

- `Egységbezárás`: Ez az adatok és a rajtuk végezhető műveletek osztályban való összezárását jelenti.

– `Polimorfizmus`: Ez a kódra vonatkozó tulajdonság, többalakúságot jelent, pontosabban azt, hogy bizonyos műveletek esetében, futásidőben dől el ezek viselkedése.

– `Származtatás, öröklés`: Új típus megadásának egyik módja.

Az ötödik paradigma, az `általánosított programozás`, amely a 90-es évek paradigmájának is tekinthető. Ez a paradigma két nagy újítással indul, az egyik az általános típus fogalma, a másik az általános algoritmus fogalma. Míg az általános típus lehetővé teszi, hogy adatstruktúráinkat típusfüggetlen módon adhassuk meg, addig az általános algoritmus lehetővé teszi, hogy alapvető rutinjainkat tárolófüggetlen módon adhassuk meg. Egyszerűbben úgy is mondhatnánk, hogy például a szekvenciális keresési algoritmust megadhatjuk úgy, hogy ez bármely tárolóra helyesen működjön. Így ugyanazzal az algoritmussal kereshetünk tömbben, láncolt listában és akár nemlineáris adatstruktúrákban is, amelynek elemei felsorolhatóak (létezik egy bejárási sorrend). Az általánosított programozás két alapeleme az osztálysablon és a függvénysablon. Ezek segítségével valósítható meg az általános típus, illetve az általános algoritmus. Ezen programozási paradigma részletes leírása megtalalálható a [2] és a [10] könyvekben.

2. FEJEZET

A Java nyelv története és tulajdonságai

Az objektumorientált nyelveket osztályozhatjuk a megengedett programozási módok alapján tiszta OO nyelvek, illetve hibrid OO nyelvek kategóriákba. Azokat a nyelveket, amelyek többféle programozási módot megengednek, hibrid nyelveknek nevezzük. Ilyen például a C++, amely a C nyelv objektumorientált kiterjesztése ([6]). A tiszta OO nyelvek csak objektumokkal dolgoznak, így nem léteznek például primitív típusok. Ilyen tiszta OO nyelv a Smalltalk. A Java nyelv e kettő között helyezkedik el. Tartalmaz ugyan primitív típusokat, de osztályon kívül nem engedélyez sem adat-, sem pedig metódus definíciót. A Java nyelv története 1991-ben kezdődik, amikor a Sun cég keretében elindul a Green nevű kutatási projekt, háztartási gépek vezérlésére alkalmas szoftver fejlesztésére. A projektvezető James Gosling volt, aki a UNIX világban az `emacs` szövegszerkesztő szerzőjeként ismert. Kezdetben C++ nyelvben fejlesztettek, de ez nem bizonyult kellőképpen hibatűrőnek, így kifejlesztettek egy hibatűrőbb nyelvet, amelyet eredetileg OAK-nak neveztek el. Mivel ez a név már márkanév volt, átnevezték JAVA-nak. Azóta a nyelv különböző átalakulásokon ment át, ezek nagyrésze új fogalmak bevezetését jelentette. Lényeges újítást hozott az 1.1-es változat, amikor újraírták az eseménykezelési modellt. Az aktuális 1.5-ös változat nagy újítása, hogy lehetővé teszi az általánosított programozást.

2.1. A Java nyelv tulajdonságai

Most áttekintjük a részletesség igénye nélkül a nyelv tulajdonságait.

1. Talán az egyik leglényegesebb tulajdonság, amelynek az óriási népszerűséget is köszönheti, az `egyszerűség`. Néhány hét alatt tökéletesen elsajátítható a nyelv. A Java nyelv nem tartalmaz mutatókat, csak hivatkozás (referencia) típusú változókat. A Java hivatkozás típus viszont merőben eltér a C++ hivatkozástól, amelyet nevezhetnénk egy szoros hivatkozásnak is. Amíg egy C++ nyelvbeli hivatkozás szorosan hozzá van ragasztva a hivatkozott objektumhoz, addig a Java hivatkozások, a mutatókhoz hasonlóan, futás közben objektumról objektumra vándorolhatnak. Szintén segítségükkel valósítható meg az osztott memória, hiszen több hivatkozásunk lehet ugyanarra az objektumra. A hivatkozott tárterületek felszabadítását a Java futásidejű környezet végzi egy úgynevezett szemétgyűjtő algoritmussal, amelyet külön programszálon futtat. A szemétgyűjtő (Garbage Collector) azokat az objektumokat szabadítja fel, amelyekre már nem létezik érvényes hivatkozás.

2. A Java `objektumorientált` nyelv, vagyis a Java programjainkat futásidő alatt egymás között kommunikáló objektumok fogják alkotni.

3. Mivel háztartási gépek vezérlésére készült, ezért alapvető szempont volt a `hibatűrő` képesség. Ezt már nyelvi szinten beépítették, így a Java a C++ nyelvvel ellentétben már a legelső változatában tartalmazta a kivételkezelési mechanizmust. A tömbök indexelése is ellenőrzötten történik és a típusátalakítások is korlátozottak.

4. A mai modern kommunikációs világban elengedhetetlenül fontos a `biztonság`. Ezt a Java több szinten is támogatja. Először a nyelv biztonsági szolgáltatásait ismertetjük:

   a) A Java nyelvben nincsenek mutatók, így a memória csak referenciákon keresztül kezelhető.

   b) A tömbök maguk is objektumok, ismerik méretüket, így nem lehet őket túlindexelni. Így a program véletlenül sem tud hozzáférni olyan memóriaterülethez, melyhez nincs joga.

c) A karakterfüzér (String) típusú objektumok létrehozásuk után nem változtathatók meg. Ez megelőzi a gyakori puffertúlcsordulásokat.

d) A Java erősen (néha idegesítően) típusos nyelv. Az explicit típusátalakítás csak összeférhető objektumok között lehetséges, így nem fordulhat elő, hogy nem létező metódus hívódjon meg objektumra.

e) A Java objektum tagjainak négy szintű láthatósági módosítója szintén emeli a biztonságosságot.

f) Az objektumok adattagjainak van alapértelmezett kezdőértéke.

5. Nagyon fontos tulajdonsága a nyelvnek a `hordozhatóság`. A Java nemcsak forrásszinten hordozható, hanem a lefordított Java kód (bájtkód) bárhol végrehajtható egy Java virtuális gépet megvalósító környezetben.

2.2. A Java virtuális gép (Java Virtual Machine)

A Java a nyelvi biztonság mellett az indirekt programvégrehajtással is támogatja a biztonságosságot. A Java kód nem közvetlenül a felhasználó számítógépén fut, hanem egy úgynevezett virtuális gépen, mely nem más mint egy program, mely értelmezi a Java bájtkódot és lefordítja azt mikroprocesszor utasításokká. Ez az indirekt programvégrehajtás teszi lehetővé a bájtkód hordozhatóságát. A virtuális gépbe egyéb biztonsági szolgáltatásokat is beépítettek, így minden osztályt betöltéskor ellenőriz.

A virtuális gépet úgy képzelhetjük el, mint egy virtuális mikroprocesszort, amely saját utasításkészlettel rendelkezik. A Java fordító a Java nyelvben megírt programot a virtuális gép utasításaira fordítja le. Ez a virtuális gép természetesen platform-független. A Java értelmező program tulajdonképpen megvalósítja a virtuális gépet az adott platform felett, átalakítva a virtuális gép-utasításokat a valós mikroprocesszor utasításaivá.

2.3. Java program készítésének lépései

Többféle Java programot is készíthetünk. Kezdetben az önállóan futtatható Java programok készítésének lépéseit ismertetjük, majd az appletek készítését.

A Java program készítése a következő lépések végrehajtását jelenti:

`Szerkesztés:` Ez történhet integrált fejlesztői környezet segítségével vagy egyszerű szövegszerkesztő programmal is. Egy egyetlen osztályt tartalmazó program esetén az eredmény egy `.java` kiterjesztésű állomány. Tételezzük fel, hogy az állomány neve `Program.java`

`Fordítás:` Sikeres fordítás esetén az eredmény egy bájtkódot tartalmazó állomány, melynek neve megegyezik a forrásállományéval és kiterjesztése `.class`, a mi esetünkben `Program.class`. A végrehajtandó utasítás integrált környezet hiányában: `javac Program.java`

`Értelmezés:` Ez a következő utasítással érhető el: `java Program`, amelynek hatására az emulált Java virtuális gépen lefut a bájtkód.

A következő program egy egyszerű, önállóan futtatható Java program.

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

`Megjegyzés:`

Az osztály neve meg kell egyezzen a forrásállomány nevével (kiterjesztés nélküli név).

`Appletek készítése:`

Habár az appletek a már elavult technológiák közé tartoznak, a teljesség kedvéért ismertetjük készítésük lépéseit. Elsősorban azt kell tudnunk, hogy az applet egy beágyazott, önállóan nem futtatható alkalmazás. Az appletek weboldalakba vannak beágyazva, így letöltésüket, illetve indításukat a kliens gép irányítja. Egyszerűbben, ha a böngésző egy olyan weboldalt jelenít meg, amely tartalmaz egy beágyazott appletet is, akkor letölti az appletnek megfelelő bájtkódot és a biztonsági ellenőrzések után elindítja azt. Az indításnak több feltétele is van. Elsősorban Java-barát böngészőt kell használnunk, amely beállítása engedélyezi appletek végrehajtását. Másodsorban a böngészők beállíthatóak úgy is, hogy csak bizonyos,

megbízható forrásból származó appletek futását engedélyezzék, így ellenőrizzük, hogy ezzel is minden rendben van. Ha mindez teljesül, megvan az esélye az applet végrehajtásának, amely a kliens gépén fog futni. Természetesen az appletek több biztonsági megkötéssel is rendelkeznek, vagyis akármit nem szabad nekik csinálni, a kliens gép védelme érdekében.

Tételezzük fel, hogy elkészítettük egy applet forráskódját és elhelyeztük az `ElsoApplet.java` nevű állományban. Ennek beágyazása a weboldalba a következő egyszerű kódrészlettel történik.

```
<HTML>
<BODY>
<APPLET CODE="ElsoApplet.class" WIDTH=300 HEIGHT=200 >
</APPLET>
</BODY>
</HTML>
```

Az applet forrásállománya:

```
import java.applet.Applet;
import java.awt.Graphics;
public class ElsoApplet{
   public void paint( Graphics g ){
      g.drawString("Hello World",50,60);
   }
}
```

Az első különbség a `main` metódus (függvény) hiánya. Mivel nem önállóan futtatható program nincs is szükség erre, hiszen a böngésző indítja és az applet kirajzolásakor meghívódik a `paint` metódus. A második különbség az `import` utasítás jelenléte. Mivel appletünknek szüksége van extra dolgokra is, amelyek a Java API-hoz tartoznak, ezeknek használatát az `import` utasítással kell jeleznünk. Ezt részletesen a Java csomagoknál fogjuk tárgyalni.

3. FEJEZET

Java alapfogalmak

A Java programozási nyelv a C/C++ programozási nyelvekre épült, ennek következtében nagyon sok típus, operátor és utasítás egyáltalán nem különbözik a C nyelvi megfelelőjétől. Ebben a fejezetben röviden ismertetjük ezeket, kiemelve azokat az entitásokat, amelyek különböznek C nyelvbeli megfelelőjüktől. Az alapfogalmak részletesebb ismertetése megtalálható a [1], [3], [9] könyvekben.

3.1. A Java programok szintaxisa

A Java programok az Unicode, 16 bites karakterkészletet használják. A számítástechnika fejlődésével és elterjedésével az ASCII 8 bites kódtábla szűkösnek bizonyult. 8 biten nem volt lehetőség a világ összes nyelvének karaktereinek kódolására. A 16 bites karakterkészlet a különböző karakterkódok számát 256-ról 65536-ra emelte.

Az unikód karaktereket leggyakrabban hexadecimális kódjukkal adjuk meg `\uhhhh` formában, ahol a `\u` az unikód ábrázolást jelentő vezérlő karakter, amelyet négy darab hexadecimális számjegy követ. A lehetséges unikód kódok `\u0000`-tól `\uffff`-ig terjednek.

A Java programban kétféle megjegyzést használhatunk, egysoros és többsoros megjegyzést. A következő kódrészlet ezeket ismerteti:

```
// Egysoros megjegyzés, C++ stílus
```

```
/*
  Többsoros
  megjegyzés,
  C stílus
*/
```

Íratlan szabályok a megnevezésekkel kapcsolatosan:

Osztály neve (azonosítója) mindig nagybetű. Amennyiben ez összetett szó, minden egyes szavát nagybetűvel írjuk. Például: `LakoKocsi`, `EgyetemiHallgato`, `NyelvTanar`.

Változó és metódus azonosítója mindig kisbetűvel kezdődik. Amennyiben ez összetett szó, a második szótól kezdődően, ezeket is nagybetűvel írjuk. Például: `peldanySzamlalo`, `emberPeldany`, `eggyelNovel`, `kettovelCsokkent`.

Állandókat (konstansokat) nagybetűsen ajánlott írni. Például: `MAXKORHATAR`, `MINKORHATAR`.

Literál

A literál egy állandó, amely beépül a program kódjába. Különböző típusú literálok léteznek, ezek a következők: egész, valós, logikai, karakter, szöveg és a `null`.

egész literál

Az egész literálokat a C nyelvhez hasonlóan adjuk meg. A decimális literál esetében használhatunk előjelet is. Az egész literálok típusa automatikusan `int`, de a literál végére helyezett `l` vagy `L` betű által a literál típusa `long` lesz.

- decimális: `+123, -543`
- oktális: `021, 077`

– hexadecimális: `0xa1, 0xff`

valós literál

A valós literálokat kétféleképpen adhatjuk meg: tizedes, illetve lebegőpontos formában. Ha `f` vagy `F` betűt teszünk a literál végére, akkor az `float` típusúként vagyis 4 bájton fog tárolódni, különben `double` típusúként.

– tizedes forma: `3.14, 2.71`

– lebegőpontos forma: `-3.4E12, 4E-7`

karakter literál

A karakter literál egy darab unikód karakter, amelyet megadhatunk a következő módon:

– unikódjával: `'\u0041', '\u0061'`

– megjeleníthető karakter formájával: `'A', 'a'`

– escape szekvenciával: `'\t', '\\'`

szöveg literál

A szöveg literál egy unikód karakterekből álló sorozat. A sorozatot idézőjelek közé tesszük és tartalmazhat escape szekvenciákat is, meg direkt unikód kódokat is.

Példák:

– egyszerű: " `Jutka`"

– unikód karaktert tartalmazó:"`Egy\u0009egy\u0009egy`"

– escape szekvenciát tartalmazó:"`Egy\tegy\tegy`"

3.2. Típusok

A Java nyelvben alapvetően kétféle típus létezik, éspedig primitív és referencia típus. A primitív típus a C nyelvből ismert típus és a Java nyelvben összesen 8 féle létezik:

- egész: `byte, short, int, long`
- valós: `float, double`
- logikai: `boolean`
- karakter: `char`

Minden egész típus negatív és pozitív számok tárolására egyaránt alkalmas. A `byte` típus egy bájt memóriát foglal, egy ilyen típusú változónak `-128` és `127` közötti értékei lehetnek.

A valós típusok ábrázolása IEEE 754 szabvány szerint történik. A `float` típus 4 bájton van ábrázolva és maximum 6-7 számjegyig terjedő tizedest tárol. A `double` ábrázolása 8 bájton történik, pontossága 14-15 jegy.

A logikai típus ábrázolása nem definiált. Egy ilyen típusú változónak két lehetséges értéke van: a `true` és a `false`.

A karakter típus ábrázolása két bájton történik.

A referencia típusú változó a C nyelv mutató típusához hasonlít, annak biztonságos változata. Többféle referenciát fogunk megkülönböztetni, éspedig tömb referenciát és objektum referenciát. Az előző egy C nyelvbeli tömb mutatóhoz hasonlít, az utóbbi pedig a struktúra mutatónak egy kiterjesztett változata, hiszen az osztály típus is a struktúra kiterjesztett változatának tekinthető. Ezt a típust a következő fejezetben ismertetjük.

3.3. Operátorok

Az operátorok nagy része C nyelvből ismert. Különbség a logikai és a bitenkénti operátorok esetében van.

Logikai operátorok: !, $\wedge$, &, |, ,&&, ,||

Operátor	Elnevezés	Asszoc.
[] . (<param>) ++ − −	unáris postfix operátorok	→
+ + − − +− !	unáris prefix operátorok	←
new (<típus>)<kif>	példányosítás, típuskényszerítés	→
* / %	multiplikatív operátorok	→
+−	additív operátorok	→
<< >> >>>	bitenkénti léptető operátorok	→
< <= > >= instanceof	hasonlító operátorok	
== !=	egyenlőségvizsgáló operátorok	→
&	logikai/bitenk. ÉS	→
^	logikai/bitenk. KIZÁRÓ VAGY	→
|	logikai/bitenk. VAGY	→
&&	logikai rövid ÉS	→
||	logikai rövid VAGY	→
?:	feltételes kiértékelés	←
= += -= *= /= %= &= |= ^= <<= >>= >>>=	értékadó operátorok	←

3.1. táblázat. Java operátorok

A Java nyelvben az & és | a hosszú kiértékelésű logikai ÉS, illetve VAGY operátorok, addig a && és || operátorok rövid kiértékelésűek. A rövid kiértékelésű azt jelenti, hogy csak addig történik kiértékelés, amíg a program el nem tudja dönteni a végeredményt. A rövid kiértékelés gyorsítja a programot. Például legyen $a = 10$ és $b = 20$ egész típusú változók, ekkor az $a > 10$ && $b > 10$ esetében a $b > 10$ már nem fog kiértékelődni, hiszen a kifejezés első része hamis, amelynek következtében az egész kifejezés hamisra fog kiértékelődni.

Bitenkénti eltolási (shift) operátorok: <<, >>, >>>

A bitenkénti operátorok esetében is van egy kis különbség, ugyanis a jobbratolás esetében két operátorunk van. Az >> operátor a C nyelvhez hasonlóan működik, míg a >>> operátor nem veszi figyelembe az előjelbitet. Pozitív számra alkalmazva mindkét operátor azonos eredményhez vezet. Például legyen a következő negatív egész szám `int b =-1` , akkor a `b >> 1` eredménye `-1` lesz, míg a `b >>> 1` eredménye $2^{31} - 1$.

Az `instanceof` operátor használatát a polimorfizmusnál szemléltetjük. Ezt az operátort arra használjuk, hogy egy referencia által hivatkozott objektum adott típusra való átalakíthatóságát vizsgáljuk. Például, ha az `r instanceof Allat` kifejezés értéke igaz, akkor az `r` által hivatkozott objektum átalakítható `Allat` típusra.

Típuskonverziók

A Java erősen típusos nyelv, a típusok összeférhetőségét a fordító ellenőrzi. Bizonyos műveletek végrehajtása előtt bizonyos operandusok típusát konvertálni kell más típusokká. A típuskonverzió lehet automatikus (implicit) vagy kényszerített (explicit). Ezen felül a konverzió iránya szerint megkülönböztetünk szűkítő vagy bővítő konverziót.

Az implicit típuskonverziót a fordító automatikusan elvégzi, az explicit típuskonverziót C nyelvhez hasonlóan végezzük. A Java fordító implicit módon csak bővítő konverziót végez. Numerikus kifejezésekben, bináris operátor esetében mindkét operandust a kettő közül a bővebb, de minimum `int` típussá konvertálja. Az eredmény típusa a közös típus lesz.

Primitív típusok esetében egy szűkebb adattípus értéke információvesztés nélkül konvertálható egy bővebb adattípus értékébe. A bővítő konverziók irányai a következők lehetnek: $byte \rightarrow short \rightarrow int \rightarrow long \rightarrow float \rightarrow double$ és $char \rightarrow int$. A szűkítő konverzió információvesztéssel járhat.

3.4. Feladatok

1. A következők közül melyek nem primitív típusúak?
   a) `"t"`
   b) `'k'`
   c) `50.5F`
   d) `"hello"`
   e) `false`
2. A következő kijelentések közül válasszuk ki a helyeseket!

a) A new, delete Java kulcsszavak

b) A try, catch, thrown Java kulcsszavak

c) A static, unsigned, long Java kulcsszavak

d) Az exit, class, while Java kulcsszavak

e) A for, while, do Java kulcsszavak

3. Válasszuk ki a nem egész típusokat!

a) boolean

b) byte

c) float

d) short

e) double

4. A következő típusok közül melyik az, amelynek értékei -2^{31} és $2^{31}-1$ között vannak?

a) float

b) double

c) int

d) long

e) byte

5. Adott a következő kódrészlet, amelyet egy metódusban helyezünk el. Erre vonatkozóan mely kijelentések igazak?

```
int a, b;
b=5;
```

a) Az a lokális változó nincs deklarálva

b) A b lokális változó nincs deklarálva

c) Az a lokális változónak nincs kezdőértéke

d) A b lokális változónak nincs kezdőértéke

e) Az a lokális változó inicializálva van, de nincs deklarálva

6. Adott a következő program. Melyek azok a sorok, amelyek szerepelhetnek a program kimenetében?

```
public class xyz {
   public static void main(String args[]) {
      for(int i = 0; i < 2; i++) {
         for(int j = 2; j>= 0; j--) {
            if(i == j) break;
            System.out.println("i=" + i + " j="+j);
         }
      }
   }
}
```

a) i=0 j=0

b) i=0 j=1

c) i=0 j=2

d) i=1 j=0

e) i=1 j=1

f) i=1 j=2

g) i=2 j=0

h) i=2 j=1

i) i=2 j=2

7. Melyek igaz kijelentések?

a) Az && operátor rövidre zárt logikai ÉS operátor

b) A ! operátor bitenkénti kizáró VAGY operátor

c) Az | operátor bitenkénti VAGY és rövidre zárt logikai VAGY operátor

d) A >> operátor előjel nélküli bitenkénti jobbra tolás operátor

8. Mely számokat írja ki a következő program?

```
public class test {
   public static void main(String args[]) {
      int i, j=1;
      i = (j>1)?2:1;
      switch(i) {
        case 0: System.out.println(0); break;
        case 1: System.out.println(1);
        case 2: System.out.println(2); break;
        case 3: System.out.println(3); break;
      }
   }
}
```

a) 0

b) 1

c) 2

d) 3

9. Mi lesz a következő program kimente?

```
public class test {
   public static void main(String args[]) {
      int i=0, j=2;
      do {
         i=++i;
         j--;
      } while(j>0);
      System.out.println(i);
   }
}
```

a) 0

b) 1

c) 2

d) Fordítási hiba az "i=++i;" következtében

4. FEJEZET

Osztályok és objektumok

4.1. Osztályok

A `típus` egy konkrét fogalom ábrázolása. A típust meghatározza az elemek halmaza és az elemeken végezhető műveletek halmaza. Az `osztály` egy felhasználói típus. Azért tervezünk új típust, hogy meghatározzunk egy fogalmat, amelynek nincs közvetlen megfelelője a nyelv beépített típusai között. Egy szövegszerkesztő programban valószínűleg szükség van egy Bekezdés (Paragrafus) típusra, hiszen a legegyszerűbb szöveg is bekezdések listájaként fogható fel. Az osztály tervezésekor alapvető szempont az adatok elrendezését elválasztani az adatokat kezelő műveletektől. Az osztály mindkettőt fogja tartalmazni, de az osztály két különböző részében, így megvalósul egyidőben az elválasztás is és az összezárás is. Végeredményben az osztály megvalósítja az adatok összezárását a rajtuk végezhető műveletekkel (kóddal). Az osztály az összezárással biztosítani tudja az `adatok védelmét`, biztosítani tudja, hogy csak az osztályban definiált műveleteken keresztül lehessen az adatokhoz hozzáférni. Az `adatok elrejtését` a láthatósági módosítókkal valósítja meg az osztály. Így az osztály egyidőben megvalósítja az adatrejtést (Data Hiding) és az adatbiztonságot (Data Security).

`ADAT+KÓD=OSZTÁLY`

Tekintsük a dátum típust. A C nyelvben ezt a típust egy struktúrával ábrázolhatnánk, a dátumkezelő műveleteket pedig függvényekkel valósíthatnánk meg. Egy lehetséges megvalósítást a következő program szemléltet.

```
struct Datum{
   int ev, ho,nap;
};

typedef struct Datum * DATUMPTR;

//Kezdeti értékadás d-nek
void datum_keszit( DATUMPTR d, int ev, int ho int nap);
void novel ( DATUMPTR d );
void csokkent( DATUMPTR d );
```

Az adattípus és a deklarált függvények között van ugyan kapcsolat, hiszen minden egyes függvénynek van egy `Datum` típusú struktúra-mutató paramétere, viszont készíthetünk ezen kívül más függvényeket is, amelyeknek átadhatunk egy dátum struktúrát és amelyek például úgy növelik a napot, hogy nem veszik figyelembe a hónapok napjainak számát. Így nincs semmilyen garanciánk arra vonatkozólag, hogy a dátum típust megfelelően fogják kezelni. Ha viszont a struktúrához csak a fenti két művelet férne hozzá, akkor szavatolhatnánk a típus biztonságát. Az osztály bevezetésével ez a fajta probléma megoldódik. Java nyelvben a dátum típust egy osztállyal valósíthatnánk meg. Kezdetben úgy az osztályt, mint annak tagjait (adatok+műveletek) ellátjuk láthatósági módosítókkal. Egyelőre csak a nyilvános (`public`) és a privát (`private`) módosítókat használjuk. Minden nyilvános tag elérhető az osztályon kívüli kód számára és minden privát tag csak az osztályon belül lesz látható.

```
public class Datum{
   //Adatmezők
   private int ev;
   private int ho;
   private int nap;

   //Metódusok
```

```
   public Datum( int ev, int ho, int nap ){ ... }
   public void novel (){ ... }
   public void csokkent(){ ... }
   ...
}
```

A leglényegesebb különbség a struktúra és az osztály segítségével történő megvalósítás között az, hogy a struktúra adattagjaihoz közvetlenül hozzáférhetünk, míg az osztály védelemmel rendelkező adattagjai osztályon kívüli kód számára elérhetetlenek. A `Datum` osztály adatmezőit a létrehozás után, csak a `novel`, `csokkent` függvényekkel lehet módosítani.

4.2. Objektumok

Az objektum egy változó, amelynek típusa egy osztály. Úgy is mondhatnánk, hogy egy osztály típusú változó. Az objektumokat még példányoknak is nevezzük, vagy néha fordítva fogalmazva egy objektumot az osztály példányosításával hozunk létre. Például a primitív típusok esetén is mondhatjuk, hogy az 1, 9, -10, 123 az egész típus példányai. A típusok is kapcsolatban állhatnak egymással. Bizonyos típusok lehetnek konkrétak, mások absztraktak, más esetben egy adott típus lehet egy másik altípusa. Tekintsünk egy konkrét példát. Egy előadáson emberek vannak. Ezek az emberek feloszthatók előadóra és hallgatókra, tehát máris megkülönböztettünk egy EMBER típust és ennek két altípusát a TANÁR és a HALLGATÓ típust. Úgy a tanárok, mint a diákok lehetnek nők vagy férfiak, vagyis osztályozhatjuk őket nemük szerint is: NŐ és FÉRFI kategóriákba. Már ebből az egyszerű példából látszik, hogy egy előadáson résztvevő ember egyidőben több típussal rendelkezik. Például az előadó tanár EMBER, FÉRFI és TANÁR is egyidőben. Ezen kívül természetesen még sok osztályozási szempontot vezethetnénk be, bővítve egyazon példány típusait. De rakjunk rendet a gondolatainkban. Programjainkban a változók deklarációjakor egyetlen típust adhatunk meg. Mivel az objektum is egy változó, létrehozáskor ezt `egyetlen osztályból fogjuk példányosítani`. Viszont az osztálynak nem kötelező csak egyetlen típust megtestesíteni, egy osztály egyidőben több típust is testesíthet meg, tehát lehet több, mint egy típus. Erre majd a későbbiekben még visszatérünk. Az osztály össze-

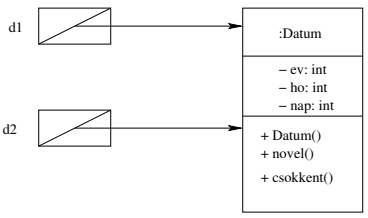
zárja az adatokat a műveletekkel, tehát pontosan meghatározza, hogy az osztály példányaival milyen műveleteket végezhetünk.

Tételezzük fel, hogy létrehoztunk két `Datum` típusú példányt. Ezek a példányok változók, tehát ábrázolásukhoz tárterületre van szükség. Minden egyes példánynak külön tárterülete lesz, amelynek mérete az adatmezők méreteinek összege lesz. Feltevődik a kérdés, hogy hogyan kapcsolódnak a műveletek az adatokhoz? Honnan tudják a műveletek, hogy melyik példányon hajtódnak végre? A `Datum` osztályban már megadtuk a műveletek szignatúráját és látható, hogy például a növelő függvénynek nincs paramétere. A válasz egyszerű: kell legyen ezeknek a műveleteknek egy láthatatlan, rejtett paraméterük, amelynek átadásáról a fordító gondoskodik és amely egyértelműen azonosítja, hogy mely objektumon hajtódik végre a kód.

Az objektumorientált programozásban az adat és a kód szerepe megváltozik. Amíg az eljárásközpontú és moduláris programozásban az adat passzív és a kód aktív szerepet játszik, addíg az OO programozásban az adat aktívvá válik és kezdeményezi a kód végrehajtását. Most már definiálhatnánk az objektumot úgy is, mint olyan változó, amely adatokat tárol és kérésre (üzenetre) tevékenységet képes elvégezni. A `Datum` osztály esetében minden egyes `Datum` típusú objektumnak három egész típusú adatmezője van és az objektum inicializálásán kívül két másik műveletet lehet velük végezni.

4.2.1. Objektum létrehozása

Objektum létrehozása egy példány elkészítését jelenti. Mi is szükséges egy példány elkészítéséhez és ki fogja ezt a tevékenységet elvégezni? Minden osztálynak rendelkeznie kell egy inicializáló művelettel, amelynek feladatköre az új példányok adatmezői kezdőértékeinek beállítása. Ezek a kezdőértékek átadhatók paraméterként, illetve ennek hiányában a mező típusának megfelelő alapértelmezett érték használható. Azt a függvényt, amely a kezdeti beállításokat elvégzi, `konstruktornak` nevezzük. Egy másik megválaszolandó kérdés, hogy ki végzi az objektumok helyfoglalását, illetve ez a helyfoglalás hogyan történik. C nyelvben is lehetőségünk volt arra, hogy futási időben tárterületeket kössünk le. Ezek a tárterületek a szabad tárból köthetők le, amit másképpen dinamikus memóriának is nevezünk. Bár ez a folyamat lassú, azzal az előnnyel rendelkezik, hogy



4.1. ábra. Két hivatkozás, egy objektum

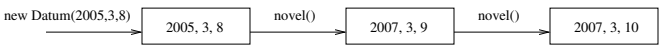
a futás közben kialakult konkrét helyzetnek megfelelően lefoglalhat kisebb vagy nagyobb memóriát, ezáltal takarékosabbá válhat programunk memóriagazdálkodása. Az olyan változókat, amelyek a szabad táron jönnek létre futási időben, dinamikus helyfoglalású változóknak nevezzük.

Java nyelvben minden objektum dinamikus helyfoglalású és ezt a `new` operátor végzi. A `new` operátor nemcsak a helyfoglalást végzi, hanem a konstruktorhívást is, így eredményül egy új példányt kapunk. Például a `new Datum(2005,3,8)` hívás eredményeként létrejött objektum ezen szöveg készítésének dátumát tárolja. A példányokat általában azért hozzuk létre, mert hivatkozni akarunk rájuk a későbbiekben, műveletet akarunk rajtuk végezni. Ehhez viszont szükség van egy azonosítóra, akárcsak az emberek világában, ahol a személyeknek nevet adnak, hogy tudják őket megszólítani. Az objektumokat hivatkozás (referencia) típusú változókkal azonosítjuk. Ez a hivatkozás típus működését illetően nagyon hasonlít a C nyelv mutatóira, csak egy picit gyengébb, vagyis kevesebbet enged meg, mint a C mutató, ezáltal biztosítva a memóriakezelés biztonságát.

```
Datum d1 = new Datum(2000,1,1);
Datum d2 = new Datum(2005,3,8);
Datum d3;
```

A fenti kódrészlet három hivatkozás típusú változót deklarál. Az első két változót el is látja kezdőértékkel, míg a harmadik egyelőre olyan, mint egy még nem inicializált mutató, nem tartalmazza egyetlen objektum címét sem. A Java nyelv szóhasználatával élve nem hivatkozik semmilyen objektumra.

A hivatkozás típusú változóknak ugyanúgy adunk kezdőértéket mint a mutatóknak:



4.2. ábra. Objektum állapota

1. helyfoglalással: `Datum d1 = new Datum( 2005, 3, 8);`

2. értékadással, ráállítjuk egy létező objektumra: `Datum d2 = d1;`

Ha a két értékadást egymás után végezzük akkor a `d1` és `d2` változók ugyanarra az objektumra fognak hivatkozni, amint a 4.1. ábra is mutatja.

Az objektum inicializálását a konstruktor végzi, amelynek a következő tulajdonságai vannak:

1. A konstruktor neve megegyezik az osztály nevével.

2. A konstruktornak nincs visszatérési értéke, deklaráláskor a nevét csak a láthatósági módosító előzheti meg.

3. Egy osztálynak akárhány konstruktora lehet. Általában a különböző inicializálási módokat különböző konstruktorokkal adjuk meg.

4. Ha egy osztályban nem definiálunk konstruktort akkor a fordító generál egyet, ezt `implicit vagy alapértelmezett` konstruktornak nevezzük. Az implicit konstruktornak nincs paramétere, így az adattagok az implicit kezdőértékükkel lesznek beállítva. Csak azt a paraméter nélküli konstruktort nevezzük implicitnek, amelyet a fordító hoz létre.

4.2.2. Objektum állapota

Az objektum állapotát az adatmezők pillanatnyi értékeinek halmaza határozza meg. Egy objektumot a `new` operátorral hozunk létre. Létrehozás után, az objetumon elvégzett műveletek hatására, az objektum különböző állapotokon megy keresztül, egészen addig, amíg meg nem semmisül. Ezt a folyamatot szemlélteti a 4.2. ábra.

4.2.3. Üzenetek

Az objektumot úgy lehet tevékenységre rábírni, hogy üzenetet küldünk neki. Például, ha az előző *d1* `Datum` típusú objektumot a holnapi napnak megfelelő dátumra akarjuk állítani, akkor a *`novel()`* üzenetet kell küldenünk neki. Az objektumot a hivatkozás típusú változón keresztül azonosítjuk, így az üzenetküldés szintaxisa a következő:

```
hivatkozas.metódusnév(aktuális paraméterek)
```

Ez konkrétan a `d1` hivatkozás típusú változón keresztül elérhető objektumra nézve a következő lenne:

```
d1.novel();
```

Természetesen, ha egy objektumra több hivatkozási lehetőségünk is van, akkor bármelyik hivatkozáson keresztül küldhetjük az üzenetet, ugyanazon az objektumon fog végrehajtódni a metódus. Amint már említettük, akárhány példányt készíthetünk egy osztályból, de amíg minden egyes példány rendelkezni fog a saját adatmezőivel, addig a metódusok csak egy példányban léteznek, ezt az objektumok közösen használják. Így a kódnak tudnia kell, hogy pontosan melyik példányon hajtódik végre, hiszen hozzá kell férnie a példány adatmezőihez (értékadás, lekérdezés). A metóduson belül a `this` hivatkozással férhetünk hozzá a példány adatmezőihez. Nézzük meg, hogyan adhatjuk meg a `Datum` osztály konstruktorát.

```
public class Datum{
   public Datum( int ev, int ho, int nap ){
      this.ev = ev;
      this.ho = ho;
      this.nap = nap;
   }
   ...
}
```

Általában az objektumokat hivatkozásokon keresztül érjük el. Megtörténhet, hogy egy objektumra csak egy üzenet erejéig van szükségünk, utána többet nem fogunk hivatkozni rá. Ilyen esetekben nem kell külön hivatkozás típusú változót bevezetnünk, hanem a példányra is meghívhatjuk a metódust. Ennek szintaxisa a következő:

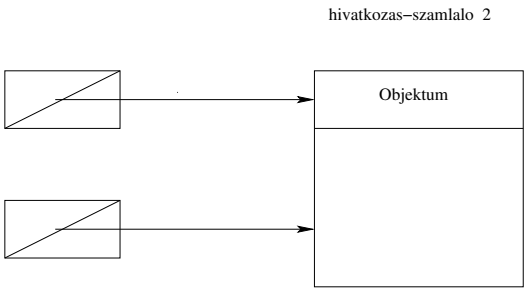
```
(new Osztaly( konstruktor paraméterek)).metódusnév( metódus paraméterek );
```

Konkrétan a `Datum` osztályra pedig

```
(new Datum(2005,3,8)).novel();
```

4.2.4. Objektumok megsemmisítése, automatikus szemétgyűjtés

A dinamikus memóriával való helyes és hatékony gazdálkodás az egyik legnehezebb probléma a C/C++ nyelvben megírt programok esetében. Azáltal, hogy a Java ezt a feladatot magára vállalta, sok bosszúságtól szabadítja meg főként a kevésbé tapasztalt programozót. Az automatikus szemétgyűjtés úgy valósul meg, hogy minden egyes objektumnak van egy hivatkozásszámlálója, és amikor ez nulla lesz, akkor az objektum felszabadítható (ld. 4.3. ábra) A felszabadítást egy külön programszál végzi (Garbage Collector), amelynek időzítését a Java futási környezet szabályozza. Más objektumorientált nyelvekben a konstruktornak van egy társ-metódusa, amelyet destruktornak nevezünk. A destruktor feladata a konstruktoréval ellentétes, a konstruktorban lekötött erőforrások felszabadítását végzi. A Java nyelvben a memória erőforrások felszabadítása megoldott, hiszen van automatikus szemétgyűjtés. A konstruktor köthet le azonban más erőforrásokat is, nemcsak memóriát, amelyek automatikusan nem szabadulnak fel, ezért ezeket explicit módon fel kell szabadítanunk. Ezt a célt szolgálja a `finalize()` metódus, amelynek definiálása nem kötelező az osztályokra nézve. Sajnos a `finalize()` metódus meghívásának pillanata nem definiált, így csak annyit tudunk, hogy mielőtt újra lefoglalódna az objektum tárterülete, ez a metódus meghívódik. Ennek a metódusnak a következő a fejléce:



4.3. ábra. Hivatkozás-számláló

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

A fenti fejléc két ismeretlen kulcsszót is tartalmaz. Az első a `protected` láthatósági módosító, amelyet a származtatásnál fogunk bevezetni, a második a `throws` kulcsszó, amely azt jelzi, hogy ezen metódus lefutása alatt kivételes állapot állhat elő, éspedig `Throwable` típusú kivétel keletkezhet. A `finalize` metódust csak akkor fogjuk használni, ha a konstruktor olyan erőforrásokat köt le, amelyeket fel kell szabadítanunk.

4.3. Osztályszintű (statikus) tagok

Az osztály metódusai kétfélék lehetnek. Léteznek `példánymetódusok` és léteznek `osztálymetódusok`. A példánymetódusok, amint nevük is mutatja a példányon dolgoznak, annak adatmezőin végeznek el valamilyen műveletet. A példánymetódus hívását üzenethívásnak is nevezzük, ilyenkor a metódus megkapja rejtett paraméterként a hivatkozást arra az objektumra, amelyen a tevékenységet el kell végeznie. A `Datum` osztályban minden metódus példánymetódus, hiszen mindenik feladata egy, vagy több adatmező módosítása.

Az osztálymetódusok nem a példányon dolgoznak. Ezekre azért van szükség, mert a Java nyelvben nem lehet függvényeket osztályon kívül deklarálni és néha szükségünk van olyan átalakító függvényekre, amelyek valamilyen típusból egy másik típusba alakítanak át. Ilyen lesz például a karakterfüzért egészszé ala-

kító függvény. Más esetben viszont szükségünk van olyan adatokra, amelyeket az osztályhoz tartozó objektumok közösen használnak- az ilyen adatokat az osztálymetódusok tudják majd kezelni. Az osztályszintű tagokat a `static` módosítóval látjuk el. A `static` módosítót általában a láthatósági módosító mögött helyezzük el.

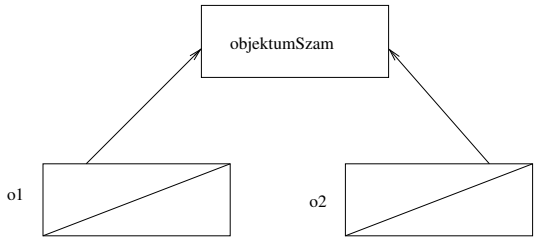
Először tekintsünk egy olyan példát, amelyben az osztály hasznos matematikai függvényeket tartalmaz. Ilyen esetben azért használjuk az osztályt, mert a Java nyelvben nem lehet osztályon kívül függvényt definiálni. Az osztály gyakorlatilag itt nem adatokat és metódusokat fog egységbezárni, hanem valamilyen szempontból rokon metódusokat, amelyek példányok létrehozása nélkül hívhatók. A következő programrészlet egy ilyen osztályra ad példát.

```
public class Matematika{
   public static int lnko( int a, int b ){
      while( a != b )
         if( a>b ) a = a-b;
         else
            b = b-a;
      return a;
   }

   public static boolean primSzam( int n ){
      if( n<=1 ) return false;
      if( n==2 ) return true;
      for( int i=2; i<=n/2; i++)
         if( n % i == 0 ) return false;
      return true;
   }
}
```

Ezt az osztályt példányosítás nélkül használjuk, hiszen az itt definiált metódusok hívása a következő szintaxis szerint történik: `osztálynév.metódusnév( aktuális paraméterlista )` Például, `Matematika.primSzam( n )`, ahol `n` egy egész típusú változó.

Tekintsük a következő példát. Készítsünk egy olyan osztályt, amely pontosan tudja, hogy hány példányt készítettek belőle. Természetesen szükségünk van egy példányszám adatmezőre, de ezt az adatmezőt nem deklarálhatjuk példány szintűnek, hiszen akkor minden példánynak lenne egy sajátja. Az sem lenne normális, ha ez nyilvános elérésű lenne, hiszen akkor bárki módosíthatná, így nem lenne biztonságban a tárolt érték. Legyen ez egy `objektumSzam` nevű mező amit osztályszintűnek fogunk deklarálni. Ki fogja ennek a változónak az értékét megváltoztatni és mikor, illetve hogyan lehet lekérdezni ezt a példányszámot? Az első kérdésre a válasz roppant egyszerű, hiszen annak a metódusnak kell ezt az értéket növelnie egy egységgel, amely csak egyszer hajtódik végre minden egyes objektumon, ez pedig a konstruktor. A másodikra sem lesz túl bonyolult a válasz, hiszen osztályszintű adatokat osztályszintű metódussal fogunk lekérdezni, így abban az esetben is helyes lesz az eredmény, ha még egyetlen objektumot sem hoztunk létre az osztályból.

```
public class ObjektumSzamlalo{
   private static int objektumSzam = 0;

   public ObjektumSzamlalo(){
      objektumSzam++;
   }

   public static int hanyObjektum(){
      return objektumSzam;
   }

   public static void main( String args[] ){
      System.out.println( ObjektumSzamlalo.hanyObjektum());
      ObjektumSzamlalo o1 = new ObjektumSzamlalo();
      System.out.println( o1.hanyObjektum());
      ObjektumSzamlalo o2 = new ObjektumSzamlalo();
      System.out.println( o2.hanyObjektum());
   }
}
```



4.4. ábra. Objektumszámláló

Az osztálymetódusok meghívhatók példány nélkül is, az osztály nevével megelőzve a metódusnevet, illetve hívhatóak példányon keresztül is, ha már létezik példánya az adott osztálynak. A kétféle módszer megkülönböztetése érdekében a továbiakban mi mindig osztálynevet fogunk használni az osztálymetódusok esetében. Továbbá megjegyezzük azt is, hogy amennyiben példányon keresztül hívjuk az osztálymetódust, az osztálymetódusnak úgy sem adódik át a hivatkozás a hívó példányra, ezért osztálymetódusokban nem használható a `this` referencia.

Az előző programban egy olyan osztályt készítettünk, amelytől bármelyik pillanatban lekérdezhető a létrehozott példányainak a száma. Ha a létező példányokat szeretnénk tárolni és nem a létrehozottak számát, akkor be kell vezetnünk a `finalize()` metódust, amely a példány fizikai eltűnésekor meghívódik és eggyel csökkenti a példányok számát. A javított változatot a következő program szemlélteti.

```
public class ObjektumSzamlalo{
   private static int objektumSzam = 0;

   public ObjektumSzamlalo(){
      objektumSzam++;
   }

   public static int hanyObjektum(){
      return objektumSzam;
   }
```

```
   protected void finalize(){
      objektumSzam--;
   }

   public static void main( String args[] ){
      System.out.println( ObjektumSzamlalo.hanyObjektum());
      ObjektumSzamlalo o1 = new ObjektumSzamlalo();
      System.out.println( ObjektumSzamlalo.hanyObjektum());
      ObjektumSzamlalo o2 = new ObjektumSzamlalo();
      System.out.println( ObjektumSzamlalo.hanyObjektum());
   }
}
```

4.4. Java program belépési és kilépési pontja

Hol kezdődik a Java program végrehajtása? A Java programok sok-sok osztályt tartalmazhatnak. Ahhoz, hogy egy osztály az indító szerepét tölthesse be, mindössze egy `main` osztálymetódussal kell rendelkeznie. Miután a Java futási környezet betölti a `main` metódust tartalmazó osztályt, elkezdi a `main` metódus végrehajtását. Tehát azt az osztályt, amely a `main` metódust tartalmazza, nem szükséges példányosítani. Nagyon gyakran egy külön osztályba helyezzük a `main` metódust, amely osztály ezen kívül semmit sem tartalmaz.

Java alkalmazásunk többféleképpen is befejeződhet. Egyik befejezési lehetőség az, ha a main metódus végrehajtása véget ér, minden utasítást végrehajtottunk belőle. Egy másik befejezési lehetőség az explicit kilépés a `System.exit(...)` hívásra. Az `exit()` a `System` osztály statikus metódusa, hatására a program futása befejeződik, beállítva a kilépési státuszt a paraméterben megadott értékre. A kilépési státuszban a program a befejeződés körülményeiről adhat tájékoztatást. Normális befejezéskor a kilépési státuszt 0-ra szokás állítani. A `main` metódus fejléce a következők egyike lehet:

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

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

1. `public`: Kötelező nyilvánosnak lennie, mert elérhető kell legyen bárhonnan.
2. `static`: Kötelező osztályszintűnek lennie, különben példány nélkül nem lehetne meghívni. A betöltés pillanatában viszont nincs egyetlen példányunk sem az osztályból.
3. `void`: Nincs visszatérési értéke.

Az egyetlen, ami változtatható, az az argumentumtömb azonosítója, amely a jelen esetben az `args` nevet viseli. A Java tömböket lehet úgy is deklarálni, hogy a zárójelpárt a típus után helyezzük: `String [] args`.

4.5. Paraméterek átadása

A Java nyelvben a paraméterátadás érték szerint történik, ami azt jelenti, hogy a hívott metódus nem változtathatja meg a parméter étékét. Ennek ellenére, ha paraméterként egy referenciát (hivatkozást) adunk át, akkor a hívott metódus megváltoztathatja a hivatkozott objektum tartalmát, a paraméterként átadott referencia változatlan marad (ugyanarra az objektumra hivatkozik).

Tekintsük a következő programot az előbbiek szemléltetésére:

```
class MyInteger{
   private int value = 0;

   public MyInteger( int value ){
      this.value = value;
   }
   public int getValue(){
      return value;
   }
   public void setValue( int value ){
      this.value = value;
  }
```

```
}

public class Main{
   public  static void incPrimitive( int i ){
      i++;
   }

   public  static void incReference( MyInteger ri ){
      ri.setValue( ri.getValue()+1);
   }

   public static void main( String args[] ){
      int i1 = 50;
      System.out.println( i1 );
      incPrimitive( i1 );
      System.out.println( i1 );

      MyInteger i2= new MyInteger( 100 );
      System.out.println( i2.getValue() );
      incReference( i2 );
      System.out.println( i2.getValue() );
   }
}
```

A `Main` osztály két metódust tartalmaz, a kétféle típusú paraméter átadásának szemléltetésére. Az első metódus az `incPrimitive`, amely egy primitív típusú paramétert ad át érték szerint. A metódus ugyan megnöveli a kapott értéket, de ezt a saját másolatán teszi, ami nem változtatja meg az eredeti változó értékét, jelen esetben a `main` metódusbeli `i1` értékét. A második metódus egy `MyInteger` típusú referenciát közvetít a hívott metódus fele. Itt gyakorlatilag a hívott másolatot kap a referenciáról, amely ugyanarra a `MyInteger` típusú objektumra fog hivatkozni, mint a `main` metódusbeli `i2` referencia. Ez a metódus megnöveli a hivatkozott objektum adattagját a `setValue` metódus segítségével. A hívott metódus ebben az esetben sem változtatta meg a paraméter értékét

hiszen hívás után is ugyanazt az objektumot fogja hivatkozni az `i2` referencia, de a hívás során megváltozott az objektum állapota.

4.6. Feladatok

1. Mi a konstruktor szerepe?

   a) Helyet foglal az objektumnak

   b) Inicializálja az objektumot

   c) Meghívja az osztály privát metódusait

   d) Elrejti az információt

2. Mely állítás hamis az alábbiak közül az osztályszintű (statikus) metódusokra vonatkozóan?

   a) A példányszintű adattagokon dolgozik

   b) Az osztályszintű (statikus) adattagokon dolgozik

   c) Használhatja a saját paramétereit

   d) Kötelező visszatérítési típust megadni deklarációjakor

3. Mely állítás igaz az osztályszintű (statikus) adattagokra vonatkozóan?

   a) Értékét csak a konstruktor változtathatja meg

   b) Kötelező ellátni a final módosítóval

   c) Csak private láthatóságú lehet

   d) Az osztály példányai közösen használják

4. Mely állítás hamis a példánymetódusokra vonatkozóan?

   a) Példányon keresztül hívódnak meg

   b) Hozzáférnek a példány privát adattagjaihoz

   c) Hozzáférnek a példány védett adattagjaihoz

   d) Osztálynév segítségével hívhatók

5. Mit jelent az egységbezárás?

   a) Különböző láthatóságú metódusok összezárását egy osztályba

   b) Különböző láthatóságú adattagok összezárását egy osztályba

   c) Az osztályszintű (statikus) és példányszintű metódusok összezárását egy osztályba

   d) Adatok és metódusok összezárását egy osztályba

6. Mit jelent a kód újrafelhasználása?

   a) Forráskód fordítása gépi kódra

   b) Gépikód fordítása forráskódra

   c) Bájtkód átalakítása forráskódra

   d) Egy már megírt kód felhasználása akár példányok létrehozására, akár osztály továbbfejlesztésére

7. Mit jelent az objektum állapota?

   a) Az objektum által hívható metódusok számát

   b) Az objektum méretét bájtban mérve

   c) Az objektum memóriacímét

   d) Az objektum attribútumainak pillanatnyi értékeinek összességét

8. Mely állítás igaz az információ elrejtésére vonatkozóan?

   a) Az információhoz senki sem férhet hozzá

   b) Csak jelszóval lehet elérni bizonyos adatokat

   c) Az objektum bizonyos információihoz kívülről nem lehet hozzáférni

   d) Az információhoz privát metódusok biztosítják a hozzáférést

9. Mely állítás igaz az interfészre vonatkozóan?

   a) Az osztály adattagjai meghatározzák annak interfészét

   b) Az osztály statikus metódusai meghatározzák annak interfészét

c) Az osztály nyilvános adattagjai meghatározzák annak interfészét

d) Az osztály privát metódusai meghatározzák annak interfészét

10. Hány referenciát és hány objektumot hoz létre a következő kódrészlet?

```
Alma a1 = new Alma();
Alma a2 = a1;
Alma a3 = new Alma();
```

a) két referencia, három objektum

b) három referencia, két objektum

c) három referencia, három objektum

d) két referencia, két objektum

11. Mely metódusban nem férünk hozzá a this referenciához?

a) A nyilvános példánymetódusokban

b) A privát példánymetódusokban

c) A védett példánymetódusokban

d) Az osztálymetódusokban (statikus metódusokban)

12. Mely kijelentés igaz a static módosítóra vonatkozóan?

a) kívülről és származtatott osztályból egyaránt el nem érhető osztályelemek bevezetésére szolgál

b) osztályszintű elemek bevezetésére szolgál

c) nem módosítható, konstans osztályelemek bevezetésére szolgál

13. Hány konstruktora lehet egy osztálynak?

a) egy

b) kettő

c) egy sem

d) nincs korlátozva

14. Hogyan kell kijavítani az `AClass` osztályt ahhoz, hogy a következő kódsor helyes legyen

```
AClass.amethod(123);

public class AClass(){
   protected void amethod( int i){...}
}
```

15. Mi lesz az eredménye a következő programnak?

```
class St {
   static int icount = 0;
   St() {
      icount++;
   }
   void countMembers() {
      System.out.print(" " + icount);
   }
   public static void main(String args[]) {
      St ob1=new St();
      St ob2=new St();
      St ob3=new St();
      ob1.countMembers();
      ob2.countMembers();
      ob3.countMembers();
   }
}
```

16. Mi lesz a következő program kimente?

```
public class X{
   public static void main( String args[] ){
      Osztaly o = new Osztaly();
      o.v = 100;
      o.m( o );
```

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

class Osztaly{
   public int v;
   public void m( Osztaly p ){
      p.v++;
   }
}
```

a) 0

b) 1

c) 100

d) 101

17. Mi lesz a következő program kimente?

```
public class MyClass{
   public static void main( String args[] ){
      double i = 10.1;
      Decrementer d = new Decrementer();
      d.dec( i );
      System.out.println( i );
   }
}

class Decrementer{
   public void dec( double what ){
      --what;
   }
}
```

a) 9

b) 10

c) 9.1

d) 10.1

18. Készítsen olyan osztályt, amelyből csak egy példány készíthető. (Egyke tervezési minta [4])

19. Elemezzétek az alábbi kódrészletet. Mi lesz az eredménye a programnak?

```
public class teszt8 {
   public static void main(String[] args) {
      MyMath m = new MyMath();
      m.E = 2.25;
      m.PI = 3.25;
      System.out.println("PI is " + m.PI);
      System.out.println("E is " + m.E);
   }
}

class MyMath {
   public static final double E = 2.7182818284590452354;
   public static final double PI = 3.14159265358979323846;
}
```

20. Mi lesz a következő program kimente?

```
public class MyClass{
   static String s;
   public static void main( String args[] ){
      System.out.println( "<<"+s+">>");
   }
}
```

a) «»

b) «null»

c) Fordítási hiba

21. Ha adottak a következő metódus deklarációk, válasszuk ki a helyes állításokat:

```
void fly(int distance)
int fly( int time, int speed ) { return time * speed; }

void fall( int time ){}
int fall( int distance ){ return distance; }

void glide( int time ){}
void Glide( int time ){}
```

a) Az első metódus-pár helyes fordítást valamint metódus túlterhelést eredményez.

b) A második metódus-pár helyes fordítást valamint metódus túlterhelést eredményez.

c) A harmadik metódus-pár helyes fordítást valamint metódus túlterhelést eredményez.

d) A második metódus-pár fordítási hibát eredményez.

5. FEJEZET

Tömbök, inicializálók, karakterfüzérek

5.1. Tömbök

A tömb egy homogén adatstruktúra: azonos típusú elemek gyűjteménye. A tömbelemek mindig folytonosan helyezkednek el a memóriában, így az elemek hivatkozása tömbnévvel és tömbön belüli pozícióval történik, akárcsak C nyelvben. Az elemek indexelése 0-tól kezdődik. A egyik nagy különbség a C tömb és a Java tömb között, az, hogy a Java tömb igazi objektum, amely tárolja a tömb méretét is, így a Java tömböket nem lehet túlindexelni. A másik különbség az, hogy a Java tömböknek van kezdőértékük, a deklaráció helyétől függetlenül. A kezdőérték a tömb típusának függvénye (numerikus-nulla; karakter-nullás kódú Unicode karakter , boolean - false, referencia - null) Tömböket alkothatunk primitív típusú értékekből, illetve hivatkozás típusú értékekből.

5.1.1. Egydimenziós tömbök

5.1.1.1. Primitív típusú elemekből alkotott egydimenziós tömb

A primitív típusú elemekből álló tömböt egy hivatkozás típusú változóval vezetjük be, amely számára majd a használat előtt helyet kell foglalnunk. A dek-

laráció kétféleképpen történhet: a zárójelek állhatnak a tömb azonosítója előtt, vagy ez után. Méretet tilos megadni, hiszen a Java nyelvben a tömbök igazi objektumok, amelyeknek dinamikusan kell végezni a helyfoglalást.

Minden tömb ismeri a saját méretét. Ezt a méretet egy `length` attribútumban tárolja, amely nyilvános elérésű. Természetesen ez a nyilvános elérésű adat állandó is, azaz egy tömbobjektum méretét nem lehet megváltoztatni. A tömb típusú hivatkozást kétféleképpen adhatjuk meg:

```
int x[];
int[] x;
```

Mindkét példa egy-egy egydimenziós tömböt deklarál. A következő példa a kétféle deklaráció közötti különbséget szemlélteti.

```
int x1[], x2;
int[] x1, x2;
```

Az első változatban csak az `x1` lesz tömb típusú, az `x2` egy egyszerű egész típusú változó, míg a második változatban úgy az `x1`, mint az `x2` tömb típusúak.

A tömbök valódi objektumok lesznek, amelyeket csak dinamikusan lehet lefoglalni, így a tömbelemek létrehozásához el kell végezni a helyfoglalást. Vajon helyfoglalás előtt lekérdezhető-e a tömb mérete? A válasz nemleges, hiszen a lekérdezéshez szükség van az objektumra, amely még ebben a pillanatban nem létezik. Ebben a pillanatban még csak egy hivatkozás típusú változó létezik. A helyfoglalást a `new` operátor végzi:

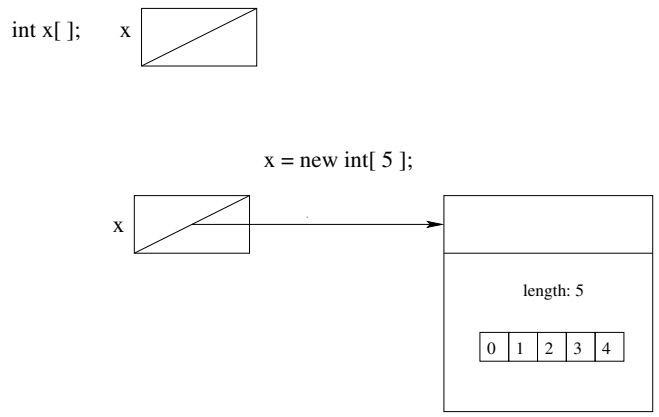
```
x = new int[5];
```

A helyfoglalás után már használható az `x` tömb. Lekérdezhetjük a méretét, beállíthatjuk elemei kezdőértékét.

```
System.out.println( x.length ); // Eredmény: 5
for( int i=0; i<x.length; i++ ) x[i]= i;
```

A fenti két lépést szemlélteti az 5.1. ábra.

Kijelenthetjük, hogy a tömbök olyan objektumok, amelyek nem változtatják méretüket. Egy tömb típusú hivatkozás, a C mutatóhoz hasonlóan, futás közben akárhány tömb típusú objektumra hivatkozhat. Például az előző `x` hivatkozásunkat most állítsuk rá egy 6 elemű tömbre:



5.1. ábra. Tömb típusú változó deklarálás után, majd helyfoglalás után

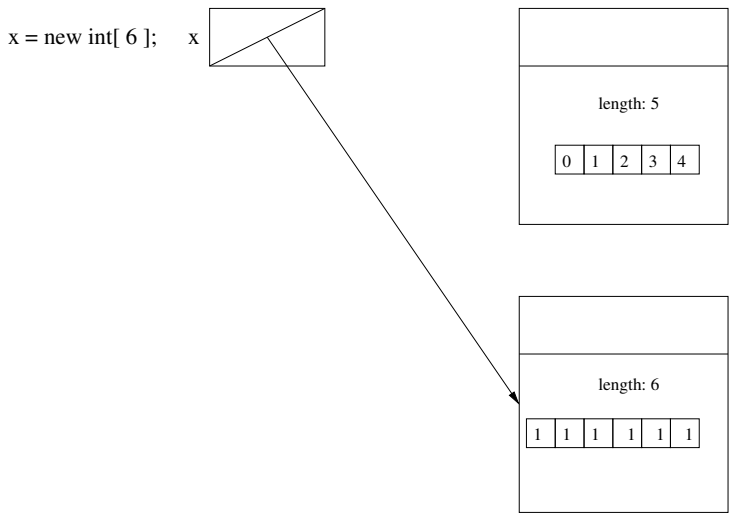
```
x= new int[6];
for( int i=0; i<x.length; i++ ) x[i] = 1;
```

Az új helyfoglalás hatására létrejön egy új tömb típusú objektum, amelyet az 5.2. ábra szemléltet. Az első 5 elemű tömb objektum elveszíti egyetlen hivatkozását, így felszabadítható állapotba kerül.

5.1.1.2. Hivatkozás típusú elemekből alkotott egydimenziós tömb

Készítsünk most `Szemely` típusú objektumokból tömböt. A személyeket név és születési év attribútumokkal fogjuk jellemezni, a metódusok közé felveszünk egy olyan metódust, amely lekérdezi az objektum állapotát és visszatéríti karakterfüzér formájában. A személynév ábrázolására szükségünk lesz a karakterfüzér típusra. A Java nyelv a karakterfüzérek ábrázolására a `String` és a `StringBuffer` osztályokat használja. Az előbbi osztály konstans karakterfüzérek ábrázolására szolgál, míg az utóbbi a változó méretű karakterfüzérek ábrázolására alkalmas.

```
public class Szemely{
   private String nev;
   private int szul_ev;

   public Szemely( String nev, int szul_ev ){
```



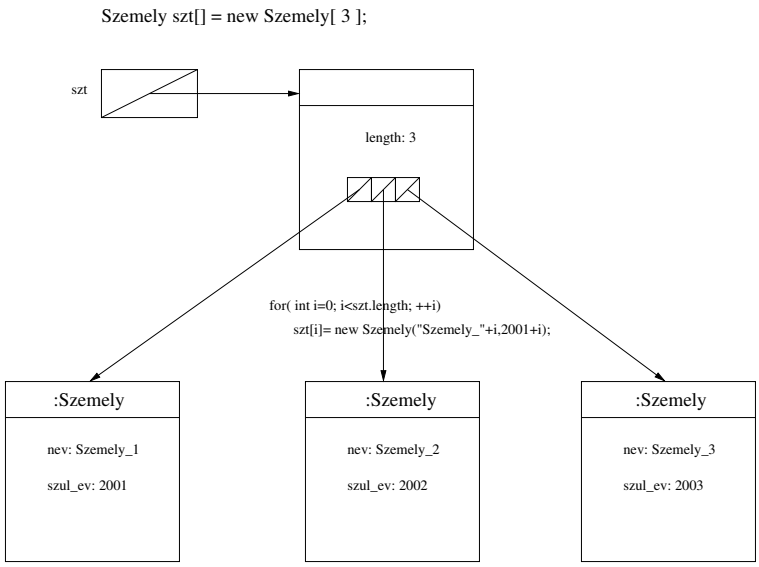
5.2. ábra. Tömb típusú változó az új helyfoglalás után

```
      this.nev = nev;
      this.szul_ev = szul_ev;
   }

   public String toString(){
      return nev+":" +szul_ev;
   }
}
```

Most pedig deklaráljunk egy három személy tárolására alkalmas tömböt, foglaljunk helyet ennek, majd töltsük fel `Szemely` típusú objektumokkal.

```
Szemely szt[] = new Szemely[3];
for( int i=0; i<szt.length; i++ )
   szt[i]= new Szemely("Szemely_"+i, 2001+i);
```

A fenti tömb három különböző Szemely típusú objektumot tartalmaz `Szemely_0, Szemely_1` illetve `Szemely_2` nevűeket. Az 5.3. ábra szemlélteti a létrehozási folyamatot.



5.3. ábra. Szemely típusú tömb létrehozása (1)

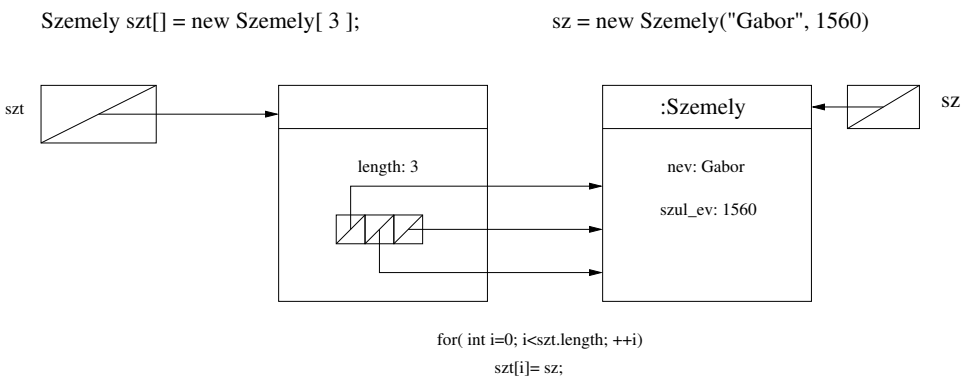
A tömb minden eleme egy `Szemely` típusú hivatkozás lesz. Az előző esetben minden egyes hivatkozást egy más-más `Szemely` objektummal inicializáltunk. Lehetőségünk van úgy tölteni fel a tömböt, hogy minden egyes hivatkozást ugyanazzal az objektummal inicializáljunk. Az 5.4. ábra ezt a fajta inicializálást szemlélteti.

```
Szemely szt[]= new Szemely[3];
Szemely sz = new Szemely("Gábor", 1560);
for( int i=0; i<szt.length; i++ )
   szt[i] = sz;
```

5.1.1.3. Tömbelemek kezdeti értékei

Ha a tömböt osztályszintű deklarációként adjuk meg, akkor a tömb típusú hivatkozás kezdeti értéke `null` lesz. Ha viszont lokális deklarációként adjuk meg, akkor definiálatlan kezdőértéke lesz.

```
class Pelda{
```



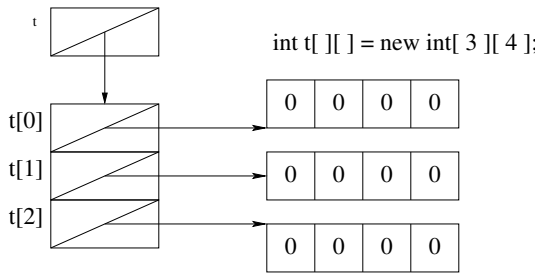
5.4. ábra. Szemely típusú tömb létrehozása (2)

```
   //osztályszintu deklaráció
   private int x[];

   public void f(){
      //Lokális deklaráció
      int y[];
      ...
   }
   ...
}
```

Az újonnan létrehozott tömb elemei (helyfoglalás után) alapértelmezés szerinti értékeket vesznek fel függetlenül attól, hogy a deklaráció osztályszintű, vagy lokális. Ha a tömb primitív típusú elemeket tartalmaz, akkor az elemtípustól függően a kezdőérték `0`, `\u0000` illetve `false`. Ha a tömb referencia típusú elemeket tartalmaz, akkor az elemek `null` értékűek lesznek.

Deklaráláskor `inicializáló blokkal` megadhatjuk a tömb elemeinek kezdeti értékeit:

```
int egeszek[]={1, 2, 3};
char maganhangzok[]={'a','e','i','o','u'};
boolean logikai[]={true, false};
```



5.5. ábra. Kétdimenziós tömb azonos méretű beágyazott tömbökkel

A fenti esetben nem szükséges a `new` operátort használni és a tömb mérete a kezdőértékek számával fog megegyezni. Mivel a tömb is hivatkozás típusú változó, ezért a tömb típusú hivatkozások (referenciák) értékadására ugyanazok a szabályok vonatkoznak, mint bármilyen más hivatkozás típusra. Vagyis két azonos hivatkozás típusú változó között lehetséges az értékadás, és ilyenkor a bal oldal a jobb oldali hivatkozás által meghatározott objektumra fog mutatni.

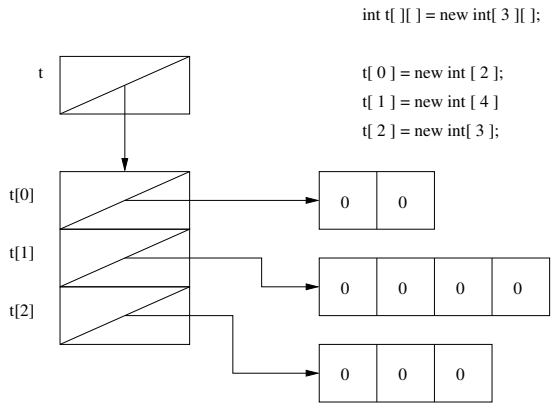
```
int x[]={1, 2, 3};
int y[]={4, 5, 6, 7, 8, 9, 10};
x = y;
```

A fenti kódrészlet lefutása után úgy az `x`, mint az `y` a hét elemet tartalmazó tömbre fog hivatkozni, vagyis ugyanazt a tömböt két név alatt is elérhetjük.

5.1.2. Kétdimenziós tömbök

A tömb elemei elvileg bármilyen típusúak lehetnek, akár tömbök is. Ha egydimenziós tömböket, mint elemeket, egy tömbbe foglalunk, rögtön kétdimenziós tömböt kapunk. Ez a séma tovább alkalmazható, így a háromdimenziós tömb olyan egydimenziós tömb melynek elemei kétdimenziós tömbök.

A kétdimenziós tömb kétféleképpen is létrehozható. Létrehozhatunk olyan kétdimenziós tömböt, amely azonos méretű egydimenziós tömbök gyűjteménye vagy létrehozhatunk olyat is, amelyben a beágyazott tömbök különböző méretűek lehetnek. Ezt a kétféle létrehozást szemléltetik az 5.5. és az 5.6. ábrák.



5.6. ábra. Kétdimenziós tömb különböző méretű beágyazott tömbökkel

5.2. Inicializálók

Inicializálók segítségével beállítható az osztályok és objektumok kezdeti állapota. A kezdeti állapot beállítható inicializáló kifejezéssel vagy inicializáló blokkal.

5.2.1. Inicializáló kifejezés

Inicializáló kifejezés segítségével egy változónak vagy konstansnak adhatunk kezdőértéket. A kifejezés kiértékelése az osztálytag esetében az osztály betöltésekor történik, míg példánytag esetében a példány létrehozásakor történik. Az inicializáló kifejezések a deklaráció sorrendjében kerülnek kiértékelésre és a kifejezésben csak a kifejezés előtti deklarációkra hivatkozhatunk. Osztálytagok nem hivatkozhatnak példánytagokra. A következő program szemlélteti az inicializáló kifejezések használatát.

```
class Pelda{
   private static int osztalyszintu_tag = 2000;
   private double peldanyszintu_tag = Math.random();
}
```

5.2.2. Inicializáló blokk

Egy osztályt az osztályinicializáló blokk, egy példányt pedig a példányinicializáló blokk fog beállítani. Az inicializáló blokk csak egyetlen egyszer fut le, osztály esetében az osztály betöltésekor, példány esetében pedig a példány létrehozásakor. Az osztályinicializáló blokkot az osztály szintű változók inicializálására szokás használni. A blokkot egy kapcsoszárójelpár határolja el, amelyet a `static` kulcsszó vezet be. A példányinicializáló blokk egyszerűen kapcsoszárójelek között adható meg és kezdőértéket ad a példányszintű tagoknak. A következő program szemlélteti mindkét fajta kezdőértékadást:

```
import java.io.*;

class Pelda{
   private static int osztaly_szintu_tag;
   private int peldany_szintu_tag_elso;
   private int peldany_szintu_tag_masodik;

   static{
      osztaly_szintu_tag = 2000;
   }

   {
      try{
        peldany_szintu_tag_elso = Integer.parseInt(
            System.console().readLine()
        );
      }
      catch( IOError e ){}
      peldany_szintu_tag_masodik = 2*peldany_szintu_tag_elso;
   }
}
```

Az inicializálások sorrendje a következő:

Osztályszintű adatok:

– először felveszik az alapértelmezés szerinti kezdőértékeket

– kiértékelődnek az osztályinicializáló kifejezések, majd lefutnak az osztályinicializáló blokkok a deklarációk sorrendjében

`Objektumszintű adatok:`

– először felveszik az alapértelmezés szerinti kezdőértékeket

– kiértékelődnek az inicializáló kifejezések, majd lefutnak a példányinicializáló blokkok a deklarációk sorrendjében

– lefutnak a konstruktorok

5.3. Karakterfüzérek

A Java a karakerfüzérek kezelésére három alapvető osztályt biztosít: `String`, `StringBuffer`, `StringTokenizer`.

5.3.1. A String osztály

A `String` osztály konstans karakterláncok tárolására alkalmas osztály. Az objektum a szöveget unikód (UNICODE) karaketerek sorozataként tárolja. A karakterfüzér karaktereit sorszámuk (indexük) alapján is elérhetjük. Az elemek sorszámozása nullától történik.

`Létrehozás`

A létrehozás ugyanúgy a `new` operátorral történik, mint minden objektum esetén.

```
String fuzer = new String("Ez egy füzér");
```

A karakterfüzér létrehozható konstans szöveg literál értékadásával is.

```
String fuzer = "Ez egy füzér";
```

A két értékadás között az a különbség, hogy az utóbbi esetében a Java rendszer optimalizációt végez úgy, hogy a megegyező karakterfüzéreket csak egyszer tárolja.

```
String ujfuzer ="Ez egy füzér";
```

Vagyis, ha az előző füzér után létrehozunk még egy másikat is ugyanazzal a kezdőértékkel, akkor új karakterfüzér objektum nem fog keletkezni, az `ujfuzer` hivatkozás is az előzőleg létrejött karakterfüzért fogja jelenteni.

Metódusok:

Mivel a füzér tartalma nem változtatható, ezért a metódusok vagy információt adnak vissza az objektumról, vagy egy új füzért állítanak elő. A leggyakrabban használt metódusok a következők:

Adott pozición levő karakter

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

Összehasonlítások

```
int compareTo(String str );
int compareToIgnoreCase( String str );
```

Összefűzés

```
String concat( String str );
```

Adott karakter vagy részfüzér kezdőpozíciója

```
int indexOf( int ch );
int indexOf( int ch, int fromIndex );
int indexOf( String str );
int indexOf( String str, int fromIndex );
```

Füzér hossza

```
int length();
```

Részfüzérek illetve más manipulált füzérek előállítása

```
String replace( char oldchar, char newchar );
String substring( int beginIndex );
String substring( int beginIndex, int endIndex );
String toLowerCase();
String toUpperCase();
String trim();
```

5.3.2. A StringBuffer osztály

Olyan karakterfüzérek esetén, amelyek tartalma változhat futás közben, a Java a `StringBuffer` osztályt használja. A `StringBuffer` osztály olyan példányok típusa, amelyek szövege manipulálható. A `StringBuffer` típusú objektum által tartalmazott szöveg aktuális hossza futás közben változhat (karaktereket tartalmazó dinamikus tömb).

Ezt két példányszintű adattaggal valósítják meg:

- kapacitás (capacity), milyen hosszú szöveg tárolására alkalmas a példány. Ezt a lefoglalt memória mérete határozza meg.
- aktuális hossz (length), az éppen tárolt szöveg hossza.

Műveletek:

Létrehozás

```
StringBuffer(); tartalma az üres lánc, kapacitása 16 karakter
StringBuffer(int size); tartalma az üres lánc, kapacitása: size
StringBuffer(String str); tartalma a megadott lánc, kapacitása
a lánc hossza + 16
```

Kapacitás, hossz, index, tartalom

```
int capacity();
int length();
void setLength( int newlength );
```

A `void setLength( int newlength )`, a hossz modosítását végzi. Szükség esetén növeli a kapacitást, ha a hosszat növeltük, az új pozíciókat `\u0000` karakterekkel fogja kitölteni

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

Bővítés

`StringBuffer append( Type value )`, bármilyen primitív és nem primitív típusú érték hozzáfűzését teszi lehetővé

`StringBuffer insert ( int offset, Type value )`, bármilyen primitív és nem primitív típusú érték beszúrását teszi lehetővé

Törlés

```
StringBuffer deleteCharAt( int index );
StringBuffer delete( int start, int end );
```

Helyettesítés és megfordítás

```
StringBuffer replace( int start, int end, String str );
StringBuffer reverse();
```

5.3.3. A StringTokenizer osztály

A `StringTokenizer` osztály lehetővé teszi egy szöveg egységekre bontását. Ilyen egységek lehetnek például: bekezdések, mondatok, szavak. Azt, hogy milyen egységekre akarjuk bontani, egy vagy több elválasztó karakter segítségével adhatjuk meg. Ha például az elválasztó karaktereknek a . ? ! karaktereket választjuk, akkor ez mondatokra való bontást tesz lehetővé. Alapértelmezésben az elválasztó karakterek a szóköz, TAB, CR, LF és FF karakterek. Az alapértelmezés szerinti elválasztókkal a mondat szavakra bomlik.

Konstruktorok:

```
StringTokenizer(String str, String delim)
```

```
StringTokenizer(String str)
```

Metódusok:

`boolean hasMoreTokens()`

Igazat térít vissza, ha van még feldolgozatlan egység.

`String nextToken()`

Visszaadja a soron következő egységet. Hívását mindig egy `boolean hasMoreTokens()` hívása kell megelőzze, különben nem létező következő egység esetében kivétel fog kiváltódni.

`String nextToken( String delim )`

Átvált a paraméterben megadott új elválasztó halmazra, majd ennek megfelelően visszaadja a következő egységet. A továbbiakban az új elválasztó halmaz lesz érvényben.

`int countTokens()`

Visszaadja, hogy hány rész van még a szövegben.

A következő program a standard bemenetről beolvasott sort felbontja egységekre és kiírja ezeket a standard kimenetre.

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

public class st{
   public static void main( String args[] ) throws Exception{
      String line = null;
      BufferedReader in = new BufferedReader( new InputStreamReader(System.in));
      line = in.readLine();
      StringTokenizer st = new StringTokenizer( line );
      while( st.hasMoreTokens() )
         System.out.println( st.nextToken() );
   }
}
```

5.4. Feladatok

1. Illusztrálja a következő kódrészlet végrehajtását lépésenként, lerajzolva a referenciákat és az objektumokat.

```
String s1 = "lma";
String s2 = "a";
String s3 = s1;
String s4 = s2 + s1;
```

2. Illusztrálja a következő kódrészlet végrehajtását lépésenként, lerajzolva a referenciákat, primitív típusú változókat és az objektumokat.

```
int[] a = {1, 2, 3};
int[] b = new int[ 2 ];
int[] c = a;
int x = c[ 0 ];
```

3. Tekintsük a következő statikus metódust, amelyet egy `Arrays` nevű osztályban helyezünk el:

```
public class Arrays{
   public static void foo( int[] z ){
      for( int i=1; i<z.length; ++i )
         z[ i ] = z[ i-1 ] + z[ i ];
   }
}
```

   Szemléltesse a következő kódrészletben szereplő tömb állapotát a metódus hívása előtt és után.

```
int[] t = {1, 2,3 };
Arrays.foo( t );
```

4. Adott a következő kódsor:

```
int [] x = new int[ 25 ];
```

A végrehajtódás után, a következő kijelentések közül melyek igazak?

a) x[24] értéke 0

b) x[24] nem meghatározott

c) x[25] értéke 0

d) x[0] értéke null

e) x.length értéke 25

5. Készítsen egy Matrix osztályt, amely tartalmazza a szabványos matrix műveleteket. Minden egyes metódust úgy tervezzen meg osztályszintűnek vagy példányszintűnek, hogy annak használata a lehető legtermészetesebb legyen.

6. Készítsen egy Arrays osztályt, amely tartalmazza az olyan klasszikus egydimenziós tömbműveleteket, mint:

   – szekvenciális keresés

   – bináris keresés

   – buborékrendezés

   – gyorsrendezés

   Osztály-, vagy példányszintű metódusoknak érdemes ezeket deklarálni?

7. Készítsen egy Polinom osztályt, amelynek célja valós együtthatós polinomok ábrázolása és a legfontosabb polinom műveletek implementációja.

8. Mely deklarációk helyesek?

   a) int i[5][];

   b) int i[][];

   c) int []i[];

   d) int i[5][5];

   e) int[][] a;

9. Adott a következő deklaráció:

```
String s = new String("xyz");
```

Válasszuk ki a helyes kifejezéseket!

a) s = 2 * s;

b) int i = s[0];

c) s = s + s;

d) s = s » 2;

10. Mi történik a következő program futtatása esetén?

```
public class MyClass{
        public static void main( String args[] ){
                int i = 0;
                int [] a = {3, 6};
                a[ i ] = i = 9;
                System.out.println( i+" "+a[0]+" "+a[1]);
        }
}
```

a) ArrayIndexOutOfBounds kivétel keletkezik

b) Kimenet: 9 9 6

c) Kimenet: 9 0 6

d) Kimenet: 9 3 6

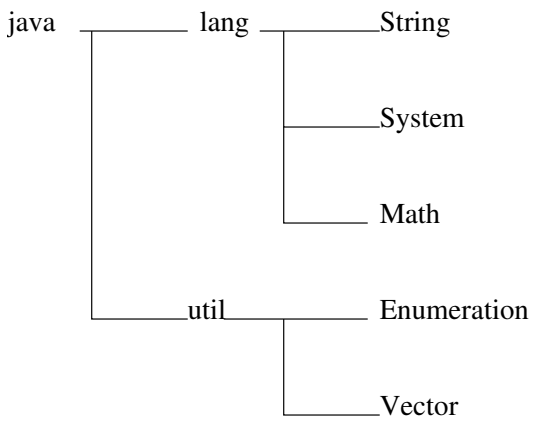
e) Kimenet: 9 3 9

6. FEJEZET

Java csomagok

Amíg az osztályok egységbe zárják az adatokat a rájuk vonatkozó metódusokkal, addig a csomagok az osztályokat zárják egységbe. C nyelven írt programjaink esetében gyakran találkoztunk a névütközés problémájával. Adott fordítási egységen belül, nem lehet két azonos nevű függvényünk, illetve nem lehet két azonos nevű típusunk. Saját fejlesztésű szoftver esetében ez nem is jelent olyan nagy gondot, de mi történik, ha két különböző forrásból származó függvénykönyvtárat használunk egyidejűleg, és mindkét függvénykönyvtár definiál egy ugyanolyan nevű függvényt? Ezt a problémát a C++ már kijavította a névterek bevezetésével. A Java egy modern tervezésű nyelv, így már a tervezés pillanatában megoldották a névütközések problémáját, bevezetve egy hierarchikus névtér szerkezetet. A Java nyelv csomagokkal dolgozik, amelyek egy fa szerkezetű hierarchiát alkotnak. A csomagok valamilyen módon összefüggő elemeket tartalmaznak. Minden csomag egy önálló névteret vezet be, amivel egyedivé teszi a benne definiált típusok neveit. A csomagok között alárendelő viszony is létezhet, így egy csomag tetszőleges számú alcsomagot tartalmazhat.

A Java csomagokat kétféle módon lehet tárolni, fájlrendszerben vagy adatbázisban. A JDK fájlrendszerben tárolja, míg az IBM VisualAge fejlesztői környezet adatbázisban. Mi az előbbit ismertetjük. A csomagok fájlrenszerbeli tárolása a hagyományosabb módszer. Ebben az esetben a kialakítandó könyvtárszerkezet követi a csomagok hierarchiáját, minden egyes alkönyvtár egy legfelső szintű csomagot reprezentál, míg az alcsomagok az adott alkönyvtár alkönyvtáraiban



6.1. ábra. Java csomagok és elhelyezésük a fájlrendszerben

helyezkednek el. Az 6.1. ábra szemlélteti a JDK csomagok leképzésének ezt a módját. A nagybetűvel kezdődő nevek osztály, illetve interfész nevek: String, System, Math, Enumeration, Vector. A csomagnevek hagyományosan kisbetűvel kezdődnek.

6.1. Csomagdeklaráció

A Java nyelvben a fordítási egység a Java állomány, amely több osztálydefiníciót is tartalmazhat. Fordításkor annyi bájtkód (bytecode - .class) állomány keletkezik, ahány osztálydefiníciót tartamaz az állomány.

Minden fordítási egység elején megadható, hogy milyen csomaghoz tartozik az adott egység. Ennek megadása a `package` kulcsszóval történik. Ez csak a fordítási egység elején lehet, más deklarációk után már nem. A Java csomag nem zárt, bármikor új elemekkel bővíthető. Természetesen csak a saját csomagjainkat bővíthetjük, a Java osztálykönyvtárhoz tartozó csomagok nem bővíthetőek.

Tekintsük a következő példát csomagdefinícióra.

```
package games;
public class NoughtCrosses{ ... }
```

Amikor a Java fordító a fenti kódot fordítja, az aktuális könyvtárban létrehoz egy `games` nevű könyvtárat és ebben helyezi el a `NoughtCrosses.class` állományt. Ha a fordítási egység elején elmarad a csomagdeklaráció, akkor a fordítási egység egy névtelen csomaghoz fog tartozni.

6.2. Importdeklaráció

A fordítási egységben a csomagdeklarációt követhetik az importdeklarációk. Egy importdeklaráció lehetővé teszi a más csomagokban deklarált nyilvános hozzáférésű típusok egyszerű, rövid névvel történő elérését. Az importdeklaráció használata nem kötelező, előnye viszont a típusnevek egyszerű használata. Ha nem használnánk az importdeklarációkat, akkor minden típusra a kiterjesztett névvel kellene hivatkoznunk. A Java rendszer a `java.lang` csomag összes nyilvános típusát automatikusan importálja. Az importdeklarációkban használhatjuk a * karaktert, ez az összes nyilvános típust jelenti az adott csomagból, nem jelenti viszont ezek alcsomagjait. Nézzünk egy példát importdeklarációra:

```
import java.util.Vector;

public class Pelda{
   public static void main( String args[] ){
      Vector v = new Vector();
...
   }
}
```

Az előző példában a `java.util.Vector` típust úgy használtuk, hogy először elhelyeztünk egy impordeklarációt a fordítási egység elejére, így a típusnévre való hivatkozás egyszerűen történt a `Vector` használatával. A következő kódrészlet ugyanazt tartalmazza, csak importdeklaráció nélkül:

```
public class Pelda{
   public static void main( String args[] ){
      java.util.Vector v = new java.util.Vector();
...
```

```
   }
}
```

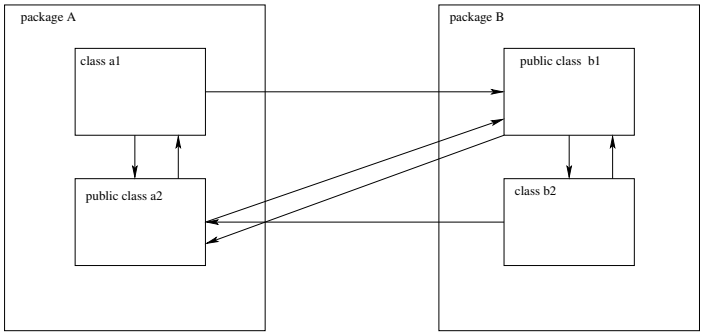
6.3. Statikus importdeklaráció

Ezt a lehetőséget az 1.5-ös változattól kezdődően vehetjük igénybe és az osztályok azon statikus tagjai importálhatók, amelyek nem privát elérésűek. Tulajdonképpen kényelmi szempontból vezették be ezt a lehetőséget, leegyszerűsítve a hivatkozás módját az így importált statikus tagokra. Az így importált statikus tagokra egyszerűen a nevükkel hivatkozhatunk, ahogyan ezt a következő program is mutatja:

```
import static java.lang.Math.PI;
import static java.lang.Math.E;
import static java.lang.Math.random;

public class Main1{
   public static void main( String args[] ){
      System.out.println( "Pi= "+PI+" E= " +E+"RND= "+random());
   }
}
```

6.4. Típusok láthatósága

Típusdeklarációnak tekintjük az osztály és az interfész deklarációkat. Minden típusnévnek megadható a láthatósága. Ha nem használunk semmilyen módosítót, akkor a típus láthatósága csomagszintű lesz. A `public` kulcsszó használatával a típus bárhol elérhető lesz, vagyis más csomagokban is. A típusok láthatóságát az 6.2. ábra szemlélteti.



6.2. ábra. Típusok láthatósága csomagon belül és azon kívül

7. FEJEZET

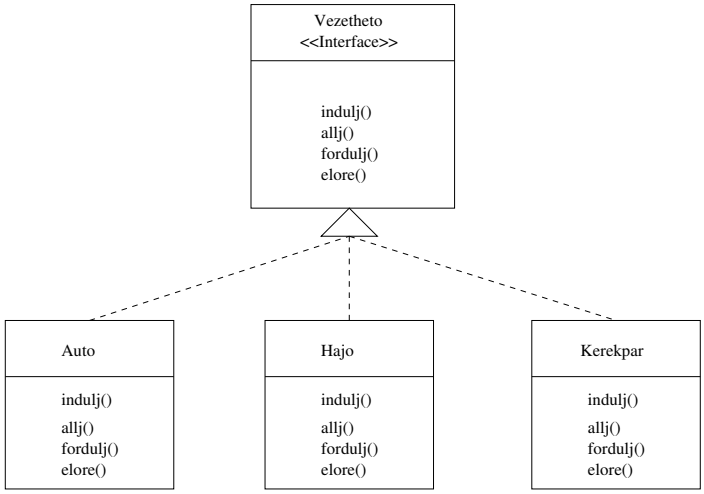
Interfészek

Amíg az osztályok általában konkrét típusokat határoznak meg, addig az interfészek absztrakt adattípusok leírására szolgálnak. Egy absztrakt adattípus (Abstract Data Type) csak a műveleteket határozza meg, semmiféle információt nem szolgáltat a típushoz tartozó adatok ábrázolására vonatkozóan, illetve a műveletek megvalósítására vonatkozóan, a konkrét típus pedig mindkettőt megadja. A Java nyelvben lehetőség van absztrakt osztályok definíciójára is, amelyek úgy konkrét mint absztrakt tagokat is tartalmazhatnak.

7.1. Interfészek deklarációja és implementálása

Tekintsük a jármű típust. A jármű egy gyűjtőnév, ide tartozik minden olyan alkamatosság, amely valamilyen módszerrel vezethető, irányítható. Tehát nem konkrét típusról van szó, hanem egy absztraktról. A vezethető tulajdonságot most megpróbáljuk leírni műveletek segítségével. Tekintsük csak a legalapvetőbb műveleteket: indulás, megállás, fordulás és az előre műveleteket. A fordulás lehetővé teszi az irányváltást, míg az előre az aktuális irányban haladást teszi lehetővé. Természetesen ezen műveleteket minden konkrét jármű biztosítja, így azt is mondhatnánk, hogy minden jármű ilyen típusú, vagyis vezethető.

A 7.1. ábra szemlélteti a `Vezetheto` interfészt és az interfészt implementáló konkrét osztályokat. A Java nyelvben az interfészt az `interface` kulcsszó vezeti



7.1. ábra. Interfészt implementáló osztályok

be. Az interfész csak deklarációkat tartalmaz. Ezek a deklarációk lehetnek adat és metódus deklarációk. Az interfész minden tagja nyilvános, így nem szüséges a `public` módosítót használni. Az adattagok pedig konstansok, ezek esetében sem szükséges a `final` és a `static` módosítókat használni. A `Vezetheto` interfészt a következő program szemlélteti:

```
public interface Vezetheto{
   void indulj();
   void allj();
   void fordulj( int szog );
   void elore( int hossz );
}
```

A fenti interfészre építhetjük a különböző jármű osztályokat, mint például hajó, kerékpár, autó. Minden konkrét járműnek biztosítania kell az interfészben megadott műveleteket, de mindenik a sajátosságainak megfelelően fogja ezt biztosítani. Nem egyformán indítjuk a kerékpárt és az autót, viszont mindkettőnek jól meghatározott indítási mechanizmusa van. A következő program egy implementációs vázat szemléltet:

```
public class Auto implements Vezetheto{
   //autó indításához szükséges műveletek
   public void indulj(){...}

   //autó megállításához szükséges műveletek
   public void allj(){...}

   //autó fordításához szükséges műveletek
   public void fordulj( int szog ){...}

   //autó előrehaladásához szükséges műveletek
   public void elore( int hossz ){...}
   ...
}
```

Amint a fenti példa is mutatja, az interfészt implementáló osztálynak meg kell valósítania minden egyes műveletet az interfészből. Mi az előnye az interfészeknek? Az előző példát megadhattuk volna úgy is, hogy három osztályt definiálunk és nem foglalkozunk a közös rész kiemelésével. A közös rész kiemelése (interfész) esetén minden jármű besorolható a Vezethető kategóriába és ezáltal különböző járművek elhelyezhetővé válnak ugyanabban a tárolóban. Például deklaráljunk egy háromelemű tömböt, amelybe belehelyezünk egy hajót, egy autót és egy kerékpárt.

```
Vezetheto tv[] = new Vezetheto[ 3 ];
tv[ 0 ] = new Hajo();
tv[ 1 ] = new Auto();
tv[ 2 ] = new Kerekpar();
```

Mivel minden jármű vezethető, ugyanazt a viselkedésmódot implementálja, ez gyakorlatilag azt jelenti, hogy ugyanazokra a parancsokra hallgat. Így lehetővé válik, hogy egységesen kezeljük a járműveket. A következő kódrészlet elindítja a tömbben elhelyezett összes járművet:

```
for( int i=0; i< tv.length; ++i )
   tv[ i ].indulj();
```

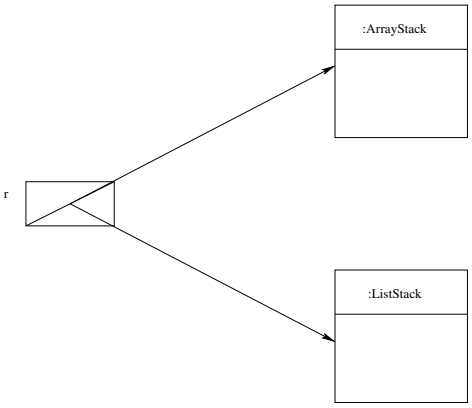
7.2. Polimorfizmus

Tekintsünk egy másik példát az interfészekre. A verem adatstruktúra az egyik legegyszerűbb adatstruktúra, amely az "utolsónak be, elsőnek ki" működési elvet követi. Ez a működési elv két művelettel megvalósítható, egy `push` és egy `pop` művelettel. Természetesen a vermet sokféleképpen lehet implementálni, de attól verem a verem, hogy betart egy adott működési elvet és nem attól, hogy hogyan ábrázolja a verembe kerülő adatokat. Nevezzük ezt a viselkedésmódot vereminterfésznek (StackInterface). Az egyszerűség kedvéért, most nem törekszünk típusfüggetlenségre, így rögzítjük az verembe helyezhető elemek típusát karakternek.

```
public interface StackInterface{
   void push( char c );
   char pop();
}
```

A fenti interfészt pedig implementálhatjuk különböző adatábrázolással. Helyezhetjük az adatokat egy egyszerű, rögzített számú elemet tartalmazó tömbbe, vagy használhatunk dinamikus tömböt. Ha nem a folytonos ábrázolás mellett döntünk, akkor az elemeket akár láncolt listában is tárolhatjuk. A lényeges dolog az, hogy a műveletek a működési elvnek megfelelően kezeljék az adatokat. Az interfész biztosítja az átjárhatóságot, vagyis ha időközben kicseréljük az adatstruktúra implementációját, ez nem változtatja meg az azokat használó programjainkat.

```
public class ArrayStack implements StackInterface{
   public void push( char c ){ ... }
   public char pop(){ ... }
}


public class ListStack implements StackInterface{
   public void push( char c ){ ... }
   public char pop(){ ... }
   ...
}
```



7.2. ábra. Interfész típusú hivatkozás

Az interfészek típusokat határoznak meg és hivatkozás típusú változók deklarációjára használjuk. Példányt viszont csak osztályból lehet létrehozni, interfészből nem. Tekintsük a következő példát az interfészek használatára:

```
1. StackInterface r= new ArrayStack( 10 );
2. r.push( 'a'); r.pop();
3. r = new ListStack();
4. r.push('x'); r.pop();
```

Az első sorban deklaráltunk egy `StackInterface` típusú hivatkozást, amelynek kezdőértéke egy `ArrayStack` típusú objektum. A második sorban műveleteket végeztünk az objektummal, az `r` hivatkozás segítségével. A harmadik sor új értéket ad az `r` hivatkozásnak, most egy `ListStack` típusú objektumot fogunk általa kezelni. A fenti példa egy szép példánya a `többalakúságnak` vagy `polimorfizmusnak`, amikor ugyanazon változón keresztül több típusú objektum is elérhető.

7.3. Üzenetküldés

Egy objektumorientált program egy objektumhalmaz, amelyből az objektumok kommunikálnak egymással egy jól meghatározott tevékenység elvégzése

7.3. ábra. Üzenetküldés

érdekében. Ez a kommunikáció üzenetküldéssel valósul meg. Tételezzük fel, hogy van két különböző típusú objektumunk, legyen az egyik objektum egy jármű, a másik pedig legyen egy járműparkot irányító objektum. A járműpark irányító objektumnak természetesen ismernie kell az összes járművet a járműparkból, különben nem tudná ellátni a feladatát. A járműpark irányítója a beérkező járműveket elhelyezi a járműparkban, amennyiben van szabad hely, illetve a kimenő járműveket kivezeti a járműparkból. Az elhelyezési stratégia most minket nem érdekel. A járművek lesznek a passzív objektumok, ők elszenvedik a járműpark-irányító objektum parancsait.

A Járműpark-irányító objektum a következő algoritmust használhatná a vezérlésre:

```
ismételd a végtelenségig
   várakozás egy kérésre
   Jarmu jarmu = jármű kiválasztása
   jarmu.indulj();
   ...
ismétlés vége
```

Az üzenetküldéshez két fél szükséges, egy küldő és egy fogadó fél. A küldő félnek ismernie kell a fogadó objektumot, vagyis lehetősége kell legyen ennek megszólítására. Úgy a küldő objektumnak, mint a fogadónak van egy típusa, ezeknek a típusoknak biztosítaniuk kell a kommunikáció lehetőségét. A mi esetünkben a `Jarmupark-irányito` osztálynak kell ismernie a járművet, szükség van egy `Jarmu` típusú referenciára a vezérléshez. Ez a `Jarmu` típusú referencia mindig azt az objektumot hivatkozza, amellyel éppen műveletet kell végeznie.

Hasonlóképpen tekinthetnénk tanár és diák objektumok közötti kapcsolatokat is. Ideális esetben itt egy kétirányú kapcsolatról van szó, vagyis úgy a tanár, mint a diák küldhet üzenetet a másiknak.

7.4. Feladatok

1. Adott két interfész és osztályok, amelyek implementálják vagy az egyik, vagy a másik, vagy pedig minkét interfészt.

```
public interface Indithato{
   void indulj();
}

public interface Megallithato{
   void allj();
}

public class A implements Indithato{...}

public class B implements Indithato, Megallithato{...}

public class C implements Megallithato{...}
```

   Adott a következő két tömb:

```
Indithato[] t1 = new Indithato[ 10 ];
Megallithato[] t2 = new Megallithato[ 10 ];
```

   Feltételezzük, hogy a tömböket feltöltjük a megfelelő példányokkal.

   - Mely osztály példányait helyezhetjük a t1 tömbbe?
   - Mely osztály példányait helyezhetjük a t2 tömbbe?
   - Készítsen kódrészletet, amely elindítja a t1 tömbben levő összes példányt.

– Készítsen kódrészletet, amely megállítja a t1 tömbben levő összes megállítható példányt. (Használja az `instanceof` operátort a példány dinamikus típusának lekérdezéséhez)

2. Adottak a következő Java típusok:

```
public interface HalmazInterfesz{
   void ujElemFelvetele( Object o);
   void elemTorlese( Object o);
   int elemSzam();
}

public class TombHalmaz implements HalmazInterfesz{
   //Adattagok
   ...
   public void ujElemFelvetele( Object o){...}
   public void elemTorlese( Object o){...}
   public int elemSzam(){...}
   //Adott pozíciójú elem lekérdezése
   public Object elem( int poz ){...}
}

public class ListaHalmaz implements HalmazInterfesz{
   //Adattagok
   ...
   public void ujElemFelvetele( Object o){...}
   public void elemTorlese( Object o){...}
   public int elemSzam(){...}
   //Beszúrás adott pozícióba
   public void beszuras( Object o, int poz ){...}
}
```

Válassza ki a helyes programrészleteket:

a) HalmazInterfesz s = new HalmazInterfesz();

b) HalmazInterfesz s = new TombHalmaz();

c) HalmazInterfesz s = new ListaHalmaz();

d) HalmazInterfesz s[] = new HalmazInterfesz[10];

e)
```
HalmazInterfesz s[] = new HalmazInterfesz[ 10 ];
for( int i=0; i<s.length; ++i)
   if( i%2 == 0 ) s[ i ] = new TombHalmaz();
   else s[ i ] = new ListaHalmaz();
```

f)
```
TombHalmaz  a = new TombHalmaz();
ListaHalmaz l = new ListaHalmaz();
HalmazInterfesz s = a;
System.out.println(s.elemSzam());
```

g)
```
TombHalmaz a = new TombHalmaz();
ListaHalmaz l = new ListaHalmaz();
HalmazInterfesz s = a;
System.out.println( s.elem( 5 ) );
```

h)
```
HalmazInterfesz s = new ListaHalmaz();
s.beszuras("almafa", 3);
```

3. Mi a hiba a következő programban?

```
interface F1{  public void M1(); }
interface F2 { public void M2(); }
class A implements F1, F2{
   public static void main( String args[] ){
      A a = new A();
      a.M1();
      a.M2();
   }
   public void M1(){ System.out.println("M1");}
}
```

8. FEJEZET

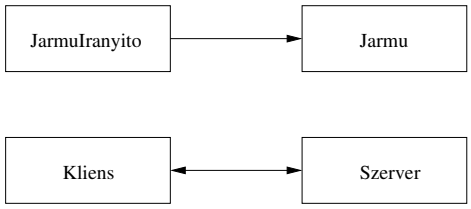
Objektumok közötti társítási kapcsolatok

Az üzenetküldés alapvető feltétele, hogy az objektumok kapcsolatban legyenek egymással. Ez a kapcsolat teszi lehetővé, hogy az objektumok üzenni tudjanak egymásnak, azaz valamilyen tevékenység elvégzésére tudja rábírni egyik objektum a másikat. Kétféle társítási kapcsolat létezik:

- ismeretségi, illetve használati kapcsolat
- tartalmazási, illetve rész-egész kapcsolat

8.1. Ismeretségi kapcsolat

Két objektum ismeretségi kapcsolatban áll egymással, ha létezésük egymástól független, és legalább az egyik ismeri, használja a másikat. Ez a kapcsolat nem kötelezően érvényes az egész futási időre. Elégséges, ha a kapcsolat a használat előtt létrejön és abban a pillanatban megszüntethető, amikor már befejeződött a kommunikáció. Ha az előző órán ismertetett járműirányító meg a jármű kapcsolatát tekintjük, itt is elmondhatjuk, hogy az irányítónak pontosan addig kell ismernie a járművet, amíg elvégzi ennek a parkolását. A kapcsolat ebben az eset-



8.1. ábra. Ismeretségi kapcsolat

ben egy egyirányú kapcsolat, azaz csak az irányítónak kell ismernie a járművet, ez fog műveleteket végeztetni a járművel.

Ha kliens-szerver alkalmazásokat tekintünk, akkor ott is megállapíthatunk ilyen jellegű kapcsolatot a kliens és a szerver objektumok között. Legelső példaként tekintsünk egy dátum szervert. Egy ilyen szerver feladata, hogy a rákapcsolódó kliensnek elküldje a pontos dátumot. Mivel egy nagyon rövid üzenetet kell elküldenie a kliensnek és rögtön utána meg is szakítja ezzel a kapcsolatot, ezért az ilyen típusú szervereket nem tervezik párhuzamos architektúrájúnak, azaz egyidőben csak egy kliens kiszolgálására alkalmasak. Egy ilyen kliens-szerver kapcsolatban mindkét félnek ismernie kell a másikat. A kliensnek azonosítania kell a szervert, ahhoz, hogy létre lehessen hozni a kapcsolatot. A szervernek pedig ismernie kell a klienset, ahhoz, hogy eljuttathassa az üzenetet hozzá. Ez a kapcsolat tehát kétirányú kapcsolat lesz.

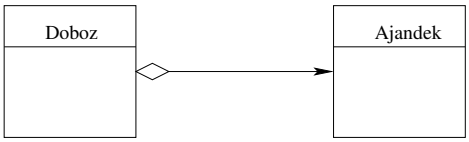
Az ismeretségi kapcsolat grafikus ábrázolását a 8.1. ábra szemlélteti.

8.2. Tartalmazási kapcsolat

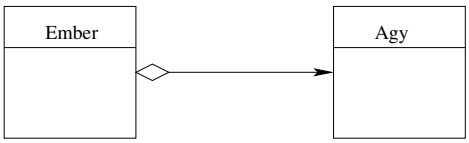
Az ismeretségi kapcsolat mellett létezik egy másik, erősebb kapcsolat, az úgynevezett `tartalmazási kapcsolat`. A tartalmazási kapcsolatot még `rész-egész` kapcsolatnak is nevezzük. Az ilyen objektumok össze vannak zárva.

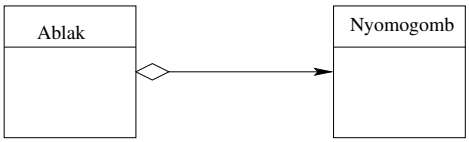
A tartalmazási kapcsolatotokat is osztályozhatjuk. A kapcsolat erőssége alapján megkülönböztetünk:

- Laza tartalmazást, ha a rész kivehető az egészből
- Szoros tartalmazást, ha a rész nem vehető ki az egészből



8.2. ábra. Laza tartalmazási kapcsolat



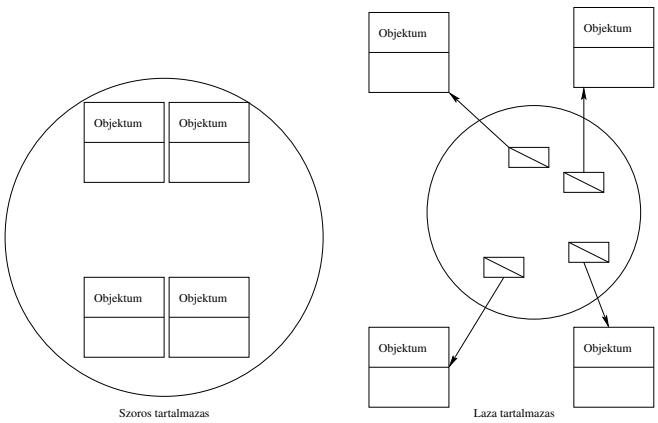


8.3. ábra. Szoros tartalmazási kapcsolat

Laza tartalmazás van például egy doboz és egy ebben elhelyezendő ajándék között (lásd 8.2. ábra). Ha a dobozból kivesszük az ajándékot, utána mindkettő önállóan is használható. Sem a tartalmazó, sem pedig a tartalmazott nem szűnik meg létezni a kapcsolat megszűnése után.

Ezzel ellentétben, szoros tartalmazás esetén, a tartalmazó megszűnése maga után vonja a tartalmazott objektum megszűnését is. A rész nem éli túl az egészet. Ilyen példákat szemléltet a 8.3. ábra is. Nincs agy ember nélkül és nincs nyomógomb ablak nélkül. Vagyis minden emberi agy hozzátartozik valamely emberhez és az emberrel együtt elhal az agya is. Minden nyomógomb hozzátartozik egy ablakhoz, az ablak megszűnése a nyomógomb megszűnéséhez is vezet.

Tartalmazási kapcsolat van a tároló és a tárolt elemek között is. A tárolók megvalósítása implementációfüggő, hiszen a tárolónak nem kötelező fizikailag is tartalmaznia a tárolt objektumokat. Bizonyos esetekben elégséges, ha az ezek elérhetőségét biztosítja referencia (vagy cím C++) által. Mindkét megvalósításnak



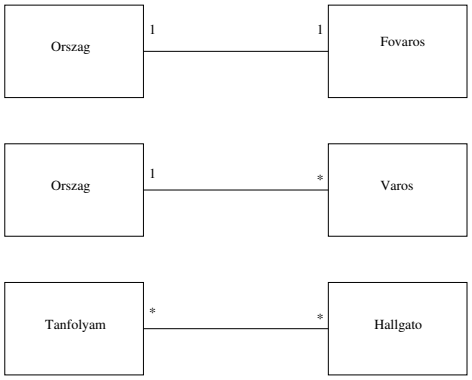
8.4. ábra. Tárolók kétféle megvalósítási módja

megvannak az előnyei és a hátrányai is egyben. Ha csak a referenciákat vesszük fel a tárolóba, akkor a tárolt objektumok még máshonnan is elérhetőek lehetnek, így állapotuk megváltoztatható a tároló tudta nélkül. Az objektumok felvétele gyors, hiszen nem igényel másolást. A Java ezt a „lazább" megvalósítást biztosítja, míg a standard C++ beépített tárolói másolatot készítenek minden egyes tárolt objektumról (kivételt képez a mutatókból alkotott tároló). A 8.4. ábra szemlélteti e kétfajta tárolási lehetőséget.

8.3. Kapcsolat foka

Amíg az interfész és az interfészt implementáló osztály között a kapcsolat egy alárendelő (függőleges) viszonyt fejez ki (általános-sajátos), addig az ismeretségi és a tartalmazási kapcsolat azonos szintű objektumok között fejez ki viszonyt (vízszintes). A kapcsolatokat osztályozhatjuk a kapcsolatok foka alapján is:

- egy az egyhez, $1 \rightarrow 1$
- egy a sokhoz, $1 \rightarrow *$
- sok a sokhoz, $* \rightarrow *$



8.5. ábra. Kapcsolatok foka

A 8.5. ábra ilyen kapcsolatokat szemléltet. Egy országnak egy fővárosa van, viszont akárhány városa lehet, sőt a városok száma időben változó. Egy oktatási rendszerben több tanfolyamot meghirdethetnek és ezekre a tanfolyamokra több hallgató is jelentkezhet. Egy tanfolyamot több hallgató látogat rendszeresen. Egy hallgató akárhány tanfolyamra jelentkezhet. Ebben az esetben egy kétirányú, `sok a sokhoz` kapcsolattal állunk szemben. Tekintsük egy FTP szerver és egy FTP kliens kapcsolatát. Az ilyen szerverek egyidejűleg több klienset szolgálnak ki, tehát ezeket párhuzamos architektúrájúaknak tervezik. A szervernek ismernie kell az összes klienset, mindeniknek az igény szerint kiválasztott állományokat kell eljuttatnia. A kliensnek viszont csak azt az egyetlen FTP szervert kell ismernie, amellyel éppen fájl átvitelt bonyolít le. A szerver irányából a kapcsolat `egy a sokhoz` kapcsolatnak tekinthető, a kliens irányából pedig `egy az egyhez` kapcsolatnak.

8.4. Kapcsolatok megvalósítása

A kapcsolatokat az osztályokban valósítjuk meg. Mivel a Java nyelv csak referenciákat biztosít, így minden típusú kapcsolatot referenciával fogunk megvalósítani.

1. `egy az egyhez` kapcsolat megvalósítása

A járműirányító és jármű kapcsolat egy egyirányú, `egy az egyhez` kapcsolat és a következőképpen valósíthatjuk meg:

```
public interface Jarmu{...}

public class Auto implements Jarmu{...}

public class Jarmuiranyito{
   private Jarmu jarmu;
   ...
}
```

A `Jarmuiranyito` osztaly `jarmu` referenciája a mindenkori mozgatandó járművet jelenti. Ha kétirányú kapcsolatunk van, akkor mindkét osztályban felveszünk egy-egy másik típusú referenciát.

2. `egy a sokhoz` kapcsolat megvalósítása

Az `egy az egyhez` kapcsolatot egy referenciával valósíthatjuk meg. Az `egy a sokhoz` vagy a `sok a sokhoz` kapcsolat viszont már tároló típusú referenciát igényel. A tároló típusa mindig problémafüggő. Ha pontosan tudjuk, hogy mennyi ez a sok, akkor használhatunk tömböt is, ha viszont ez nem nyilvánvaló, akkor dinamikus (változó kapacitású) tárolót ajánlott használni. Ha rendezett sorrendben akarjuk tárolni az objektumainkat, akkor rendezettséget biztosító tárolóra van szükségünk. Az ország-város kapcsolat egy lehetséges megvalósítását a következő példa szemlélteti.

```
public class Varos{ ... }

public class Orszag{
   private Varos [] varosok;
   ...
}
```

Egy másik megvalósítás esetében dinamikus tömböt fogunk használni. Ez az utóbbi megvalósítás ajánlott, ha a városok száma időben változó.

```
import java.util.Vector;

public class Varos{ ... }

public class Orszag{
   private Vector varosok;
   ...
}
```

3. `sok a sokhoz` kapcsolat megvalósítása

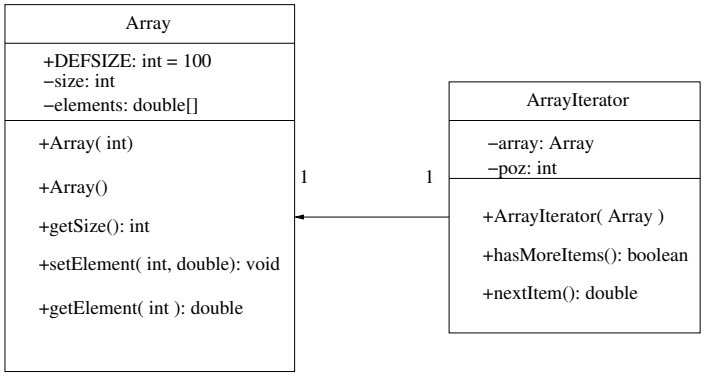
Erre az esetre tekintsük a tanfolyam-hallgató kapcsolatot. Mindkét esetben, mivel nem rögzített sem a hallgatók száma egy tanfolyamon, sem pedig az egy hallgató által felvehető tanfolyamok száma, dinamikus tömböt fogunk használni.

Tanfolyam.java

```
import java.util.Vector;

public class Tanfolyam{
   private Vector hallgatok;
   ...
}
```

Hallgato.java

```
import java.util.Vector;

public class Hallgato{
   private Vector tanfolyamok;
   ...
}
```



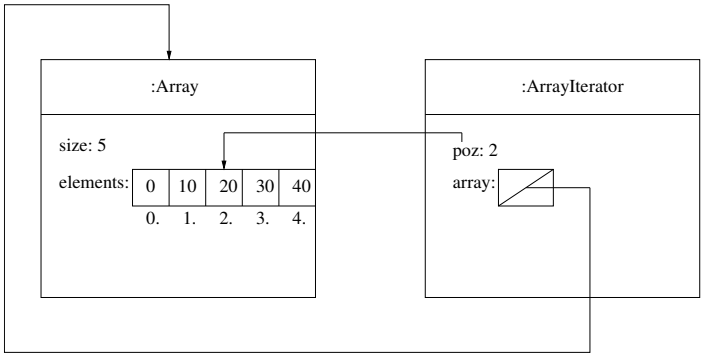
8.6. ábra. Tároló és bejáró osztálydiagram

8.5. A bejáró (iterátor) tervezési minta

A bejáró feladata, hogy szekvenciális hozzáférést biztosítson egy adott tároló elemeihez anélkül, hogy annak belső ábrázolását nyilvánossá tenné. Ezt úgy valósíthatja meg, hogy eltárolja a tároló aktuális pozícióját és a műveletein keresztül lehetővé teszi ezen pozíción elhelyezkedő elem lekérdezését, illetve a pozíció léptetését a tárolón belüli következő pozícióra. Egy lehetséges megoldás, hogy maga a tároló tartalmazzon egy ilyen aktuális pozíció adattagot. Ennek a megoldásnak viszont az lenne a hátránya, hogy egyidejűleg a tárolónak csak egy bejárását végezhetnénk. Egy tárolót pedig egy párhuzamos futási környezetben nyugodtan használhatnak különféle objektumok, amelyek egyenként a tároló különböző pozícióiban lehetnek. Ezt viszont csak úgy lehet megvalósítani, ha a bejárási funkciókat elkülönítjük egy külön osztályba, amelyből annyi példányt hozhatunk majd létre, ahány bejárásra van szükségünk.

A bejáróra egy nem túl általános megvalósítást fogunk szemléltetni, amely jó példa az `egy az egyhez` típusú ismeretségi kapcsolatra is. Tekintsünk egy valós számok tárolására alkalmas `Array` osztályt és az ennek bejárását megvalósító `ArrayIterator` osztályt.

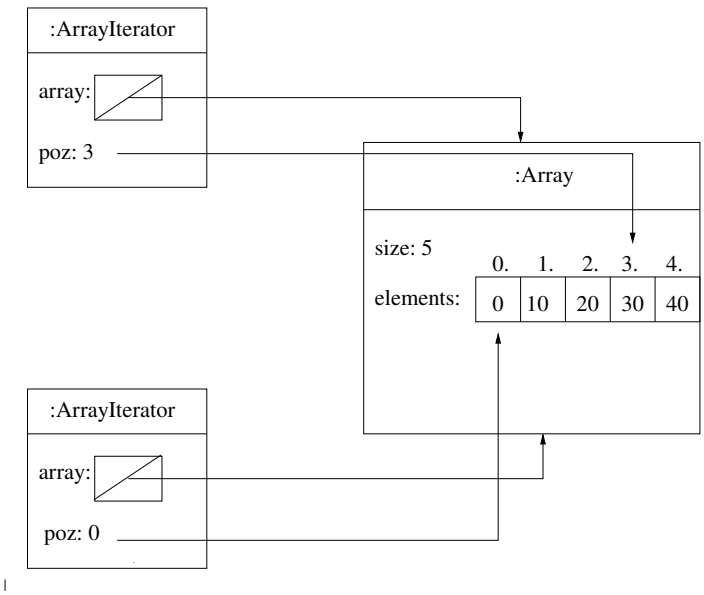
Az osztálydiagramot a 8.6. ábra szemlélteti, míg az objektumok közötti kapcsolatot a 8.7. ábra.



8.7. ábra. Tároló és bejáró kapcsolata

Mivel egy tárolóhoz több bejárót is rendelhetünk, ez azt jelenti, hogy több iterátor objektumunk lesz, amelyek mindenike `egy az egyhez` típusú ismeretségi kapcsolatban van a tároló objektummal (8.8. ábra).

```
public class Array{
   public static final int DEFSIZE = 100;
   private int size;
   private double[] elements;

   public Array(){
      this.size = DEFSIZE;
      this.elements = new double[ size ];
   }

   public Array( int size ){
      this.size = size;
      this.elements = new double[ size ];
   }

   public int getSize(){
      return size;
   }
```



8.8. ábra. Egy tároló, több bejáró kapcsolata

```
   public void setElement( int poz, double val ){
     if( poz >=0 && poz<size )
        elements[ poz ] = val;
   }

   public double getElement( int poz ){
      return elements[ poz ];
   }
}

class ArrayIterator{
   private Array array;
   private int poz;

   public ArrayIterator( Array array ){
```

```
      this.array = array;
      this.poz = 0;
   }

   public boolean hasMoreItems(){
      return poz<array.getSize();
   }

   public double nextItem(){
      return array.getElement( poz++ );
   }
}
```

A fenti programban megadott típusok használatát a következő kódrészlet szemlélteti:

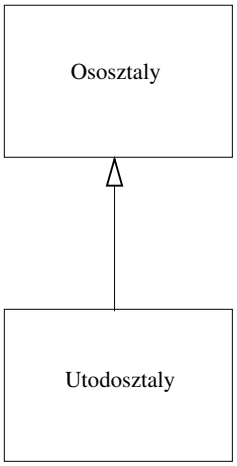
```
Array a= new Array( 5 );
for( int i=0; i<a.getSize(); ++i )
   a.setElement( i, i * 10 );
ArrayIterator it = new ArrayIterator( a );
while( it.hasMoreItems() )
   System.out.println( it.nextItem() );
```

9. FEJEZET

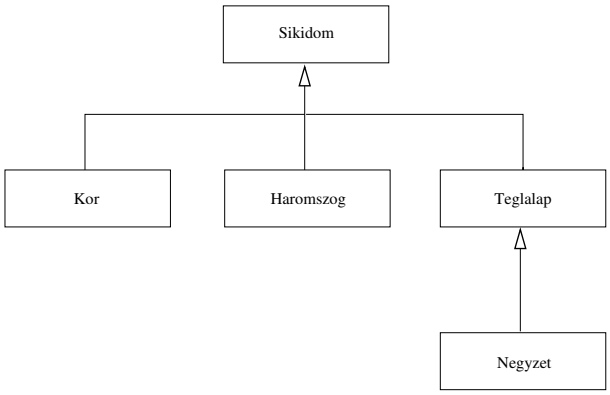
Származtatás

A fogalmak nem önmagukban, hanem más fogalmakkal együtt léteznek. Az objektumok közötti ismeretségi, illetve tartalmazási kapcsolat is objektumok közötti viszonyt fejeznek ki. Amíg e két kapcsolat azonos szintű fogalmak közötti viszonyt fejez ki, addig a származtatással alárendeltségi (általános-sajátos) viszonyt tudunk kifejezni. Amikor több fogalomból kiemeljük a közös részt, akkor `általánosításról` beszélünk, illetve amikor egy általános fogalomhoz egyedi jellemzőket adunk, akkor `sajátosításról`. A négyzetben és a körben az a közös, hogy mindkettő síkidom (általánosítás). A téglalap olyan paralelogramma, amelynek van egy derékszöge (sajátosítás). Származtatott osztállyal egy sajátosítási viszonyt fejezünk ki, vagyis egy általánosabb típusból származtatunk egy sajátosabbat. Az általánosabb típust, amiből származtatunk, ősosztálynak nevezzük, míg a sajátos típust, amit származtatunk, utódosztálynak. A származtatást UML diagramon egy nyíllal ábrázoljuk, amely az utódosztálytól az ősosztály fele mutat (ld. 9.1. ábra).

A 9.2. ábra a síkidom, kör, négyzet, téglalap, háromszög fogalmak viszonyát szemlélteti.



9.1. ábra. UML jelölése az öröklésnek



9.2. ábra. Osztályhierarchia

9.1. A származtatás szabályai

Származtatással egy már létező osztály tulajdonságait és viselkedésmódját bővíthetjük, illetve ezeket módosíthatjuk. A tulajdonságokat az adatok határozzák meg, míg a viselkedésmódért a metódusok a felelősek. Ennek következtében a származtatás a következőket teheti:

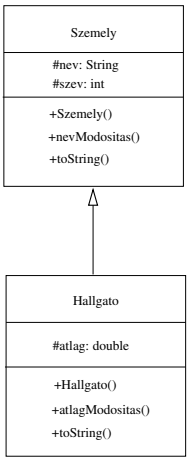
- új adatokkal bővíti a létező osztályt
- új metódusokkal bővíti a létező osztályt
- létező adatokat elfed
- létező metódusokat felülír

Tekintsük most a személy és a hallgató típusok közötti kapcsolatot. Nyilvánvaló, hogy minden hallgató egy személy is egyben, tehát a személy az általánosabb fogalom, a hallgató pedig a sajátosabb. Ezt a kapcsolatot származtatással fogjuk megvalósítani. A személy típus esetében csak a nevet és a születési évet fogjuk tárolni. A hallgató pedig kibővíti a tulajdonságokat egy tanulmányi átlaggal. A viselkedésmódot illetően nagyon minimalisták leszünk, ezért a konstruktor mellé még két-két műveletet adunk meg. A személy típus esetében névmódosítást engedélyezünk, illetve az objektum állapotának lekérdezését, míg a hallgató esetében a tanulmányi átlag módosítását és szintén az objektum állapotának a lekérdezését. Ha az utódosztálynak engedélyezni akarjuk az ősosztály adataihoz való hozzáférést, akkor ezeket az adatokat védettnek kell deklarálni. A védett tagokat a `protected` kulcsszó vezeti be. A többi láthatósági módosítóhoz hasonlóan ez is úgy adattagokra, mint metódusokra használható. A 9.3. ábra a két osztály és a köztük lévő származtatási viszony UML diagramját szemlélteti.

Ez a származtatási kapcsolat a származtatás négy lehetőségéből a következőket használta:

- új adattaggal bővítette a `Hallgato` osztályt: `atlag`
- új metódussal bővítette a `Hallgato` osztalyt: `atlagModositas()`
- felülírta a `toString()`metódust

A 9.3. ábrának megfelelő Java kódot a következő program szemlélteti:



9.3. ábra. Személy-Hallgató osztályok

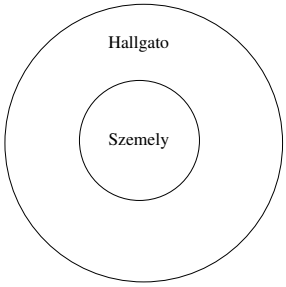
```
public class Szemely{
   protected String nev;
   protected int szev;

   public Szemely( String nev, int szev){
      this.nev = nev;
      this.szev = szev;
   }

   public void nevModositas( String ujnev ){
      this.nev = ujnev;
   }

   public String toString(){
      return this.nev+" : "+this.szev;
   }
}

public class Hallgato extends Szemely{
```

```
   protected double atlag;

   public Hallgato( String nev, int szev, double atlag){
      super( nev, szev );
      this.atlag = atlag;
   }

   public void atlagModositas( double ujatlag ){
      this.atlag = ujatlag;
   }

   public String toString(){
      return nev+" : "+szev+" : "+atlag;
   }
}
```

A `Hallgato` osztály konstruktorában a `super( nev, szev )` az ősosztály konstruktorának meghívását jelenti, jelen esetben a `Szemely(nev, szev)` konstruktorét. Az ősosztály konstruktorának hívása első utasításként kell szerepeljen a származtatott osztály konstruktorában. A következőkben a származtatott osztály példányosítását fogjuk elemezni. Tekintsük a következő kódot példány létrehozására:

```
Hallgato h =new Hallgato("Peter Pal",1980,8.25);
```

A példány létrehozásakor először meghívódik a `Hallgato` osztály konstruktora, majd ez meghívja a `Szemely` osztály konstruktorát. Miután lefut a `Személy` osztály konstruktora, befejeződik a `Hallgato` osztály konstruktorának hívása is. A mi esetünkben az ősosztály konstruktorának hívása expliciten történt, mert ennek paraméterekre volt szüksége. Ez a hívás történhet implicit módon is, ha az ősosztálynak van paraméter nélküli konstruktora. Egy `Hallgato` típusú objektum mindig tartalmaz egy `Szemely` típusú objektumot is, hiszen annak minden egyes adattagját elörökölte. Természetesen felülírhat egyes adattagokat, ebben az esetben két példány lesz a felülírt adattagból. Alapértelmezetten a felülírt változat lesz használatban, de a `super` hivatkozással az ősosztály bármely nem privát



9.4. ábra. Hallgató objektum

adattagja elérhető. A 9.4. ábra szemlélteti a `Hallgato` és `Szemely` objektumok közötti viszonyt.

9.2. Statikus típus, dinamikus típus és polimorfizmus

Egy hivatkozás típusú változó esetében megkülönböztetjük ennek statikus típusát, illetve dinamikus típusát. A változó `statikus típusa` a deklarálási típus, a dinamikus típusa pedig a hivatkozott objektum tényleges típusa. A dinamikus típus minden esetben vagy a statikus típus vagy annak leszármazottja. Ha a statikus típus egy interfész, akkor a dinamikus típus egy olyan típus, amely implementálja az adott interfészt. Minden változónak egyetlen statikus típusa van és több dinamikus típusa lehet a program futása alatt. Emlékezzünk vissza a `Vezetheto` interfészre és az ezt implementáló `Auto, Hajo` és `Kerekpar` osztályokra.

```
Vezetheto v;        // v statikus típusa Vezetheto
v = new Hajo();     // v dinamikus típusa Hajo
v.indulj();
v = new Kerekpar();// v dinamikus típusa Kerekpar
v.indulj();
v = new Auto();     // v dinamikus típusa Auto
v.indulj();
```

Az előző kódrészletben a `v` változónak egyetlen statikus típusa van (`Vezetheto`) és egymás után háromféle dinamikus típusa volt. Ezt neveztük polimorfizmusnak (sokalakúságnak), azaz ugyanazon a változón keresztül különböző típusú objektumokat érhettünk el. Természetesen a dinamikus típus bonyodalmakat fog okozni a fordítóprogramoknak is, hiszen a `v.indulj()` kódrészlet háromféle viselkedést kell eredményezzen a három különböző hívás alkalmával. A megoldás egyszerű, de csak futásidőben oldható meg, hiszen akkor derül ki a változó által hivatkozott objektum dinamikus típusa. Ilyen esetben a döntés futásidőre halasztódik, ekkor fog kiderülni, hogy pontosan melyik osztály `indulj()` metódusát kell meghívni. Az ilyen kötést (metódus hívás - metódus címe) `kései kötésnek` vagy `dinamikus kötésnek` nevezzük. Hátránya a klasszikus függvényhívásokhoz képest az, hogy lassítja a végrehajtást, hiszen futásidőben kell eldönteni a pontos típust és ennek alapján meghívni a megfelelő metódust. A Java minden egyes metódusnál dinamikus kötést alkalmaz, kivételt képeznek a statikus metódusok, ahol egyértelmű, hogy melyik metódust kell meghívni. A nem dinamikus kötést `statikus kötésnek` nevezzük. Ebben az esetben már fordításkor eldől a hívott függvény címe, így ez gyors metódushívást eredményez. Statikus metódusoknál viszont nincs lehetőség a polimorfizmusra.

Az interfészeknél ismertetett polimorfizmus a származtatásra is érvényes, hiszen minden utódosztályból példányosított objektum egyben ősosztály típusú is. Így lehetőség van ősosztály típusú referencián keresztül utódosztályból példányosított objektumokat kezelni. Tekintsünk erre a következő példát:

```
Szemely sz; // sz statikus típusa
sz = new Szemely(''Pal Peter'', 1986 );
System.out.println( sz );
sz = new Hallgato( ''Sandor Klara'', 1970);
System.out.println( sz );
```

A fenti kódrészletben az `sz` egy alaposztálybeli referencia, amely rendre egy `Szemely` típusú objektum illetve egy `Hallgato` típusú objektum kezelését teszi lehetővé. A `System.out.println( sz )` kódrészlet a hivatkozott objektum állapotát fogja megjeleníteni úgy, hogy automatikusan meghívja a hivatkozott objektum típusának megfelelő `toString()` metódust. A `Hallgato` osztályban elérhető

az ősosztály `toString()` metódusa a következőképpen: `super.toString()`. Például:

```
public String toString(){
   System.out.println( "\H{O}sosztályé: "+super.toString());
   return nev+":"+szev+":"+atlag;
}
```

Az előző `Szemely-Hallgato` példában láthattuk a polimorfizmus azon megnyilvánulását, amelyben egy ősosztály típusú referencián keresztül utódosztály típusú objektumokat is kezelhettünk. A `Szemely` típusú `sz` referenciának értékül adhattunk `Hallgato` típusú objektumot is. Az értékadás után kiírattuk az objektum állapotát és, ahogyan elvártuk, ez a `Hallgato` osztaly `toString()` metódusát hívta. Ha viszont most a `Hallgato` objektum `atlagModositas()` műveletét kellene meghívnunk, akkor ez nem lenne lehetséges az `sz` referencián keresztül, mert ezen keresztül csak a `Szemely` típusnak megfelelő műveletek végezhetőek (ezek természetesen polimorfikusan). Ha mégis szükség van a művelet elvégzésére, akkor explicit konverziót kell végezni a következőképpen.

```
Szemely sz = new Hallgato("Pal", 10);
Hallgato h = (Hallgato) sz;
h.atlagModositas( 7.50 );
```

Tulajdonképpen egy objektum annyiféle típusú referencián keresztül kezelhető, ahány típust megtestesít (öröklési hierarchia+implementált interfészek), de az objektummal csak azokat a műveleteket lehet végezni, amelyeket a referencia típusa(statikus típus) biztosít. A referencia típusa fogja meghatározni, hogy milyen viselkedésre bírható rá az objektum.

Az explicit konverzió végzése előtt érdemes megvizsgálni, hogy az átalakítandó objektum átalakítható-e a kért típusra, vagyis összeférhető-e a két típus. Erre a Java nyelv `instanceof` operátora használható. A fenti kódrészlet a következőképpen alakulna:

```
Szemely sz = new Hallgato(''Pal'', 10);
Hallgato h = null;
if( sz instanceof Hallgato ){
```

Módosító	Saját osztály	Saját csomag	Utódosztály	Külvilág
private	Igen			
csomagszintű	Igen	Igen		
protected	Igen	Igen	Igen	
public	Igen	Igen	Igen	Igen

9.1. táblázat. Hozzáférési kritériumok

```
   h = (Hallgato) sz;
   h.atlagModositas( 7.50 );
}
```

9.3. Láthatósági módosítók

Mivel már ismertettük a négyféle hozzáférési (láthatósági) szintet, a 9.1 táblázatban összefoglaljuk ezeket. Amennyiben egy tagot nem látunk el hozzáférési módosítóval, ezt a fordító alapértelmezettnek módosítójunak tekinti. Az alapértelmezett módosítót még csomagszintűnek is nevezzük.

9.4. Metódusok felülírása

Ahogyan az előző alfejezetekben már szemléltettük, a metódusok felülírása képzi a polimorfizmus alapját. Van azonban egy pár szabály, amelyeket be kell tartanunk ezzel kapcsolatosan. Tulajdonképpen a metódusok felülírása biztosítja, hogy egy utódosztály megváltozathassa egy ősosztály viselkedését.

1. Visszatérési típus: A felülírás szabálya egészen a J2SE 5.0 verzióig az volt, hogy az utódosztálybeli metódus neve, visszatérési típusa és argumentumlistája tökéletesen meg kellett egyezzen az ősosztályban deklaráltéval. A

J2SE 5.0 verziótól kezdődően a visszatérési típus lehet az ősosztályban megadott visszatérési típus, vagy annak valamely leszármazott típusa.

2. Láthatósági módosító: A felülíró metódus láthatósága nem lehet szűkebb, mint a felülírté. Ennek következménye, hogy `public` láthatóságú metódust csakis hasonlóval lehet felülírni; `protected` láthatóságút pedig `protected`, illetve `public láthatóságúval`; csomagszintűt pedig csomagszintű, `protected, public` láthatóságúval és végül a `private` láthatóságút bármilyen mással.

3. Ellenőrzött kivételek: Habár a kivételkezelést csak egy későbbi fejezetben szemléltetjük, a teljesség érdekében megemlítjük, hogy a felülíró metódus csak ugyanolyan típusú vagy ezen típusok leszármazottjai típusú ellenőrzött kivételeket dobhat, mint a felülírt.

4. Végleges metódusok: A `final` módosítóval ellátott metódusokat végleges metódusoknak nevezzük. Az ilyen metódusokat nem lehet felülírni.

9.5. Metódusok túlterhelése

Amíg a metódusok felülírása vagy felülbírálása csak akkor lehetséges, ha van legalább két osztályunk, amelyek örökítési kapcsolatban vannak egymással, addig a metódusok túlterhelése egy osztályon belül érvényes fogalom. Akkor beszélünk túlterhelésről, amikor ugyanazon osztálynak több azonos nevű, de különböző szignatúrájú metódusa van. Azt már tapasztaltuk, hogy a `System.out.print(valtozo)` bármilyen típusú változóra meghívható. Ez pedig úgy valósul meg, hogy a `System.out`, avagy szabványos kimenet, típusa a `PrintStream` osztály, több túlterhelt formáját tartalmazza ennek a metódusnak:

```
void print( int i );
void print( char c );
void print (String s );
..
```

Túlterhelt név használatánál az aktuális paraméterek száma és típusa szerint választható ki, hogy pontosan melyik metódust kell meghívni (a metódus

visszatérési típusa érdektelen). A metódus aktuális paramétereinek illeszkednie kell valamely létező metódus szignatúrájához. Amíg a metódusok felülírása egy olyan fajta polimorfizmust indukál, amelyet csak dinamikus kötéssel, futásidőben lehet feloldani, addig a metódusok túlterhelését már fordítási időben fel lehet oldani.

9.6. Konstruktorok

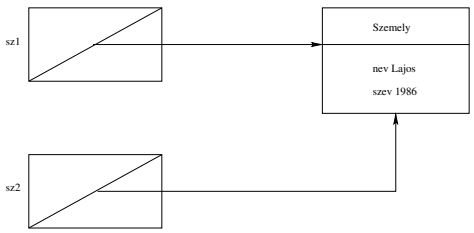
Annak ellenére, hogy az utódosztály örökli az ősosztály metódusait és adatait, ennek konstruktorait nem örökli. Minden osztálynak van konstruktora, amelyet vagy a programozó készít, vagy pedig amennyiben a programozó nem készítene, akkor a fordító generál egyet. A fordító által generált konstruktort alapértelmezett konstruktornak nevezzük.

Minden konstruktor meghívja az ősosztálya konstruktorát. Ez a hívás történhet expliciten a `super(paraméterlista)` hívással, vagy pedig impliciten, amennyiben az ősosztálynak van paraméter nélkül hívható konstruktora. A `super` kulcsszóval hívott ősosztály konstruktor hívását nem előzheti meg semmilyen más utasítás.

A metódusokhoz hasonlóan a konstruktorok is túlterhelhetők és ezek egymást is hívhatják a `this(paraméterlista)` segítségével. Ennek érdekében tanulmányozzuk a következő példaprogramot:

```
import java.util.*;

public class Alkalmazott{
   private static final double ALAPFIZETES=1000;
   private String nev;
   private double fizetes;
   private Date szulDatum;

   public Alkalmazott( String nev, double fizetes, Date szulDatum ){
      this.nev = nev;
      this.fizetes = fizetes;
      this.szulDatum = szulDatum;
```

```
   }

   public Alkalmazott( String nev, double fizetes ){
      this( nev, fizetes, new Date());
   }

   public Alkalmazott( String nev, Date szulDatum ){
      this( nev, ALAPFIZETES, szulDatum);
   }

   public Alkalmazott( String nev){
      this( nev, ALAPFIZETES, new Date());
   }

   public String toString(){
      return nev+" - "+fizetes+" - "+szulDatum;
   }

   public static void main( String args[] ){
      Alkalmazott a1 = new Alkalmazott("Kati", 2000, new Date(100,01, 31));
      Alkalmazott a2 = new Alkalmazott("Laci", 3000 );
      Alkalmazott a3 = new Alkalmazott("Mari", new Date(20,12,31));
      Alkalmazott a4 = new Alkalmazott("Jani");
      System.out.println( a1 );
      System.out.println( a2);
      System.out.println( a3 );
      System.out.println( a4 );
   }
}
```



9.5. ábra. Egy példányra két referencia esetében sz1 == sz2 értéke is igaz, és sz1.equals(sz2) is igaz

9.7. Az Object osztály

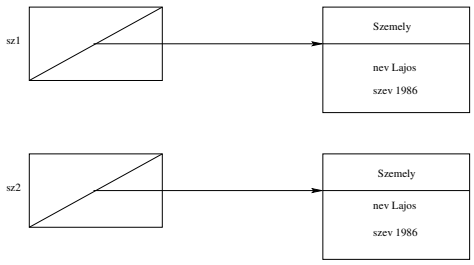
Az `Object` osztály minden osztály közös őse, a Java osztályhierarchia gyökere. Ha egy osztály definíciójakor nem adjuk meg az osztály ősét, automatikusan az `Object` lesz az őse. Ennek következtében az `Object` osztáy metódusait minden osztály örökli, biztosítva az ennek megfelelő viselkedésmódot. Bizonyos metódusokat nem lehet felülírni, ilyenek a `wait(), notify()`, `notifyAll()`, amelyeket a párhuzamos végrehajtás esetében (Java szálaknál) használunk. A `getClass()` metódus sem írható felül, ezzel tudjuk meghatározni egy objektum osztályát.

Az `Object` osztály `toString()` és `equals()` metódusait majdnem minden osztály esetében felül kell írnunk. A `toString()` az objektum állapotát adja vissza karakterlánc formájában. Az `Object` osztályban megvalósított `equals()` metódus úgy vizsgálja a két objektum egyenlőségét, hogy ha a két objektum referenciája megegyezik, akkor az objektumokat egyenlőknek nyilvánítja. Tekintsük a következő példát:

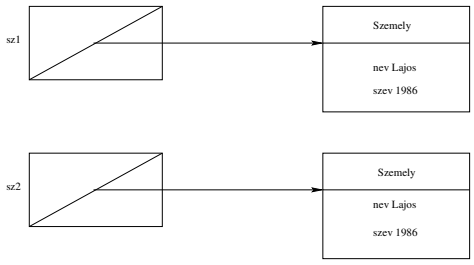
```
Szemely sz1 = new Szemely(''Lajos'', 1986 );
Szemely sz2 = sz1;
System.out.println(sz1.equals(sz2));
sz2 = new Szemely(''Lajos'', 1986 );
System.out.println(sz1.equals(sz2));
```

Az első összehasonlítás eredménye igaz, míg a másodiké hamis. A két állapotot a 9.5. és a 9.6. ábrák szemléltetik.

Természetesen két objektum egyenlőségét tartalmi összehasonlítással illik végezni. Ennek érdekében kiegészítjük a `Szemely` osztályt egy `equals` metódussal:



9.6. ábra. Az `equals` metódus felülírása nélkül: sz1 == sz2 is hamis, és sz1.equals(sz2) is hamis



9.7. ábra. Az `equals` metódus felülírása után: sz1 == sz2 hamis, és sz1.equals(sz2) igaz

```
public boolean equals( Object o){
   if( o == null || !(o instanceof Szemely) ) return false;
   Szemely p = (Szemely) o;
   return  nev.equals(p.nev) && szev == p.szev;
}
```

A felülírás utáni helyzetet a 9.7. ábra szemlélteti.

A JDK osztályai felülírják az `equals` metódust, ezért ez a metódus mindig tartalmi összehasonlítást fog végezni. Érdekességképpen megemlítünk egy optimalizálást, amelyet a Java a `String` típusú objektumok esetében végez. Ennek szemléltetésére tekintsük a következő programot:

```
public class p1{
```

```
   public static void main( String args[] ){
      String s1 ="Hello";
      String s2 ="Hello";
      System.out.print(s1 == s2);
      System.out.print(" "+s1.equals(s2));
      String s3 = new String("World");
      String s4 = new String("World");
      System.out.print(" "+(s3 == s4));
      System.out.println(" "+s3.equals(s4));
   }
}
```

A fenti program kimenete: `true true false true`. Ennek magyarázata a következő: A Java a karakterlánc literálokat egy olyan tárolóban helyezi el, amelyben csak egyedi karakterláncok vannak. Így az s2 inicializálása ugyanazzal a karakterlánccal történik, mint az s1 referenciáé. Eért lesz az `s1 == s2` értéke igaz. Amennyiben a karakterlánc inicializálása a `new` operátorral történik, akkor a Java mindig egy másolatot készít a saját tárolójában elhelyezett karakterlánc literálról. Ezért, az s3 is és az s4 is egy-egy különálló karakterlánc példányra hivatkozik aminek következtében az `s3 == s4` hamis.

Általánosan elfogadott szabály az, ha egy osztály felülírja az `equals` metódust, akkor felülírja a `hashCode` metódust is. Amennyiben az adott osztály példányait hasító táblázatokban (`HashSet`, `HashMap` stb.) szeretnénk elhelyezni, akkor ez a `hashCode` függvény értéke alapján fog történni. Az egyetlen szabály amit be kell tartani az, hogy az `equals` metódus szerint egyenlőnek minősített példányokra a `hashCode` függvény ugyanazt az értéket szolgáltassa. További részleteket a [5] és a [8] könyvekben találunk.

Az `Object` osztály másik fontos metódusa a `clone()` metódus. Ennek segítségével az objektumról másolatot tudunk készíteni. Abban az esetben, ha csak primitív típusú adattagjai vannak az objektumnak, akkor az `Object` osztály által biztosított megvalósítás tökéletesen megfelelő. Ha viszont referencia típusú adattagok is vannak, akkor a másolatba csak a referencia másolódik át, így az eredeti objektum és a másolat referencia típusú mezői mindig ugyanarra az objektumra fognak hivatkozni. A `clone()` metódus védettként van megvalósítva, így közvetlenül nem lehet használni, felül kell írni. Minden osztály, amely biztosítani

akarja a klonózást a példányainak, implementálnia kell a `Clonable` interfészt. A `clone()` metódus használatára tekintsük a következő kódrészletet:

```
Date d1 = new Date();
System.out.println( d1 );
Date d2 = (Date)d1.clone();
System.out.println( d1 );
System.out.println( d2 );
System.out.println( d1.equals( d2));
```

9.8. Absztrakt osztályok

Az absztrakt osztályok az interfészekhez hasonlóak. Amíg az interfészben deklarált minden metódus alapértelmezetten absztrakt, hiszen nem adjuk meg az implementációját, addig az absztrakt osztályban lehetnek absztrakt metódusok és implementált metódusok is. Minden olyan metódust, amelyeknek nem adhatjuk meg az implementációját, mert nem lehetséges az adott szinten, absztraktnak kell deklarálnunk. A metódust úgy deklaráljuk absztraktnak, hogy a láthatósági módosító mellett elhelyezzük az `abstract` kulcsszót. Ha egy osztálynak van egy absztrakt metódusa, akkor az osztály is absztrakt, tehát az osztálydefiníciónál is használnunk kell az `abstract` kulcsszót. Tekintsük a következő típusokat: síkidom, négyzet és kör. Amíg a síkidom egy absztrakt fogalom, addig a négyzet és a kör két konkrét síkidom. Azt viszont tudjuk, hogy minden síkidom jellemezhető területtel. Ha interfészt használnánk a síkidom típus ábrázolására, nem lenne lehetőségünk a terület adattag deklarálására, hiszen az interfészek nem teszik lehetővé ezt. Az absztrakt osztály viszont megengedi. A következő program tartalmazza a három fogalom egyszerű Java implementációját.

```
public abstract class Sikidom{
   protected double terulet;
   public abstract void rajzol();

}
```

```
public class Negyzet extends Sikidom{
   protected double oldalhossz;

   public Negyzet( double oldalhossz ){
      this.oldalhossz = oldalhossz;
      this.terulet = oldalhossz * oldalhossz;
   }


   public void rajzol(){
      System.out.println("Negyzet vagyok, teruletem: "+terulet);
   }
}

public class Kor extends Sikidom{
   protected double sugar;

   public Kor( double sugar ){
      this.sugar = sugar;
      this.terulet = 3.14 * sugar * sugar;
   }

   public void rajzol(){
      System.out.println("Kor vagyok, teruletem: "+terulet);
   }
}
```

A következő kódrészlet példát ad az előző programban definiált osztályok használatára.

```
Sikidom t[] = new Sikidom[ 2 ];
t{[0] = new Negyzet( 3 );
t{[1] = new Kor( 2 );
for( int i=0; i<t.length; ++i )
   t[i].rajzol();
```

A fenti példából láthatjuk, hogy az interfészek és az absztrakt osztályok ugyanazt a célt szolgálják. Amíg az interfészben csak állandókat deklarálhattunk, addig az absztrakt osztályban deklarálhatunk példányváltozókat. Az interfészben minden metódus alapértelmezetten absztrakt, az absztrakt osztályban vegyesen fordulhatnak elő az absztrakt és konkrét metódusok. Egyszerűen fogalmazva az interfész egy teljesen absztrakt típust definiál, míg az absztrakt osztályban kötelező módon jelen van legalább egy absztrakt elem.

9.9. Végleges osztályok és metódusok

Bizonyos osztályok és metódusok működésének megváltoztatása veszélyes következményekkel járhat. Ilyen például a `getClass()` metódus. Az ilyen metódusok felüldefiniálását a `final` kulcsszóval lehet megakadályozni. A `final` módosítóval ellátott metódusokat nem lehet felülírni az utódosztályokban.

```
public class Object{
   public final Class getClass(){ ... }
}
```

Ha egy osztály kiterjesztését akarjuk megakadályozni, akkor az osztályt látjuk el a `final` modosítóval.

```
public final class System{ ... }
```

Értelemszerűen a `final` és az `abstract` módosítók kizárják egymást.

9.10. Feladatok

1. Készítsen egy halmaz interfészt, amely tartalmazza a halmazokra jellemző legfontosabb műveletek deklarációit:
   - új elem felvétele a halmazba
   - adott elem törlése
   - adott elem hozzátartozásának ellenőrzése

– halmaz méretének lekérdezése

Az interfész legyen teljesen általános. (használjon `Object` típust!)

2. Készítse el az 1. feladatban definiált interfész implementációját. Az osztály használjon `Object` típusú tömböt a halmaz elemeinek tárolására.

3. Készítse el az 1. feladatban definiált interfész implementációját. Az osztály használja a Vector dinamikus tömböt a halmaz elemeinek tárolására.

4. A 2. feladatban definiált osztályból származtasson egy rendezett halmaz ábrázolására alkalmas osztályt. Írja fölül azokat a metódusokat, amelyeket másképpen kell végezni rendezett halmaz esetében.

5. Mi a következő program eredménye?

```
class X{
   public void doIt(){ System.out.println("X");}
}
class Y extends X{
   public void doIt(){ System.out.println("Y"); }
}
public class P{
   public static void main( String args[] ){
      X  obj = new Y();
      obj.doIt();
   }
}
```

6. Feltételezve, hogy a C osztály kódja helyes, mi a hiba a következő programban ?

```
public class A{
   private int a;
   public A( int a ){ this.a = a; }
}
```

```
public class B extends A{
   private int b;
   public B( int b ){ this.b = b; }
}

public class C{
   public static void main( String args[] ){
      A oa = new A( 10 );
      A ob = oa;
   }
}
```

7. Mit jelent a metódusok túlterhelése?

   a) Azonos nevű, különböző paraméterezésű metódusok

   b) Ősosztály metódusának felülírása utódosztályban

   c) Ősosztály metódusának elrejtése

   d) A `toString` metódus deklarálása

8. Az int f(int i, int j) függvényt hogyan nem lehet túlterhelni?

   a) int f(int i, int j, int k)

   b) int f(float i, float j)

   c) float f(int i, int j)

   d) int f(int i, int j, float k)

9. Leszármazott osztály nem fér hozzá ....

   a) Az ősosztály privát adattagjaihoz

   b) Az ősosztály nyilvános adattagjaihoz

   c) Az ősosztály védett adattagjaihoz

   d) Az ősosztály nyilvános metódusaihoz

10. A példánymetódusok felülírása azt jelenti, hogy

a) Egy osztályon belül több azonos nevő, különböző paraméterezésű metódust deklarálhatunk

b) Egy osztálynak több konstruktora lehet

c) Egy utódosztályban megadunk egy olyan nevű és paraméterezésű metódust, amilyen már van az ősosztályban

d) Egy osztály annyi metódust deklarál, ahány attribútumot

11. Mely állítás hamis absztrakt osztályra vonatkozóan?

a) Absztrakt osztályt nem lehet példányosítani

b) Absztrakt osztály örökítési célokat szolgál

c) Absztrakt osztályból nem lehet absztrakt osztályt származtatni

d) Absztrakt osztálynak van legalább egy absztrakt metódusa

10. FEJEZET

Kivételkezelés

10.1. Futásidejű hibák

A kivételkezelés célja a futás közben fellépő hibák orvosolása. Ha egy program nem végez kivételkezelést és futása közben kivételes helyzet adódik, például index-túlcsordulás, akkor a program végrehajtása befejeződik egy hibaüzenettel, amely tartalmazza a hiba okát. Ha például egy tömböt túlindexelünk, akkor `ArrayIndexOutOfBoundsException` szöveggel áll le a programunk, ha pedig egy nem inicializált referencián keresztül valamilyen metódushívást végzünk, akkor `NullPointerException` a kiírt szöveg.

```
int i;
double t[] = new double[ 100 ];
int i = //beolvasunk 200-at
System.out.println(t[i]);
//ArrayIndexOutOfBoundsException

String s=null;
System.out.println( s.compareTo(''üres'' ));
//NullPointerException
```

10.2. Kivétel keletkezése

A Java nyelvben a futásidejű hibákat kivételeknek nevezzük és ezeket kivételobjektumok reprezentálják. A kivételobjektum információt tárol a kivételes állapotról. Mint bármilyen más objektumról, a kivételobjektumról is lekérdezhető a kivétel típusa és típustól függően egyéb információk is. A kivételobjektum létrejötte után ez a Java-futtató környezet felügyelete alá kerül. Kivétel háromféle módon keletkezhet:

- A program futása közben valami rendellenes dolog történt. Ilyen például a nullával való osztás, vagy erőforrások hiánya.
- A program `throw` utasítással váltotta ki a kivételt
- Aszinkron kivétel (párhuzamos végrehajtás esetében)

A kivételt vagy a futtatott kód váltja ki, vagy pedig a futtató környezet (Java virtuális gép). Akárhogyan is keletkezett a kivétel, a Java-futtatási környezet megszakítja az utasítások végrehajtásának normális menetét és kivételes módon folytatódik a végrehajtás. Ha a futtatott program nem tartalmaz kivételkezelést, akkor egyszerűen visszakerül a kivételobjektum a Java virtuális gép szintjére és ez leállítja programunkat.

10.3. Kivétel lekezelése

A kivételek létrehozását és lekezelését a Java nyelvi szinten biztosítja. A létrehozás vagy kivételdobás a `throw` utasítással végezhető, míg a lekezelés a `try catch finally` utasítással. Kivételobjektumot ugyanúgy hozunk létre, mint bármilyen más objektumot, vagyis a `new` operátorral. Minden kivételosztály közvetlen vagy közvetett módon az `Throwable` osztály leszármazottja. Nem kötelező, de illik a kivételt úgy elnevezni, hogy a név tartalmazza az `Exception` szöveget is. Így világosan látszik a típus szerepe.

Létrehozás:

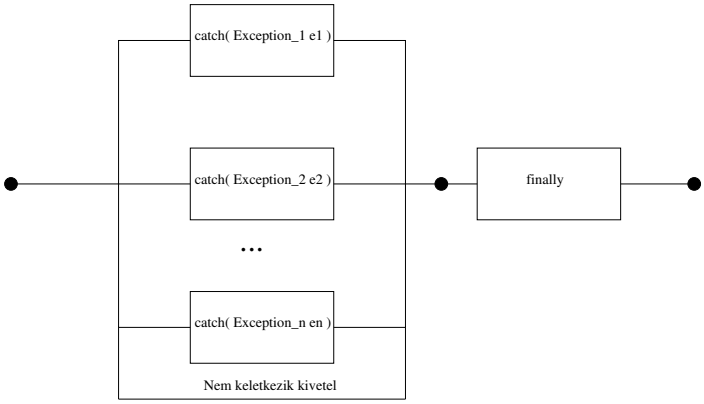
```
throw new Exception("Én egy kivétel vagyok");
```

Kivételkezelés:

```
try{
   //Utasításblokk-Kivételforrás
}
catch( Exception_1 e1){
   //Utasításblokk- Exception_1 lekezelésére
   //e1-a kivételobjektum referenciája
}
catch( Exception_2 e2){
   //Utasításblokk- Exception_2 lekezelésére
   //e2-a kivételobjektum referenciája
}
...
catch( Exception_n en){
   //Utasításblokk- Exception_n lekezelésére
   //en-a kivételobjektum referenciája
}
finally{
   //Utasításblokk-ez mindenképpen végrehajtódik
}
```

A kivételkezelő utasítás végrehajtását a 10.1. ábra szemlélteti.
Megjegyzések:

- Egy kivételkezelő utasításban a `try` után következő utasításblokkot kivételforrásnak is nevezzük. Ez az a kódrészlet, amelynek végrehajtása során kivételes állapot állhat elő. Lehetőleg minimális utasítást tegyünk egy ilyen blokkba, mert kivétel keletkezése esetén csak a kivétel keletkezési helyéig hajtódnak végre a blokkbeli utasítások.
- Egy `try` utasítás után akárhány `catch` utasítás következhet.
- A `finally` rész nem kötelező, viszont ha szerepel, akkor az utána levő utasításblokk mindenképpen végrehajtódik.



10.1. ábra. try-catch-finally utasítás végrehajtási váza

– A `catch` utasításokban elhelyezett kivételtípusok sorrendjét körültekintően kell megadni a kivételtípusok hierarchiája miatt. A `catch` utasítások végrehajtása hasonló a `switch` utasításéhoz. Először azt ellenőrzi a rendszer, hogy a kivételobjektum típusa összeférhetője az `Exception_1` típussal (`Exception_1` típusú vagy annak leszármazottja). Ha összeférhető típusú akkor végrehajtódik az első `catch` utáni utasításblokk és utána a `finally`. Ha az első `catch` utasításban például az `Exception` (minden ellenőrzött kivétel ősosztálya) típusnév szerepel, akkor biztos, hogy a többit fölöslegesen írtuk, mert azok az ágak soha nem kerülnek végrehajtásra.

– A kivételek kétfélék lehetnek, ellenőrzöttek és nem ellenőrzöttek. Ellenőrzött kivétel például az `InterruptedException` és nem ellenőrzött az `ArrayIndexOutOfBoundsException`. Az ellenőrzött kivételeket kötelező lekezelni. Ez azt jelenti, hogy a fordításidőben ellenőrzi a fordító az ilyen kivételek lekezelését és kényszeríti a programozót ezek lekezelésére. Azok a metódusok, amelyekben ilyen ellenőrzött kivétel keletkezhet vagy lekezelik helyben a kivételt, ha ez lehetséges, vagy pedig továbbdobják őket. A kivétel továbbdobását a `throws` kulcsszóval kell jelezni. Például:

```
void f() throws CloneNotSupportedException{
   Szemely szemely = new Szemely("Ibolya", 1978);
   Szemely ujszemely =(Szemely)szemely.clone();
```

```
    }
```

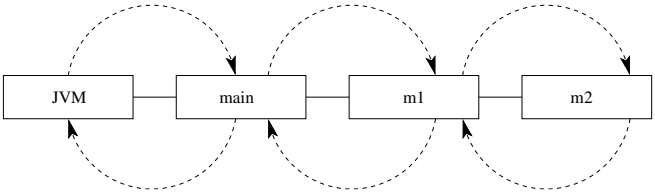
Tekintsük a következő kódrészletet:

```
public static void main( String args[] ){
   double s = 0;
   for( int i=0; i<args.length; ++i )
   try{
      s+=Double.parseDouble( args[ i ] );
   }
   catch( NumberFormatException e ){
      System.out.println(i+"."+args[i]);
   }
   System.out.println(''Osszeg: ''+s);
}
```

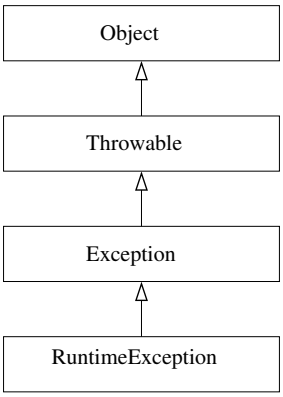
A fenti program egy nem ellenőrzött kivétel forrása lehet, éspedig akkor, ha valamelyik argumentuma nem numerikus. A kivételkezelés lehetőségét felhasználva viszont hibatűrővé tettük, ennek következtében programunk a numerikus argumentumok összegét fogja kiszámítani, kiírva az összes nem numerikus argumentumot a szabványos kimenetre. Kivételkezelés nélkül a program leállna az első nem numerikus argumentum feldolgozásakor.

10.4. Kivétel lekezelésének helye

A Java program végrehajtása úgy történik, hogy a Java virtuális gép egy folyamatként fut és ez hajtja végre a bájtkód utasításokat. Tehát a folyamat kódszegmensébe a JVM van betöltve, míg az adatszegmensbe a végrehajtandó bájtkód. A JVM meghívja a `main()` metódust. A `main()` metódus is meghívhat egy `m1()` metódust, illetve az `m1()` metódus egy `m2()` metódust. Ezt a hívási sorozatot szemlélteti a 10.2. ábra. Tételezzük fel, hogy a kivétel az `m2` metódusban történik. A Java futtatási környezet megszakítja a program normális végrehajtását és elkezdődik a legközelebbi kivételkezelő blokk keresése, ahová eljuttatja a kivételobjektumot. Ez a blokk lehet az `m2` metódusban vagy az `m1` metódusban,



10.2. ábra. Kivétel lekezelési lehetőségek



10.3. ábra. Kivételek hierarchiája

esetleg a `main` metódusban. Ha az `m2` metódus tartalmaz a kivétel lekezelésére utasítást, akkor itt fogja a kivételt lekezelni, kölönben a hívási sorrenddel ellentétes sorrendben fogja keresni a kivételkezelő blokkot. Ha egyáltalán nincs kivételkezelés, akkor a kivételobjektum a JVM szintjére fog visszajutni, amely leállítja a program futását.

10.5. Saját kivételosztály készítése

Saját típusainkhoz készíthetünk saját kivételtípust. Ezt természetesen osztályként fogjuk megvalósítani származtatás segítségével.

A 10.3. ábra a kivételek alaposztályait szemlélteti. Amennyiben az új kivételtípust a `RuntimeException` osztályból származtatjuk, a kivétel alapértel-

mezetten a nem ellenőrzött kivételek kategóriájába fog tartozni. Az ilyen fajta kivételek lekezelése a felhasználóra van bízva, azaz nem kötelező ezeket lekezelni. Amennyiben a kivételt az `Exception` osztályból származtatjuk, akkor ez az ellenőrzött kivételek részévé válik és lekezelését a futtatási környezet ellenőrzi.

Készítsünk egy verem programmodult és lássuk el ezt ellenőrzött kivételtípussal:

```
public class StackException extends Exception{
   public StackException(String text){
     super( text );
   }
}

public class Stack{
   private final int SIZE = 100;
   private Object st[];
   private int size;
   private int sp;

   public Stack(int size){
      this.size = size;
      this.st = new Object[ size ];
      this.sp = -1;
   }

   public void push( Object o ) throws StackException{
      if( sp== this.size-1 )
         throw new StackException("Tele verem");
      this.st[++this.sp ] = o;
   }

   public Object pop( ) throws StackException{
      if( sp== -1 ) throw new StackException("Ures verem");
      Object o = this.st[this.sp];
      this.sp--;
```

```
      return o;
   }

   public boolean isempty(){
      return this.sp == -1;
   }
}
```

A következő programrészlet példát ad az előző programban definiált verem osztály használatára.

```
public class Main{
   public static void main(String[] args ){
      Stack s = new Stack(10);
      for (int i=0; i<=10; ++i )
      try{
         s.push(new Integer(i));
      }
      catch( StackException e1){
         System.out.println( e1 );
      }
      while( ! s.isempty() ){
         try{
            System.out.println( (Integer)(s.pop()) );
         }
         catch(StackException e1){
            System.out.println( e1 );
         }
      }
  }
}
```

A kivételkezelés lehetőséget teremt a hibakezelő és az egyéb kód szétválasztására. Ennek következtében áttekinthetőbb lesz a kód.

10.6. Feladatok

1. Készítsen saját kivételosztályt az Arrays osztályhoz (5.4 alfejezet, 6. feladat). Ha nem rendezett tömbre akarjuk alkalmazni a bináris keresést, váltódjon ki kivétel.

2. Lássa el kivételosztállyal a Matrix osztályt (5.4 alfejezet, 5. feladat).

3. Lássa el kivételosztállyal a halmaz osztályt (9.10 alfejezet). Létező elem beszúrása-, illetve nem létező elem törlése esetén váltódjon ki kivétel.

4. Mi az eredménye a következő parancssornak: `java p1 10`

```
public class p1{
   public static void main( String args[] ){
      int i=0;
      try{
         i = Integer.parseInt( args[ 0 ] );
      }
      catch( NumberFormatException e ){ i= 12; }
      finally{ i++; }
      System.out.println("i="+i);
   }
}
```

5. Tudva, hogy az ArithmeticException a RuntimeException osztályból származik, mi lesz a következő program kimenete?

```
public class MyClass{
   public static void main( String [] args ){
      int k =0;
      try{
         int i = 5 / k;
      }
      catch( ArithmeticException e ){
         System.out.print( "1 ");
      }
```

```
      catch( RuntimeException e ){
         System.out.print( "2 ");
      }
      catch( Exception e ){
         System.out.print( "3 ");
      }
      finally{
         System.out.print( "4 ");
      }
      System.out.println( "5 ");
   }
}
```

a) 5

b) 1 4

c) 1 2 4

d) 1 4 5

e) 1 2 4 5

f) 3 5

6. Adott a következő program. Erre vonatkozóan melyek igazak a következő kijelentések közül?

```
public class Exceptions{
   public static void main( String args[] ){
      try{
         if( args.length == 0 ) return;
         System.out.println(args[ 0 ]);
      }
      finally{
         System.out.println("The end");
      }
   }
}
```

a) Argumentum nélküli futás esetében a programnak nincs kimenete

b) Argumentum nélküli futás esetében a program kimenete: The end

c) A program ArrayIndexOutOfBounds kivételt dob

d) Ha egyetlen argumentummal futtatjuk, akkor kiírja azt az egyetlen argumentumot

e) Ha egyetlen argumentummal futtatjuk, akkor kiírja azt az egyetlen argumentumot és a The end szöveget

7. Mi a hiba a következő kódban?

```
public class MyClass{
   public static void main( String args[] ){
      try{
         f();
      }
      finally{
         System.out.println("Done.");
      }
      catch( A e ){
         throw e;
      }
   }

   public static void f() throws B{
      throw new B();
   }
}

class A extends Throwable{}
class B extends A{}
```

a) A main metódusnak deklarálnia kellene, hogy B típusú kivételt dobhat

b) A finally blokkot kötelező a catch blokk után elhelyezni

c) A main metósban elhelyezett catch blokkban az A típusú kivétel helyett B típusút kellene specifikálni.

d) Egy try blokk után vagy csak catch blokkok vannak, vagy pedig finally blokk.

e) Az A osztály deklarációja illegális

11. FEJEZET

Beágyazott osztályok

A Java 1.1 verziója bevezette a beágyazott osztályok fogalmát. A beágyazás helye szerint megkülönböztetünk

- osztály szintű
- metódus szintű
- utasítás szintű

beágyazott osztályokat. Ha például csak egy példány létrehozása erejéig van szükségünk egy típusra, akkor utasítás szintű beágyazást fogunk használni. Ebben az esetben névtelen osztályt fogunk használni, hiszen fölösleges típusnevet hozzárendelni egy olyan típushoz, amelyre csak egy utasítás erejéig van szükségünk. Ha egy típus hatókörét egy metódusra akarjuk korlátozni, akkor a metódus szintjén kell azt bevezetni. Ha pedig az egész beágyazó osztályban láthatónak kell lennie, akkor osztály szintű beágyazást fogunk használni, vagyis a beágyazott osztályt a beágyazó osztály többi tagjainak szintjén fogjuk elhelyezni. A beágyazott osztályok segítségével a típusok közötti szoros kapcsolatot tudjuk kifejezni, ugyanakkor pedig lehetővé válik a típus elrejtése is. A beágyazott osztályok láthatóságára vonatkozóan ugyanazok a szabályok érvényesek, mint az osztály adat-, illetve metódus tagjaira. Ha egy beágyazott osztályt a `private` láthatósági módosítóval látjuk el, akkor ez a típus csak a beágyazó osztályban lesz látható és láthatatlan lesz a külvilág számára.

A beágyazásnak módja szerint megkülönböztetünk

– statikusan

– dinamikusan

beágyazott osztályokat.

A beágyazás statikus vonatkozása: a beágyazott osztályokra a definíciójuk helyén érvényes hozzáférési szabályok vonatkoznak, ennek következtében a beágyazott osztályok hozzáférhetnek a befoglaló osztályok privát tagjaihoz. A beágyazott osztály hatásköre viszont leszűkül a befoglaló osztályra, utasításblokkra, vagy egyetlen példányosítás pontjára.

A beágyazás dinamikus vonatkozása a beágyazott osztály példányainak és az ezt létrehozó példány kapcsolatát jelenti. Konkrétabban ez azt jelenti, hogy a beágyazott osztály példánya dinamikusan kapcsolódik az őt létrehozó aktuális példányhoz és ez a kapcsolat a példányosítást végrehajtó metódus elhagyása után is fennmarad. Ebben rejlik a beágyazás igazi ereje.

11.1. Tagosztályok

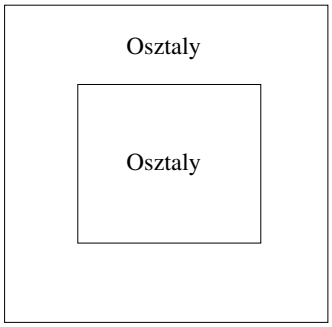
Ha egy osztályt egy másik osztályba ágyazunk be az osztálytagok szintjén, akkor ezt az osztályt `tagosztálynak` fogjuk nevezni, úgy, ahogy az adatokat adattagoknak, illetve a metódusokat metódustagoknak nevezzük. Ezt az esetet szemlélteti a 11.1. ábra.

```
Osztály=(Adattagok, Metódustagok, Osztálytagok)
```

A tagosztályokat is kétféleképpen lehet megadni: statikusan és nem statikusan. A statikus tagosztályokat a `static` minősítővel látjuk el. A kétféle tagosztály között az alapvető különbség az, hogy csak a nem statikus osztályok esetén érvényes a beágyazás dinamikus vonatkozása.

11.1.1. Statikus tagosztályok

Statikus tagosztályt abban az esetben érdemes deklaráni, ha egy segédosztályt el akarunk rejteni a külvilág elől és a beágyazott osztálynak nem kell ismernie a beágyazót, vagyis egyirányú kapcsolatot szeretnénk kialakítani a beágyazó



11.1. ábra. Egymásba ágyazott osztályok

11.2. ábra. Lista-Listaelem

és beágyazott között. A beágyazó természetesen hozzáfér a beágyazott privát mezőihez. A beágyazott osztálynak viszont csak a beágyazó osztály statikus tagjaihoz van hozzáférése.

Tekintsük a láncolt lista adatstruktúrában a `lista` és a `listaelem` kapcsolatát. Ebben az esetben a felhasználónak nem kell hozzáférnie a listaelemhez, hiszen a listának kell biztosítania az összes műveletet, amelyeken keresztül a lista használható. A listaelemet akarjuk elrejteni a külvilág elől, így a lista lesz a beágyazó, a listaelem pedig a beágyazott. A beágyazás következtében a lista hozzáfér a listaelem privát adattagjaihoz, de a listaelem nem fér hozzá a lista adattagjaihoz. A két osztály között egyirányú a kapcsolat, a lista használja a listaelemet. Ezt a kapcsolatot szemlélteti a 11.2. ábra.

A lista implementációt a lehető legegyszerűbben végezzük, a teljesség igénye nélkül. Ennek következtében csak minimális számú művelettel látjuk el a listát.

```
public class Lista{
      private Elem elso;

      private static class Elem{
```

```
      private Object adat;
      private Elem kovetkezo;

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

      public Elem( Object adat){
         this.adat = adat;
         this.kovetkezo= null;
      }
   }

   public void beszurelsonek( Object adat ){
      elso = new Elem( adat, elso);
   }

   public void listaz(){
      Elem seged = elso;
      while( seged != null ){
         System.out.println(seged.adat);
         seged = seged.kovetkezo;
      }
   }
}
```

A következő programrészlet szemlélteti az előző programban definiált lista használatát:

```
Lista l = new Lista();
for( int i=0; i<10; ++i )
   l.beszurelsonek( new Integer(i));
l.listaz();
```

11.1.2. Nem statikus tagosztályok

Abban az esetben, ha a két osztály között a kapcsolat kétirányú, vagyis a beágyazó ismeri a beágyazottat és fordítva, akkor ezt a kapcsolatot nem statikus tagosztállyal érdemes megvalósítani. Példaként szintén a lista osztályt fogjuk használni. Tételezzük fel, hogy egy általánosabb megvalósítását szeretnénk adni ennek az adatstruktúrának és külön akarjuk választani a lista bejáráshoz szükséges műveleteket a lista karbantartó (beszúrás, törlés, módosítás) műveleteitől. A bejárást egy külön osztállyal szeretnénk megvalósítani, amelynek az a feladata, hogy lehetővé tegye a listaelemek szekvenciális feldolgozását. Ennek érdekében biztosítania kell egy lekérdező műveletet, amely az aktuális listaelem adatmezőjét téríti vissza, illetve egy léptető műveletet, amelynek segítségével a lista következő elemére léptethetjük az aktuális elemmutatót. Ezzel a bejáróval (iterátorral) meg lehetne valósítani az előző részben megadott `listaz` műveletet is, de akár a listaelemek más feldolgozása is lehetővé válik. A két osztály kapcsolatát a 11.3. ábra szemlélteti. Ennek alapján egy listához akárhány iterátort rendelhetünk, de minden iterátor egy listát fog ismerni, pontosan azt, amelyik őt létrehozta, vagyis a bejárandó listát. A Java nyelv tartalmaz egy `Iterator` nevű interfészt, amely a `java.util` csomagban található és a Java beépített tárolói ezt az interfészt használják. Az iterátor egyben egy tervezési minta, amelyről részletesebben a [4], [7] könyvekben olvashatunk.

```
public interface Iterator{
   boolean hasNext();
   Object next();
   void remove();
}
```

Ez az interfész három műveletet tartalmaz. A `hasNext()` művelet ellenőrzi, hogy van-e még bejárandó elem, azaz az itetrátor osztály segítségével ábrázolt aktuális pozicíció évényes-e. A `next()` megvalósítja az aktuális elem visszatérítését és az aktuális pozíció léptetését is. Ezt a műveletet úgy kell megvalósítani, hogy lementjük a tároló aktuális pozíciójában levő elemet, léptetjük az aktuális pozíciót, majd visszatérítjük a lementett elemet. A `remove()` metódus pedig azt az elemet törli, amelyet az előző `next()` hívás visszatérített. Ez a művelet

11.3. ábra. Lista-Iterátor

lehetővé teszi a bejárás közbeni törlését a tárolt elemeknek, opcionális művelet, ezért mi nem fogjuk megvalósítani. A `Lista` és a `ListaIterator` osztályok implementációt a következő program szemlélteti.

```
import java.util.Iterator;

public class Lista{
   ...
   private class ListaIterator implements Iterator{
      //Beágyazott osztály hozzáfér a
      //beágyazó privát tagjaihoz
      private Elem akt = elso;

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

      public Object next(){
         Object adat = null;
         if( akt != null){
            adat = akt.adat;
            akt = akt.kovetkezo;
         }
         return adat;
      }

      public void remove(){
      }
   }
```

```
   public Iterator bejaras(){
      return new ListaIterator();
   }
   ...
}
```

Figyeljük meg, hogy a `ListaIterator` típust privát tagosztályként adtuk meg. Így ez a típus csak a `Lista` osztályon belül lesz használható. A megvalósításban csak a `bejaras()` művelet használja ezt a típust, amely példányosítja az osztályt és `Iterator` típusként téríti vissza a példányt (ez lehetséges, mert a `ListaIterator` megvalósítja az `Iterator` interfészt). Az `Iterator` típus viszont egy nyilvános típus, amelynek műveletein keresztül bejárhatóvá válik a mi listánk is. A következő kódrészlet a lista használatra ad példát:

```
Lista l = new Lista();
for( int i=0; i<10; ++i )
   l.beszurelsonek( new Integer(i));
Iterator it = l.bejaras();
while( it.hasNext())
  System.out.println( it.next());
```

A nem statikus tagosztályok csak példányszintű tagokat tartalmazhatnak és `final static` módosítójú tagokat. Tehát ezekben az osztályokban nem lehet osztályszintű tagokat deklarálni, hacsak nem látjuk el ezt a `final` módosítóval is.

11.2. Lokális és névtelen osztályok

Új típus megadható metóduson belül is. Ez utóbbi esetben a típus láthatósága az adott metódusra korlátozódik és ez bevezethető akár név nélkül is. Ha a típusra csak egy példányosítás erejéig van szükségünk akkor nem érdemes típusnevet rendelni hozzá. Azokat az osztályokat, amelyek metóduson belül vannak bevezetve és nevet is rendeltünk hozzájuk, lokális osztályoknak nevezzük, amelyeket pedig név nélkül vezetünk be, névtelen osztályoknak nevezzük. A nem

statikus tagosztályokat, a lokális osztályokat és a névtelen osztályokat együttesen belső osztályoknak is nevezzük.

Tekintsünk egy példát névtelen osztályra. Az előző részben ismertetett `Lista` osztály bejárását a `java.util.Iterator` interfésznek megfelelően végeztük. Bevezettük a `ListaIterator` típust, amelyet pontosan egyszer használtunk, éspedig a `bejaras()` műveletben egy példányosítás kapcsán. Ha egy típust csak egyszer kell használnunk, nem érdemes típusnevet rendelni hozzá. Ilyen esetben használhatjuk a névtelen osztályokat. Természetesen amikor a példányt létrehozzuk, akkor a `new` operátort használjuk. A `new` operátor után meg kell adnunk azt a típust, amelyet példányosítani akarunk. Névtelen osztály esetében ennek ősosztályát vagy pedig a névtelen osztály által implementált interfészt kell megadnunk típusnévként.

A `Lista` osztály megvalósítása névtelen iterátor típusú osztály segítségével:

```
import java.util.Iterator;

public class Lista{
   ...
   public Iterator bejaras(){
      return new Iterator(){
         //Névtelen osztály blokkja
         private Elem akt = elso;

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

         public Object next(){
            Object adat = null;
            if( akt != null){
               adat = akt.adat;
               akt = akt.kovetkezo;
            }
            return adat;
         }
```

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

A lokális és névtelen osztályok is csak példányszintű tagokat tartalmazhatnak és `final static` módosítójú tagokat. Ezekből az osztályokból is elérhető a beágyazó osztály minden tagja és a lokális változók közül csak a `final` módosítóval ellátottak. Amennyiben egy lokális vagy névtelen osztályt statikus blokkban deklarálunk, akkor csak a beágyazó osztály statikus tagjaihoz lesz hozzáférés, illetve a lokálisak közül csak a `final` módosítóval ellátottakhoz.

11.3. Feladatok

1. Készítsen el egy egydimenziós tömb ábrázolását és legfontosabb műveleteit megvalósító osztályt. Készítsen hozzá iterátor osztályt.

2. A halmaz osztályhoz (9.10 alfejezet) készítsen iterátor osztályt.

3. A Matrix osztályhoz (5.4 alfejezet) készítsen egy iterátor osztályt. Az iterátor biztosítsa a sorfolytonos bejárását a matrixnak.

4. Mi lesz a következő program eredménye?

```
public class Main{
   public static void main( String args[] ){
      Outer r = new Outer( 1000 );
      System.out.println( r.createInner().getValue() );
   }
}

class Outer{
   private int value;
```

```
   Outer( int value ){
      this.value = value;
   }

   class Inner{
      int getValue(){
         return value;
      }
   }

   Inner createInner(){
      return new Inner();
   }
}
```

a) Fordítási hiba, mert az Inner osztály nem deklarálható az Outer osztályban

b) Fordítási hiba, mert az Inner típus nem érhető el a Main osztályban

c) Fordítási hiba, mert a value mező nem érhető el az Inner osztályban

d) A program eredménye 1000

5. Mi lesz a következő program eredménye?

```
public class Main{
   public static void main( String args[] ){
      B.C o = new B().new C() ;
   }
}

class A{
   int value;
   A(int value) {this.value =value;}
}

class B extends A{
```

```
   int value = 1;
   B(){ super(2); }

   class C extends A{
      int value = 3;
      C(){
         super(4);
         System.out.println(B.this.value);
         System.out.println(C.this.value);
         System.out.println(super.value);
      }
   }
}
```

a) Fordítási hiba.

b) 2 3 4

c) 1 4 2

d) 1 3 4

e) 3 2 1

6. Válasszuk ki a helyes állításokat!

```
public class Outer{
   public void f(){
   }
   public class Inner{
      public void f(){
      }
   }

   public static void main( String args[] ){
      new Outer().new Inner().f();
   }
}
```

a) Az Inner osztály f metódus felülírja az Outer osztály f metódusát

b) Az Inner osztály f metódus túlterheli az Outer osztály f metódusát

c) Az Inner osztály f metódus elfedi az Outer osztály f metódusát

d) Az Inner osztály teljes neve Outer.Inner

7. Válasszuk ki a helyes állításokat!

a) A legfelső szintű osztályokat lehet statikusnak is deklarálni.

b) A legfelső szintű osztályokban deklarált tagosztályok lehetnek statikusak is.

c) A lokális osztályokat lehet statikusnak is deklarálni.

d) A név nélküli osztályokat lehet statikusnak is deklarálni.

e) Osztályokat nem lehet statikusnak deklarálni.

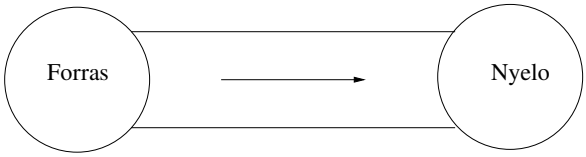
12. FEJEZET

Adatfolyamok (csatornák)

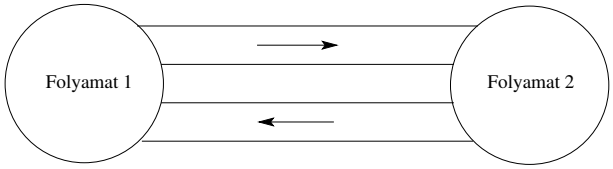
12.1. Bevezetés

Programjaink működésük közben kommunikálnak a külvilággal. Ez a kommunikáció vagy adatok olvasását jelenti vagy pedig azoknak írását. Egy programnak azt a képességét, amivel a külvilággal kommunikál, bemenet/kimenet (input/output:I/O) kezelésnek nevezzük. Rendkívül sokféle bemeneti és kimeneti eszköz létezik, amelyeket programjainkból el kell érnünk. Az eszközök sokfélesége indokolttá teszi, hogy ezeket programjainkból egységesen kezeljük, különben túl bonyolult lenne, ha minden eszközt különböző módon kellene kezelni. Az egységesítést az adatfolyamok bevezetésével oldották meg és az első ilyen rendszer kidolgozója Dennis Ritchie volt, aki 1984-ben megalkotta a Unix operációs rendszer adatfolyam alapú I/O rendszerét.

Az adatfolyam tulajdonképpen egy csatorna két entitás között, amely lehetővé teszi az adatok áramlását az egyik végponttól a másikig. A Java nyelvben a csatornák egyirányúak. Azt a végpontját a csatornának, amely oldalán adatokat írunk a csatornába, `adatforrásnak` nevezzük, illetve a másik végpontját `adatnyelőnek` (ld. 12.1. ábra). Az adatforrás feladata adatokat írni a csatornába, míg az adatnyelőnek adatokat kell kiolvasnia a csatornából. A csatorna az adatokat olyan sorrendben juttatja el az adatnyelőig, amilyen sorrenben ezeket az adatforrás a csatornába helyezte.



12.1. ábra. Adatfolyam



12.2. ábra. Két folyamat kommunikációja

Programjainkból kezelt csatornák egyik végpontja mindig maga a program (folyamat), így csak a másik végpont megadása szükséges. Ha például állományba akarunk írni adatokat, akkor a csatorna a folyamat szemszögéből nézve kimeneti csatorna lesz, amelynek az adatforrás végpontja maga a program, az adatnyelő végpontja pedig az állomány lesz. Fordított művelet esetében, amikor állományból kell adatokat olvasnunk, egy bemeneti csatornára lesz szükségünk, amely esetben az állomány tekintendő adatforrásnak és az adatokat kiolvasó program pedig adatnyelőnek. Ha két folyamatnak kell kölcsönösen kommunikálnia egymással, akkor ezt a Java nyelvben két adatfolyammal oldhatjuk meg, így lehetőséget teremtve mindkét folyamatnak az írásra és olvasásra is. Az az adatfolyam, amely az egyik folyamatnak kimeneti adatfolyama lesz, a másik folyamatnak bemeneti adatfolyama lesz. Ezt a fajta kommunikációt szemlélteti a 12.2. ábra.

Az adatfolyamatok többféleképpen osztályozhatók. Irányítottság alapján megkülönböztetünk bemeneti, illetve kimeneti adatfolyamokat. A csatornában szállított adat típusa alapján beszélhetünk bájtcsatornákról, illetve karaktercsatornákról. A karaktercsatornákban az adat egysége a 16 bites UNICODE karakter. Egy harmadik osztályozási szempont az ellátott feladat alapján alsó, illetve felső szintű csatornákat különböztet meg. Az alsó szintű csatornák az adatforrást vagy az adatnyelőt határozzák meg a csatorna irányítottságától függően. A felső szintű csatornák az adatátvitel módját határozzák meg. A felső szintű csatornák

kényelmi szempontokat is szolgálnak, hiszen ezek teszik lehetővé, hogy ne csak bájt-, illetve karaktersorozatokat írhassunk és olvashassunk, hanem tetszőleges adattípus írását és olvasását is lehetővé teszik.

A Java nyelv I/O lehetőségeit biztosító típusokat a `java.io` csomag tartalmazza. A legfontosabb típusokat az 12.3. ábrán szemléltetjük.

12.2. Standard adatfolyamok

A konzol kezelésére a Java nyelv három adatfolyamot biztosít. Ezeket az adatfolyamokat nem kell nyitni és zárni, mindkét művelet automatikusan történik. A standard bemenet a `System.in, InputStream` típusú adatfolyamán keresztül érhető el. A standard kimenet elérését a `System.out`, illetve a standard hibát a `System.err` biztosítja. Mindkettő `PrintStream` típusú. Amíg az `InputStream` egy alacsony szintű adatfolyam típus, addig a `PrintStream` egy magas szintű adatfolyam. Az `InputStream` a bájt alapú adatfolyamok ősosztálya, míg a `PrintStream` az `OutputStream` leszármazottja és különböző Java típusok írását biztosítja. Egy másik sajátossága a `PrintStream` osztálynak, hogy soha nem dob kivételt, helyette egy kapcsoló állapotát változtatja meg. A három standard csatornát a 12.4. ábra szemlélteti.

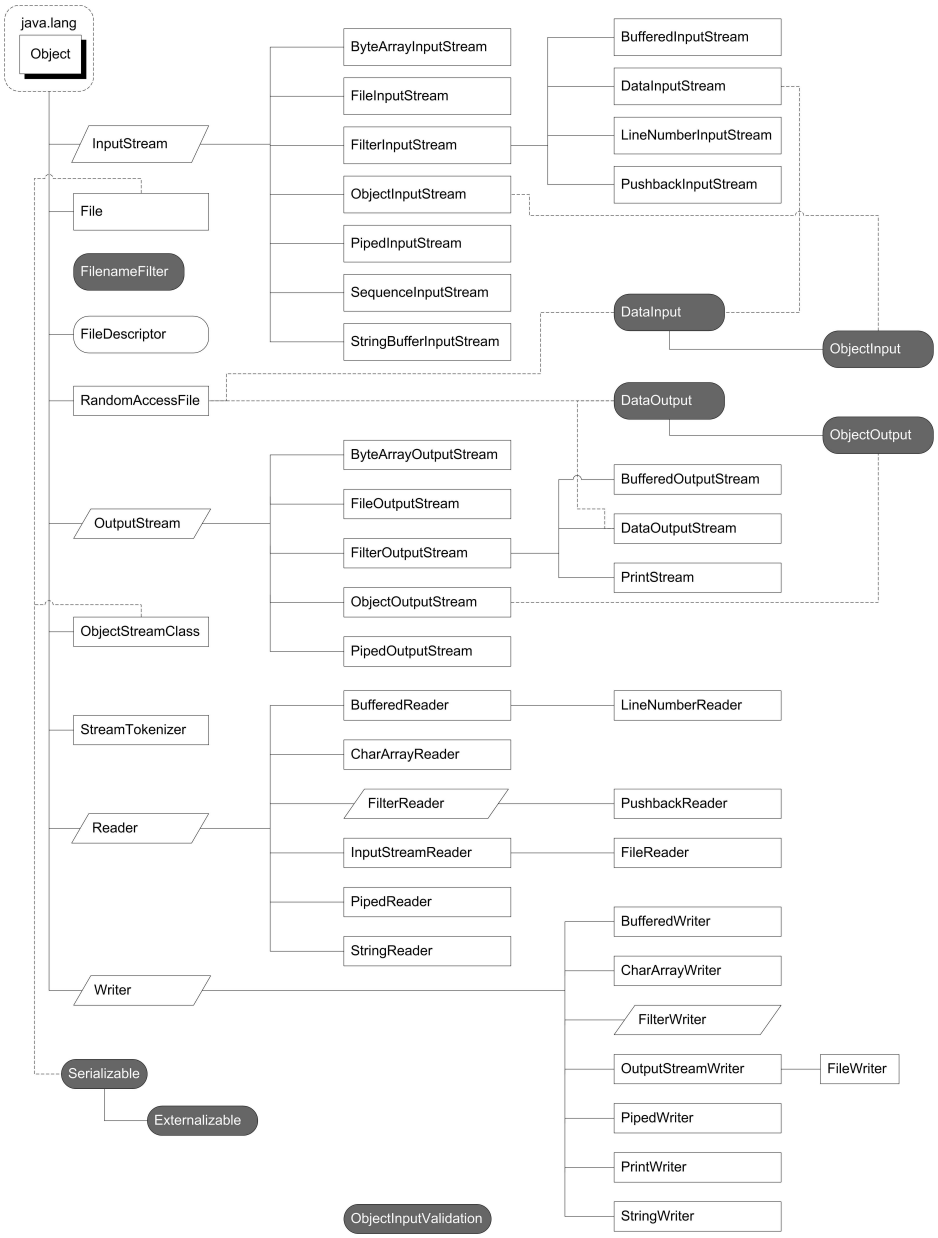
12.3. Műveletek adatfolyamokkal

12.3.1. Nyitás, lezárás, puffer ürítése

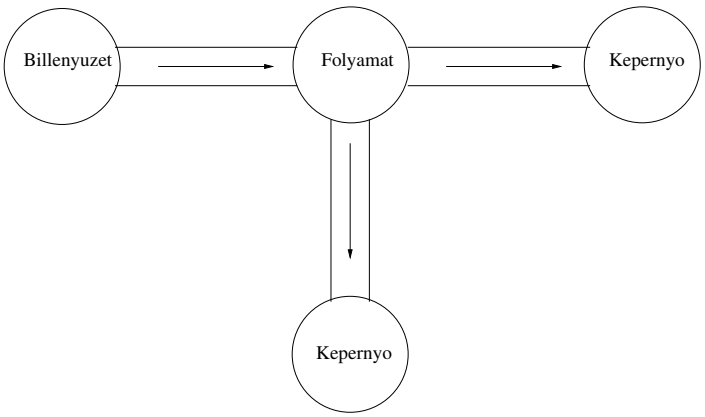
A három standard csatorna kivételével az összes többit használat előtt meg kell nyitni. Az adatfolyamok nyitását a konstruktor végzi. Természetesen lehetséges, hogy egy adatfolyam objektumot egyszer létrehozzunk, majd utána többször becsukjunk és kinyissunk. A nyitásra rendelkezésunkre áll az `open(String filename)` művelet, a bezárásra pedig a `close()` művelet.

Ha például egy állományba karaktereket akarunk írni, akkor ennek megnyitását a következőképpen végezhetjük:

```
FileWriter fout = new FileWriter("kimenet.txt");
```



12.3. ábra. A java.io csomag



12.4. ábra. Standard adatfolyamok

Kimeneti csatornák esetén a lezárás előtt automatikusan meghívódik a `flush()` művelet, amely a puffer tartalmát beírja az adatfolyamba. A `flush()` művelet szükség esetén expliciten is hívható. Adatfolyam lezárása után I/O művelet már nem végezhető az adatfolyammal, ellenkező esetben `IOException` kivétel váltódik ki.

12.3.2. Írás

Bármely bájtszervezésű kimeneti csatornára használható a `write` metódus három túlterhelt változata.

```
void write ( int i ) throws IOException;
void write ( byte b[] ) throws IOException;
void write ( byte b[], int from, int length) throws IOException;
```

Az első metódus az `i` egész típusú paraméter alsó bájtját írja ki. A második alak egy bájttömb kiírására szolgál, míg a harmadik alak lehetővé teszi egy bájttömb valamely pozíciójától valamely hossznyi bájtsorozat beírását adatfolyamba. Tekintsük a következő példát a metódusok használatára, amelyben mindhárom metódus segítségével rendre ugyanazt a tartalmat egy adatfolyamba írjuk. A kódrészletből hiányzik a kivételkezelés!

```
byte b[] = new byte[ 1024];
//x feltöltése
FileOutputStream fout = new FileOutputStream( "bytes.dat");
fout.write( x );
fout.write( x, 0, x.length );
for( int i=0; i<x.length; ++i )
   fout.write( x[ i ] );
```

Karaktercsatornák esetében hasonló metódusok állnak rendelkezésünkre. Ebben az esetben viszont a paraméter `char[]` vagy `String` típusú.

```
void write ( int i ) throws IOException;
void write ( char t[] ) throws IOException;
void write ( char t[], int from, int length) throws IOException;
void write ( String str ) throws IOException;
void write ( String str, int from, int length) throws IOException;
```

12.3.3. Olvasás

Az olvasást csak a bájtszervezésű csatornákra szemléltetjük. Az íráshoz hasonlóan, olvasásra is három túlterhelt alakja lesz a `read` metódusnak.

```
int read() throws IOException;
int read( byte b[] ) throws IOException;
int read( byte b[], int from, int length ) throws IOException;
```

Az első alak egy bájtot olvas és ezt a visszatérített egész alsó bájtjában helyezi el. Adatfolyam vége esetén `-1` lesz a visszatérített érték. A második alak, a tömb hosszának megfelelő számú bájtot próbál beolvasni, míg a harmadik alak `length` darab bájt beolvasását próbálja meg. Amennyiben vége van az adatfolyamnak, mindkét alak a sikeresen beolvasott bájtok számát téríti vissza. Olvasásnál megkülönböztetünk `üres csatornát`. Ilyen esetben blokkolódik az olvasási művelet egészen addig, amíg új adatok nem érkeznek. A `csatorna vége` állapot akkor következik be, ha például állományhoz csatlakoztatott csatorna esetében állomány vége van, illetve hálózati alkalmazások esetében az egyik fél lezárja a csatornát.

Az eddigi ismereteink alapján már elkészíthetünk egy általános célú másoló függvényt, amely bárhonnan bárhová átmásol egy bájtsorozatot. A függvény használatára pedig két példát is adunk. Az első a standard bementről a standard kimenetre másol, a második állományok másolását fogja megvalósítani:

```
import java.io.*;

public class copy{
   public static void mycopy( InputStream in, OutputStream out )
                                                 throws IOException{
      byte buffer[] = new byte[1024];
      int bytes;
      while( (bytes = in.read( buffer ) ) >0 )
         out.write( buffer, 0, bytes );
      out.flush();
   }

   public static void main( String args[] ){
      System.out.println("Stdin->Stdout");
      try{
         mycopy( System.in, System.out);
      }
      catch( IOException e ){
         System.out.println("I/O error");
      }
      System.out.println("File copy");
      try{
         FileInputStream fin =
            new FileInputStream("copy.java");
         FileOutputStream fout=
            new FileOutputStream("masolat.java");
         mycopy( fin, fout);
      }
      catch( FileNotFoundException e1 ){
         System.out.println("File not found"+e1 );
```

```
      }
      catch( IOException e ){
         System.out.println("I/O error");
      }
   }
}
```

12.3.4. Rendelkezésre álló bájtok

Az `InputStream` osztály lehetővé teszi a rendelkezésre álló bájtok lekérdezését egy csatorna objektumra. Ezt a következő függvény biztosítja:

```
public int available() throws IOException;
```

A függvény felhasználásával készítsünk programot, amely egy állomány méretét lekérdezi:

```
import java.io.*;

public class FileSize{
   public static void main( String args[] ){
      if( args.length != 1 ){
         System.out.println(
            "Hasznalat java FileSize allomanynev");
         System.exit( 0 );
      }
      FileInputStream fin=null;
      try{
        fin = new FileInputStream( args[0]);
      }
      catch( FileNotFoundException e ){
         System.out.println("Nem letezo allomany");
         System.exit( 0 );
      }
      int size=0;
      try{
```

```
         size = fin.available();
      }
      catch( IOException e1 ){
      }
      System.out.println("Meret:"+size+" bájt");
   }
}
```

12.4. Alsó szintű adatfolyamok

12.4.1. Állományhoz kapcsolt adatfolyamok

Alsó szintű adatfolyamok esetén azt határozzuk meg, hogy az adatok milyen fizikai adattárolóra képződjenek le. Ha állományhoz kapcsolt adatfolyamról van szó, akkor az adatfolyam létrehozásakor meg kell adnunk egy hivatkozást az állományra. Ez többféleképpen is megvalósítható. Az állománynak megadhatjuk a nevét vagy egy `File` típusú objektumot is átadhatunk paraméterként. Bájtszervezésű, állományhoz kapcsolt adatfolyamokhoz `FileInputStream` és `FileOutputStream` osztályok használhatók. Karakterszervezésű állományok esetében pedig a `FileReader` és a `FileWriter` osztályok. Bemeneti fájlfolyam megnyitása esetében, ha a megnyitandó állomány nem létezik, `FileNotFoundException` kivétel keletkezik. Ha pedig nincs meg a szükséges jogosultságunk az elvégzendő művelethez, akkor `SecurityException` fog keletkezni.

```
import java.io.*;
public class Pelda{
   public static void main( String args[] ){
      FileWriter fout=null;
      try{
        fout= new FileWriter("out.txt");
      }
      catch( IOException e1 ){}
      char t[]={'a','l','m','a','f','a'};
      try{
```

```
         fout.write( t );
         fout.close();
      }
      catch( IOException e2){}
   }
}
```

12.4.2. Bájt vagy karaktertömbhöz illesztett csatorna

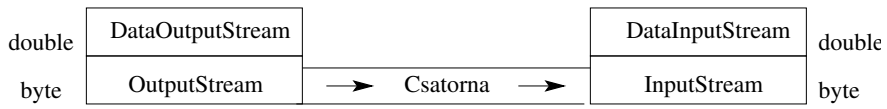
A csatornát kapcsolhatjuk egy bájttömbhöz is bájtszervezésű csatornák esetében, illetve karaktertömbhöz karaketerszervezésű csatornáknál. Így bármely bájttömböt vagy karaktertömböt ugyanúgy kezelhetünk, mintha állomány lenne. Tekintsünk egy példát karaktertömb ilyen jellegű feldolgozására:

```
char t[]={ 'V','a','k','á','c','i','ó'};
CharArrayReader in = new CharArrayReader( t );
int c;
while(( c=in.read()) != -1 )
   System.out.println( c+" ");
```

12.4.3. String objektumhoz illesztett karaktercsatorna

Karaktercsatornát `String` illetve `StringBuffer` objektumokhoz is csatlakoztathatunk. A következő példa egy karakterlánc feldolgozását szemlélteti.

```
String str="hello world";
StringReader strin = new StringReader( str );
int c;
try{
   while( (c = strin.read()) != -1 )
      System.out.println( c+ " : "+(char)c=);
}
catch( IOException e2){
}
```



12.5. ábra. Szűrő típusú csatornák

12.5. Felső szintű adatfolyamok (szűrők)

A szűrő típusú csatornaosztályokkal nem a csatorna végpontját határozzuk meg, hanem azt, hogy `hogyan` akarunk írni, illetve olvasni. Ezekkel az osztályokkal egy meglévő csatornát ruházhatunk fel különleges tulajdonságokkal. Az eddig használt csatornákat csak bájt-, illetve karaktersorozatként írhattuk, illetve olvashattuk. Ez hasonló a C nyelv alacsony szintű fájlkezeléséhez. Bizonyos esetekben kényelmetlen lenne ilyen egységekben kezelni az adatainkat. Sokszor numerikus adatokat, például valós számokat akarunk írni, illetve olvasni csatornákba, csatornákból. Ezt megtehetjük a `DataInputStream` meg a `DataOutputStream` szűrő típusú csatornák használatával. A csatornák egymásra helyezését szemlélteti a 12.5. ábra. A `DataInputStream` osztály különböző Java típusok olvasására tartalmaz műveleteket, a `DataOutputStream` pedig ezen Java típusok írására. Például a valósakat `writeFloat` és a `writeDouble` metódusokkal írhatjuk. Ugyanezen típusok olvasását a `readFloat` és `readDouble` műveletekkel végezhetjük.

12.5.1. Valós számok írása és olvasása

```
DataOutputStream dout = null;
try{
   dout = new DataOutputStream(
      new FileOutputStream("valosak.dat");
}
catch( Exception e1 ){
   System.out.println("Fajl nyitasi problema";
}
for( int i=0; i<10; ++i )
try{
```

```
   dout.writeDouble( Math.random() {*} 100 );}
}
catch( IOException e2){
DataInputStream din = null;
try{
   din = new DataInputStream(
   new FileInputStream("valosak.dat"));
}
catch( FileNotFoundException e3 ){
   System.out.println("Fajl nyitasi problema");
}
for( int i=0; i<10; ++i )
try{
   System.out.println( din.readDouble());
}
catch( IOException e4){
}
```

12.5.2. Szöveges állományok írása és olvasása soronként

A következő kódrészlet szöveges állomány soronkénti feldolgozását szemlélteti. A kivételkezelést szándékosan elhagytuk!

```
BufferedReader br = new BufferedReader(
               new FileReader("int.txt"));
String s;
int i = 1;
while( ( s = br.readLine()) != null ){
   System.out.println(i+". "+s);
   ++i;
}
```

Szöveges állomány írására használhatjuk a `PrintWriter` osztályt. Ennek `print` és `println` metódusaival tetszőleges típusú adatokat írhatunk szöveges állományba.

```
PrintWriter pw = new PrintWriter( new FileWriter("out.txt"));
pw.println("Hello"+(12*3));
```

12.6. Bájt típusú adatfolyam átalakítása karakter típusúvá

Az `InputStreamReader` és az `OutputStreamWriter` osztályok megteremtik a kapcsolatot a karakterszervezésű és a bájtszervezésű csatornák között. Feladatuk a bájtsorozatok karaktersorozatokká alakítása. A konstruktoroknak második paraméterként megadhatjuk, hogy milyen karakterkódolási szabványt szeretnénk használni. Például a 8859_2 az ISO Latin-2 karakterkódolási szabvány és tartalmazza a magyar karaktereket is.

Tekintsük a következő két példát az adatfolyamok használatára. Az első az `Ő` karaktert írja ki, míg a második a szabványos bemenetről olvas be adatot egy egész számot. Mivel a `System.in` típusa `InputStream` és mi sorvégjelig szeretnénk olvasni, olyan felső szintű adatfolyamot kell használnunk, amely biztosítja a sorvégjelig való olvasást. Ez a `BufferedReader` osztály. Az egyetlen probléma, hogy csak azonos szervezésű adatfolyamokat lehet egymásra helyezni. Ahhoz, hogy a `BufferedReader` osztályt rá tudjuk helyezni az `InputStream` típusúra, az utóbbit át kell alakítanunk karakter szervezésűvé. Ezt a célt szolgálja az `InputStreamReader` csatorna.

1. Példa

```
try{
   Writer w = new OutputStreamWriter(System.out);
   w.write(336);
   w.close();
}
catch( UnsupportedEncodingException e1 ){}
catch( IOException e2 ){}
```

2. Példa

```
BufferedReader bin = new BufferedReader(
```

```
      new InputStreamReader( System.in ));
try{
   String line = bin.readLine();
   int i = Integer.parseInt( line );
   System.out.println( i );
}
catch( IOException e ){}
```

12.7. Objektumok szerializációja

Objektumaink élettartama a Java virtuális gép futásidejére van korlátozva. A Java virtuális gép leálltával elhal minden objektum is. Jó lenne, ha objektumaink élettartama meghaladhatná a virtuális gép élettartamát. Erre teremt lehetőséget az objektumok szerializációja.

Objektum szerializációja az objektum állapotának elmentését jelenti bájtsorozatként, illetve a bájtsorozatból a lementett állapotú objektum visszaállítását. Azon objektumokat, amelyek életciklusa meghaladja a virtuális gép életciklusát, perzisztens objektumoknak nevezzük.

Ahhoz, hogy egy objektumot szerializáljunk, az objektum típusának ezt lehetővé kell tennie. Ezt pedig úgy érhetjük el, ha az objektum osztálya megvalósítja a `Serializable` interfészt. Ez az interfész nem tartalmaz metódusdeklarációkat, úgy tekinthető, mint egy kapcsoló, amely lehetővé teszi, hogy az osztály példányai bájtsorozatokká alakíthatók legyenek.

Tekintsük a következő programot, amelyben egy `Dátum` típust deklarálunk úgy, hogy lehetővé tesszük a típus példányainak szerializációját. Ezután létrehozunk egy dátumot, amelyet szerializálunk, ezt követően pedig visszaolvassuk a bájtsorozatból, visszaállítva az objektumot. Objektum szerializációját a `writeObject` metódussal végezzük, visszaállítását pedig a `readObject` metódussal.

```
import java.io.*;

class Datum implements Serializable{
   static final long serialVersionUID = 6;
```

```
   private int ev, ho, nap;

   public Datum( int ev, int ho, int nap ){
      this.ev = ev;
      this.ho = ho;
      this.nap = nap;
   }

   public String toString(){
      return ev+"-"+ho+"-"+nap;
   }
}
```

A `Datum` osztályt elláthatjuk egy szerializálás azonosítóval. Ezt az azonosítót mindaddig változatlanul hagyjuk, amíg nem hajtunk végre lényeges módosítást az osztályon. Ilyen nem lényeges módosítás például egy új metódus felvétele az osztályba, vagy valamely metódus szignatúrájának megváltoztatása. Ezek a módosítások nem befolyásolják az objektum állapotát. Amennyiben már új adatmezőt veszünk fel az osztályba, meg kell változtatnuk a szerializációs azonosítóját is (`serialVersionUID`).

Példa objektumok szerializációjára:

```
Datum d = new Datum( 2006, 4, 24 );
ObjectOutputStream dout = null;
try{
   dout = new ObjectOutputStream(
      new FileOutputStream("out.txt"));
   dout.writeObject( d );
   dout.close();
}
catch( FileNotFoundException e1 ){}
catch( IOException e2 ){}

ObjectInputStream din = null;
```

```
try{
   din = new ObjectInputStream(
     new FileInputStream("out.txt"));
   System.out.println( (Datum) din.readObject() );
   din.close();}
}
catch( ClassNotFoundException e1 ){}
catch( IOException e2 ){}
```

A továbbiakban a következő három kérdésre próbálunk válaszolni:

1. Mi történik szerializációnál az osztály szintű tagokkal?
2. Hogyan lehet megakadályozni egy példány szintű adatmező bájtsorozattá alakítását?
3. Az előző részben példát adtunk az automatikus szerializációra. Hogyan lehet megvalósítani az objektum kézi szerializációját?

1. A statikus osztálytagok egyáltalán nem mentődnek szerializációkor. Olvasáskor, az objektum felépítésekor, ezen mezők a típusnak megfelelő implicit kezdőértéket kapnak.
2. Egy példányszintű adatmező kiíratását megakadályozhatjuk a `transient` módosító használatával. Ez azt jelenti, hogy ez a mező nem maradandó, nem perzisztens. A szerializáció úgy történik, hogy elmentődik az összes nem statikus de perzisztens adatmező. Az adatmezők közül is először az örökölt mezők mentődnek, majd a sajátok, deklarálási sorrendben. Ha az objektumnak van hivatkozás típusú adatmezője, akkor a hivatkozott objektumra is meghívódik a mentési mechanizmus. Ha a hivatkozott objektum már ki van mentve (más objektum is tartalmazott rá hivatkozást), akkor nem mentődik ki még egyszer.
3. Ha nem megfelelő az automatikus szerializációs mechanizmus, akkor saját szerializációs mechanizmust adhatunk meg az adott típushoz,

a `readObject()` és a `writeObject()` megadásával. Amennyiben egy osztály meghatározza ezeket a metódusokat, mentéskor és visszaállításkor ezek fognak meghívódni. Sokszor, amikor ezt a két műveletet definiáljuk egy osztályhoz, csak kiegészíteni szeretnénk az alapértelmezett mechanizmust. Ezt pedig megtehetjük az objektumcsatornák `defaultReadObject()`, illetve `defaultWriteObject()` metódusok hívásával. Ezeket a metódusokat csak `readObject`, illetve a `writeObject` metódusokban szabad meghívni, ellenkező esetben kivétel váltódik ki. Vigyáznunk kell arra az esetre is, amikor egy olyan osztályt teszünk szerializálhatóvá, amelynek bázisosztálya nem rendelkezik ezzel a tulajdonsággal. Ebben az esetben nem mentődnek el az örökölt adattagok.

Ha nem megfelelő az alapértelmezett szerializáció egy adott típusra, akkor a típusban meg kell adnunk a következő két metódust:

```
private void writeObject(
   ObjectOutputStream out ) throws IOException;

private void readObject(
   ObjectInputStream in ) throws IOException,
                      ClassNotFoundException;
```

Vegyük észre, hogy mindkét metódus privát metódus. Ezeket a metódusokat a Java virtuális gép fogja meghívni. Egy adott osztály privát metódusait osztályon kívül egyedül a Java virtuális gép hívhatja meg. Ezekben a metódusokban meghívjuk az alapértelmezett szerializációs mechanizmust, majd ezt kiegészítjük egyéb műveletekkel is. Így oldható meg például a jelszó típusú mezők mentése titkosítva, vagy ezek kiolvasása, elvégezve a titkosításnak megfelelő dekódolást. Az előző `Datum` típus esetében ez a következő módon nézne ki:

```
class Datum implements Serializable{
   static final long serialVersionUID = 1L;
   private int ev, ho, nap;

   public Datum( int ev, int ho, int nap ){
      this.ev = ev;
```

```
      this.ho = ho;
      this.nap = nap;
   }

   public String toString(){
      return ev+"-" +ho+"-"+nap;
   }

   private void writeObject( ObjectOutputStream out )
                                    throws IOException{
      out.defaultWriteObject();
      System.out.println("Extra iras muvelet");
   }

   private void readObject( ObjectInputStream in )
         throws IOException, ClassNotFoundException{
      in.defaultReadObject();
      System.out.println("Extra olvasas muvelet");
   }
}
```

A következő program létrehoz egy Datum típusú objektumot, szerializálja, utána pedig visszaolvassa és kiíratja a standard kimenetre.

```
import java.io.*;

public class Szerializacio{

   public static void main( String args[] ){
      Datum d = new Datum(2006,4,25);
      try{
         ObjectOutputStream out = new ObjectOutputStream(
            new FileOutputStream("obj.dat"));
         out.writeObject( d );
         out.close();
```

```
         ObjectInputStream in = new ObjectInputStream(
            new FileInputStream("obj.dat"));
         d = ( Datum )in.readObject();
         System.out.println( d );
      }
      catch( IOException e1 ){
         e1.printStackTrace();
      }
      catch( ClassNotFoundException e2 ){
         e2.printStackTrace();
      }
   }
}
```

12.7.1. Saját szerializációs protokoll készítése

Amennyiben teljesen kézbe akarjuk venni a szerializációs mechanizmust, az `Externizable` interfészt kell implementálnunk, amely két metódust tartalmaz:

```
public void writeExternal( ObjectOutput out )
                          throws IOException;
public void readExternal( ObjectInput in )
  throws IOException, ClassNotFoundException;
```

Ha ezt az utat választjuk, akkor egyben a legnehezebbet is választjuk. Ennek ellenére ez az út biztosítja a legnagyobb szabadságot a szerializáció megvalósításában. Ezen metódusok hívása is automatikusan történik a `readObject` és `writeObject` metódusokon keresztül. A programozónak tehát nem feladata ezen metódusok explicit hívása.

12.8. Közvetlen elérésű fájlok

A közvetlen elérésű fájlot megvalósító osztály a `RandomAccessFile`. Ez a típus gyakorlatilag egy külső tárolón elhelyezett bájttömbként fogható fel. Indexelni nem lehet, de létezik egy `seek` pozícionáló művelet, amellyel az állomány

bármely pozíciójából olvashatunk, illetve írhatunk. Mivel ez az osztály megvalósítja a `DataInput` és a `DataOutput` interfészt, tetszőleges adattípus írása és olvasása válik lehetővé. Egy ilyen típusú állomány megnyitásakor nemcsak a fájlrendszer adott állományát kell azonosítanunk, hanem a konstruktor második paraméterében a megnyitás módját is jeleznünk kell. Ez pedig egy `String` típusú paraméter, amelynek értéke `"r"` vagy `"w"` vagy `"rw"` lehet attól függően, hogy olvasásra, írásra vagy mindkettőre akarjuk megnyitni állományunkat. Ezen osztály használatát a következő program szemlélteti.

```
import java.io.*;

public class RAF{
   public static void main( String args[] ){
      RandomAccessFile f=null;
      try{
         f = new RandomAccessFile("adatok.dat","rw");
      }
      catch( FileNotFoundException e1 ){}
      try{
         for( int i=0; i<10; ++i )
            f.writeDouble( Math.random()*100 );
            f.seek(0L);
      }
      catch( IOException e2){
         e2.printStackTrace();
      }
      do{
         try{
            System.out.println( f.readDouble());
         }
         catch( EOFException e3 ){
            System.exit( 0 );
         }
         catch( IOException e4 ){}
```

```
      }while( true );
   }
}
```

12.9. Feladatok

1. Mely osztályok absztraktok az alábbiak közül?

   a) InputStream

   b) PrintStream

   c) Reader

   d) FileInputStream

   e) FileWriter

2. Mi lesz a következő kódrészlet kimenete?

```
FileOutputStream fos = new FileOutputStream("bytes.dat");
for( byte b=10; b<50; b++)
   fos.write( b );
fos.close();
RandomAccessFile raf = new RandomAccessFile("bytes.dat", "r");
raf.seek(10);
int i =raf.read()'
raf.close();
System.out.println("i= "+i);
```

   a) A kimenet i= 30

   b) A kimenet i= 20

   c) A kimenet i= 10

   d) Kivétel keletkezik

3. Egy állományt az alábbi kódrészlettel hozunk létre. Az állomány tartalmának kiolvasását mely megoldás teszi lehetővé?

```
File fos = new FileOutputStream("file");
DataOutputStream dos = new DataOutputStream( fos );
for( int i=0; i<500; ++i )
   dos.writeInt( i );
```

a) FileInputStream->DataInputStream, readInt()

b) FileReader, readInt()

c) PipedInputStream, readInt()

d) RandomAccessFile, readInt()

e) FileReader->DataInputStream, readInt()

4. Mi történik a következő program fordítása és futtatása esetén?

```
 1. import java.io.*;
 2.
 3. public class IO{
 4.    public static void main( String args[] ){
 5.       try{
 6.          FileOutputStream fos = new FileOutputStream("ser.dat");
 7.          ObjectOutputStream oos = new ObjectOutputStream( fos );
 8.          oos.writeObject( new Object() );
 9.          oos.close();
10.          fos.close();
11.       }
12.       catch( Exception e ){
13.         System.out.println( e );
14.       }
15.   }
16.}
```

a) Fordítási hiba a 8. sorban

b) Futásidejű kivétel a 8. sor miatt

c) Futásidejű kivétel a 9. sor miatt

d) Helyes végrehatódás

5. Adott a következő osztály. Szerializáláskor mely adattagok kerülnek lementésre?

```
public class X implements java.io.Serializable{
   public    String f1;
   private   String f2;
   static    String f3;
   transient String f4;
   volatile  String f5;
   ...
}
```

   a) f1

   b) f2

   c) f3

   d) f4

   e) f5

6. Milyen típusú argumentuma lehet a DataInputStream osztály konstruktorának?

   a) File

   b) FileReader

   c) FileInputStream

   d) RandomAccessFile

13. FEJEZET

Grafikus felhasználói felületek készítése

Az interaktív programok jellemzője, hogy futás közben biztosítják a kommunikációt a felhasználó és a futó program között. Természetesen nem minden programnak kell interaktívnak lennie, hiszen egy műveletsorozat elvégzése például elképzelhető úgy is, hogy a bemenő adatokat állományból olvassa a program, míg a kimenőket szintén állományba írja. Egy másik példa nem interaktív programra a szerver típusú alkalmazások, amelyek nem felhasználóval, hanem kliens programokkal kommunikálnak. A programok nagy részénél viszont interaktív kommunikációra van szükség (szövegszerkesztő, táblázatkezelő, levelezőprogram). Az interaktivitás megvalósítható szöveges üzemmódban is, de leggyakrabban grafikus felhasználói felületen keresztül történik.

A grafikus felhasználói felület (GUI-Graphical User Interface) szerepe a bevitel és kivitel biztosítása esztétikus formában. A grafikus felhasználói felületet grafikus komponensek alkotják, amelyeket szerepük alapján csoportosíthatunk beviteli-, vezérlő- és megjelenítő komponensekre.

A grafikus interfészek létrehozásához szükséges típusokat a `java.awt` csomag tartalmazza. Az `AWT (Abstract Window Toolkit)` Absztrakt Ablakozó Készletet azért nevezzük absztraktank, mert a felületelemek konkrét megjelenítését nem ez a készlet, hanem az operációs rendszer ablakozó rendszere végzi. Ennek következményeként ez a készlet csak azokat a komponenseket tartalmazza, amelyeket

minden egyes ablakozó rendszer meg tud jeleníteni, így a készlet szegényes. A másik hiányossága ennek a megközelítésnek, hogy az interfész alakja a futattó környezettől fog függeni, vagyis nem független a platformtól. Ez ellentmond a Java platformfüggetlen alapelvének. A probléma megoldására létrehozták a `Swing` nevű grafikus programkönyvtárt (`javax.swing` csomag). A `Swing` csomag az `AWT`-re épül, de a komponenseket Javaban írták és a komponensek gondoskodnak a saját megjelenítésükről, ezáltal függetlenítve magukat a grafikus felületet megjelenítő operációs- és ablakozó rendszertől.

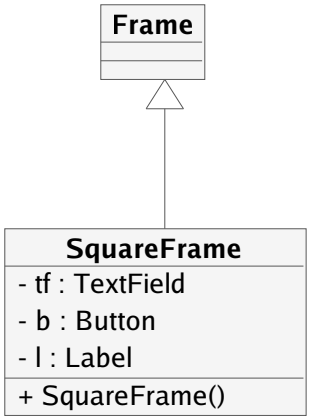
A felülettel kapcsolatosan három probléma merül fel: a megépítés, használat és bezárás.

13.1. Felületek megépítése, használata és bezárása

13.1.1. Felület megépítése

Célunk egy egyszerű primitív felület megépítése, amely egy ablak felületén megjelenít egy szövegdobozt, egy nyomógombot és egy címkét. A szövegmező lesz a beviteli komponens. Miután beírtuk a számot, megnyomjuk a nyomógombót, amelynek hatására a címkén megjelenik a begépelt szám négyzete. A címke egy megjelenítő komponens, és mivel a nyomógomb segítségével vezéreltük a programot, ez lesz a programunkban a vezérlő komponens. Az ablak lesz a konténer, mert ez fogja tartalmazni a komponenseinket. A megvalósításhoz szükségünk van egy új osztályra, amelyben megadjuk a mi sajátos ablakunkra vonatkozó részleteket. Elsősorban ez az új osztály egy ablakot fog reprezentálni, így a létező keretes ablak osztályból fogjuk származtatni (`Frame`). A komponensek osztályai rendre `TextField` (szövegdoboz), `Button` (nyomógomb), `Label` (címke). Az osztálydiagramot a 13.1. ábra szemlélteti.

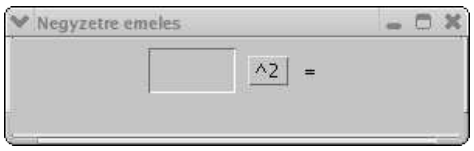
A komponenseket kétféleképpen helyezhetjük el a felületen, kézi elrendezéssel vagy automatikus elrendezéssel. Mi az automatikus elrendezést ismertetjük, mert ez az egyszerűbb. Ebben az esetben szükségünk van egy elrendezésszervező objektumra, amelynek felelőssége a komponensek elrendezése. A `java.awt` csomagban többféle elrendezésszervező létezik. Mi a legegyszerűbbet fogjuk most használni, éspedig a komponensek egymás utáni, folytonos elrendezését. Ezt biztosítja a



13.1. ábra. Square_Frame osztálydiagram

`FlowLayout` típusú elrendezésszervező. Az elrendezésszervezőknek nemcsak az a feladatuk, hogy a kezdeti elrendezést biztosítsák, hanem az is, hogy futásidőben a megváltozott méretű konténerben újrarendezzék a komponenseket. Minden elrendezésszervezőnek saját stratégiája van a komponensek elhelyezését illetően. Minden egyes komponensnek van egy ideális mérete, amelyet a komponenstől le lehet kérdezni. A 13.1. ábrán levő osztálydiagramnak megfelelő kódot a következő program szemlélteti.

```
import java.awt.*;
public class SquareFrame extends Frame{
   private TextField tf;
   private Button b;
   private Label l;

   public SquareFrame(){
      this.setTitle("Negyzetre emeles");
      this.setLayout( new FlowLayout());
      tf = new TextField( 6 ); this.add( tf );
```



13.2. ábra. Az első grafikus felületünk

```
      b = new Button("^2"); this.add(b);
      l = new Label("=      "); this.add(l);
   }
}
```

Most pedig elkészítjük az első példányunkat az újonnan definiált osztályból és megjelenítjük. A megjelenítéshez szükséges a méret megadása és utána az ablak láthatóvá tétele. A méretet a `setSize()` metódussal adjuk meg, a megjelenítést pedig `setVisible()` metódussal végezzük. Az ablak ebben az esetben a képernyő bal felső sarkában fog megjelenni. A program által megjelenített ablakot a 13.2. ábra szemlélteti.

```
SquareFrame f = new SquareFrame();
f.setSize(300,100);
f.setVisible(true);
```

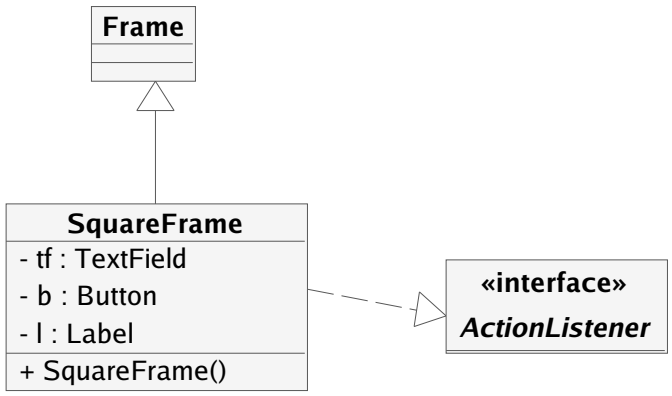
13.1.2. Felület használata

Az így elkészített felületünk még egyáltalán nem reagál a felhasználó által kiváltott eseményekre, azaz nem lehet bezárni és a nyomógomb hatására sem történik semmi. Nem is várhatjuk el, hiszen az eddig megírt kód még semmit sem tartalmaz ezen eseményekre vonatkozóan. A felhasználó a felülettel események formájában kommunikál. Ha például lenyomunk egy nyomógombot, akkor egy adott típusú eseményobjektum keletkezik, amely eljut az esemény lekezelésért felelős objektumhoz és ez fogja lefuttatni az esemény lekezelését célzó kódrészletet. Általában a konténerek szokták lekezelni a komponenseik eseményeit, de ennek nem kötelező így lennie. A programozó feladata meghatározni minden figyelt esemény esetében, hogy ki legyen az esemény lekezelésért felelős objektum. Egy

eseménynek több figyelője is lehet, ilyen esetben minden figyelőt értesítenek az esemény bekövetkeztéről. Az eseményfigyelőknek fel kell iratkozniuk az eseményt kiváltó objektumnál, ha egy adott eseményben érdekeltek. Az eseményt kiváltó objektum ismeri az összes eseményfigyelőjét. Ez az elv hasonló a hírlevelekhez. Ha kíváncsiak vagyunk valamilyen típusú hírre, akkor a megfelelő hírlistára feliratkozunk, és a feliratkozás pillanatától kezdődően megkapjuk a híreket.

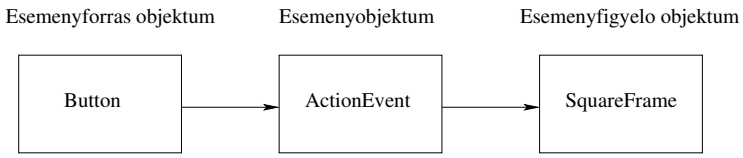
Az alkalmazásunkban a nyomógomb lenyomását szeretnénk figyelni. A nyomógomb lenyomásakor egy `ActionEvent` típusú eseményobjektum keletkezik. Válasszuk eseményfigyelőnek az ablak objektumot és lássuk el az ablakot az ilyen típusú események feldolgozási képességével. Minden egyes eseménytípusnak megvannak a sajátos eseményfeldolgozó műveletei. Ezeket egy-egy eseményfigyelő interfész tartalmazza. Az `ActionEvent` típusú eseménynek az `ActionListener` interfész a megfelelője. Ez az interfész egyetlen metódust tartalmaz, egy `actionPerformed(ActionEvent)` nevűt. Mivel az ablakunk fogja lekezelni ezt az eseményt, neki kell implementálnia ezt az eseményfigyelő interfészt. Ennek megfelelően a diagram a 13.3. ábrán látható módon alakul. Természetesen ez a kód módosulását is maga után vonja. Elsősorban a `SquareFrame` osztálynak implementálnia kell az `ActionListener` interfészt, aminek következményeként tartalmaznia kell az `actionPerformed()` metódust. Ez utóbbi pedig tartalmazni fogja az eseményt lekezelő kódot, vagyis a szövegdoboz tartalmának a négyzetre emelését és ennek megjelenítését a címke objektumon. Az eseményforrás, esetünkben a nyomógomb, felelőssége az eseményfigyelők regisztrálása. Ez az `addActionListener()` metódussal történik, amelynek szerepe összekötni az eseményforrást az eseményfigyelővel. A módosításokat a következő program szemlélteti, az eseményhez szükséges objektumokat és a köztük levő kapcsolatokat pedig a 13.4. ábra.

```
import java.awt.*;
import java.awt.event.*;
public class SquareFrame extends Frame implements ActionListener{
   private TextField tf;
   private Button b;
   private Label l;

   public SquareFrame(){
```



13.3. ábra. A módosított osztálydiagram

```
        this.setTitle("Negyzetre emeles");
        this.setLayout( new FlowLayout());
        tf = new TextField( 6 ); this.add( tf );
        b = new Button("^2");
        this.add( b );
        l = new Label("      ");
        this.add(l);
        b.addActionListener( this );
    }

    public void actionPerformed( ActionEvent e ){
        double d = Double.parseDouble( tf.getText());
        l.setText(""+(d*d));
    }
}
```

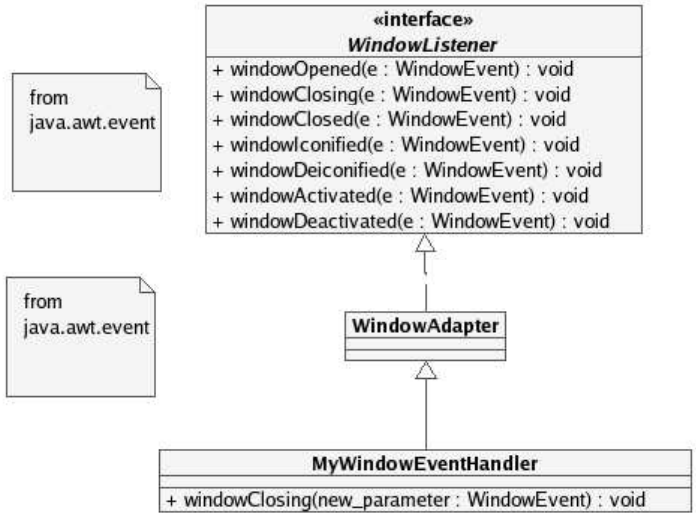


13.4. ábra. Esemény-Forrás-Figyelő

13.1.3. Felület bezárása

A felület bezárásához az ablakkal kapcsolatos eseményekre kellene figyelnünk. Az ablakok esetében elég sokféle esemény bekövetkezhet, mint például az ablak megnyitása, bezárása, aktívvá tétele, inaktívvá válás stb. Minket most az ablak bezárása érdekel, éspedig szeretnénk, ha befejeződne a Java alkalmazás, ha a felhasználó rákattint az ablak jobb sarkában levő X szimbólumra. Az ablak eseménykezelő metódusai a `WindowListener` interfészben vannak deklarálva a következőképpen:

```
public interface WindowListener extends EventListener{
   void windowActivated(WindowEvent e)
   //Ablak aktiválásakor hívódik

   void windowClosed(WindowEvent e)
   //Ablak bezárása után hívódik(dispose() hívásakor)

   void windowClosing(WindowEvent e)
   //Ablak bezárása

   void windowDeactivated(WindowEvent e)
   //Ablak inaktívvá válásakor hívódik

   void windowDeiconified(WindowEvent e)
   //Átváltás minimalizált állapotból normál méretű állapotra

   void windowIconified(WindowEvent e)
   //Átváltás normál méretű állapotból minimalizált állapotra}
```



13.5. ábra. Eseményadapter kiterjesztése

```
   void windowOpened(WindowEvent e)}
   //Ablak láthatóvá válásakor aktiválódik}
}
```

Ha csak egy esemény is érdekel a fenti hét közül, az eseménykezelő objektumunknak implementálnia kell a fenti interfész. Az implementálás következményeként az interfész mind a hét metódusát meg kell valósítanunk, még akkor is ha nem érdekelnek azon események. Ilyen esetben használható az üres metódusblokkal történő megvalósítás, de ez is fárasztó. Az ilyen esetek kényelmes kezelésére vezették be az adapter osztályokat. Az eseményadapter osztályok megvalósítják az eseményinterfészeket, minden metódust üres blokkal. Ha egy ilyen eseményadapter osztályt kiterjesztünk (örökítünk belőle), akkor csak azokat a metódusokat kell felülírnunk, amelyek számunkra érdekesek.

Programunkban csak a felület bezárásában vagyunk érdekeltek. A felület bezárásának az eseményforrása maga a felület (ablak) lesz. Az eseménykezelést

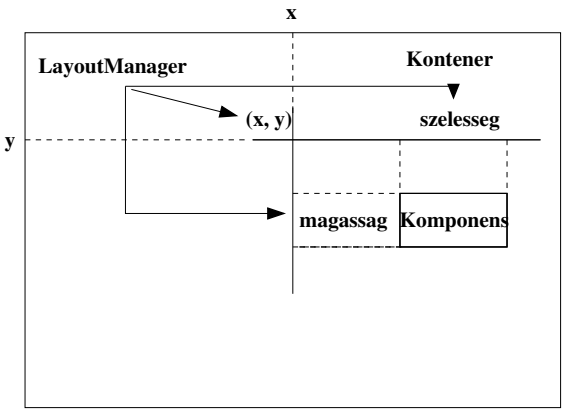
ráruházzuk egy olyan objektumra, amelyet a `WindowAdapter` osztályból származtatott osztályból hozunk létre, természetesen belső osztályt használva a típusra, hiszen csak egy példány létrehozása erejéig van szükségünk e típusra. Az előző osztályból csak a konstruktort fogjuk módosítani, hozzáadva egyetlen utasítást:

```
public SquareFrame ( ){
   ...
   this.addWindowListener( new WindowAdapter(){
      public void windowClosing( WindowEvent e){
         System.exit( 0 );
      }
   });
}
```

Az egyetlen utasítás, amelyet hozzáadtunk a konstruktorhoz, rengeteg mindent csinál. Elsősorban meghatároz egy új, névtelen típust, amely rendelkezik az ablakesemény kezelésének képességével és ablakbezárás esetén a Java virtuális gép lezárásához vezet. Ebből a névtelen típusból létrehoztunk egy példányt, amelyet az ablak objektumunk (`this`) regisztrált, mint eseményfigyelőt.

13.2. Konténerek és elrendezési stratégiák

Minden grafikus komponens a `Component` osztály leszármazottja és egy téglalap alakú területet foglal el. A komponenseket konténerekbe szervezzük. A konténerek maguk is komponensek, amelyek legfőbb feladata más grafikus komponensek összefogása egy csoporttá. A konténer tartalmazási kapcsolatban van a komponensekkel, amelyekkel együtt egy kompozíciót alkot. A konténer elemei a tartalmazó konténer megjelenítési felületén talalhatóak. A konténer is, a többi grafikus komponenshez hasonlóan, reagálhat a felhasználó által kiváltott eseményekre. Egy konténeren belül elhelyezhetünk egy másik konténert is, ennek következtében a konténer elemei egy fa struktúrát alkotnak. Így a legfelső szintű konténer lesz a fa gyökere, míg a levelek mindig egyszerű komponensek lesznek. A felsőszintű konténerek tartják a kapcsolatot az operációs rendszer ablakozó rendszerével és eseménysoruk is csak nekik van. A konténerek rendelkeznek a kompo-



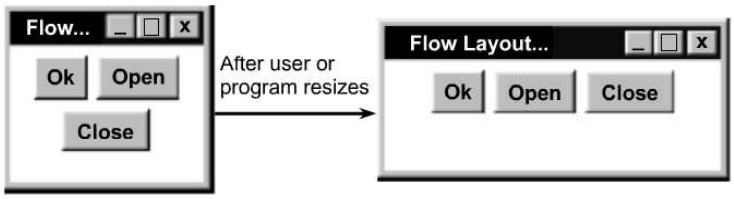
13.6. ábra. Elrendezési stratégia

nensek minden tulajdonságaival, hiszen azoknak a leszármazottjai (a `Container` osztály a `Component` osztály leszármazottja). Minden konténer a következő alapvető tulajdonságokkal rendelkezik:

- komponens felvétele és eltávolítása a konténerbe(-ből) (`add-remove` metódusok)
- a komponensek lekérdezése a konténertől (`getComponents` metódus)
- konténer adott pontján pozicionált komponens lekérdezése (`findComponentAt` metódus)

Míg a felső szintű komponenseknek van eseménysora, addig az alsószintűek nem rendelkeznek ezzel. Ilyen alsószintű komponens például a `Panel`, míg a felsőszintűek közül a `Frame`-et használtuk. A konténerekben a komponenseket különféleképpen rendezhetjük el. Használható a kézi elrendezési stratégia, amelyben minden egyes komponensnek megadjuk a pozícióját és a méretét, vagy használhatunk elrendezészervezőt, amely valamilyen jól meghatározott stratégia alapján elrendezi a komponenseket a felületen. Az automatikus elrendezésszervezés előnye, hogy a konténer átméretezésénél újraméretezi a komponenseket az aktuális konténerméretnek megfelelően.

Az elrendezési stratégiák közül az egyszerűbbek a `FlowLayout`, `BorderLayout` és `GridLayout` stratégiák. A `FlowLayout` elrendezési straté-



13.7. ábra. FlowLayout elrendezési stratégia ([11] nyomán)

gia a komponenseket sorokba szervezi, azok optimális méretét véve figyelembe. Egy sorba addig kerülnek be a komponensek, amíg az be nem telik, utána a következő sorral folytatódik az elhelyezés (ld. 13.7. ábra).

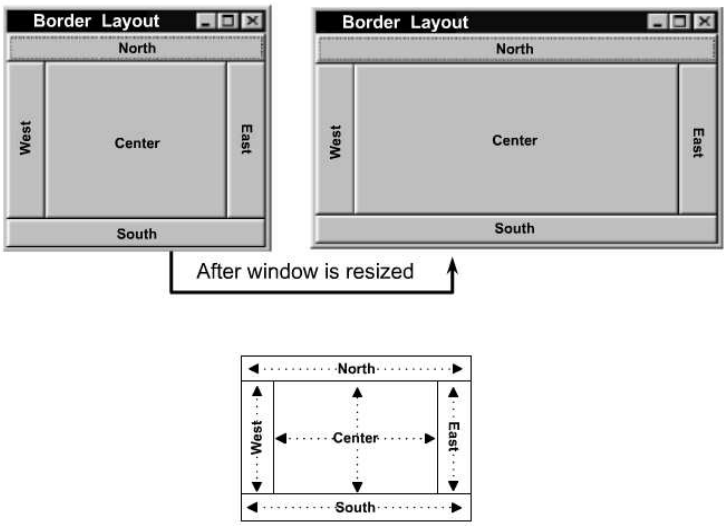
```
container.setLayout( new FlowLayout() );
container.add( new Button(''Ok''));
container.add( new Button(''Open''));
container.add( new Button(''Close''));
```

A `BorderLayout` elrendezési stratégia egyszerre legfeljebb öt komponenst tud a konténerben elhelyezni, ezeket a négy szélhez, illetve középre igazítva (ld. 13.8. ábra).

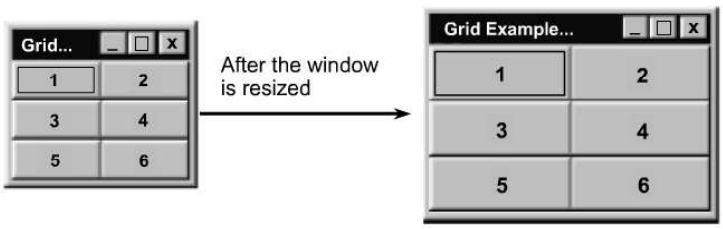
```
container.setLayout( new BorderLayout());
container.add( new Button(''North''),''North'');
container.add( new Button(''South''),''South'');
container.add( new Button(''East''),''East'');
container.add( new Button(''West''),''West'');
container.add( new Button(''Center''),''Center'');
```

A GridLayout elrendezési stratégia a felületet sorokra és oszlopokra osztja, egyenlő méretű cellákat határozva meg. Minden komponens azonos méretű lesz (ld. 13.9. ábra).

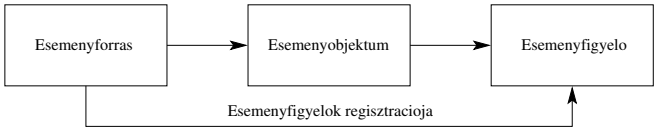
```
container.setLayout( new GridLayout(3,2));
for( int i=1; i<=6; i++ )
  container.add( new Button(i+""));
```



13.8. ábra. BorderLayout elrendezési stratégia ([11] nyomán)



13.9. ábra. GridLayout elrendezési stratégia ([11] nyomán)



13.10. ábra. Eseményforrás-Eseményobjektum-Eseményfigyelő

Minden egyes konténernek van egy alapértelmezett elrendezésszervezője, amelyet át lehet állítani a `setLayout(LayoutManager)` metódussal. Ha nem akarunk elrendezésszervezőt használni, akkor a `setLayout(null)` metódushívást kell végezni a konténerre. Ezután viszont minden komponenst méretezni és pozicionálni kell a konténerben.

13.3. Események

A Javaban az objektumok események segítségével kommunikálnak egymással. Például egy nyomógomb értesíti a programot, hogy a felhasználó megnyomta őt. Ez az értesítés esemény formájában történik. Minden eseménynek van egy eseményforrása (generátora) és egy-vagy több eseményfigyelője. Az esemény bekövetkeztekor létrejön az eseményobjektum, amelyet a Java futtatási környezet eljuttat minden egyes eseményfigyelőhöz. Az esemény átadásakor az eseményfigyelő objektumban végrehajtódik egy metódus, az eseménykezelő metódus. Maga az esemény is egy objektum, amely információkat tartalmaz az adott eseménnyel kapcsolatosan. Az eseménykezelő metódusoknak egyetlen paraméterük van, az eseményobjektum.

Az eseményforrás feladata regisztrálni az eseményfigyelőket. Az eseményfigyelőknek viszont képeseknek kell lenniük az események fogadására, ez pedig azt jelenti, hogy tartalmazniuk kell eseménykezelő metódusokat, amelyeket végrehajtanak az illető esemény bekövetkeztekor. Az eseményforrás, figyelők és objektum kapcsolatát a 13.10. ábra szemlélteti. Az eseménykezelést nagyon sokféleképpen lehet megvalósítani. Az egyik legegyszerűbb változat, hogy a grafikus komponensekkel bekövetkezett összes eseménynek a konténer legyen a figyelője. Ebben az esetben összesen egy eseményfigyelőnk lenne. Egy másik véglet,

hogy minden egyes eseménynek legyen külön figyelőobjektuma, amelynek csak az eseményfigyelés-kezelés a feladata.

Javaban minden eseménytípus a `java.util.EventObject` osztály leszármazottja kell legyen. Ezen osztály nyilvántartja az esemény forrását, melyet a `getSource()` metódussal lehet lekérdezni. Az `AWT` által használt események ősosztálya a `java.awt.AWTEvent` absztrakt osztály, amely a `java.util.EventObject` leszármazottja. Minden `AWT` eseménynek van egy azonosítója, amelyet a `getID()` metódussal lekérdezhetünk.

13.4. Eseményfigyelők

Az eseményfigyelők is objektumok, de a konkrét eseménykezelést mindig egy metódus látja el. Az eseményfigyelő metódusok interfészekben vannak deklarálva, eseménytípusonként külön interfészekben csoportosítva. Minden egyes eseményinterfész a `java.util.EventListener` interfész leszármazottja. A 13.1. táblázat olyan eseményeket tartalmaz, amelyek bármely komponenssel illetve konténerrel bekövetkezhetnek, a 13.2. táblázat pedig komponensspecifikus eseményeket tartalmaz.

Az eseményregisztráció lehetővé teszi, hogy az események eljussanak az eseményforrástól az eseményfigyelőig. Minden eseményforrás objektumnak rendelkeznie kell eseményfigyelő karbantartási műveletekkel, új figyelő felvétele, létező figyelő törlése. Ezen műveletek neve eseménytípus függő:

```
public void add<ListenerType>( <ListenerType> listener);
public void remove<ListenerType>( <ListenerType> listener);
```

A Java eseménymechanizmus nem írja elő, hogy miként függ a program viselkedése eseményfigyelők hozzáadásának, illetve törlésének sorrendjétől, viszont azt előírja, hogy minden regisztrált eseményfigyelőnek meg kell kapnia az eseményt.

Esemény	Eseményforrás	Eseményfigyelő interfész	Eseménykezelő metódus
ComponentEvent	Komponensek	ComponentListener	componentResized() componentMoved() componentShown() componentHidden()
FocusEvent	Komponensek	FocusListener	focusGained() focusLost()
KeyEvent	Komponensek	KeyListener	keyTyped() keyPressed() keyReleased()
MouseEvent	Komponensek	MouseListener, MouseMotionListener	mouseClicked() mousePressed() mouseReleased() mouseEntered() mouseExited() mouseDragged() mouseMoved()
ContainerEvent	Konténerek	ContainerListener	componentAdded() componentRemoved()

13.1. táblázat. Komponensek és konténerek eseményei

13.5. Felhasználó által kiváltott események

A felhasználó beviteli eszközök segítségével kommunikál a számítógéppel. A szokásos beviteli eszközök a billentyűzet és az egér. A felhasználó által kiváltott események tartalmazzák az esemény keletkezésének időpontját is, amelyet a `getWhen()` metódussal le lehet kérdezni. A felhasználó által kiváltott eseményobjektum tárolja a beviteli eszközök aktuális módosító állapotát is, amelyet a `getModifiers()` metódussal le is lehet kérdezni. A billentyűzet esetében a módosító állapoton valamely vezérlőbillentyű lenyomott állapotát értjük (`Ctrl`, `Alt`, `Shift`), míg az egér esetében ez az egérgombok lenyomott állását tartalmazza. A felhasználói eseményekért felelős osztályok hierarchiáját a 13.11. ábra szemlélteti.

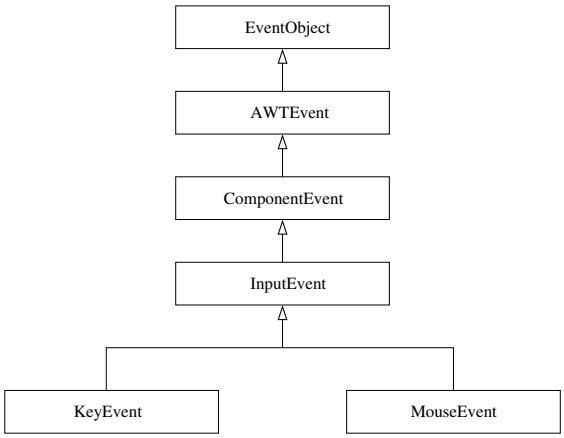
13.5.1. Billentyűzet események

Minden billentyűeseményt egy `java.awt.event.KeyEvent` típusú objektum reprezentál. Ez az osztály háromféle billentyűzeteseményt tud megkülönböztetni:

– billentyűleütés (`keyPressed`)

Esemény	Eseményforrás	Eseményfigyelő interfész	Eseménykezelő metódus
ActionEvent	TextField Menuitem List Button	ActionListener	actionPerformed()
ItemEvent	List CheckBox Choice CheckboxMenuItem	ItemListener	itemStateChanged()
AdjustmentEvent	ScrollPane Scrollbar	AdjustmentListener	adjustmentValueChanged()
TextEvent	TextArea TextField	TextListener	textValueChanged()
WindowEvent	Frame Dialog	WindowListener	windowOpened() windowClosing() windowClosed() windowIconified() windowDeiconified() windowActivated() windowDeactivated()

13.2. táblázat. Komponensek sajátos eseményei



13.11. ábra. Eseményobjektumok hierarchiája

– billentyűfelengedés (`keyReleased`)

– billentyűmegnyomás (`keyTyped`)

Billentyűleütés akkor keletkezik, ha a felhasználó a billentyűzeten egy billentyűt lenyom. Ebben az esetben a `getID()` metódus a `KeyEvent` osztályban definiált `KEY_PRESSED` értéket adja vissza. Billentyűfelengedés természetesen csak leütés után következhet. Ebben az esetben `KEY_RELEASED` lesz a `getID()` által visszaadott érték. Billentyűmegnyomás esemény keletkezik, ha a felhasználó lenyom, majd felenged egy billentyűt. Ilyenkor `KEY_TYPED` lesz a `getID()` által visszaadott érték.

A billentyűmegnyomás magasabb szintű esemény a leütés és felengedésnél, hiszen ezen két esemény következtében fog kiváltódni. Ezen felül módosító billentyűk esetében nem is váltódik ki. `Billentyűleütés és felengedés esetén,` az esemény `getKeyCode()` metódusa által visszatérített egész szám egy virtuális billentyűkódnak felel meg, amelynek segítségével egyértelműen azonosítható minden egyes billentyű. A `KeyEvent` osztály tartalmazza a billentyűknek megfelelő állandókat. Pl.

`KeyEvent.VK_0 - KeyEvent.VK_9`: a számjegybillentyűket jelölik

`KeyEvent.VK_F1- KeyEvent.VK_F24`: a funkcióbillentyűket jelölik

Ha például lenyomjuk az `x` karakternek megfelelő billentyűt, akkor a következő sorrendben generálódnak a billentyűzet események: `KEY_PRESSED, KEY_TYPED, KEY_RELEASED`. A billentyűzetet rendszerint szöveges információ bevitelére használjuk, ezért karaktert kell rendelni a billentyűzet-eseményhez. Adott billentyűzet-eseményhez rendelt karaktert annak `getKeyChar()` metódusával lehet lekérdezni. Ha módosító billentyű volt lenyomva, akkor a metódus `KeyEvent.CHAR_UNDEFINED` karaktert fog visszaadni.

13.5.2. Egéresemények

Minden egérrel kapcsolatos eseményt egy `java.event.MouseEvent` objektum reprezentál. Az egéresemény objektum tartalmazza az egéresemény pozícióját, amelyet a `getX()` és `getY()` metódusokkal kérdezhetünk le. Az egéreseményekhez szükséges metódusok két interfészbe vannak csoportosítva, a `MouseListener` és a `MouseMotionListener` interfészekbe.

A `MouseMotionListener` interfész két eseménykezelő metódust tartalmaz, éspedig az egérelmozdulást (`mouseMoved`) és az egérrel történő mozgatást (`mouseDragged`). Egérelmozdulás akkor keletkezik, ha a felhasználó az egeret elmozdítja egérgomb lenyomás nélkül. Egérrel történő mozgatás akkor keletkezik, ha a felhasználó az egeret elmozdítja és eközben le volt nyomva valamelyik egérgomb.

A `MouseListener` interfész öt egéresemény lekezelését szolgáló metódust tartalmaz. A `mousePressed()` metódus célja az egérgomb-lenyomás eseményének lekezelése, amely akkor keletkezik, ha a felhasználó az egér egyik gombját lenyomja. Az egérgomb-felengedés eseményét a `mouseReleased()` metódussal kezelhetjük le. Ha pedig lenyomtunk és fel is engedtünk egy egérgombot, akkor egérgomb-kattintás esemény keletkezik, amelyet a `mouseClicked()` metódussal kezelhetünk le. Ha az egér belép valamely grafikus komponens felületére, akkor egérbelépési esemény keletkezik, ha pedig kilép valamely komponens felületéről, akkor egérkilépési esemény keletkezik. Ezek lekezelését szolgálja a `mouseEntered()` illetve a `mouseExited()` metódusok.

13.6. Események feldolgozása

Minden eseményvezérelt programnak van egy eseménysora, amelybe az ablakozó rendszer helyezi be az alkalmazáshoz tartozó eseményeket. A Java futtatási környezet minden eseményvezérelt programban elindít egy külön programszálat, amely az események feldolgozásáért felelős. Például ha billentyűlenyomás esemény keletkezett egy szövegdobozban, az esemény először az eseménysorba kerül, majd ez közvetíti az eseményforrás (esetünkben szövegdoboz) felé, amely megjeleníti a lenyomott billentyűnek megfelelő karaktert. Ez az eseményfeldolgozás ebben az esetben alapértelmezett, de szükség esetén megváltoztatható. A szövegdobozban történt billentyűlenyomás elsősorban egy `KeyEvent` eseményt hoz létre majd ezt követően egy `TextEvent` eseményt. Ha az Enter billentyűt nyomunk le, akkor elmarad a `TextEvent` esemény és egy `ActionEvent` esemény keletkezik. A programozó feladata eldönteni, hogy melyik eseményt szeretné lekezelni. További részleteket a [3], [9] könyvek tartalmaznak.

14. FEJEZET

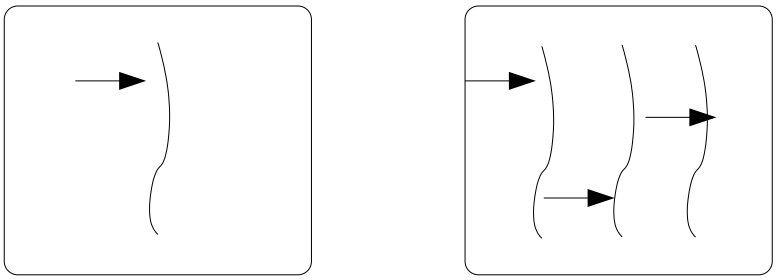
Párhuzamos programozás - Programszálak

14.1. Folyamatok, szálak

A modern programozási nyelvek absztrakciós eszközöket biztosítanak a következőkre:

- absztrakt adattípus
- kivételkezelés
- párhuzamos feldolgozás

A Java nyelv mindháromra biztosít eszközöket. Elsősorban az osztályok és interfészek segítségével lehetővé vált az absztrakt adattípus definiálása. Másodsorban a Java nyelv támogatja a futásidejű hibák kivételkezeléssel történő megoldását. A Java nyelvi szinten támogatja a párhuzamos feldolgozást, külön száltípus (`Thread`) bevezetésével, illetve szinkronizációs mechanizmussal (`synchronized` kulcsszó, monitor). A szálak között létezik egy jelzési mechanizmus is, ezt a lehetőséget már az `Object` osztály támogatja (`wait, notify, notifyAll` metódusok). A hatvanas évektől elkezdődött az olyan eszközök kidolgozása, amelyek segítségével az eljárások szinkronizálhatók. Ilyen eszközök például:



14.1. ábra. Egy szálat, illetve több szálat futtató folyamat

- szemafor (Dijkstra)
- monitor (Hoare)
- szinkron üzenetátadás (Hoare)

A multitaszking operációs rendszerek a folyamatok váltogatásával próbálják biztosítani a processzor maximális kihasználtságát. Amíg az egyik folyamat valamilyen erőforrás hiányában várakozásra kényszerül, addig a processzor egy másik folyamatot futtathat. A központi egység egyszerre csak egy folyamatot hajt végre, de ha elég gyakran váltogatja ezeket, akkor a párhuzamos végrehajtás illúzióját kelti. Igazi párhuzamosságot természetesen csak többprocesszoros rendszerekben lehet elérni.

A szálak hasonlítanak a folyamatokhoz, mindkettőnek alapvető feladata egy utasítássorozat végrehajtása. Amíg a folyamat kernel szintű fogalom, addig a szál felhasználó szintű fogalom. Ennek következtében a folyamatleíró struktúra minden része a kernelben van, a szálleíró struktúra pedig felhasználói területen. A folyamathoz alapvetően háromféle memória tartozik: kód, verem, adat.

Minden szálnak megvan a végrehajtandó utasítássorozata, de ez közös, így egy folyamaton belüli bármely szál hozzáférhet. Minden szálnak viszont saját veremterülete és saját utasításmutatója van. Egy folyamaton belüli szálak ugyanazon memóriaterületeket látják. Ha egy szál megváltoztat egy folyamatszintű adatot, akkor a folyamat összes szála érzékelni fogja a változást a legközelebbi hozzáféréskor. A 14.1. ábra összehasonlítja az egy szálat, illetve a több szálat futtató folyamatot.

Egy hagyományos egyszálas program végrehajtásakor, ha valamilyen eszközre várakozni kell, akkor a központi egység várakozó állapotba kerül. A többszálas rendszere lehetővé teszi, hogy amíg az egyik szál az adatokra várakozik, addig a többi szál folytathassa munkáját, azaz ha a folyamat egy részének várakoznia is kell, nem kell az egész folyamatot blokkolni.

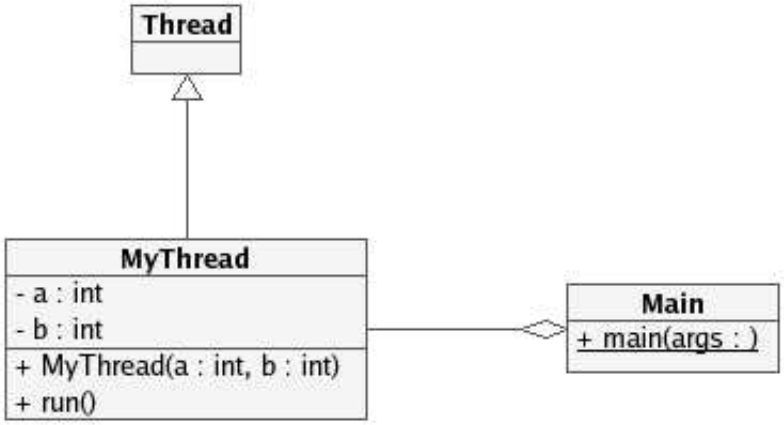
Bizonyos típusú alkalmazások jellegükből adódóan párhuzamos szerkezetűek. Ilyenek a kiszolgálást végző, szerver típusú alkalmazások. Ezeknek az alkalmazásoknak egyidőben több kérelmet kell feldolgozniuk és kiszolgálniuk. Ilyen jellegű alkalmazások például a fájlszerverek vagy webszerverek. Egy másik közismert alkalmazás például a csevegést lebonyolító szerver. Egy csevegő alkalmazás kliens részében is találunk párhuzamosan végrehajtható dolgokat. Például egy csevegő kliens programnak egyszerre kell a felhasználóra figyelni, aki éppen üzenetet gépel, illetve a szervertől érkező üzeneteket fogadni és megjeleníteni a felületen.

14.2. Java szálak

A Java nyelvben kétféleképpen lehet szálat létrehozni. Az első lehetőség a `Thread` osztályból való származtatás. A `Thread` osztályt a `java.lang` csomag tartalmazza. Ez a származtatott osztály szál viselkedésű lesz, amelyben a viselkedés sajátosítására felül kell írni a `run` metódust. A `Thread` osztálybeli `run` metódus üres, mindig az utódosztályban felülírt változata fogja meghatározni, hogy milyen műveleteket kell külön szálon végrehajtani. Az első fajta létrehozást szemlélteti a 14.2. ábra.

A szál indítása a `start` metódussal történik. Ez inicializálja a szálat és meghívja ennek a `run` metódusát. A `run` metódus szerepe hasonló az egyszálon futó alkalmazások `main` metódusához. A következő program egy olyan szál osztályt definiál, amely egy adott intervallumból kiírja a számokat. Ebből az osztályból két példányt készítünk és elindítjuk őket.

```
class MyThread extends Thread{
   private int a, b;

   public MyThread( int a, int b ){
      this.a = a;
```



14.2. ábra. Szál létrehozása származtatással

```
      this.b = b;
   }

   public void run(){
      for( int i=a; i<=b; ++i ){
         System.out.println( getName()+": "+i);
         try{
            sleep( 1);
         }
         catch( InterruptedException e ){}
      }
   }
}

public class MainMyThread1{
   public static void main( String args[] ){
      MyThread t1, t2;
      t1 = new MyThread(1, 5000 );
```

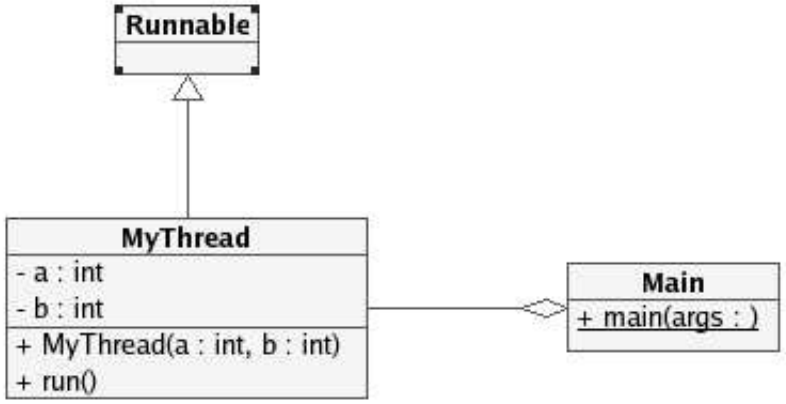
```
      t2 = new MyThread(5001, 10000);
      t1.start();
      t2.start();
   }
}
```

A `sleep` metódus felfüggeszti a szál futását a paraméterben megadott időintervallumra. Az időintervallumot ezredmásodpercben adjuk meg.

Az előző szállétrehozási módszernek az a hátránya, hogy amennyiben egy osztályt a `Thread` osztályból származtatunk, nem lehetséges, hogy ez egy másik osztályt is kiterjesszen, hiszen a Java nyelv nem támogatja a többszörös öröklést. Például, ha egy HTML oldalba beágyazható kisalkalmazást akarunk készíteni, akkor az `Applet` osztály kiterjesztése kötelező. A kisalkalmazásokat pedig igazán illik külön szálra tervezni, főképp ha valamilyen animációt akarunk megvalósítani ennek segítségével.

Az ilyen esetek megoldására vezettek be egy második szál létrehozási módszert. Ez a módszer a `Runnable` interfész megvalósítására támaszkodik. A `Runnable` interfész egyetlen metódust definiál, a `run()` metódust. Implementálva az interfészt meg kell adnunk a `run()` metódust, amely a külön szálon végzendő tevékenységeket fogja tartalmazni. Ebben a második esetben a szálat úgy fogjuk létrehozni, hogy a `Thread` osztály konstruktorának átadjuk a `Runnable` típusú objektumot. Amikor a szál elkezd futni, a `Runnable` objektum `run()` metódusa kapja meg a vezérlést. A következő program ezt a második létrehozási módot szemlélteti.

```
class MyRunnable implements Runnable{
   private int a, b;

   public MyRunnable( int a, int b ){
      this.a = a;
      this.b = b;
   }

   public void run(){
```



14.3. ábra. Szál létrehozása a `Runnable` interfész segítségével

```
      for( int i=a; i<=b; ++i ){
         System.out.println(Thread.currentThread().getName()+":"+i);
         try{
            Thread.sleep(1);
         }
         catch( InterruptedException e ){}
      }
   }
}
```

Összefoglalva a következőket állíthatjuk a Java szálakról:

- Minden szálnak egyetlen belépési pontja van és egy jól meghatározott utasítássorozatot kell végrehajtania.
- Minden szál a többitől függetlenül hajtja végre a saját kódrészletét és az alkalmazástól függ, hogy a szálak együttműködnek-e vagy sem.
- A szálak párhuzamosan hajtódnak végre, egy ütemezési algoritmus vezérli a végrehajtási sorrendet.

– Java nyelvben háromféle változóhoz férhetnek hozzá a szálak:

  – lokális változók - ez teljesen privát, egyik szál sem fér hozzá a másik szál lokális változóihoz. Amennyiben több szál ugyanazt a kódrészletet hajtja végre, akkor minden szálnak külön másolata van az adott változóról.
  – példány szintű adattagok - ezek megoszthatók a különböző szálak között. A megosztást lehetővé kell tenni.
  – statikus adattagok - ezek automatikusan minden szál által elérhetőek

14.3. Szálcsoportok

A szálcsoportok lehetővé teszik, hogy több szálat egyetlen csoportobjektumba fogjunk össze. Ennek következtében, ha a szálcsoportnak küldünk üzenetet, azt a csoporthoz tartozó összes szál meg fogja kapni. Lehetőség nyílik például a szálcsoporthoz tartozó összes szál leállítására egyetlen üzenet segítségével.

Amikor a szálat létrehozzuk, lehetőségünk van arra, hogy az új szálat egy megadott csoporthoz adjuk, vagy a Java futási rendszerre bízzuk, melyik csoportba tegye azt. Egy szálcsoporton belül más szálcsoportok is létrehozhatók, így az egy programon belül futó szálaknak egy hierarchikus fastruktúrája lesz. A szálcsoportok szemléltetésére tanulmányozzuk a következő programot:

```
class MyThread extends Thread{
   public MyThread( String name ){
      super( name );
   }

   public void run(){
     try{
         for( int i=0; i<500; ++i )
            sleep( 100 );
     }
     catch( InterruptedException e ){}
   }
```

```
}

public class MyThreadGroup{
   public static void main( String args[]){
      MyThread t[] = new MyThread[20];
      for( int i=0; i<20; ++i )
         t[i] = new MyThread(i+".szal");
      for( int i=0; i<20; i+=2 )
         t[i].start();
      ThreadGroup grp = Thread.currentThread().getThreadGroup();
      int instanceCount = grp.activeCount();
      Thread th[]= new Thread[instanceCount];
      int activeCount = grp.enumerate( th );
      System.out.println("Csoport neve: "+grp.getName() );
      System.out.println("Csoporthoz tartozo szalak: "+instanceCount);
      System.out.println("Csoporthoz tartozo aktiv szalak:"+activeCount);
      System.out.println("Csoporthoz tartozo aktiv szalak nev szerint:");
      for( int i=0; i<activeCount; ++i )
         System.out.println( th[i].getName());
      grp.interrupt();
   }
}
```

14.4. Futásvezérlés

14.4.1. Szálak indítása

Minden szálat a `start()` metódussal indítunk, ez az egyetlen lehetőség. Ez a kezdeti inicializálások után végrehajtja a `run()` metódust. Amennyiben a `run()` metódust közvetlenül hívnánk, ez egy normális metódusként hajtódna végre a hívó szálon, nem pedig egy külön szálon.

Tekintsük a következő programot, amelyben a `main` szálon is, meg egy külön szálon is kiíratjuk az első három természetes számot. Minden szám kiírását egy

`sleep()` metódushívás követi, ezzel biztosítjuk, hogy biztosan ütemezésre kerül a másik futó szál is.

```
public class MyThread extends Thread{

   public void run(){
      for( int i=0; i<3; ++i ){
         System.out.println(getName()+". "+i);
         try{
            sleep(1);
         }catch( InterruptedException e ){}
      }
   }


   public static void main( String args[]){
      System.out.println( Thread.currentThread().getName()+" indul");
      MyThread t = new MyThread();
      t.start();
      for( int i=0; i<3; ++i ){
         System.out.println(Thread.currentThread().getName()+": "+i);
         try{
            Thread.currentThread().sleep(1);
         }
         catch( InterruptedException e ){}
     }
   }
}
```

Egy lehetséges kimenet a következő:

```
main indul
main: 0
Thread-0. 0
main: 1
Thread-0. 1
```

```
main: 2
Thread-0. 2
```

Amennyiben a `t.start()` hívást kicseréljük a `t.run()` hívásra, először lefut a `run()` metódus, utána pedig folytatódik a `main` metódus, tehát teljesen szekvenciálisan fog végrehajtódni a két kiíratás. Ebben az esetben programunk a következő kimenetet eredményezi:

```
main indul
Thread-0. 0
Thread-0. 1
Thread-0. 2
main. 0
main. 1
main. 2
```

14.4.2. Szálak leállítása

Szálat leállítani a `stop()` metódussal lehet, de nem ajánlott. Azért nem ajánlott, mert könnyen holtpontba juttathatja a rendszert, hiszen a `stop()` hívása nem biztosítja az erőforrások felszabadítását. Amennyiben a leállított szál olyan erőforrásokat kötött le, amelyekre más szálak várakoztak, ez holtpontot eredményez. A holtpont elkerülése végett ajánlott a szálosztályba bevezetni egy plusz állapotváltozót és egy olyan metódust, amelyen keresztül módosíthatjuk ezen állapotváltozót. A `run()` metódusban pedig a végrehajtást ezen változótól tesszük függővé. Tekintsük erre a következő programot.

```
public class ThreadStop extends Thread{

   private volatile boolean isRunning = true;

   public void stopp(){
      isRunning = false;
   }

   public void run(){
```

```
      while( isRunning ){
         System.out.println(getName()+" szal fut...");
         try{
            sleep( 100 );
         }
         catch( InterruptedException e ){}
      }
   }

   public static void main( String args[] ){
      ThreadStop t = new ThreadStop();
      t.start();
      try{
         Thread.sleep( 1000 );
      }
      catch( InterruptedException e ){}
      System.out.println(t.getName()+" szal leallitasa");
      t.stopp();
   }
}
```

Az `isRunning` adattagot a `volatile` módosítóval láttuk el. Ez azt jelenti, hogy kényszeríti a JVM-et, hogy az így deklarált változón végezzen atomi vizsgálat- és beállítás műveleteket. Pontosabban azt jelenti, hogy a változó olvasásakor ez mindig a memórából töltődik be és íráskor az új érték mindig a memóriába íródik. Azért van szükség erre a védelemre, mert bizonyos változókat lehet a CPU regisztereiben is, illetve a gyorsító tárban (cache) is tárolni.

14.4.3. Démon szálak

A démon szálak ugyanolyan jellegűek, mint a démon folyamatok és ugyanolyan célokra használjuk ezeket, mint a háttér folyamatokat. A démon szálak általában valamilyen szolgáltatást biztosítanak és csak akkor állnak le, ha az összes többi nem-démon szál leállt. A démon szál leállításáról a futtató környezet gondoskodik. A szál démon jellegét a `setDaemon()` metódussal állíthatjuk.

14.4.4. Szálak összekapcsolása

Bizonyos nagy méretű feladatok megoldásához a szálaknak együtt kell működniük. Lehetőséget kell teremteni arra, hogy részfeladatokat külön szálra helyezhessünk és amennyiben az egyik részfeladat egy másik részfeladat eredményeitől függ, akkor ennek befejeződését be kell várnia. A várakozást a `join()` metódussal lehet megoldani. A váró szálnak mindig kell ismernie azt a szálat, amelyet be kell várnia. A `join()` metódusnak két túlterhelt formája is van. Az egyiknek nincsen paramétere, ez feltétel nélkül várakozik egészen addig, amíg a másik szál be nem fejeződik. A másik formának megadható egy időintervallum századmásodpercben, ez legfeljebb a megadott időintervallum lejártáig várakozik. A `t.join()` hívás például azt a szálat fogja várakoztatni, amelyik éppen ezt a kódrészletet futtatja és a `t` referenciájú szálra fog várakozni.

14.5. Szálak ütemezése

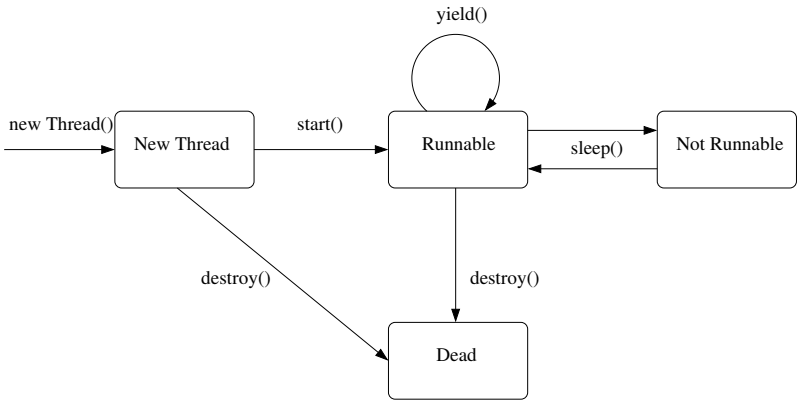
A Java nyelvben az ütemezés azt jelenti, hogy egy szálat végrehajtunk a Java virtuális gépen. Két jellemző befolyásolja a szálak ütemezését: a szálak állapota és a szálak prioritása.

14.5.1. Szál állapota

A Java szálak sokféle állapotban lehetnek. A főbb állapotok az 14.4. ábrán láthatók.

`New Thread - Új szál állapot -` Ebbe az állapotba a szál a `new Thread()` művelet következtében kerül. Az új szál állapotban a szál még nem fut, ehhez a `start()` metódus hívása szükséges.

`Runnable - Futtatható állapot -` Ebben az állapotban a szálak ütemezhetők hiszen mindennel rendelkeznek, ami a végrehajtásához szükséges. Egy processzor esetén egyszerre csak egy szál rendelkezhet a központi egységgel. Egy szál többféle képpen kerülhet nem futtatható (blokkolt) állapotba. Erre a következő lehetőségek vannak:



14.4. ábra. Szálak állapotdiagramja

-A `sleep()` metódus hívása felfüggeszti a szál futását a paraméterben megadott időintervallum erejéig, utána visszakerül a szál futtatható állapotba.

-A szálat várakoztathatjuk egy adott feltétel teljesüléséig a `wait()` metódus hívásával.

-Ha egy szál éppen I/O műveletet hajt végre, egy `read()` metódus blokkolhatja a szálat. Az olvasás befejeztével a szál ismét futtatható állapotba kerül.

-Szál futását fel is függeszthetjük a `suspend()` metódussal, ahonnan a `resume()` hívás következtében fog visszatérni.

`Not Runnable - Blokkolt állapot` - Az ilyen állapotban levő szálak nem ütemezhetők. Blokkolt állapotból a szál többféle képpen kerülhet vissza futtatható állapotba.

-Ha feltételre várakozik a szál, akkor futtatható állapotba egy `notify()` hívás juttathatja.

-Ha a `read()` művelet hatására blokkolódott, akkor ennek befejeződése fogja futtatható állapotba vinni a szálat.

-Ha a `sleep()` blokkolta, akkor a várakozási idő lejárta viszi vissza a szálat, vagy egy `interrupt()` művelet is megteheti. Ez utóbbi esetben `InterruptedException` is ki fog váltódni.

`Dead - Halott állapot` - Egy szál két módon halhat el. Az egyik lehetőség erre a normális befejeződés, vagyis a `run()` metódus befejeződése, a másik

lehetőség pedig a szál leállítása a `stop()` metódus segítségével vagy valamilyen más alternatív módon.

14.5.2. Szál prioritása

A JVM prioritásokat alkalmaz annak eldöntésére, hogy melyik szál futhat előbb. A Java nyelvben tíz féle prioritási szintet különböztetünk meg, a `Thread.MIN_PRIORITY`-tól a `Thread.MAX_PRIORITY`-ig. A JVM a legnagyobb prioritású szálat indítja el a futási sorból. Amennyiben több azonos prioritású szál van, ezeket a Round-Robin ütemezési algoritmussal fogja ütemezni. A Round-Robin egy körkörös ütemezési algoritmus. Egy szál prioritását a `setPriority()` metódussal állíthatjuk. Amennyiben nem állítjuk, ez az őt létrehozó szülő száltól örökölt prioritással fog maradni.

Amennyiben több szál van futásra kész állapotban, a futtató rendszer a legmagasabb prioritású szálat választja. Ez a szál pedig addig fog futni amíg vagy befejeződik, vagy elalszik, vagy megszakítják, vagy pedig blokkolt állapotba kerül. Az operációs rendszerek időosztással harcolnak az ilyen önző szálak ellen. Az időosztás azt jelenti, hogy a központi egységet egy szál vagy folyamat csak egy időszelet erejéig kaphatja meg. A JVM viszont nem garantálja az időosztást, így nem szabad erre támaszkodnunk. Lehetőség van egy szálat udvariassá tennünk, úgy, hogy ez explicit módon szakítsa félbe a munkáját a többiek javára. Ezt a célt szolgálja a `yield()` metódus. Így lehetővé válik, hogy programunk az operációs renszertől függetlenül időosztásosan fusson.

14.6. Szinkronizáció

14.6.1. Miért van szükség szinkronizációra?

Egy banki alkalmazásnak számlakezelést is kell biztosítania. Mivel a számlák kezelését automatával is végezhetjük, úgy kell megtervezni egy ilyen rendszert, hogy egy adott számlához ne engedjen meg párhuzamos hozzáférést. Nézzük meg, hogy miért is baj, ha lehetőség van párhuzamos hozzáférésre. Elsősorban egy automatával pénzkivétel esetén a következő műveleteket végezzük:

1. Azonosítjuk magunkat a banknál és bevisszük a kívánt összeget (bankszámla azonosítása)
2. Összeg levétele a számláról
3. Pénz kiadása
4. Papír kinyomtatása

Tekintsünk példát egy `Szamla` osztályra. Minket főképp a `levesz` metódus érdekel, így csak ezt szemléltetjük:

```
public class Szamla{
   private float egyenleg;

   public boolean levesz( float osszeg ){
      if(osszeg <= egyenleg ){
         egyenleg -= osszeg;
         return true;
      }
      return false;
   }
   ...
}
```

Látható, hogy a levesz metódusban először ellenőrződik, hogy van-e megfelelő mennyiségű pénz a számlán. Amennyiben van, csökkentjük az egyenleget és visszatérítünk egy logikai igaz értéket. Ellenkező esetben hamis lesz a visszaadott érték. Az automata osztály a következőképpen nézne ki:

```
class Automata{
   ...
   public void kivesz( float osszeg ){
      Szamla sz = szamlaAzonositas();
      float osszeg = osszegBekeres();
      if( sz.levesz() ){
         penzkiadas( osszeg );
```

```
            szamlanyomtatas();
        }
    }
    ...
}
```

A fenti kódolással megtörténhet, hogy egy férj és egy feleség, akiknek közös számlájuk van, ugyanazon időben, egymástól függetlenül, két különböző automatán keresztül, kiürítik a számlát. Ez azért történhetik meg, mert a `levesz` metódus megszakítható, vagyis végrehajtása nem atomikusan történik. Ha ez nem lenne megszakítható, a fenti rendellenesség nem következhetne be. Tekintsünk egy példát a rendellenes végrehajtásra:

1. Férj szála - belép a `levesz` metódusba és végrehajtja a feltételvizsgálatot, amely alapján lehetővé válik az összeg kivétele.
2. Feleség szála - belép a `levesz` metódusba és ellenőrzi, hogy a kívánt összeg levehető-e
3. Feleség szála - folytatja a végrehajtást és leveszi a számláról a kívánt összeget
4. Férj szála - folytatódik a feltételhez kötött blokk végrehajtásával, és már a feleség által megváltoztatott egyenlegből fogja kivonni a kért összeget

Azért, hogy a fenti eset ne fordulhasson elő, a `levesz` metódus végrehajtását úgy kellene végezni, hogy egyszerre csak egy programszál hajthassa végre.

14.6.2. Objektumok és zárak

Egy Java program minden egyes objektumához tartozhat egy zár. Ennek a zárnak a feladata biztosítani, hogy az objektumhoz egyszerre csak egy szálnak legyen hozzáférése. Amennyiben valamely szál megszerezte ezt a zárat, más szálnak nem lesz hozzáférése az objektumhoz, így biztosítható, hogy a szál állapotát egyszerre csak egy szál módosíthatja. Amikor a szál, amely lekötötte a zárat, befejezte a tevékenységét, elengedi a zárat, így más szálak is megszerezhetik azt. Ezt a zárolási mechanizmust szinkronizálásnak nevezzük.

Szintaktikailag két egységet láthatunk el a `synchronized` kulcsszóval, kódblokkot és metódust. Mivel a zár objektumhoz tartozik, amennyiben kódblokkot látunk el a `synchronized` kulcsszóval, azt is meg kell mondani, hogy mely objektum zárolódik. Ez lehetőséget biztosít, hogy bármely metódusban bármely objektumot zároljunk. A metódus szinkronizálása úgy történik, hogy egyszerűen ellátjuk a metódust a `synchronized` kulcsszóval. Ez viszont nem tartozik hozzá a metódus szignatúrájához, vagyis ez a módosító nem öröklődik. Tekintsünk mindkettőre egy-egy példát:

```
synchronized void f(){ ... }

void f(){
   ...
   synchronized(obj){ ... }
}
```

Ha metódus-szinkronizációt használunk, a metódust hívó objektum egészen addig zárolódik, amíg le nem fut a metódus. A kódblokk-szinkronizáció általánosabb, hiszen lehetőséget teremt, hogy ne csak a metódust hívó objektumot zároljuk, hanem bármelyiket, sőt, a zárolást korlátozhatjuk egy kódblokk lefutásának idejére.

Az objektum zárolása és elengedése automatikusan történik a szinkronizált blokk elején és végén. Így nem történhet meg, hogy elfelejtjük elengedni a zárt.

14.6.3. Monitorok

A Java szálak monitorokat használnak az alkalmazás kritikus szekcióihoz való hozzáférés vezérlésére. A monitor alapjában véve egy ZÁR, melyet egy bizonyos kódrészlet védelmére használunk. Amikor egy szál belép egy kritikus kódrészletbe, lefoglal egy monitort. Ha egy másik szál ugyanebbe a kritikus szekcióba akar belépni, akkor neki várnia kell, amíg a monitor felszabadul. A monitor a MUTEX (`Mut`ual `Ex`clusive - kölcsönös kizárású zár) zár egy típusa. A mutex zárakat viszont kifejezetten le kell foglalni, illetve fel kell szabadítani. A Java nyelvben a monitor lefoglalását és elengedését a környezet (JVM) látja el. A programozónak a feladata kijelölni a kritikus szekciókat és ellátni ezeket a `synchronized` kulcsszóval.

Monitora minden Java objektumnak, illetve osztálynak lehet. Egy szál akkor foglalhat le egy objektum-monitort, amikor annak szinkronizált metódusába lép be. Amennyiben egy osztálynak van osztályszintű (statikus) metódusa, és egy szál ennek végrehajtását kéri, akkor a szál ezt az osztálymonitort fogja lefoglalni. A Java monitorok újra hívhatók (`re-entrant`), ami azt jelenti, hogy ha egy szál lefoglalt egy monitort egy adott metódus hívásának következményeként, utána ugyanazon objektum összes többi szinkronizált metódusát hívhatja.

```
public class Reentrant{
   public synchronized void m1(){
      m2();
      System.out.println("m1 metódus hívása");
   }


   public synchronized void m2(){
      System.out.println("m2 metódus hívása");
   }

}
```

MONITOR = ZÁR + VÁRAKOZÁSI_LISTA

A Java monitorok nemcsak zárak, hanem várakozási listák is egyidejűleg, azaz tárolják a monitorra várakozók listáját. A várakozási lista kezelését a JVM végzi. Ez a várakozási lista olyan szálakból áll, amelyeket a `wait()` metódushívás blokkolt.

14.6.4. Szálak közötti jelzésrendszer

Bizonyos esetekben nemcsak szinkronizációra van szükség, hanem egy jelzésrendszerre is a szálak között. Jó lenne például szálakat várakoztatni adott feltétel teljesüléséig, a feltétel teljesülésekor pedig jelet kapna a szál. Ezt más szálcsomagok feltételváltozók bevezetésével oldják meg, a Java nyelv pedig három metódussal segíti a jelzésrendszert: `wait(), notify(), notifyAll()`. Ezeket a metódusokat csak szinkronizált kódrészletből lehet hívni.

A `wait()` hatására a következők történnek:

- A JVM a szálat a megfelelő objektum várakozási listájába helyezi.
- A `wait()` metódust hívó objektum elengedi a zárat.

A `notify()` metódus következménye:

- Ha létezik várakozó szál az objektum várakozási listájában, akkor ezek közül a JVM kiválaszt egyet és ez megpróbálja megszerezni a monitort.
- A kiválasztott szálnak is legalább addig kell várakoznia, amíg a `notify()`-t hívó szál elenegedi a monitort. Amennyiben több szál van a várakozási listában, akármelyiket kiválaszthatja a JVM.
- A kiválasztott szál onnan folytatja tevékenységét ahol azt előzőleg abbahagyta.

A `notifyAll()` a `notify()` metódushoz hasonlóan működik azzal a különbséggel, hogy ilyenkor az összes várakozó szál jelzést kap és mindenik megpróbálja megszerezni a monitort. Mivel mindenik ugyanarra a monitorra várakozik, csak egynek fog sikerülni a továbblépés.

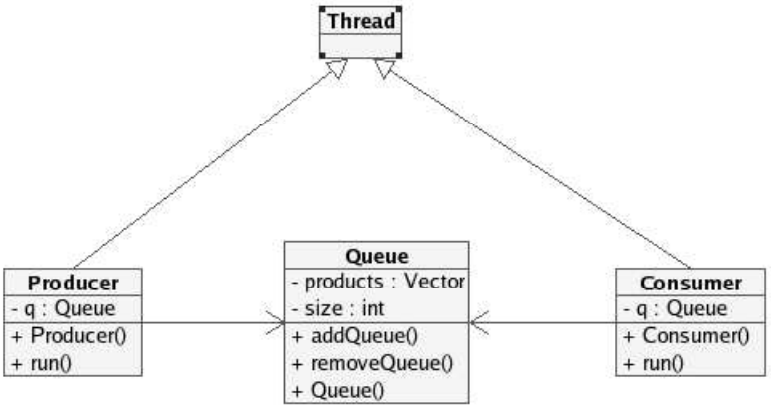
Amennyiben több szál kap jelzést, ezek mindenike megpróbálja újra lefoglalni a monitort. Természetesen ezek közül csak egy szerezheti meg a monitort. Pontosan ezért a jelet kapó szálnak ismét ellenőriznie kell a feltételt. Ezt úgy szokás megoldani, hogy ciklusba helyezzük a `wait()` metódus hívását. Például:

```
while( feltétel )
   wait();
```

Amikor egy feltételre váró szál jelet kap, pontosan ott folytatja a végrehajtást ahol ezt abbahagyta, vagyis a ciklust fogja folytatni és továbblépés csak a feltétel megváltozásakor következik be.

14.6.5. Termelő-Fogyasztó probléma

Tegyük fel, hogy adott egy korlátos méretű sor, amelyet közösen használ egy termelő és egy fogyasztó szál. A termelő szálnak az a feladata, hogy termeljen, és a termékeket helyezze a sorba. Mivel a sor korlátos méretű, amennyiben



14.5. ábra. Termelő-Fogyasztó probléma

ez betelik, a termelőnek várakoznia kell. Ha viszont a fogyasztó túl gyorsan fogyasztja a termékeket, a sor kiürül és ennek következtében a fogyasztó kényszerül várakozásra. A probléma megoldásában két dologra kell figyelnünk:

(1) A sor lesz a kritikus szekció a programunkban, tehát a sorhoz való hozzáférést szinkronizálnunk kell. A sort alapvetően a betesz és a kivesz műveletek segítségével kezeljük.

(2) Használnunk kell a jelzésrendszert a termelő és a fogyasztó szál közötti kommunikációra.

A feladat osztálydiagramját az 14.5. ábra szemlélteti.

```
import java.util.*;
public class Queue{
   private Vector products;
   private int size;

   public Queue( int size ){
      this.size = size;
      products = new Vector( size );
   }
```

```
   public synchronized void addQueue( Object o ) {
      while( products.size() == size ){
         System.out.println(Thread.currentThread().getName()+
                                 " várakozik");
         try{
            wait();
         }
         catch( InterruptedException e ){}
      }
      products.addElement( o );
      System.out.println(Thread.currentThread().getName()+
                                " beteszi "+( String )o);
      notify();
   }

   public synchronized void removeQueue() {
      while( products.size() == 0 ){
         System.out.println(Thread.currentThread().getName()+
                                     " várakozik ");
         try{
            wait();
         }
         catch( InterruptedException e ){}
      }
      Object o = products.remove(0);
      System.out.println( Thread.currentThread().getName()+
                                   " kiveszi "+ (String)o);
      notify();
   }
}

public class Producer extends Thread{
   Queue q;
   public Producer( Queue q ){
```

```
      this.q = q;
   }

   public void run(){
      for( int i=0; i<10; i++ ){
         String s = "Product "+i;
         q.addQueue( s );
      }
   }
}

public class Consumer extends Thread{
   Queue q;

   public Consumer( Queue q ){
      this.q = q;
   }

   public void run(){
      for( int i=0; i<10; i++ )
         q.removeQueue();
   }
}

public class Main{
   public static void main( String args[] ){
      Queue queue= new Queue( 3 );
      Producer p = new Producer( queue );
      Consumer q = new Consumer( queue );
      p.start(); q.start();
   }
}
```

14.7. Feladatok

1. Mikor fejeződik be egy Java program futása?

   a) Amikor végrehajtódott a main összes utasítása

   b) Amikor minden nem démon típusú szál befejeződik

   c) Amikor kivétel keletkezik

   d) Amikor befejeződik a run metódus

2. Hogyan lehet az osztott változók biztonságát garantálni egy többszálas környezetben?

   a) Minden változót a synchronized kulcsszóval vezetünk be

   b) Minden változót a volatile kulcsszóval vezetünk be

   c) Csak statikus változókat használunk

   d) A változókat szinkronizált metódusokon keresztül érjük el

3. Mit nem lehet szinkronizálni?

   a) metódusban kódblokkot

   b) statikus metódust

   c) statikus metódusban kódblokkot

   d) osztályt

4. Hány zár tartozik egy példányhoz?

   a) Egy

   b) Egy-egy darab minden egyes metódusához

   c) Egy-egy darab minden egyes szinkronizált metódusához

   d) Egy-egy darab minden egyes szinkronizált, nem statikus metódusához

5. Válasszuk ki az egyetlen helyes kijelentést az alábbiak közül!

```
import java.util.Vector;

class A extends Vector implements Runnable{
   public void run( String msg){
      System.out.println(msg);
   }
}

public class B{
   public static void main( String args[] ){
      A a = new A();
      Thread t = new Thread( a );
      t.start();
   }
}
```

a) Fordítási hiba, mert az A osztály helytelenül implementálja a Runnable interfészt

b) Fordítási hiba, mert a Thread osztálynak nincsen paraméteres konstruktora

c) Helyes fordítás, de futásidőben kivétel váltódik ki

d) Helyes fordítás, helyes végrehajtás

6. Válasszuk ki az egyetlen helyes kijelentést az alábbiak közül!

```
class A extends Thread{
   A(){ setPriority( 10 );}
   public void run( ){
      System.out.println(getName()+" is running");
      while( true ){}
   }
   public static void main( String args[] ){
      A t1 = new A();
      A t2 = new A();
      A t3 = new A();
```

```
        t1.start();
        t2.start();
        t3.start();
    }
}
```

a) Csak a t1 szál fog futni, a másik kettő soha nem fogja megkapni a vezérlést

b) Mindhárom szál lefut és sikeresen befejeződik a program

c) Mindhárom szál párhuzamosan fog futni egy-egy időszelet erejéig

d) Nem lehet pontosan előrejelezni, hogy mi fog történni.

7. Válasszuk ki az egyetlen helyes kijelentést az alábbiak közül!

```
public class MyThread{
    static Thread makeThread( final String id, boolean daemon){
        Thread t = new Thread(id){
            public void run(){
                System.out.println( id+" ");
            }
        };
        t.setDaemon(daemon);
        t.start();
        return t;
    }

    public static void main( String args[] ){
        Thread t1 = makeThread("T1", false );
        Thread t2 = makeThread("T2", false );
        System.out.print("End ");
    }
}
```

a) A T1 szöveget soha nem írja ki

b) A T2 szöveget mindig kiírja

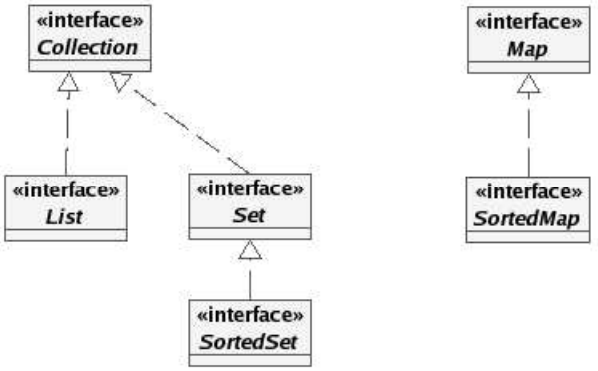
c) Egy lehetséges kimenet T1 End T2

15. FEJEZET

Tárolók

15.1. Bevezetés

A Java csomagok közül kiemelt szerepe van a Java tárolóknak. Ezeket a tárolókat a `java.util` csomag tartalmazza a szükséges interfészekkel együtt. A Java 2 előtt is tartalmazott a Java tárolókat, de ezek nagyon primitívek voltak, például a `java.util.Vector` vagy a `java.util.Hashtable`. A Java 2 által bevezetett tárolókat még gyűjtemény keretrendszernek is nevezzük (`Collections Framework`). Az 15.1. ábra a fontosabb interfészeket szemlélteti. A `Collection` interfész és leszármazottai alapvetően egy értékhalmaz tárolására alkalmasak, míg a `Map` és leszármazottai `(kulcs, érték)` elempárok tárolására alkalmasak. A `Map` interfész egy asszociatív tároló megvalósításához szükséges műveleteket definiálja. Az interfészeket megvalósító legfontosabb osztályokat a 15.1. táblázat tartalmazza. A táblázat tartalmazza a Java 2 előtti osztályokat is a második oszlopban.



15.1. ábra. Interfészek tárolókhoz

Interfész	Java 2 osztály	Java 2 előtti osztály
`Set`	`HashSet`	
	`TreeSet`	
`List`	`ArrayList`	`Vector`
	`LinkedList`	`Stack`
`Map`	`HashMap`	`Hashtable`
	`TreeMap`	`Properties`

15.1. táblázat. Java tárolók

Metódus	Szerepe
`add (Object)`	új elem hozzáfűzése
`addAll (Collection)`	egy egész tároló hozzáfűzése
`clear ()`	az összes elem törlése
`boolean contains (Object o)`	egy elem bennfoglalásának ellenőrzése
`boolean containsAll (Collection)`	egész tároló bennfoglalásának ellenőrzése
`boolean isEmpty ()`	tároló ürességének ellenőrzése
`Iterator iterator ()`	bejáró visszatérítése
`remove (Object o)`	adott elem törlése
`removeAll (Collection)`	a paraméterben megadott összes elem törlése
`retainAll (Collection)`	összes elem törlése a paraméterben megadottak kivételével
`int size()`	a tároló elemeinek száma
`Object[] toArray ()`	tároló átalakítása objektumtömbbé

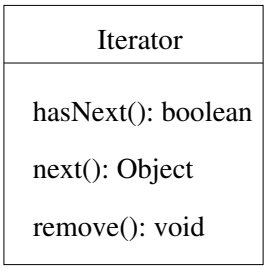
15.2. táblázat. A `Collection` interfész metódusai

15.2. A Collection interfész

A `Collection` interfész metódusait a 15.2. táblázat tartalmazza.

Az interfészek elősegítik az absztrakciót, segítségükkel lehetővé válik, hogy átalános kódrészleteket készítsünk. Ezek a kódrészletek működni fognak bármely olyan tárolóra, amely a `Collection` interfészt implementálja. Készítsük el például egy tároló bejárását általánosan. A tároló bejárását az Iterator interfész segítségével fogjuk végezni. Ez szintén a Java 2 újítása és a 15.2. ábra szemlélteti.

```
public static void printCollection( Collection c ){
   Iterator it = c.iterator();
   while( it.hasNext() ){
      System.out.println(it.next());
```



15.2. ábra. Iterator interfész

```
   }
}
```

15.3. A List interfész

Ez az interfész a lista műveleteket deklarálja. A lista bármilyen elemeket, bármilyen sorrendben tartalmazhat. A `Collection` interfészhez képest tartalmaz néhány kiterjesztést, éspedig elemek pozíció szerinti elérése, keresés, egy új listaiterátor és részlista kezeléshez szükséges műveletek.

Az `add` és az `addAll` műveletek a megadott elemeket a lista végéhez adják hozzá. A listák iterátorai `ListIterator` típusúak, ez pedig több műveletet tartalmaz, mint az `Iterator` típus. A `next` művelet mellett létezik egy `previous` művelet is a fordított sorrend biztosításához, illetve a `hasNext` műveletnek van egy `hasPrevious` megfelelője. A `subList` művelet két megadott index közé eső elemekből részlistát képez.

A `List` interfésznek két implementációját tartalmazza a Java, az `ArrayList` és a `LinkedList` osztályokat. Példaként képezzünk ezen osztályok felhasználásával veremszerű, illetve sorszerű listát:

```
import java.util.*;

class MyStack extends ArrayList{
   public void push( Object o ){add(o);}
   public Object pop(){return remove(size()-1);}
```

```
   public Object top(){return get(size()-1);}
}

class MyQueue extends LinkedList{
   public void enqueue( Object o ){addLast( o );}
   public Object dequeue(){return removeFirst();}
}

public class MainList{
   public static void main( String args[] ){
      MyStack s = new MyStack();
      for( int i=0; i<10; ++i )
         s.push( new Integer( i ));
      while( !s.isEmpty() ){
         System.out.println( (Integer) s.pop() );
      }

      MyQueue q = new MyQueue();
      for( int i=0; i<10; ++i )
         q.enqueue( new Integer( i ));
      while( !q.isEmpty() ){
         System.out.println( (Integer) q.dequeue());
      }
   }
}
```

A fenti program nagy hibája, hogy származtatást használ, és ennek következtében az utódosztály átveszi az ősosztály viselkedésmódját is. Ennek következtében nincs semmiféle garancia arra nézve, hogy a `MyStack` objektumokat csak a veremműveleteken keresztül kezeljük, illetve a `MyQueue` objektumokat csak a sorműveleteken keresztül. A `MyStack` objektumokra alkalmazható lesz az `ArrayList` összes művelete és a `MyQueue` objektumokra pedig a `LinkedList` összes művelete. Ha viszont nem származtatást használnánk, hanem tartalmazási kapcsolatot, akkor az így kapott osztályok példányai viselkedését nem befolyásolná az

`ArrayList`, illetve a `LinkedList`. Következtetésképp nagyon óvatosnak kell lenni a származtatással, lehetőleg megfontoltan használjuk.

Most pedig készítsük el a `MyStack` és a `MyQueue` osztályok megvalósítását tartalmazási kapcsolattal. A `main` metódus nem módosul, az előző működik az új típusokra is:

```
class MyStack{
   private ArrayList elements=new ArrayList();

   public void push( Object o ){
      elements.add(o);
   }

   public Object pop(){
      return elements.remove(elements.size()-1);
   }

   public Object top(){
      return elements.get(elements.size()-1);
   }

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

class MyQueue{
   private LinkedList elements = new LinkedList();

   public void enqueue( Object o ){
      elements.addLast( o );
   }

   public Object dequeue(){
      return elements.removeFirst();
```

```
   }

   public boolean isEmpty(){
      return elements.isEmpty();
   }
}
```

Figyeljük meg a különbséget a két implementáció között. Amíg az első változatban az üres állapotot lekérdező `isempty()` metódust örökléssel megkapta az új típus, addig a második változatban ezt explicit módon deklarálnunk kell. A második változat előnye, hogy az osztályok garantáltan csak az általunk megadott viselkedésmódot fogják biztosítani.

15.4. A Set interfész

A `Set` interfész kiterjeszti a `Collection` interfészt oly módon, hogy ez egy matematikai halmazt ábrázoljon. A matematikai halmaz sajátossága, hogy minden elem pontosan egyszer szerepelhet benne. A beszúrási metódusok az `equals` metódussal fogják ellenőrizni, hogy a beszúrandó elem megegyezik-e a halmaz valamely elemével.

A `Set` interfész nem tartalmaz új metódusokat a `Collection` interfészhez képest. A `Set` interfésznek két implementációját is tartalmazza a `java.util` csomag, éspedig a `HashSet` és a `TreeSet` osztályokat. A `HashSet` osztályt rendezetlen halmazok ábrázolására használjuk és a `TreeSet` osztályt rendezett halmazokra. A `TreeSet` kiegyensúlyozott bináris fával van ábrázolva, így ennek következtében az alapvető műveletek, mint beszúrás, törlés és keresés logaritmikus időigényűek. Ha rendezettségre is szükség van, általában gyorsabb egy `HashSet`-et felépíteni és ebből létrehozni egy `TreeSet` példányt. A következő program ezt szemlélteti:

```
import java.util.*;

public class HashSetMain{

   public static void printCollection( Collection c ){
```

```
      Iterator it = c.iterator();
      while( it.hasNext() )
         System.out.print( it.next()+" ");
      System.out.println();
   }

   public static void main(String args[]){
      HashSet m = new HashSet();
      m.add("Marika");
      m.add("Boti");
      m.add("Kriszti");
      m.add("Kinga");
      printCollection( m );
      TreeSet t = new TreeSet( m );
      printCollection( t);
   }
}
```

A program kimenete:

```
Kinga Kriszti Boti Marika
Boti Kinga Kriszti Marika
```

15.5. A Map interfész

A `Map` interfész asszociatív tárolót definiál, amelybe (kulcs, érték) elempárokat helyezhetünk. Minden kulcs csak egyszer szerepelhet a tárolóban és minden kulcsnak csak egyetlen értéket lehet megfeleltetni. Természetes a megfeleltetett érték bármely típusú lehet, akár tároló típusú is. Ezzel lehet megoldani az $1 \rightarrow n$ megfeleltetést. Az interfész metódusait három csoportba sorolhatjuk:

- karbantartó műveletek: beszúrás, törlés
- lekérdező műveletek
- különböző nézetek létrehozását célzó műveletek

A karbantartó műveletek a következők:

```
Object put( Object key, Object value );
Object remove( Object key );
void putAll(Map m );
void clear();
```

A lekérdező műveletek esetében is annyira "beszédes" neveket adtak a metódusoknak, hogy nem szükséges magyarázni a viselkedésüket.

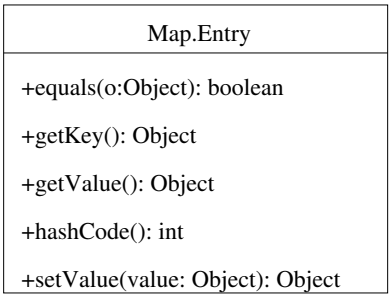
```
Object get( Object key );
boolean containsKey(Object key);
boolean containsValue( Object value );
int size();
boolean isEmpty();
```

Egy asszociatív tárolóról háromféle nézetet kaphatunk meg. Kérhetjük csak a kulcsokat halmaz (`Set`) típusként, vagy csak a kulcsokhoz rendelt értékek halmazát `Collection` típusként, vagy a harmadik lehetőség a (kulcs, érték) párok halmazként való lekérdezése.

```
Set keySet();
Collection values();
Set entrySet();
```

A `Map` interfész tartalmaz egy belső statikus interfészt a `(kulcs, érték)` párokkal végezhetető műveletekre. Ezeket a műveleteket szemlélteti a 15.3. ábra. A `setValue()` metódus beállítja értékként a paraméterben megadott objektumot és visszatéríti a régit. A többi művelet nem szorul magyarázatra.

A gyűjtemény keretrendszer az asszociatív tárolók két megvalósítását kínálja . Mindkettő a `Map` interfészt implementálja, az egyik a `HashMap`, a másik pedig a `TreeMap`. Amennyiben csak beszúrás, törlés és keresés műveletekre akarjuk használni a tárolót, a legjobb választás a `HashMap` lesz. Ha rendezett sorrend előállítása is gyakori, akkor a `TreeMap` lesz a megfelelő választás. A `HashMap` osztály használata feltételezi, hogy a tárolandó elemek típusának van egy jóldefiniált `hashCode()` metódusa. A `TreeMap` pedig feltételezi, hogy a tárolandó



15.3. ábra. A Map.Entry interfész

elemek összehasonlíthatók. A `HashMap` teljesítményét hangolhatjuk a kezdeti kapacitás és a telítettségi faktor beállításával. A `TreeMap` teljesítményét nem lehet hangolni, hiszen ez egy kiegyensúlyozott keresőfa.

Most pedig tekintsünk egy példát asszociatív tároló használatára. Készítsünk szóelőfordulási statisztikát egy szövegállományról. Ez azt jelenti, hogy programunk bemenete egy szövegállomány lesz, míg a kimenet egy (szó, előfordulás_szám) halmaz. A szó típusának `String` típust fogunk használni, az előfordulások számának pedig bevezetünk egy saját egész típust, ugyanis az `Integer` típus csak konstans egész értékeknek biztosít burkolatot, a mi feladatunkban viszont az előfordulások száma futásidőben változik. Legyen a saját egész típusunk neve `MyInteger`, amelynek lesz beállító (`setValue()`) meg lekérdező (`getValue()`) metódusa egyaránt:

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

class MyInteger{
   private int i;

   public MyInteger( int i ){
      this.i = i;
   }

   public void setValue( int i ){
```

```
      this.i = i;
   }

   public int getValue(){
      return i;
   }

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

A feladat most már egy banális feladattá alakult:

- Soronként olvassuk a szövegállományt.
- A sorokat szavakra bontjuk a `StringTokenizer` osztály segítségével.
- Amennyiben az asszociatív tárolónk nem tartalmazza az aktuális szót, hozzádjuk ezt a tárolóhoz 1-es előfordulásszámmal. Ha a tároló tartalmazza az aktuális szót, növeljük az előfordulások számát.
- A legvégén kiíratjuk a (kulcs, elem) párok halmazát.

A teljes megoldást a következő program szemlélteti:

```
public class MainMap{
   public static void main(String args[]) throws Exception{
      TreeMap m = new TreeMap( );
      BufferedReader br = new BufferedReader(
         new FileReader("in1.txt"));
      String line;
      while( (line = br.readLine()) != null ){
         StringTokenizer stk = new StringTokenizer(line);
         while(stk.hasMoreTokens()){
            String word = stk.nextToken();
            if( m.containsKey( word ) ){
```

```
                MyInteger ival = (MyInteger)(m.get(word ));
                int v = ival.getValue()+1;
                ival.setValue( v );
             }
             else
                m.put(word, new MyInteger(1));
          }
       }
       //Kimenet
       Iterator it = m.entrySet().iterator();
       while( it.hasNext() ){
          Map.Entry e = (Map.Entry) it.next();
          String word = (String)e.getKey();
          MyInteger frequency =(MyInteger)e.getValue();
          System.out.println( word+" : "+frequency );
       }
    }
}
```

Az asszociatív tárolóktól nem lehet direkt módon iterátor objektumot lekérni. Biztosítanak viszont három metódust, amelyek segítségével más tárolóvá lehet alakítani a bennük tárolt elemeket. Ez a másik tároló vagy `Set` vagy pedig `Collection` típusú, amelytől már lehet kérni iterátort.

15.6. Rendezés

Egy gyűjtemény rendezése kétféleképpen történhet. Vagy maga a gyűjtemény rendezett és a gyűjtemény bejárásával már meg is van a rendezett sorrend, vagy a gyűjtemény rendezetlen és valamely rendezési algoritmussal előállítjuk a gyűjtemény rendezett állapotát (pl. `Collections.sort`). A növekvő sorrend előállításához szükséges, hogy a tartalmazott elemek típusán értelmezve legyen egy rendezési reláció. Természetesen megköveteljük, hogy a tároló azonos típusú elemeket tartalmazzon, amelynek következménye, hogy ezek az elemek összehasonlíthatóak. A Java 2 bevezette a `Comparable` típust. A `Comparable` interfészt a

`java.lang` csomag tartalmazza és ez közös őst biztosít az összehasonlítható típusoknak. A Java beépített típusai implementálják ezt az interfészt. Alapértelmezetten ez numerikus típusoknál növekvő sorrendet, karakterláncoknál alfabetikus sorrendet és dátumoknál kronologikus sorrendet jelent.

```
public interface Comparable{
   public int compareTo( Object o );
}
```

Ha egy `x` objektum `Comparable` típusú, akkor

$$x.compareTo(o) = \begin{cases} + & , ha\ x > o \\ 0 & , ha\ x = o \\ - & , ha\ x < o \end{cases}$$

Amennyiben egy saját típust rendezett tárolóba akarunk betenni, akkor ezt úgy is biztosíthatjuk, hogy a típusunkat úgy tervezzük, hogy implementálja a `Comparable` interfészt.

Elképzelhető viszont egy olyan helyzet is, hogy a típusnak van egy alapértelmezett összehasonlítsi módja, azonban bizonyos esetekben más szempontok szerint kell rendezni. Például a `String` típus alapértelmezetten alfabetikusan rendez, de sokszor arra van szükségünk, hogy ne tegyen különbséget kis és nagy betűk között vagy éppen csökkenő sorrendre van szükség. Az ilyen esetekre vezették be a `Comparator` interfészt:

```
public interface Comparator{
   public int compare( Object o1, Object o2 );
   public boolean equals( Object o);
}
```

Az `equals` metódus az aktuális `Comparator` objektum és a paraméter egyenlőségét vizsgálja. Akkor térít vissza igaz értéket, ha a paraméter is `Comparator` típusú és ugyanazt a rendezettséget implementálja, mint az aktuális objektum.

Például, ha egy tárolóba karakterfüzéreket akarunk betenni és szeretnénk, ha a karakterfüzér hossza szerint végezné a rendezést, akkor elkészítünk egy `Comparator` interfészt megvalósító típust, amelyben megadjuk ezt a fajta rendezést, ezt a típust pedig átadjuk a tárolónak. Tekintsük erre a következő példát,

amelyben létrehozunk egy-egy tárolót, egyikben a `String` osztály alapértelmezett rendezését használjuk, a másodikban megadunk egy saját rendezési kritériumot:

```
import java.util.*;

class StringLengthComparator implements Comparator{
   public int compare( Object o1, Object o2 ){
      String s1 = (String) o1;
      String s2 = (String) o2;
      return s1.length()-s2.length();
   }
}

public class MainComparator{
   public static void printCollection( Collection c ){
      Iterator it = c.iterator();
      while( it.hasNext() )
         System.out.print(it.next()+" ");
      System.out.println();
   }

   public static void main(String args[]){
      TreeSet t1 = new TreeSet();
      TreeSet t2 = new TreeSet(new StringLengthComparator());
      String [] t = {"alma", "eperfa", "cseresznye", "dio"};
      for( int i=0; i<t.length; ++i ){
         t1.add(t[i]);
         t2.add(t[i]);
      }
      System.out.print("Alfabetikus sorrend: ");
      printCollection( t1 );
      System.out.print("Hosszusag szerinti sorrend:");
      printCollection( t2 );
   }
}
```

Az előző program eredménye:

```
Alfabetikus sorrend: alma cseresznye dio eperfa
Hosszusag szerinti sorrend: dio alma eperfa cseresznye
```

15.7. Kiegészítő eszközök

Ezek az eszközök a `java.util.Collections` osztályban vannak elhelyezve. (Nem tévesztendő össze a `java.util.Collection` interfésszel)

A burkoló implementációk olyan objektumok, melyek egy meglévő gyűjtemény- vagy asszociatív tároló típusú objektumot magukba foglalnak, kiegészítve ezt új funkcionalitásokkal. Ezek a burkoló implementációk a díszítő tervezési mintára ([4]) épülnek. Kétféle burkoló implementációt fogunk ismertetni, az első egy szinkronizációs burokkal látja el a tárolót, a második pedig egy módosíthatatlansági burokkal.

A Java `Vector` és `Hashtable` osztályai biztosítottak szinkronizációt, a Java 2 által bevezetett tárolók viszont nem biztosítanak. A tárolók tervezői mondtak le a szinkronizációról a hatékonyság érdekében. Ezzel a felelősséget a programozóra hárították, egyben megteremtve a lehetőségét annak, hogy egy tárolót átalakíthassunk szinkronizálttá. A következő hat metódus biztosítja a szinkronizált tárolópéldányok létrehozását:

```
Collection synchronizedCollection( Collection );
List synchronizedList( List );
Set synchronizedSet( Set );
Map synchronizedMap( Map );
SortedSet synchronizedSortedSet( SortedSet );
SortedMap synchronizedSortedMap( SortedMap )'
```

Amennyiben több szálban akarunk egy tárolóhoz hozzáférni, akkor ennek szinkronizált változatát hozzuk létre. Viszont a szinkronizált változatot mindig a nem szinkronizált objektumból képezzük. Ilyen esetben legyünk óvatosak, ha van már szinkronizált változata a tárolónak, akkor mindenképpen csak ezen a referencián keresztül kezeljük a tárolót, ne is tartsunk meg referenciát a nemszinkronizált változatból. A tárolók szinkorinizálttá tehetők, de a tárolókhoz

tartozó iterátorokra ez nem vonatkozik. Ezért az iterátorok használatát külön szinkronizációs blokkba kell helyezni, zárolva a tárolót a hívás idejére:

```
import java.util.*;

public class TestSyncCollection{
   public static void main(String args[]){
      //nem tartjuk meg a referenciat
      //a nem szinkronizalt peldanyra}
      Set s = Collections.synchronizedSet( new HashSet());
      s.add("alma");
      s.add("dio");
      s.add("mogyoro");
      synchronized( s ){
         Iterator i = s.iterator();
         while(i.hasNext())
            System.out.println(i.next());
      }
   }
}
```

A nem módosítható tárolókat a szinkronizáltakhoz hasonlóan hozzuk létre azzal a különbséggel, hogy a `synchronized` szó helyett az `unModifiable` szót használjuk.

15.8. Feladatok

1. Szavakat olvasunk be a szabványos bementről. Bontsuk anagramma osztályokra a beolvasott szavakat.

Bemenet: `suta, kap, tusa, pak, suta, kép, utas, pék`

Kimenet:

1. `kap, pak`

2. `kép, pék`

3. `suta, tusa, utas`

Megoldási vázlat

Szavanként olvassuk a bemenetet.

Minden egyes szónak megfeleltetünk egy olyan szót, amely a beolvasott szó karaktereit rendezett sorrendben tartalmazza. Pl. *kap* → *akp*

Egy asszociatív tárolót fogunk használni, amelyben a kulcsnak a rendezett karaktereket tartalmazó szót fogjuk használni, értéknek pedig egy `Vector` típusú tárolót.

```
suta - astu
kap  - akp
tusa - astu
pak  - akp
suta - astu
kép  - ékp
utas - astu
pék  - ékp
```

kulcs	érték
akp	kap, pak
astu	utas, tusa, suta
ékp	kép, pék

A megoldást a következő program tartalmazza:

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

public class Anagramma{
   public static void main( String args[] )
                                throws Exception{
```

```
      BufferedReader bin = new BufferedReader(
              new InputStreamReader(System.in));
      String word=null;
      //Feltöltés
      TreeMap t = new TreeMap();

      while( (word = bin.readLine() ) != null ){
         char c[]= word.toCharArray();
         Arrays.sort(c);
         String sortedWord= new String ( c );
         if( !t.containsKey( sortedWord )){
            Vector v = new Vector();
            v.addElement( word );
            t.put(sortedWord, v );
         }
         else{
            Vector v = (Vector) t.get( sortedWord);
            v.addElement( word );
         }
      }

      //Kiíratás
      Iterator it = t.entrySet().iterator();
      while( it.hasNext() ){
        Map.Entry entry = (Map.Entry)it.next();
        String w = ( String )entry.getKey();
        Vector v = ( Vector )entry.getValue();
        System.out.print( w +":");
        for( int i=0; i<v.size(); ++i )
           System.out.print( (String) v.elementAt(i)+"");
        System.out.println();
      }
   }
}
```

2. Készítsen helyesírásellenőrző programot egy adott szótárhoz.

A következő program ezt szemlélteti feltételezve, hogy adott egy angol szótár az ENGLISH.TXT nevű állományban, amely egy olyan szöveges állomány, amelynek minden sora egy angol szót tartalmaz. Az alábbi megvalósítás egy adott szöveg minden szavára ellenőrzi, hogy benne van-e a szótárban. A szótárt előzetesen betölti egy gyors keresést biztosító tárolóba.

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

class Dictionary{
   private HashSet words;

   public Dictionary( String fileName){
      words = new HashSet();
      int nrlines = 0;
      try{
         BufferedReader bin = new BufferedReader(
                        new FileReader( fileName ));
         String word;
         while((word = bin.readLine())!=null)
            words.add( word.toLowerCase() );
      }
      catch( Exception e ){
         System.out.println("File open exception:"+e);
      }
   }

   public int size(){
      return words.size();
   }

   public boolean check( String word ){
      return words.contains( word );
   }
```

```
}

class SpellChecker{
   private Dictionary dict;
   public SpellChecker( String dictionaryFile ){
      dict = new Dictionary( dictionaryFile );
   }
   public boolean checkWord( String word ){
      return dict.check( word.toLowerCase() );
   }
   public Vector checkLine( String line ){
      Vector output = new Vector();
      StringTokenizer stk = new StringTokenizer(line," ,.:?!");
      String word = null;
      while(stk.hasMoreTokens()){
         word = stk.nextToken();
         if(!dict.check(word.toLowerCase()))
            output.addElement( word );
      }
      return output;
   }
   public HashMap checkFile( String fileName )
                            throws IOException{
      HashMap output = new HashMap();
      int lineCounter = 0;
      BufferedReader file = new BufferedReader(
                  new FileReader(fileName));
      String line = null;
      while( (line = file.readLine() ) != null ){
         ++lineCounter;
         Vector errors = checkLine( line );
         if( errors.size() > 0 )
            output.put( new Integer(lineCounter),errors );
      }
      return output;
```

```
   }
   public void printErrors( Vector v ){
      Iterator it = v.iterator();
      while( it.hasNext() )
         System.out.print( it.next() +"");
      System.out.println();
   }
   public void printErrors( HashMap map ){
      TreeMap m = new TreeMap( map );
      Iterator it = m.entrySet().iterator();
      while( it.hasNext() ){
         Map.Entry entry = (Map.Entry)it.next();
         System.out.print("Line "+entry.getKey()+":");
         printErrors( (Vector)entry.getValue());
     }
   }
}

public class DictionaryMain{
   public static void main( String args[] ){
      SpellChecker spk = new SpellChecker("ENGLISH.TXT");
      BufferedReader bin = null;
      try{
         bin = new BufferedReader(new InputStreamReader(System.in));
      }
      catch( Exception e){}
      String what = null;
      while( true ){
         System.out.print("What to find (type quit to exit):");
         try{
            what = bin.readLine();
            Vector v = spk.checkLine( what );
            for( int i=0; i<v.size(); ++i )
               System.out.print( v.elementAt( i ));
            System.out.println();
```

```
            }
            catch( IOException e ){}
            if( what.equals("quit")) break;
        }
        try{
           spk.printErrors( spk.checkFile("in.txt") );
        }
        catch( Exception e ){}
    }
}
```

SZAKIRODALOM

[1] ANGSTER, Erzsebet, Objektumorientált tervezés és programozás, 4KÖR Bt., 2002.

[2] CZARNECKI, Krzysztof, EISENECKER, Ulrich W., Generative Programming, Addison-Wesley, 2000.

[3] ECKEL, Bruce, Thinking in Java, Prentice Hall, 1999.

[4] GAMMA, Erich, HELM, Richard, JOHNSON, Ralph, VLISSIDES, John, Programtervezési minták, Kiskapu, 2004.

[5] HELLER, Philip, ROBERTS, Simon, Complete Java 2 Certification, Fifth Edition, SYBEX, 2005.

[6] JOSUTTIS, Nicolai M., Object-Oriented Programming in C++, John Wiley, 2003.

[7] LISKOV, Barbara, GUTTAG, John, Program Development in Java, Addison Wesley, 2001.

[8] MUGHAL, Khalid A., RASMUSSEN, Rolf W., A Programmer's Guide to JAVA Certification, Second Edition, Addison-Wesley, 2004.

[9] NYÉKINÉ, Gaizler Judit és társai, Java 2 útikalauz programozóknak: 1.3, ELTE TTK Hallgatói Alapítvány, 2001.

[10] STROUSTRUP, Bjarne, A C++ programozási nyelv, Kiskapu, 2001.

[11] ***, Fundamentals of Java Programming v1.1, Cisco Systems, 2002.