Értékünk az ember
Humánerőforrás-fejlesztési Operatív Program

Varjasi Norbert

PROGRAMOZÁS III.

Objektumorientált programozás Java nyelven

SZÉCHENYI ISTVÁN EGYETEM
GYŐR

Magyarország célba ér
NEMZETI FEJLESZTÉSI TERV

Készült a HEFOP 3.3.1-P.-2004-09-0102/1.0 pályázat támogatásával.

Szerző: Varjasi Norbert
egyetemi tanársegéd

Lektor: Pusztai Pál
egyetemi adjunktus

Marton László emlékére

A dokumentum használata

Mozgás a dokumentumban

A dokumentumban való mozgáshoz a Windows és az Adobe Reader megszokott elemeit és módszereit használhatjuk.

Minden lap tetején és alján egy navigációs sor található, itt a megfelelő hivatkozásra kattintva ugorhatunk a használati útmutatóra, a tartalomjegyzékre, valamint a tárgymutatóra. A ◀ és a ▶ nyilakkal az előző és a következő oldalra léphetünk át, míg a Vissza mező az utoljára megnézett oldalra visz vissza bennünket.

Pozícionálás a könyvjelzőablak segítségével

A bal oldali könyvjelző ablakban tartalomjegyzékfa található, amelynek bejegyzéseire kattintva az adott fejezet/alfejezet első oldalára jutunk. Az aktuális pozíciónkat a tartalomjegyzékfában kiemelt bejegyzés mutatja.

A tartalomjegyzék és a tárgymutató használata

Ugrás megadott helyre a tartalomjegyzék segítségével

Kattintsunk a tartalomjegyzék megfelelő pontjára, ezzel az adott fejezet első oldalára jutunk.

A tárgymutató használata, keresés a szövegben

Keressük meg a tárgyszavak között a bejegyzést, majd kattintsunk a hozzá tartozó oldalszámok közül a megfelelőre. A további előfordulások megtekintéséhez használjuk a Vissza mezőt.

A dokumentumban való kereséshez használjuk megszokott módon a Szerkesztés menü Keresés parancsát. Az Adobe Reader az adott pozíciótól kezdve keres a szövegben.

Tartalomjegyzék

Előszó

A programozási nyelvek története nem ölel fel nagy időintervallumot. Az utóbbi ötven év alatt nagy fejlődési ívet jártak be mind a hardver-, mind a szoftvertervezési módszertanok. A számítógépek elterjedésével a piac egyre több alkalmazásfejlesztési igényt támasztott. A számítógépes programok termeléséhez szigorú szabályok, egységes módszertanok szükségesek. A szoftverek tervezéséhez és elkészítéséhez a modellező rendszerek ismerete szükséges. A szoftverfejlesztés során egy-egy elméleti síkon ábrázolt feladat magasszintű modelljéből kell leképezni a számítógépen tárolható adatstruktúrákat és algoritmusokat az adott hardver- és szoftverkörnyezetben.

A modellalkotásnak számos elmélete ismert. Az 1960-as években megjelenő, majd széles körben a '80-as években elterjedő objektumorientált modellezési és programozási szemlélet új megközelítési módot adott az adatszerkezetek és az algoritmusok tervezésébe. Az új megközelítés szerint az élet számítógépektől független, természetes jelenségeit kell megfigyelni, és ezeket a megfigyeléseket programozási környezettől függetlenül kell modellezni. Az így előállított általános modellt azután olyan programozási nyelven kell megvalósítani, amelyek támogatják ezt az objektumorientált szemléletet. Ha nem ilyen programnyelveket használunk, akkor a programozás nehézkessé válhat, az objektumorientált rendszer elemei sérülhetnek, vagy legrosszabb esetben a felvázolt modellt torzítjuk el, így pedig nem a modell szerinti feladat lesz a végeredmény, hanem valami más.

Az objektumorientált szemléletmód szakítva a strukturált programozás elvével – ahol egy adott feladat megoldása függ az implementáló nyelvtől, mert a feladatokat mindig az adott programnyelv ábrázolható adatstruktúrájával és utasításkészletével készíthetjük el – egy gépektől és programozási nyelvektől független modellszemléletet, modellalkotást követel meg a fejlesztőtől. Az előálló objektumorientált modellek azután, egy-egy adott objektumorientált nyelven megvalósíthatóak.

A Java a '90-es évek és napjaink dinamikusan fejlődő programnyelve. Az Internet korában – amikor lehetségessé vált az egymással összeköttetésben álló gépek használata – a Java nyelv képes az összekapcsolt, nyílt rendszereket használó szoftverek fejlesztésére. Hétköznapi szemlélő által is megfigyelhető az a folyamat, amelyben az eddig csak különálló gépeken futó egyedi programokat, a hálózatos, kliens-szerver, vagy a napjainkra

elterjedt közvetlen adatcserét lehetővé tevő (peer-to-peer) szoftverek váltják fel.

A Java nyelv alkalmas az objektumorientált alapelveknek teljes mértékben megfelelő, architektúra és operációs rendszer független, hordozható programkód fejlesztésére. A megalkotott szoftverek bármely más gépen lefuttathatóak lesznek, amelyek támogatják és ismerik a Java virtuális gépet – a Java értelmezőjét.

A nyelv nyílt forrású, és fejlett eszközkészlettel rendelkezik, amelyek segítik a hatékony és gyors programfejlesztést, melyhez mindig adottak a legfrissebb, részletesen kidolgozott leírások, igaz csak angol nyelven.

A jegyzet első fejezetét a Java programozási nyelv alapelemeinek vizsgálatával, alapvető eszközkészletének bemutatásával kezdjük. Ezután megismerkedhetünk az objektumorientált szemléletmóddal, annak fogalomkészletével és legfontosabb alapelveivel – osztály, objektum, egységbezárás, adatrejtés, öröklődés és többalakúság. Ez a fejezet még nem tartalmaz konkrét nyelvi elemeket, megoldásokat, hanem az absztrakt és magasszintű modellalkotás szerkezetét és elemeit tárgyalja. Továbbá nem tartalmazza az objektumorientált elemzési és modellezési (OOA, OOD) módszertanokat, illetve az UML teljes specifikációját, csupán az objektumorientált modellek programozásához szükséges osztálymodelleket tárgyaljuk.
Az elméleti ismeretek után rátérünk azok Java környezetbeli tárgyalására. Az objektumorientált elméleti alapelemeknél felrajzolt ívet követve megismerjük, hogy a Java nyelv a magas szintű modellek leképezéséhez milyen eszköztárat ad a programozó kezébe.

Az ezt követő fejezetek a Java keretrendszer egy-egy különálló fejezetének leírását tartalmazzák. Megismerkedünk többek között a magas szintű adatábrázolás eszközeivel, a fájl- és a hálózatkezelést támogató csomagokkal, és minden olyan nyelvi elemmel, amely a leggyakrabban előforduló algoritmizálási és programtervezési feladatokban felhasználható.

A jegyzetben a hivatkozásoknál a következő jelölésrendszert alkalmaztam: A nyomtatott irodalmakat a szerző nevével és a kiadás évszámával közlöm pl: [Angster01], az internetes hivakozásokat sorszámozással jelölöm pl: [2], illetve abban az esetben, ha egy konkrét Java kiadásra jellemző fogalmat tárgyalunk, azt (Java1.4) alakban jelzem.

A jegyzet célja, hogy az objektumorientált programfejlesztési technikákkal ismerkedő informatikus hallgatók átfogó képet kapjanak a Java programnyelvről és a mögötte álló szemléletről. A témakört nem tárgyaljuk a teljesség igényével, hiszen az elmúlt évtizedekben számos magyar és

angol nyelvű könyv, [Angster01, Flanagan05, Kondorosi04, Lemay02, Nyéky01, Schildt03, Vég99] és online segédanyag [1], [8], [9] került az olvasók elé. Ezt az elekronikus jegyzetet a felsorolt művek alapján rendszereztem. A tárgyalás során igyekeztem a legfrissebb kiadások (Java5, Java6) újdonságaira is kitérni [12], [13].

1. A Java nyelv alapjai

Az alkalmazásfejlesztéshez valamilyen konkrét nyelvi interpretációt kell készíteni. Az objektumorientált elveket a '70-es évektől kezdődően kezdték beépíteni az egyes programnyelvekbe, és ekkor készült az első tisztán objektumorientált nyelv is. A hatékony, objektumorientált nyelvek sorában a '90-es években jelent meg a Java, amely – mint az internet-korszak egyik úttörő nyelve – az utóbbi években látványos fejlődésen ment keresztül, és napjainkban is nagy népszerűségnek örvend. Először tekintsük át a Java nyelv kialakulásának rövid történetét.

1.1. Történeti áttekintés

Patrick Naughton és James Gosling (SUN Microsystems – Green Team) 1991-től egy mini kommunikációs nyelvet kezdtek kifejleszteni [10]. Céljuk egy hatékony, architektúra-független, kis erőforrás-igényű rendszer kifejlesztése volt, melyet egyszerű elektronikai eszközökben szerettek volna felhasználni (digitális kábel-TV végponti eszközök).

Alapötletük lényege (Niclaus Wirth pascal nyelvi kódja alapján) a következő modellen alapult: egy elképzelt, hipotetikus gépre közbenső kódot generálnak (Java Virtual Machine – JVM), amely azután minden olyan architektúrán és operációs rendszeren működőképes, amely rendelkezik ezzel az interpreterrel.

> Az interpreter (értelmező) olyan program, amely a forrásprogramnak egyszerre egyetlen utasítását értelmezi. Az utasítást natív kóddá alakítja, és azonnal végrehajtja. A lefordított kódot nem jegyzi fel, hanem végrehajtás után közvetlenül eldobja, és a következő utasítás feldolgozásába kezd.

A virtuális gép és az interpreter jelleg azért volt döntő és hatékony lépés, mert az elektronikai piac az 1980-as, '90-es években ugrásszerű fejlődésen ment keresztül, a hardverek terén nagyfokú architekturális eltérés mutatkozott. Ezt a különbséget hidalták át azzal, hogy a nyelvet általánossá tették, és minden használt architektúrára elkészítették a virtuális gépet. Így a programok nemcsak forrás szinten, hanem a lefordított tárgykódok szintjén váltak szabadon hordozhatóvá. Ez az újítás – az olyan hátrányai ellené-

re, mint például a lassúság – fontos szerepet játszott később, a nyelv gyors elterjedésében.

A virtuális gép a Java alkalmazások futtatására képes, általában szoftveresen megvalósított működési környezet. A virtuális gép feladata a Java osztályokat tartalmazó *„.class”* állományok betöltése, kezelése, valamint az azokban található – Java kódban adott – műveletek gépi kódú utasításokká történő átalakítása.

Gosling ezt az alapelvet az „írd meg egyszer, futtasd bárhol” kifejezéssel magyarázta. Mivel a kutatásban részt vevő fejlesztők C++ programozók voltak, így a kifejlesztett új nyelvhez a C++ nyelv alapjait használták fel kiindulásként, ezt egyszerűsítették, alakították. Céljuk egy olyan nyelv megalkotása volt, ami tisztán objektumorientált, és amelyben a gyakran előforduló programozási hibák már fordítási időben kiderülnek. Kezdetben Oak (tölgy) névvel illették az elkészült nyelvet, de ez a szó már foglalt volt. A Java névválasztás a fejlesztők kávézási szokásából ered. (Jáva szigetéről származó kávét fogyasztottak az új név kiválasztásakor.)

Az új nyelv elkészítése még nem jelentett egyértelműen széleskörű elterjedést akkor sem, ha az a nyelv, illetve a kidolgozott technológia hatékony. Az elkészült eszköz (Set Tob Box – interaktív kábeltelevíziós rendszer) messze megelőzte a korát, mert a ’90-es évek elején az ipar és a piac (konkrétan a kábeltelevíziós rendszerek) még nem ismerték fel ebben az új technológiában rejlő lehetőségeket.

A nyelv ismertsége és gyors elterjedése annak volt köszönhető, hogy a 90-es években rohamosan fejlődő web-alkalmazásokra is felkészítették. Ekkoriban vált ugyanis népszerűvé a hipertext, és a web-böngészőkbe belecsempészett, beépülő JVM-ek (HotJava Browser, Netscape stb.) életre keltették az addig statikus lapokat. Vagyis a Java-képes böngésző motorok képesek voltak a HTML-oldalakba épített, előre lefordított bájtkódok futtatására, animációk, mozgó ábrák, zenék és videók formájában (az appletek 1995-96-tól) jelentek meg.

A Java programozási nyelvet 1995 májusában adta ki a Sun Microsystems, ekkor már a főbb gyártó és feljesztőcégek bejelentették a Java technológiával való együttműködést (Netscape, Oracle, Toshiba, Lotus, Borland, Macromedia, Silicon Graphics, Symantec, IBM, Adobe, Microsoft stb.).

1998-tól működik a Java Community Process (JCP), mint a Java fejlesztéseket összefogó, több nagyvállalatot és számos kutatóintézetet, valamint egyéni fejlesztőt összefogó független szervezet.

1999-ben a Java2 rendszer kiadásakor már három független részre bontották a Java technológiát. Így az ME (Micro Edition) – a beágyazott eszközök, a SE (Standard Edition) – az általános desktop-programok, és az EE (Enterprise Edition) – a kliens-szerver, illetve nagyteljesítményű vállalati alkalmazások fejlesztéséhez nyújt hatékony környezetet. Azonban hamarosan negyedik fő irányként vizsgálhatjuk a „Java Card" - rendszereket, vagyis az intelligens chipkártyák programozási platformját.

A nyelv nagyon gazdag eszközkészlettel támogatja a programozói munkát, a beágyazott rendszerek programozásától, mint a mobiltelefonok és kéziszámítógépek (PDA-k) az elosztott és nagy erőforrásigényű üzeti webszolgáltatásokig, illetve a kliens-szerver alkalmazásokig. A Java nyelv fejlesztése nem állt meg. A JCP keretében a mai nap is közel ezer, egymással párhuzamosan folyó projektet fejlesztenek. Napjainkban a Java a tízéves jubileumát ünnepli, a honlapján közzétett adatok szerint az aktív Java programozók száma 4,5 millió körüli, és jelenleg forgalomban van körülbelül 2,5 milliárd Java-képes eszköz. A felsorolt számadatok azt vetítik előre, hogy a Java technológia nagy iramú fejlődése nem áll meg, és a jövőben nagy szükség lesz a nyelvet ismerő szakemberekre.

Az alábbiakban a Java programnyelv alapvető felépítését és építőköveit igyekszem bemutatni.

1.2. A Java programok és a virtuális gép szerkezete

Mint minden programozási nyelvhez, úgy a Java nyelvhez is szükséges valamilyen fejlesztői környezet, amely a forrásprogramok fordítását, illetve futtatását lehetővé teszi. A Java fejlesztői csomag (Java Software Development Kit – SDK) telepítése, és a szükséges környezet beállítása után (a részletes telepítési útmutatót lásd a Mellékletekben) az elkészült forráskódot lefordíthatjuk.

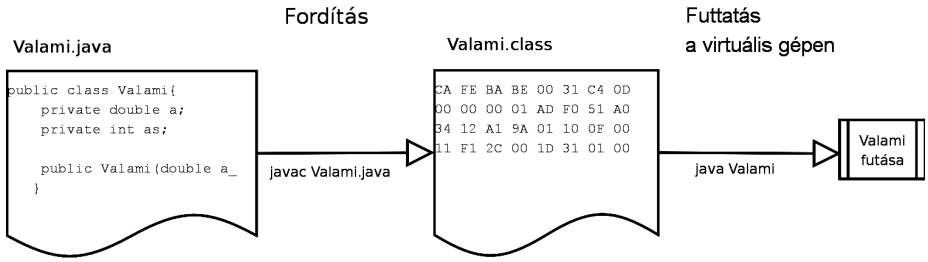
A nyelv megalkotói a virtuális gépet egy képzeletbeli számítógépként írják le. Ezt a számítógépet szimuláljuk szoftveresen valamilyen konkrét rendszeren. A virtuális gép már előzetesen lefordított tárgykódú, (bájtkódú) bináris állományokkal (továbbiakban class fájl) dolgozik. A virtuális gép indításakor a paraméterként megadott publikus osztály műveleteit kezdi el végrehajtani. A végrehajtás során a virtuális gép más osztályokat is betölthet, objektumokat példányosíthat, üzeneteket küldhet stb.

Ugyanaz a Java forrásprogram bármilyen fordítóval lefordítva ugyanazt a class fájlt állítja elő bájtról-bájtra. A virtuális gépen futtatva pedig ugyanazt az eredményt kapjuk bármilyen környezetet is használjunk (architektúra, operációs rendszer). A közös bájtkód használata tehát hordozható kódot nyújt. Továbbá a kis méretek miatt – a tárgykód tömör, és optimalizált – a class fájlok hatékonyan továbbíthatóak hálózatos környezetben. A Java gyors elterjedésének ez a platformfüggetlenség és hatékonyság volt az egyik oka.

Az interpretereken kívül természetesen léteznek olyan eszközök is *(Java processzorok),* amelyek hardveresen támogatják a tárgykódú fájlokat, vagyis az eszköz gépi kódja maga a JVM (mobiltelefonok, kézi számítógépek, célhardverek). Továbbá léteznek olyan eszközök is, amelyek a Java kódot egy adott rendszer bináris kódjára fordítják.

Egy konkrét java forráskód lefordításához a java környezet `/bin` könyvtárában található `javac (javac.exe)` fordítóprogramot használjuk.

A fordítás után egy class fájlt kapunk. A class állományokat a számítógépünk közvetlenül nem tudja feldolgozni, szüksége van egy közbenső interpreterre, amely a bináris kódot az adott architektúra számára értelmezhető utasítássorozatra alakítja. Ez az eszköz a virtuális gép (JVM). A virtuális gép a Java környezet `/bin` könyvtárában található, és a `java (java.exe)` paranccsal indítható. A parancs paramétere egy már lefordított Java osztály neve lehet.



1. ábra: Java programok fordítása és futtatása a virtuális gépen

A virtuális gép a class fájl betöltésekor az érkező kódot szigorúan ellenőrzi, majd a szükséges memóriaterületet inicializálja, és az érkező utasítássorozatot végrehajtja.

A fejlesztők a nyelv, és a java környezet tervezésénél a következő tizenegy szempontot tartották szem előtt:

- **Egyszerű:** A nyelv a C++ (mint a '90-es évek legnépszerűbb és legelterjedtebb környezete) egyszerűsített változata. A Java nyelv sokkal kevesebb nyelvi eszközt sokkal nagyobb kötöttségekkel tartalmaz.
- **Objektumorientált:** A Java tisztán objektumorientált nyelv, egy Java alkalmazás nem tartalmaz globális, illetve osztályokhoz nem köthető kódot.
- **Elosztott:** A Java alkalmazások képesek hálózatos környezetben, az internet egységes erőforrás azonosítóival (URL – Uniform Resource Locator) egyértelműen azonosított objektumokat kezelni. A Java rendszer továbbá képes a távoli eljáráshíváson alapuló kliens-szerver jellegű alkalmazások kifejlesztésére.
- **Robusztus:** (hibatűrő, megbízható): A nyelv tervezésekor fontos szempont volt a programozói hibák már fordítási időben való kiszűrése, a futási idejű hibák minimalizálása.
- **Biztonságos:** A nyelv már kezdeti szakaszában támogatta a hozzáférések és jogosultságok kezelését. A Java környezet a hálózatos alkalmazások és elosztott rendszerek támogatása mellett hatékony biztonsági megoldásokkal rendelkezik. Az osztályok virtuális gépbe töltésekor (Class loader) hibaellenőrzés történik, és a virtuális gép az adott operációs rendszer felett az ún. homokverem (sandbox) modell szerint működik, amely azt jelenti, hogy a futó programok az operációs rendszer egy kijelölt és lahatárolt területén futnak, így képtelenek a gazdagép erőforrásaihoz közvetlenül hozzárérni.
- **Hordozható:** A Java nyelv és a programozási környezet nem tartalmaz implementációfüggő elemeket. Egy adott forrásból minden Java fordító ugyanazt a tárgykódot állítja elő.
- **Architektúra-független** (gépfüggetlen): A fordítás során az azonos típusok minden környezetben azonos méretű memóriaterületet foglalnak, a fordítás után előálló class fájlok bármelyik szabványos virtuális gépen futtathatóak. A futtató környezet a tárgykódú állományokat azonos módon hajtja végre. Az egyes architektúrákhoz és operációs rendszerekhez elkészített virtuális gépek alakítják át a class állományokat natív kódokra.
- **Interpretált:** Az egyes hardvereken futó natív kódot a virtuális gép futási időben hozza létre, a lefordított tárgykódú állományokból.
- **Nagy teljesítményű:** Ez az utolsó jellemző még nem került teljes mértékben megvalósításra. A Java környezet kritikusai az interpretált

jellegből adódó lassú futást és az optimalizálás nehézkességét nagy hibának róják fel, azonban a processzorok fejlődésével, és a Just In Time (JIT) fordítók kutatásába fektetett erőforrásokkal a Java fejlesztői a jövőben ezt a szempontot is meg szeretnék valósítani. A JIT fordítók a lefordított tárgykódokat a futás alatt megőrzik, optimalizálják.[5]

- **Többszálú:** A Java környezet támogatja a párhuzamos programozást, és alkalmas párhuzamos algoritmusok megvalósítására. Ezt az adott hardveren úgy valósítja meg, hogy egy időben egymástól függetlenül több programrészt futtat. Egyprocesszoros rendszereknél ezt természetesen egymás utáni végrehajtással, a szálak ütemezésével oldja meg.
- **Dinamikus:** Az osztálykönyvtárak szabadon továbbfejleszthetők, és már a tervezésnél figyelembe vették, hogy egy esetleges későbbi bővítés a már megalkotott programok működését ne akadályozza.

A Java programok forrásszövege – a C, C++ nyelvhez hasonlóan – egy tiszta karaktersorozat. Az egyes nyelvi elemeket fehér karakterekkel (szóköz, tabulátor, soremelés) tagoljuk, illetve választjuk el egymástól. A program írásakor be kell tartanunk a nyelv szabályait, és érdemes figyelembe venni a Java nyelv kódolási konvencióit (lásd: Kódolási szabványok).

A szintaktikai szabályok szigorúan meghatározzák, hogy milyen elemeket, azonosítókat milyen sorrendben alkalmazhatunk a forráskódban. Szintaktikai hiba például, ha egy azonosítót hibásan írunk le. A szintaktikailag hibás program nem fordítható le érvényes class fájl formátumba.

A szemantikai szabályok a program működési elvét, futását határozzák meg. Ezek a hibák fordításkor általában nem deríthetőek ki, csak a futás közben jelennek meg. Például ha a programban egy `a/b` alakú kifejezésben 0-val való osztást kísérelünk meg végrehajtani.

1.3. Kérdések

- Milyen formátumú egy lefordított java forrás?
- Mi a Java technológia három fő iránya?
- Melyek az interpretált programok főbb jellemzői?
- Mi a szerepe a virtuális gépnek?
- Mi a különbség a szintaktikai és a szemantikai programhiba között?

2. Az objektumszemlélet elméleti megközelítése

Az előző fejezetben megismerkedtünk a Java környezettel, áttekintettük a történeti vonatkozásokat és a működés főbb alapelveit. Maga a Java nyelv egy tisztán objektumorientált programozási nyelv. Az objektumorientált programozás elsajátításához szükségünk van a nyelv jelkészletének és működési elvének megismerésére.

Az objektumorientált alapelvek és modellek azonban érvényesek konkrét programozási nyelv nélkül is, hiszen alkalmazásuk magasabb szintű, általános megoldásokat nyújt. Ebben a fejezetben ennek a gondolkodásmódnak – másképpen programozási paradigmának – alapelveivel ismerkedünk meg.

A programozás „hőskorában” a programok csupán néhány száz soros, alacsonyszintű utasítássorozatból álltak. A magasszintű, strukturált programnyelvek megjelenése – az 1960-as évektől kezdve – könnyebb és hatékonyabb programozási technikákkal látta el a szakembereket. A konstansok, változók vezérlési szerkezetek és a szubrutinok megjelenése nagyobb és összetettebb problémák megoldását tette lehetővé: a feladatok magas szintű modellezését és algoritmizálását. A kalsszikus strukturált programozás szabályai szerint a megoldandó feladatokat át kell alakítani számítástechnikai modellé, azaz a megoldáshoz felhasznált gép számára értelmezhető adatokra, illetve az ezekkel az adatokkal, a feladatot elvégző algoritmusra. A megvalósításához számos programnyelv készült, és ahogy nőtt a számítógépekkel elvégzett feladatmegoldások száma, az egyes programozási nyelvek is úgy váltak összetettebbeké, általánosabbá.

Megjegyzés: Struktúráltnak tekinthető az a program, amely vezérlési szerkezetként megengedi a szubrutinhívásokat, a szekvencia, a szelekció, az iteráció utasításokat, de más vezérlésátadó utasításokat (pl. ugró utasítás) nem. [Marton02]

A 80-as évekre az elkészült programok mérete és összetettsége akkorára nőtt, hogy mindinkább szembeötlöttek a strukturált programtervezés korlátai. Ezt az időszakot a szoftverkrízis időszakának is nevezték, gyakori volt a határidők csúszása, az eredménytelen fejlesztés és a költségvetések

túllépése, hiszen a túlságosan bonyolulttá váló rendszerek modellezésére, majd tesztelésére – illetve a hibák javítására nem fordítottak kellő erőforrást. Ezek a rendszerek tipikusan „egy emberes" munkák voltak, és ebből adódóan nagyobb, átfogóbb feladatokhoz már nem voltak hatékonyak, hiszen a programok száma és mérete nagyságrendekkel gyorsabban nőtt, mint a programozói kapacitás. A felmerülő problémákat igyekeztek valamilyen szabványosítás vagy új módszertan segítségével megoldani, illetve újrahasznosítható kódokat tervezni.

A programozási nyelvek történetében az 1960-as évektől a '80-as évek elejéig (Simula, Smalltalk nyelvek) figyelhető meg az a folyamat, amikor a strukturált programozási módszerek mellett megjelent egy új elgondolás, egy új szemlélet, amelyet objektumorientált programozás névvel illetnek mind a mai napig.

A megoldás nem technikai jellegű – a hangsúly nem ott van, hogy az egyes programnyelvekbe új elemeket építenek – hanem módszertani kísérlet volt annak a problémának a feloldására, hogy a modellalkotásban a hagyományos síkon nem mindig lehet az adott problémákat hatékonyan megközelíteni. Magát a valós világot kell minél hűebben leírni, modellezni. Az objektumorientált modell a valós elemeket, tárgyakat, objektumokkal ábrázolja, amelyeket egyes állapotaik, jellemzőik, adataik és az ezekhez az objektumokhoz egyértelműen hozzárendelhető műveletek írnak le.

Az objektumorientált modellalkotás és programozás megjelenése új fogalmakat is bevezetett. Az osztály, az egységbezárás, az öröklés, a többalakúság, a futás alatti kötés, az üzenetküldés, az együttműködés, az interfészek, a csomagok és komponensek fogalma mind-mind az új szemlélet jegyében születtek. Az új fogalom- és szóhasználat egy másfajta gondolkodásmódot jelez. Az objektumorientált modell egy új szemléletmódot vezetett be a feladat- és rendszermodellezésbe is. A modellalkotók egy kevésbé absztrakt és a valós problémákhoz jobban illeszkedő és könnyebben áttekinthető módszert alkottak. A '80-as '90-es évek alatt számos meglévő programozási nyelvbe beszűrődött ez a szemléletváltás (Object Pascal, Delphi, C++, Ada, C#, PHP), illetve megszülettek a tisztán objektumorientált nyelvek is (Smalltalk, Eiffel, Java).

A programozási nyelveknek ez a gyors átalakulása nem abból az okból ered, hogy a hagyományos módszertanok hibás alapokra épültek, vagy mára már elavulttá váltak. A választ inkább a számítástechnikai eszközök gyors terjedésében, a megoldandó feladatok összetettségében és a hatékony és újrahasználható programokra irányuló piaci igény oldalán célszerű

keresni. Az objektumorientált programokban – hiszen építőkövei a strukturált programozásból származnak – éppúgy megtaláljuk a strukturált alapelemeket, algoritmusokat, mint a hagyományos programokban. Mégis egy másik világban találjuk magunkat.

Az objektumorientált programozás a mai szoftverfejlesztés legelterjedtebb módja. Ennek a legfőbb oka az, hogy az osztályhierarchiák segítségével hatékonyan támogatja az absztrakt objektumok kialakításával. Az újrafelhasználható kódok, csomagok és komponensek támogatásával nagymértékben tudja csökkenteni a fejlesztéshez szükséges időt.

Az alábbiakban egy tömör áttekintést találunk az objektumorientált szemlélet legfontosabb alapelveiről. A fogalmak tisztázása során még nem lesz szó egyes programnyelvi elemekről, hiszen az alábbi elveket minden nyelv majd a maga módján valósítja meg.

2.1. Az osztály és az objektum

A valós világ leírásában – az emberi gondolkodást követve – osztályozhatjuk, csoportosíthatjuk a látottakat, a modellezés során absztrakt fogalmakat és általánosított tételeket állíthatunk fel. Az absztrakciós modellezéskor az összetett és bonyolult dolgokat egyszerűsítjük, és csak a feladat számára információval bíró részleteit használjuk fel.

Ha például egy útkereszteződés forgalmát figyeljük, akkor nem fontos, hogy egy jármű hány alkatrészből áll, mennyire összetett és hogy hány különböző jellemzője van, pontszerűnek tekinthetjük. Annyi azonban fontos lehet, hogy a jármű személy-, vagy tehergépkocsi.

Más esetben a világ összetett dolgait részekre kell bontanunk. A specializáló modellezéskor ezeket a részeket valamilyen szabályok mentén összekapcsoljuk, hiszen a probléma megoldásához elengedhetetlennek bizonyul.

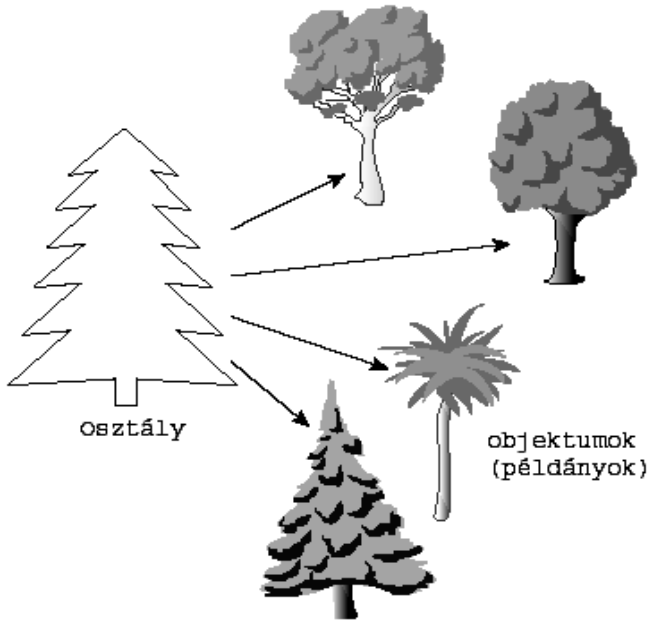
Fenti példánkat folytatva egy gépkocsi diagnosztikai műhely számára már nem elégséges a fenti modellalkotás, itt már egy gépkocsit elemeire kell szednünk, és meg kell figyelnünk ezen alkatrészek tulajdonságait, működését.

Az objektumorientált modellalkotásban objektumoknak nevezzük azokat a dolgokat, amelyekhez a valós világban egy-egy önálló dolgot (jelentést, tartalmat, vagy fogalmat) rendelhetünk. A szakirodalomban egyaránt találkozunk az objektum, a példány, vagy az egyed kifejezéssel is. Az objektumokat valamilyen mérhető tulajdonságával jellemzünk, és megfigyeljük, hogy egy külső eseményre (üzenetre) hogyan reagálnak. A reakció az

objektumok tulajdonságait megváltoztathatja, vagy valamely más objektum felé újabb üzenet küldésére indíthatja.

Az objektumok információkat tárolnak, és kérésre feladatokat hajtanak végre. Programozási szempontból egy objektumot adatok, és az ezen adatokon végrehajtható műveletek összességével jellemzünk. Minden objektum egyedileg azonosítható.

A modellezés folyamata az adott feladat absztrakciós és specializáló feldolgozásából áll, végeredménye mindig valamilyen osztályozás. Ez az osztályozás követi a hagyományos emberi gondolkodási struktúrákat. A hasonló vagy ugyanolyan tulajdonságokkal (adattagok, attribútumok) rendelkező és hasonlóan, vagy ugyanúgy viselkedő (metódusok) objektumokat egy csoportba, egy osztályba soroljuk. A példányok rendelkeznek a saját osztályaik sajátosságaival, valamint az érkező üzenetekre hasonló módon reagálnak. Tömören fogalmazva elmondhatjuk, hogy minden egyes objektum egy-egy osztály egyetlen, és egyedi példánya.



2. ábra: Egy osztály és objektumai (modell)

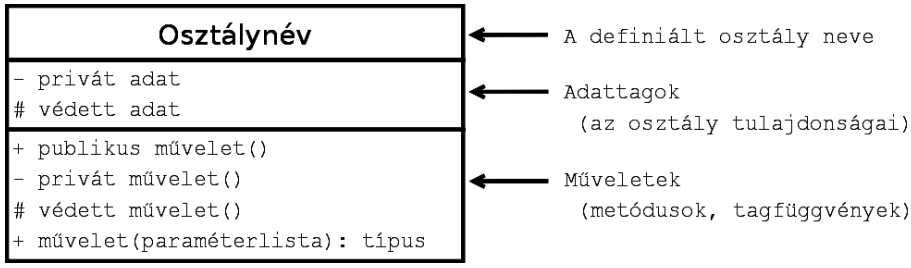
Az ábrán fákat vizsgálunk. Minden fa egyedi, hiszen megállapíthatunk közöttük eltéréseket színre, formára stb. de mégis a növények egy jól körülhatárolható osztályát alkotják. A fák osztálya már egy absztrakt fogalom, hiszen magas szinten foglakozik az osztály jellegzetességeivel, tulajdonságaival. (Programozási szempontból az osztályokat típusdefiníció segítségével írjuk le.)

Az osztály az ugyanolyan jellemzőkkel, tulajdonságokkal (adattagok) és ugyanolyan működéssel (metódusok, tagfüggvények) rendelkező objektumok magasabb szintű egysége. Minden objektum valamilyen osztály példánya, rendelkezik az osztály tulajdonságaival, műveleteivel.

Tekintsünk az alábbi példát: Egy iskola tanulóit vizsgáljuk, itt minden gyerek egy-egy önálló, egyedi példány, önálló objektum. Ha ezeket a tanulókat osztályozni szeretnénk, akkor többféle megközelítést is választhatunk. Első ránézésre – általánosító modellezés – megállapíthatjuk főbb tulajdonságaik és viselkedésük alapján, hogy mindannyian egyetlen osztályba sorolhatóak: „gyerekek”, hiszen mindegyik azonos iskolába jár, és hasonló tevékenységekben vesznek részt és egyértelműen elkülöníthetők más osztályok egyedeitől (objektum), például a tanáraiktól. Mégis, ha jobban szemügyre vesszük ezt a csoportot – specializáló modellezés – akkor nagyon könnyen újabb osztályokba sorolhatjuk őket mondjuk életkor szerint. Itt már megfigyelhetőek lesznek az életkori sajátosságok szerinti különbségek, az azonos utasításokra való másfajta reagálások.

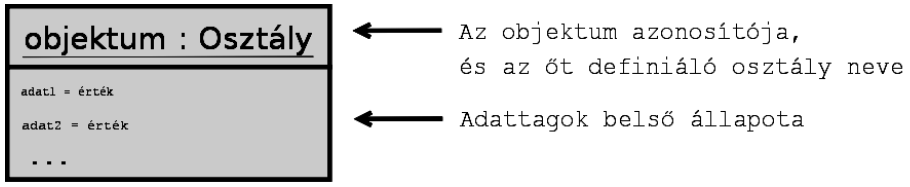
Az osztályozást azonban mindig az adott feladatnak megfelelően kell elvégezni az általánosító, vagy a specializáló modellezés segítségével. A megoldások során célszerű a feladat szempontjából leghatékonyabb osztályozást megadni.

Megjegyzés: Az osztályozás nem minden esetben triviális művelet. Az OOA (Object Oriented Analysis), és az OOD (Object Oriented Design) modellező ismeretek segítségével a tervezés átláthatóbb, ezért minden szempontból érdemes mélyebben megismerni ezeket a módszertanokat.



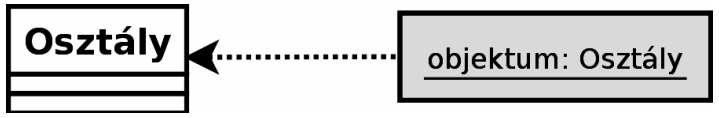
3. ábra: Az osztály jelölése

Maga az osztály tehát egy absztrakt adat- és művelethalmaz. Másként fogalmazva az osztály egy olyan típusdeklaráció, amely mintául szolgál az ezen típushoz tartozó objektumok (példányok) létrehozásához.



4. ábra: Az objektum (példány) jelölése

A programfejlesztés szempontjából a példányosítás folyamata során létrehozzuk az egyes osztályok konkrét példányait. Itt már megtörténik a tárhelyfoglalás, és a példányosított objektum már egyértelműen ismeri az osztályához tartozó műveleteket. Másként fogalmazva a példányosítás után minden egyes objektum tudja, hogy melyik osztályhoz tartozik.



5. ábra: A példányosítás

Az objektumok élettartamuk során mindig valamilyen belső állapotban vannak. A feladatvégzést minden esetben egy kezdőállapotból kiindulva értelmezzük. Az objektum ezt a kezdőállapotát, illetve a későbbiekben a belső állapotát egy-egy üzenet hatására módosíthatja.

Az üzenet az objektumok közötti kommunikáció, adatcsere, illetve egy feladat elvégzésére való felkérés.

Az egymással kapcsolatban álló objektumok kommunikálhatnak, üzenetet küldhetnek egymásnak. Ezeket a műveleteket az objektumorientált nyelvek többnyire az objektum metódusainak (más terminológia szerint tagfüggvényeinek) hívásával valósítják meg. Az üzenethívást mindig egy olyan objektum kezdeményezi, amelyik kapcsolatban áll a hívandó objektummal. Ezt a szakirodalom ismertségi, vagy tartalmazási kapcsolatnak definiálja.
Az osztály és az objektum fogalmának tárgyalása után már megfogalmazhatjuk, hogy mit is takar egy objektumorientált program.

Egy objektumorientált program egymással kommunikáló objektumok összessége, melyben minden egyes objektumnak jól meghatározott feladatköre van. Az objektumok közötti kommunikációt a szabványos üzenetküldés valósítja meg. Az objektumorientált program nem sértheti meg az objektumorientált alapelveket.

2.1.1. Jelölésrendszer

Az osztályok grafikus megjelenítéséhez az egységes modellező nyelv - az UML (Unified Modeling Language) modellezési sémáit használjuk. (Részletesen lásd: [3]).

Az osztályok és a példányok leírását osztálydiagramokkal, az objektumok és magasabb szinten, az osztályok közti üzenetcserét együttműködési diagramokkal ábrázoljuk. Itt külön nem térünk ki az UML alapú tervezés minden egyes lehetőségére (esethasználati-, objektum-, szekvencia-, átmenet- és aktivitás diagramok).

2.2. Egységbezárás

Az egységbezárás (encapsulation) az egyik legfontosabb objektumorientált alapfogalom. Az adatstruktúrákat és az adatokat kezelő műveleteket egy egységként kell kezelni. Az objektumok belső állapota – adattagjainak futás alatti pillanatnyi értéke – és viselkedése minden egyes objektum belügye. Minden objektumnak védenie kell belső adatait. Az egyes műveleteket úgy definiáljuk, hogy az kizárólag a belső adatokkal dolgozzon. Minden egyes adatot és minden egyes műveletet egyértelműen valamilyen osz-

tály részeként kell értelmezni. Ezzel a megközelítéssel az objektumorientált rendszerek a valós világ modellezésének egy hatékony és áttekinthető leírását adják.

Az egy osztályban definiált adattagok és a hozzájuk tartozó műveletek (metódusok) összetartoznak, zárt és szétválaszthatatlan egységet alkotnak.

Az egyes hibrid objektumorientált nyelvekben (Object Pascal, Delphi, C++, C#, PHP) az egységbezárást a hagyományos, összetett adatszerkezetek (struktúrák, rekordok) kibővítésével valósították meg. A tisztán objektumorientált nyelvekben a modellek leírását csak osztályokban adhatjuk meg, önálló, vagy globális adatokat és műveleteket nem definiálhatunk.

2.3. Adatrejtés

A második objektumorientált alapfogalom az adatrejtés (data hiding) szorosan kapcsolódik az egységbezáráshoz. Előzőleg utaltunk rá, hogy az adatokat minden egyes objektum elrejti a külvilág elől. Ez azért szükséges, hogy egy adatot kívülről ne lehessen módosítani, manipulálni, csak valamilyen ellenőrzött üzeneten keresztül. Segítségével egy objektum csak olyan üzenetre reagál amit ismer, és az érkező üzenetet ellenőrzés után végrehajtja. Az adatok ellenőrzése azonban itt is a programozó feladata! Az adatrejtés elvének betartásával – mellékhatásként – a program többi részét is védhetjük az adott objektumban fellépő hibáktól, így a fejlesztések során a hiba forrása könnyen azonosítható lesz.

Minden objektum védi a belső adatait, így adatstruktúrája kívülről nem érhető el. Ezekhez az adatokhoz csak a saját műveletei férhetnek hozzá.

Az egyes nyelvekben az adatrejtést a láthatósági szabályok definiálják. A hozzáférési kategóriák mind az adatok, mind a metódusok számára használhatók. A legfőbb kategóriák a nyilvános (public), jelölése: „+"; a védett (protected), jelölése: „#" és a privát (private), jelölése: „–". Az egyes programnyelvek megadhatnak ennél több, illetve kevesebb kategóriát, illetve definiálhatnak másfajta hozzáférési módokat is. Az adatrejtés elvének megsértése olyan szemantikai hibákhoz vezethet, amelyek nehezen felderíthetők, illetve csak huzamosabb használat után „véletlenül" okoznak

futási hibákat. (Több nyelv már a fordítás során hibaként jelzi az ilyen jellegű hibákat.)

2.4. Öröklődés

Az objektumorientált szemlélet nagyon fontos eleme az öröklődés (inheritance). A modellezés során a valós világ elemeit – az objektumokat – valamely tulajdonságuk alapján osztályokba szervezhetjük. A fentiekben már utaltunk az általánosító, vagy a specializáló modellezési módszerekre, azaz egy adott osztály által reprezentált tartalmat szűkíthetjük, illetve részletezhetjük.

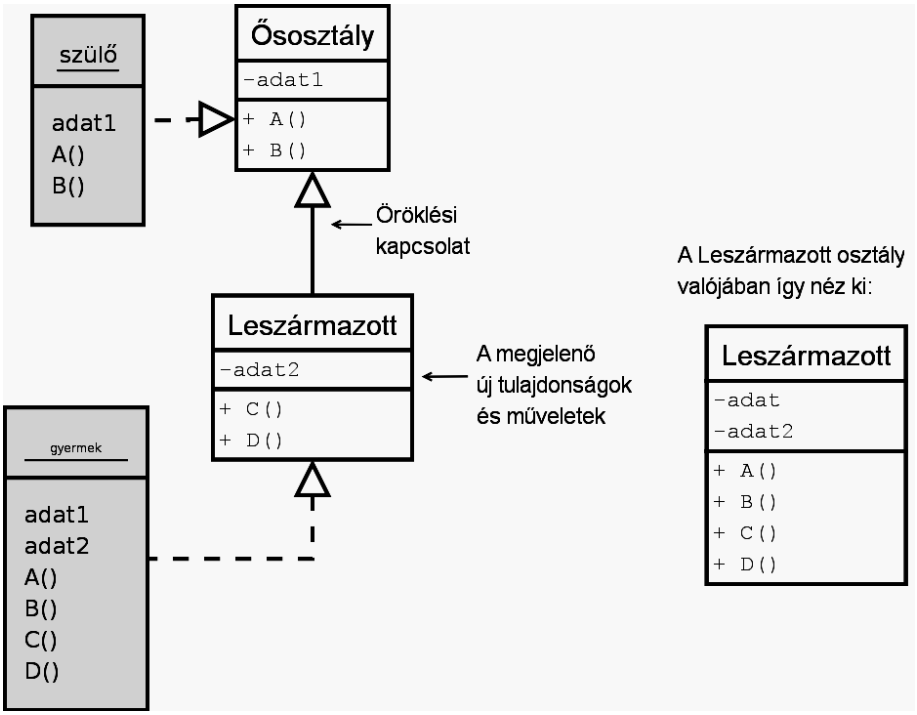
Az öröklődés segítségével egy osztályhoz alosztályokat rendelhetünk. Az újonnan megalkotott osztály minden esetben egy másik osztály kiterjesztése. A szokásos terminológia szerint az osztályokat ős-leszármazott, szülő-gyerek fogalompárokkal jellemezzük.

Az öröklődés általános fogalmához hasonlóan, a leszármazott osztály is örökli ősének minden tulajdonságát és műveleteit. A leszármazott osztály objektumai tehát „származásuk miatt" reagálnak az ősben definiált üzenetekre. Az öröklés során a leszármazott osztályok kiterjeszthetőek, új tulajdonságokkal rendelkezhetnek és új műveleteket is végezhetnek. Így a leszármazott osztály példányai a kiterjesztés során a definiált saját tulajdonságokon túl az őstől örökölt adattagokat és metódusokat is elérhetik.

Az osztályok leszármaztatásával a leszármazott osztály örökli ősének minden egyes tulajdonságát, műveletét. A leszármazott osztály új tulajdonságokkal és új műveletekkel bővíthető.

Legyen N osztály a természetes számokat leíró osztály. Ez az osztály tartalmaz egy nemnegatív egész adattagot, és értelmezhetjük rajta az összeadás, a szorzás, illetve korlátozottan a kivonás műveletét. Ennek leszármaztatásával, kiterjesztésével megalkothatjuk az egész számok ábrázolására szolgáló Z osztályt. Itt már előjeles egész számokkal dolgozunk. A leszármazott osztály örökli az összeadás és a szorzás műveletét. A kivonás műveletét pedig felüldefiniálhatjuk, hiszen a művelet itt már – a matematikai megközelítés analógiája alapján – korlátozások nélkül használható. Ezután a Z osztályt tovább specializálhatjuk, létrehozva a további számhalmazokat leíró osztályokat.

Egy meglévő osztályt felhasználhatunk örökítési céllal újabb osztályok definiálására. Az ősben szereplő tulajdonságokat, vagy műveleteket a leszármazottakban újra fel tudjuk használni ismételt definíció nélkül is.



6. ábra: Öröklődés

Az örökítési módok az egyes programrendszereknél többféleképpen valósíthatóak meg. Minden objektumorientált programrendszer lehetővé teszi az egyszeres öröklést, ahol minden leszármazott osztálynak egy és csakis egy őse lehet. Egyes rendszerek lehetővé teszik a többszörös öröklést is, ekkor egy leszármazott két, vagy több ősosztály leszármazottja is lehet. A többszörös öröklődésnél is érvényesül az öröklési szabály, azaz minden leszármazott osztály örökli az ősosztályok adattagjait és metódusait. A leszármazott osztályok objektumai (példányai) elérhetik a saját, illetve az ősosztály műveleteit.

Programozási szempontból az egyes osztályok definiálásánál a „keresd az őst” módszert alkalmazhatjuk. Azaz minden olyan művelet, amely az osztályhierarchia egy adott szintjén több osztályban is megjelenik, azt már az ősosztályban célszerű definiálni, hiszen így minden leszármazott örökli

azt, így a műveletet elég egyszer megadni. Az öröklődés ezen tulajdonságainak kihasználása segít elfelejteni azt a hibás programozói gyakorlatot, amikor a hasonló műveletek elvégzéséhez redundáns kódot írnak.

2.4.1. Absztrakt osztályok

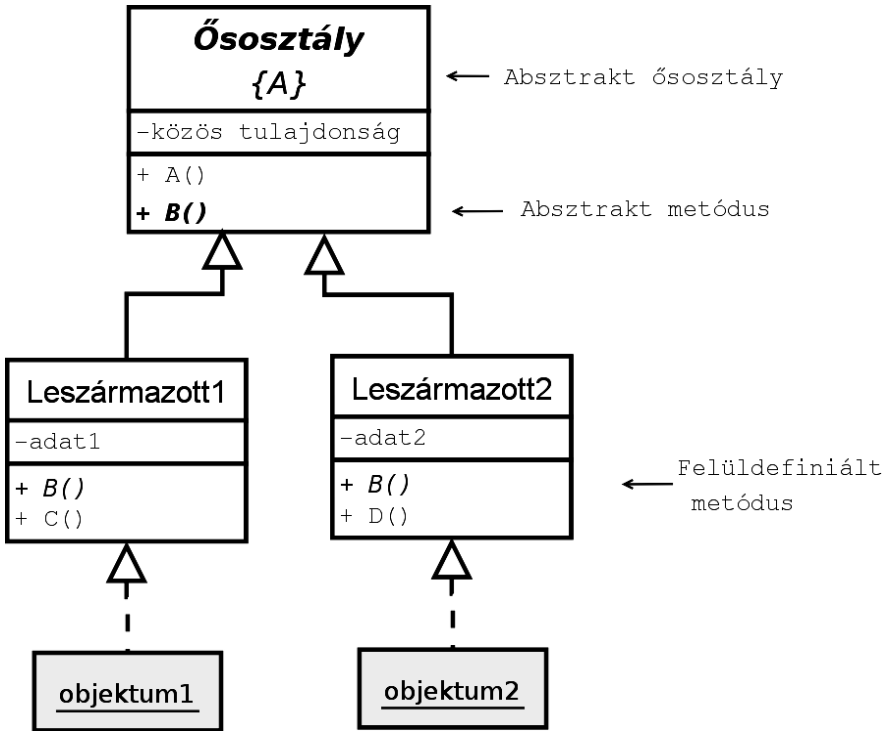
Az objektumhierarchia tervezése során – a konkrét programnyelvi megvalósítástól függetlenül – kialakítjuk a feladatot leíró típusdefiníciókat. Sok esetben a programtervezés során szükségünk van egy közös ősre, amiből kiindulva az összes leszármazott értelmezhető. Tehát egy olyan osztályt kell definiálnunk, amely egységbe foglalja, könnyen kezelhetővé teszi a leszármazott osztályhoz tartozó objektumokat. Ezek az osztályok általában még általánosak, sokszor csak az osztályhierarchia számára szóló üzenetek szabályait definiálják. (A hierarchia különböző szintjéhez kapcsolódó objektumokat így azonos üzenetekkel érhetjük el.) Ezeket az osztályokat absztrakt osztályoknak nevezzük.

Az absztrakt osztályok kizárólag örökítési céllal létrehozott osztályok. Az absztrakt osztályokhoz nem rendelhetőek objektumok, nem példányosíthatóak.

Az absztrakt osztályok megvalósításában gyakran hiányoznak az adattagok, és sokszor találkozunk üres metódusokkal, melyek csak egy egységes interfészt, a leszármazott objektumok számára küldhető azonos üzeneteket definiálják.

Az absztrakt metódusok mindegyike üres törzsű virtuális metódus. Ezeket az absztrakt metódusokat majd a leszármazottak felüldefiniálják, megvalósítják. Absztrakt metódus csak absztrakt osztályban adható meg.

Az absztrakt osztályok nem példányosíthatók, hiszen tartalmaznak olyan műveleteket, amelyek még definiálatlanok.



7. ábra: Absztrakt osztályok használata

Az absztrakt osztályok segítségével osztályhierarchiát is felépíthetünk, de a leszármazott osztályokat mindaddig absztraktnak kell tekinteni, míg az összes absztrakt metódust meg nem valósítják (nem adnak rá konkrét definíciót). Ennek megértéséhez azonban meg kell ismerkednünk a polimorfizmus tulajdonságaival.

Egy példában egy olyan osztályhierarchiát szeretnénk megadni, amely a matematikában megismert függvények kezelését teszi lehetővé. Az osztályozás során megadhatjuk a konstans-, a lineáris-, és a másodfokú stb. függvényeket. A függvényértékek eléréséhez a matematikában ismert y(x) alakú üzeneteket használunk majd, ahol y(x) az x helyen felvett függvényértéket jelenti. A közös üzenetküldésnek az összetettebb objektumorientált programokban nagyon fontos szerepe lesz. Az egyes konkrét osztályok közös ősének egy absztrakt Függvény osztályt definiálunk, amelynek csak egy absztrakt y metódusa lesz. A leszármazottakban ezt az y metódust fogjuk rendre felüldefiniálni.

2.4.2. Interfészek

Az absztrakció fogalmánál néhány pillanatot elidőzve, egy fogalmat még tisztáznunk kell. Az objektumorientált modellezés során az absztrakció egy magasabb szintű megvalósítása lehetővé teszi az olyan osztályok (interfészek) definiálását, amelyek már konkrét objektumhoz köthető adattagot sem tartalmaznak, csupán azoknak az üzeneteknek a formátumspecifikációját adják meg, amelyek majd csak később, a leszármazottakban fognak megjelenni. Az interfészek használatával egy magas szintű, a konkrét megvalósítástól független felületet lehet a feladat számára megadni. A tervezés fázisában az interfészek használata nagyfokú rugalmasságot, könnyű módosíthatóságot nyújt. Az absztrakt osztályokhoz hasonlóan az interfészekben deklarált műveleteket is a leszármazottak valósítják meg.

2.5. Többalakúság

A többalakúság (polimorphism) szintén objektumorientált alapfogalom. Mivel ez a fogalom többféle jelentéssel bír, ezért mélyebb vizsgálatot igényel. Először is meg kell különböztetnünk a változók és a műveletek többalakúságát.

A változók polimorfizmusa azt jelenti – a dinamikus típusokat támogató nyelvekben (Smalltalk, PHP) – hogy egy változót dinamikus típusúnak is deklarálhatunk, a hagyományos statikus típusok mellett. A dinamikus típus azt jelenti, hogy egy változó élettartama során más-más osztályhoz tartozó objektumokat is értékül kaphat. Ezeknél a nyelveknél a felvázolt megoldás nagyon gyakori, ám számos, nehezen feltárható hibalehetőséget rejt magában.

Változók többalakúsága

A dinamikus típust közvetlenül nem támogató nyelvekben is van lehetőségünk polimorfikus változók deklarálására, az öröklődés egyik hasznos tulajdonságának kihasználásával. Az automatikus típuskonverziót, illetve a típuskényszerítést kihasználva egy leszármazott osztály példányait az ősosztály egy előfordulásának is tekinthetjük (bizonyos megszorításokkal) így egy ősosztályúként deklarált változó egy leszármazott objektumot is felvehet értékül.

Egy változó, élettartalma alatt – az implementáló nyelv szabályai szerint – más-más, vele öröklési viszonyban álló osztálybeli objektumot is értékül vehet fel.

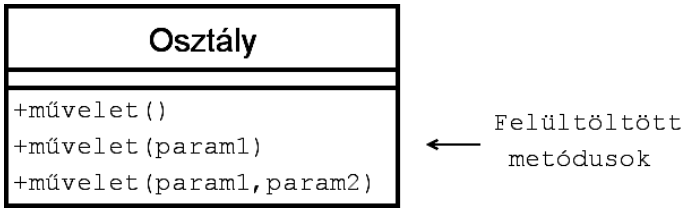
A műveletek többalakúsága

Sokkal gyakoribb és elterjedtebb a műveletek többalakúságának használata. A feladatmegoldások során sokszor kell hasonló műveleteket elvégezni. Az objektumorientált terminológia szerint az egyes objektumoknak hasonló üzeneteket kell küldeni. A műveletek polimorfizmusát általában kétféleképpen értelmezzük.

Felültöltés

Az első értelmezés szerint egy objektum a hozzá érkező azonos üzeneteket az üzenetben érkező adat típusa (osztálya) szerint másként hajtja végre. Ezt a megoldást felültöltésnek nevezzük (overloading). Az egyes programozási nyelvek lehetővé teszik az azonos nevű, ám eltérő paraméterezettségű metódusok definiálását egy osztályon belül. Itt a metódus neve mellett a metódusok szignatúrája (paraméterek típusa és száma) is hozzátartozik az egyértelmű azonosításhoz. Ezután minden egyes metódushívás érvényes lesz, amely megfeleltethető valamelyik felültöltött metódusnak. A hívás pedig – az aktuális paraméterlista alapján – már fordítási időben ellenőrizhető, majd a futás során végrehajtható.

Egy objektum egy-egy üzenetre többféleképpen reagálhat. A felültöltött metódusok (overload) segítségével a hasonló feladatokat azonos névvel illethetjük.

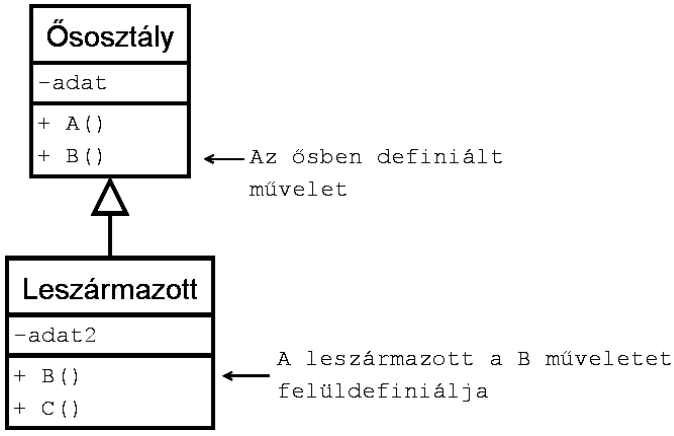


8. ábra: Felültöltött metódusok egy osztályban

Felüldefiniálás

A többalakú műveletek másik csoportja a műveletek felüldefiniálása (overriding). Az öröklés tulajdonságát úgy definiáltuk, hogy a leszármazott osztály örökli az ősosztály minden műveletét is. Ahol az ősosztály műveletei használhatók, ott a leszármazott műveletek is felhasználhatók lesznek. Azonban az öröklést sokszor azért használjuk, hogy a leszármazott osztályt specializáljuk, mert a leszármazottak egy-egy műveletet már másként hajtanak végre. Egy osztály akkor definiál felül egy metódust, ha az öröklési láncban megtalálható az adott metódus, így azt teljes mértékben örökölné, de az adott osztály ugyanarra a metódusra saját definíciót ad.

Egy objektum attól függően, hogy az osztályhierarchia melyik szintjén levő osztály példánya, egy-egy üzenetet másképp hajt végre. A metódusok felüldefiniálása (override) ezt teszi lehetővé.



9. ábra: Felüldefiniált metódusok az osztályhierarchiában

Ha feladatunk egy olyan program elkészítése, amely három különböző típusba sorolható adatot tárol veremszerkezetben, akkor strukturált módon a három verem számára minden egyes művelethez három-három eltérő nevű szubrutint kell írni. Az objektumorientált szemlélet szerint pedig egyszerűbb azonos üzenetként – azonos nevű, felültöltött metódusként – definiálni a műveleteket, amelyeket a többalakúság tulajdonságának kihasználásával hatékonyan érhetünk el. A polimorfizmus hatékony kihasz-

nálása átláthatóbb és kevésbé szerteágazó programozási modellek építését teszi lehetővé.

2.5.1. A futás alatti kötés

Az egyes programozási nyelvek kétféle üzenet szerepkört definiálnak. Beszélhetünk statikus és dinamikus műveletekről. A hagyományos programozási nyelvekben statikus kötésű szubrutinokkal találkozunk. Ez azt jelenti, hogy az adott fordítóprogram már a fordítás során (fordítási időben) a szubrutin kapcsolási címét egyértelműen meghatározza (early binding – korai kötés). Így egy adott nevű szubrutin hívása egyértelmű, a hagyományos programozási nyelvek nem lennének alkalmasak arra, hogy hatékonyan használják ki a műveletek felüldefiniálását, illetve felültöltését, vagyis azt, hogy egy objektum módosított metódusait elérjük. (A statikus kötésű metódusokat az objektumorientált programozásban osztálymetódusnak nevezik.)

Az objektumorientált szemlélet viszont más megoldásokat vár, hiszen az egyes objektumokhoz szorosan hozzá kell rendelni az egyes műveleteket, tehát olyan megoldást kell adni, amely futási időben képes az azonos üzeneteket – legyen az egy felültöltött, vagy felüldefiniált metódus hívása – egyértelművé tenni. Ehhez a megoldáshoz a műveletek dinamikus összekapcsolása szükséges.

A többalakúság közös tulajdonsága, hogy a műveletek azonos névvel hívhatók, ám a végrehajtáskor a futás alatti kötés, vagy késői kötés (late binding, runtime binding) segítségével válik egyértelművé, hogy egy objektum egy üzenetre melyik műveletével reagál.

Tekintsük azt az esetet, amikor – statikus hozzárendeléssel definiálva – egy osztályban (továbbiakban ősosztály) definiáljuk az A és B metódust, ahol az A metódus a B-t hívja. Öröklléssel létrehozunk egy leszármazott osztályt. Az öröklés hatására a leszármazott is rendelkezik mind az A, mind a B metódussal. A leszármazott osztályban a B metódust felüldefiniáljuk. Ha az ősosztályt példányosítjuk, és a létrejövő példányt felkérjük, hogy hajtsa végre az A műveletet, akkor az elkezdi az ősosztály A műveletét végrehajtani, amely az ősosztály B metódusát hívja meg, majd az abban definiált műveleteket végrehajtja. Ez eddig hibátlanul működött. Ha azonban a leszármazott osztály egy példányával kívánjuk ezt a fenti műveletet

végrehajtani, akkor kudarcot vallunk. Hiszen a leszármazott osztály példánya (ha felkérjük az A művelet végrehajtására) mivel a saját osztályában nem talál A metódust, így az öröklési láncon visszafelé haladva megtalálja és végrehajtja az ősosztály A metódusát. Az ősosztály A metódusa pedig a B metódust hívja meg. És itt történik a hiba, hiszen a statikus szerepek miatt az ősosztály csak a saját metódusait ismeri, a vezérlés „nem találja meg" a leszármazott példányhoz tartozó felüldefiniált B metódust, így azt az ősosztályból hívja. A leszármazott példányhoz viszont a felüldefiniált B műveletet kellett volna elvégezni.

A dinamikus összekapcsolást az egyes programnyelvekben eltérően végzik, de vázlatosan a következő szempontok szerint járnak el:

- Azokat a műveleteket, amelyeknél ki szeretnénk használni a többalakúságot, virtuális műveletnek kell definiálni.
- Ha egy műveletet egy öröklési ágban virtuálisnak minősítettünk, akkor annak az összes leszármazottjában is virtuálisnak kell lennie. A virtuális metódusok kezeléséhez minden osztályhoz az ún. virtuális metódus táblákat (VMT) rendelnek hozzá. A VMT nem más, mint az egyes osztályhoz rendelt – mind az ősosztályból származó, mind a felüldefiniált, ill. felültöltött – metódusok címeit tartalmazó adatstruktúra.
- Általánosan kijelenthető, hogy minden definiált osztályhoz (amely használ virtuális metódusokat) tartozik egy VMT, amelyet a példányosításkor hozzá kell rendelni az új objektumhoz.

  Ezt a műveletet a továbbiakban konstrukció névvel illetjük.

A dinamikusan összekapcsolt többalakú változó- és metódushasználat során nagyon fontos szempont az objektumok tervezett élettartamuk végén való teljes memória-felszabadítás. A helyes programtervezéshez hozzátartozik a munka utáni „rendrakás". Itt a konstrukcióval ellentétes irányú műveletet kell végrehajtanunk, melyet destruktornak hív a szakirodalom.

A VMT kezelését az egyes programozási környezetek mind másként valósítják meg, működését elrejtik a programozó elől. A hagyományos hibrid programrendszereknél (Object Pascal, Delphi, C++) a virtual kulcsszóval hozhatunk létre dinamikusan összekapcsolt metódusokat, a tisztán objektumorientált nyelveknél (Smalltalk, Eiffel, Java) minden metódus alapértelmezés szerint virtuális.

2.6. Osztályok és objektumok együttműködése, kapcsolatai

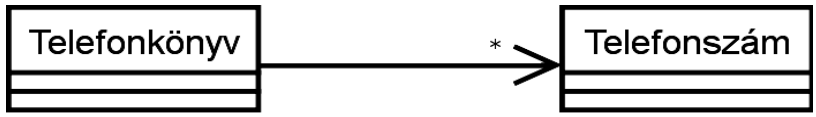
A valós világ modellezésénél az objektumok osztályozása és az osztályok hierarchikus felépítése számos feladatra nyújt megoldást. Az egyszerűbb feladatok modelljei után egy valós probléma megoldásakor rájövünk, hogy az osztályok között nemcsak hierarchikus kapcsolatokat építhetünk, hanem egymástól gyökeresen eltérő osztályok között kell megteremtenünk valamilyen kommunikációt úgy, hogy az objektumorientált alapfogalmakat ne sértsük meg.

Az objektumoknak valamilyen kapcsolatban kell állniuk egymással, ha kommunikálni szeretnének egymással. A kapcsolatokat értelmezhetjük konkrét objektumok között, vagy általánosítva az osztályok között. Az osztályok közötti kapcsolatok esetén a kapcsolatok is természetesen általános értelmet nyernek.

Az objektumok kommunikációját két nagy csoportba: ismertségi, illetve tartalmazási kapcsolatba sorolhatjuk. Az objektumok illetve az osztályok közötti kapcsolatok és a kommunikáció ábrázolásához az UML objektum- és osztálydiagramjait használjuk.

2.6.1. Ismertségi kapcsolat

Két objektum abban az esetben van ismertségi kapcsolatban (asszociáció), ha mindkettő egymástól függetlenül létezik, és legalább az egyik ismeri, vagy használja a másikat. A tartalmazó fél valamilyen mutatón, vagy referencián keresztül éri el a másik objektumot. Az ismertségi kapcsolatban álló objektumok üzenetet küldhetnek egymásnak. Amennyiben a program élettartama során egyes, a kapcsolatban szereplő, hivatkozott objektumokra már nincsen szükség, akkor az objektumok megszüntethetőek, de figyelni kell arra, hogy az objektum megszüntetéskor a kapcsolatot is meg kell szüntetni.



10. ábra: Ismertségi kapcsolat

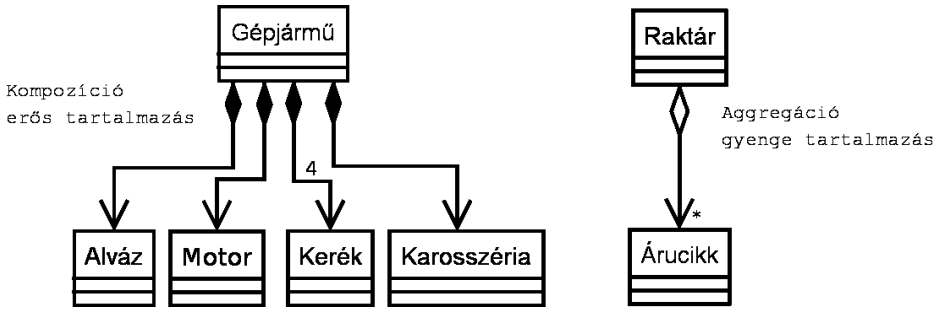
2.6.2. Tartalmazási kapcsolat

A tartalmazási kapcsolat – egész-rész kapcsolat – esetén két objektum nemcsak ismeri egymást, hanem az egyik objektum fizikailag is tartalmazza a másikat. Másként fogalmazva az összetett objektum része egy másik, ún. részobjektum. A tartalmazási kapcsolatokat további két részre bonthatjuk.

Gyenge tartalmazás (aggregáció) esetén a részobjektum az összetett objektumtól függetlenül is létezik. Gyenge tartalmazás figyelhető meg egy raktár és a benne raktározott árucikkek között, hiszen az árucikkek a raktártól függetlenül léteznek, felhasználhatóak.

Erős tartalmazás (kompozíció) esetén a részobjektum önállóan nem fordul elő. Egy gépjármű osztály és a hozzátartozó alkatrészek között erős tartalmazási kapcsolat van, hiszen az alkatrészek önmagukban nem alkotnak működőképes járművet.

A tartalmazási kapcsolatot az UML-jelölés szerint az összetett objektum osztályától egy rombuszból kiinduló nyíl jelöli. Gyenge tartalmazás esetén üres, erős tartalmazás esetén teli rombuszt rajzolunk.



11. ábra: Tartalmazási kapcsolatok

2.7. Konténer osztályok

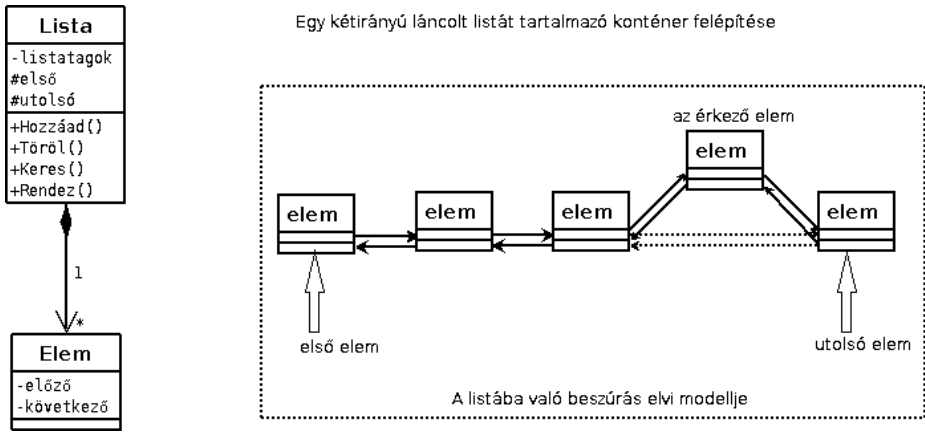
Gyakran szükségünk van olyan osztályokra, amelyek gyenge tartalmazási kapcsolattal sok, a program futási ideje alatt változó számú – akár egymástól különböző osztályba tartozó – objektumot tartalmaznak.

Amíg az objektumok csak 1:1, vagy 1:N kapcsolatban állnak – ahol N értéke konstans – addig a hagyományos ismertségi, vagy tartalmazási kapcsolattal is dolgozhatunk. Ha több objektumot 1:* kapcsolat köt össze – vagyis a program tervezésekor még ismeretlen a számosság, vagy konstans értékkel nem korlátozható – akkor ún. konténer objektumokat haszná-

lunk. A konténer objektumokat definiáló osztályokat összefoglaló névvel konténereknek nevezzük.

A konténer osztályokból példányosított objektumok szerepe és célja, hogy a bennük elhelyezett objektumokat kezelje, és adatkarbantartó műveleteket végezzen (beszúrás, törlés, rendezés, keresés stb.). A konténer osztályok – és az ezeket absztrakt szinten definiáló interfészek – használatával egy magas szintű, a tervezési és a megvalósítási szinteket jól elkülöníthető eszközrendszer áll a programozó rendelkezésére. A konténer osztályok interfészeinek definiálásakor megadható a konténer szabályrendszere, és az üzenetek egységesíthetők. Az egységesítés azért célszerű, mert a gyakran használt adatkezelő műveletek,

Az olyan fejlett adatstruktúrák használatához, mint a dinamikus tömbök, a verem, az egyszerű és összetett listák, a bináris és többágú fák, a kupac, a halmaz, az asszociatív tömbök, a hasítótáblák, gráfok stb. használatához elengedhetetlen a konténer osztályok használata. [Marton02, Cormen01]



12. ábra: Egy konténer osztály lehetséges megvalósítása

A konténer osztályok szerepét egy nagyon egyszerű példával szeretném bemutatni. Egy cég „A” városból vasúton szeretné a termékeit elszállítani „B” városba. Ebben az esetben a konténer objektum szerepét maga a vasúti szerelvény tölti be, hiszen az áru berakodása a vagonokba, az utazás, majd a kirakodás mind jól definiált és a termékektől elkülönülő magas szintű művelet. A konténer független a szállított terméktől.

2.8. Komponensek

Az objektumorientált programtervezés során célszerű a logikailag összetartozó és együttműködő osztályokat csoportosítani. Az így előálló csomagok, vagy más terminológia szerint komponensek újrafelhasználható, más programokba beilleszthető kódot alkotnak. A komponensek egymásba ágyazhatóak, így tetszőleges mélységű komponens-struktúra építhető fel.

> Komponens: egy adott rendszer részét képező különálló, általános célú osztálygyűjtemény (amely önmagában is zárt egységet képez), de más programokba is szabadon beépíthető.

Az elkészített komponensek szabványos, összeszerkesztett, tesztelt építőelemek. Az egyes objektumorientált programnyelvekben definiált komponensek használata hatékonyabb és szabványosabb fejlesztést tesz lehetővé. Ilyen komponens lehet például: egy adott adatbáziskezelő rendszer szabványos elérését, vagy éppen egy grafikus vezérlőelemet megvalósító komponens.

2.9. Kérdések

- Melyek a legfőbb különbségek egy osztály és egy objektum között?
- Miket tartalmaz egy osztály?
- Mit nevezünk egységbezárásnak?
- Mit jelent az adatrejtés elve?
- Milyen főbb hozzáférési kategóriákat definiál az objektumorientált szemlélet?
- Mi a különbség az adattag és a metódus között?
- Mit jelent az öröklődés?
- Mi a többalakúság?
- Mit jelent a futásidejű (késői) kötés fogalma?
- Mikor nevezünk egy osztályt absztraktnak?
- Miért nem példányosítható egy absztrakt osztály?
- Mit értünk objektumok együttműködési kapcsolatán?
- Mi a különbség a gyenge és az erős tartalmazási kapcsolat között?
- Mi célt szolgálnak a konténer osztályok?

2.10. Feladatok

Tervezze meg az alábbi, egymástól független osztályokat, és rajzolja meg az UML osztálydiagramokat! (Lehetőleg gyűjtse össze a legjellemzőbb adatokat és azokat a műveletet, melyek az adott osztályhoz tartozó példányokon elvégezhetők.):

- Alkalmazott
- Árucikk
- Gépjármű
- Dobókocka
- Ébresztőóra
- Bejárati ajtó
- Lézernyomtató
- Tintasugaras nyomtató
- MP3-zeneszám
- Fénykép
- Hibajelenség

3. A Java technológiáról közelebbről

3.1. Keretrendszerek

A Java fejlesztésekhez mindig valamilyen fejlesztő környezetet használunk, legyen az a legegyszerűbb parancssori fordító, vagy a legmodernebb grafikus fejlesztőkörnyezet.

A Sun Microsystems által kifejlesztett szabványos környezet a JDK (Java Development Kit). Ez tartalmazza a fejlesztéshez alapvetően szükséges részeket (fordító, nyomkövető, futtató környezet és egyéb segédprogramok), illetve a szabványos Java osztálykönyvtárakat (továbbiakban API – Application Programming Interface) amely jelenleg kb. 6500 különböző osztályt és interfész hiearchikus felépítését jelenti.

Az API magának a Java nyelvnek a teljes – nyílt forráskódú – osztálykönytár-struktúráját tartalmazza. (A telepített rendszer futtató környezetében a tömörített `/jre/lib/rt.jar` fájlban, illetve a forrása az `src.zip` fájlban található.) A tervezésnél a logikailag könnyen átlátható, hibamentes és könnyen felhasználható osztályhierarchia megalkotását tűzték ki célul. A programozók részére rengeteg, jól elkülönülő osztályt adnak a kezükbe feladataik elvégzésére. Ezek az osztályok (kb. 6500 db) azonban csak jól felépített struktúra alapján lesznek átláthatók és hatékonyan felhasználhatók.

A forráskódok szerkesztéséhez olyan szövegszerkesztőt célszerű használni, amely az egyes nyelvi jellegzetességeket felismeri, átlátható és kiemeléseket használ, emellett gyors, és kis helyet foglal. (SciTe, WsCite, TextPad, Emacs, JEdit, UltraEdit stb.) [4]

Az egyes cégek által kifejlesztett integrált környezetek nagyban segítik a programozási munkát. Tartalmaznak többek között szövegszerkesztőt, fordító, futtató, nyomkövető eszközt és beépített súgót. A legmodernebb (fizetős) környezetek ezeken felül tartalmaznak beépített adatbázis-, web-, alkalmazás-, portál- és azonnali üzenetküldés szervereket a hatékony csapatmunka elősegítésére.

A legelterjedtebb keretrendszerek (IDE: Integrated Development Engine), a teljesség igénye nélkül: Sun-JavaStudio, Eclipse, NetBeans, Borland-JBuilder, IBM-Websphere, JCreator, Oracle-JDeveloper.

3.2. Nyelvi alapelemek

3.2.1. Kulcsszavak és azonosítók

A forráskódban a nyelvi elemeket két csoportra bontjuk: kulcsszavak és azonosítók. A Java kulcsszavai olyan szavak, amelyek a Java programok szerkezetét és vezérlését meghatározzák, és amelyeket azonosítókként nem használhatunk fel.

`abstract`	`boolean`	`break`	`byte`	`case`
`catch`	`char`	`class`	`const`	`continue`
`default`	`do`	`double`	`else`	`extends`
`final`	`finally`	`float`	`for`	`goto`
`if`	`implements`	`import`	`instanceof`	`int`
`interface`	`long`	`native`	`new`	`package`
`private`	`protected`	`public`	`return`	`short`
`static`	`super`	`switch`	`synchronized`	`this`
`throw`	`throws`	`transient`	`try`	`void`
`volatile`	`while`			

1. táblázat: A Java kulcsszavai

A kulcsszavak közül a `goto` és a `const` a nyelvben nem értelmezett kulcsszó, de foglalt minősítést kapott, így azokat programjainkban nem használhatjuk azonosítóként. A nyelv alapértékeit jelző literálokkal sem képezhetünk azonosítókat (`true`, `false`, `null`).

Az azonosítóképzés szabálya nagyon egyszerű. Egy azonosítóval egy elemet (osztály, adattag, metódus, változó) azonosítunk, és valamilyen betűvel kezdődő, és betűvel vagy számjegyekkel folytatódó karaktersorozatként adunk meg. Az azonosítók tartalmazhatnak _ karaktert is.

Mivel a Java a unicode szabvány szerinti kódolást használ (UTF-8), az azonosítókban megengedettek az ékezetes karakterek is, azonban bizonyos esetekben nem használhatóak platformfüggetlenül. (A csomagok, osztályok és a fájlok nevei nem mindenhol képezhetőek le azonos módon az adott operációs rendszerek szabályai szerint.)

Az azonosítók képzésénél azt is figyelembe kell venni, hogy a Java nyelv kis- és nagybetű érzékeny. Az ajánlások szerint a hosszabb nevű

azonosítóknál a második szótól kezdve az első betűket nagybetűvel írjuk, javítva az olvashatóságot:

```
születésiDátum, nagyonHosszúnevűAzonosító.
```

Megjegyzés: Az azonosítóképzés ajánlott szabályait a 12. fejezetben vagy a [6]-ban találjuk meg részletesebben.

3.2.2. Adattípusok

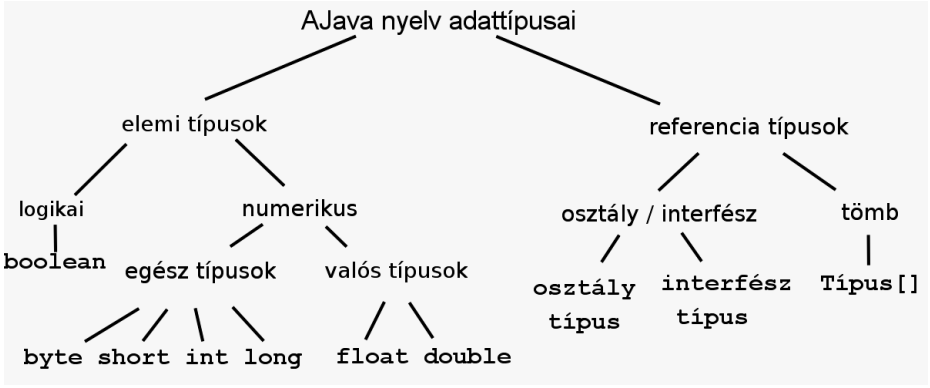
Mielőtt elmélyülnénk a Java nyelv osztályainak vizsgálatában, tisztában kell lennünk a Java nyelv típusképzési szabályaival. A nyelv a következő alaptípusokkal dolgozik:

Alaptípus	Csomagoló osztály	Értékkészlet	Leírás, méret
boolean	Boolean	true, false	logikai típus (1 bájt)
char	Character	65535 db: '\u0000'…'\uffff'	unicode karakter (2 bájt)
byte	Integer	-128…127	előjeles egész szám (1 bájt)
short	Integer	-32768…32767	előjeles egész szám (2 bájt)
int	Integer	-2147483648… 2147483647	előjeles egész szám (4 bájt)
long	Long	-9223372036854775808 … 9223372036854775807	előjeles egész szám (8 bájt)
float	Float	$1.4\ 10^{-45}$ … $3.4\ 10^{38}$	lebegőpontos valós szám (4 bájt)
double	Double	$4.9\ 10^{-324}$ … $3.8\ 10^{308}$	lebegőpontos valós szám (8 bájt)

2. táblázat: A Java alaptípusai (elemi típusok)

Az alap-, vagy elemi típusokhoz tartozó változók – a C nyelvhez hasonlóan – adott típushoz tartozó értékeket tárolnak. Az ilyen változók használata megegyezik a C nyelv szabályaival (deklaráció, kezdőérték adás stb.).

A táblázat második oszlopában látható csomagoló osztály kifejezés azt jelenti, hogy ezeket az alaptípusú változókat egy, a típust leíró osztály objektumaként is felfoghatjuk, és az objektumokkal azonos módon kezelhetjük őket (bővebben lásd: 3.2.11. fejezet).



13. ábra: A Java adattípusai

A Java adattípusai gépfüggetlenek, vagyis minden architektúrán, ill. operációs rendszeren azonos formában, azonos ábrázolási tartománnyal lesznek tárolva, feldolgozva.

A C nyelvből ismert void kulcsszó – a Java nyelvben – nem típust jelöl, hanem csupán azt, hogy egy metódus nem ad vissza értéket a hívó rutinnak.

A változók deklarációja során legalább a változó típusának, és azonosítójának szerepelni kell. Több változónevet vesszővel választunk el, illetve a deklaráció során kezdőértéket is rendelhetünk a változókhoz. Ebben az esetben a változó definíciós deklarációjáról beszélünk.

```
int x;
int y, z;
short r=4, p=0;
```

Az osztályok adattagjai automatikus kezdőértéket kapnak (amennyiben nem rendeltünk hozzájuk semmit). Az automatikus értékek a következők:

```
boolean                 false
char                    '\u0000'
byte, short, int, long  0
float, double           0.0
objektum referencia     null
```

Az egyes metódusok lokális változói azonban határozatlanok lesznek, az inicializálásról a programozónak kell gondoskodnia!

A Java nyelv objektumait a referencia típusú változókkal érjük el. Az osztályok példányosítása során valahol a memóriában helyet foglalunk az

objektumnak és egy változóval – referencia – ezt a memóriaterületet elérhetjük.

```
Teglalap t2 = new Teglalap (2,5);
```

Azonban nem szabad összekevernünk referenciát a C nyelv mutató típusával. Ott a mutató egy adott memóriacímre való hivatkozás, amelynek adattartalmát indirekcióval érhetjük el. A Java nyelvben azonban a referencia különbözik a hagyományos mutatóktól abban, hogy a hozzárendelés után ezt a címet azonnal „elrejti” a programozó elől, és automatikusan az egész lefoglalt objektumot jelenti. A példányosítás során tehát a `new` operátor által előállított referenciát hozzárendeli a megfelelő típusú azonosítóhoz (`t2`). Ezután az ezzel az azonosítóval hivatkozott objektum nyilvános tagjai hivatkozhatók.

Egy speciális referencia érték a `null`, amely értékül adható minden referencia típusú változónak. A `null` olyan referencia, amely nem mutat egyetlen objektumra sem. A `null` értékű referenciákon keresztül nem lehet egyetlen objektumra sem hivatkozni, metódust hívni (ha mégis megtennénk futási idejű kivétel esemény keletkezik: `NullPointerException`; a kivételkezelésről bővebben a 7. fejezetben lesz szó).

Abban az esetben, ha egy objektumra már nincsen többé szükségünk, célszerű a lefoglalt memóriát felszabadítani. Sok programozási nyelvben ezt a programozónak kell kezdeményeznie – destruktor hívással. A Java nyelv azonban nem használ destruktort, hanem az ún. szemétgyűjtő mechanizmussal (garbage collector) szünteti meg a nem használt objektumokat.

A szemétgyűjtés a virtuális gép része, működése elve a következő: minden egyes objektumot addig tekint használatban lévőnek, amíg arra valamilyen referencia hivatkozik. Amint ezt a hivatkozást megszüntetjük, az objektum már a programból elérhetetlen – hiszen megszűnt az összekapcsolás – a tárterületet a szemétgyűjtő felszabadítja.

3.2.3. Megjegyzések

A forráskódban fontos szerep jut a megjegyzéseknek. Ezeket a részeket a fordítóprogram figyelmen kívül hagyja, csak a fejlesztők számára – az elemzés megkönnyítésére – hordoznak információt a forráskód működéséről. A forrásban minden olyan helyen lehet megjegyzés, ahol az a program szintaktikáját, ill. szemantikáját nem befolyásolja.

A Java három fajta megjegyzést ismer.

- A legegyszerűbb a kettős törtjel: `//`. A jel után következő karaktersorozat a sor végéig megjegyzésnek minősül.
- A második a C nyelvből ismert `/* */` zárójelpár, ahol a közrezárt rész szintén nem kerül fordításra. Ez a megjegyzés több sorból is állhat.
- A második típus speciális változata a `/** */`alakú, ún. dokumentációs megjegyzés. Ezeket a dokumentációs megjegyzéseket egy definiált elem (osztály, adattag, metódus, csomag) előtti sorokban használjuk. Ezeknek a megjegyzéseknek különleges jelentősége van, hiszen a Java fejlesztőkörnyezet javadoc programjával html formátumú dokumentációt generálhatunk. (A javadoc program leírását lásd a 13.2. fejezetben)

Megjegyzés: Maga a Java API dokumentációja is ezzel a segédprogrammal készült.

3.2.4. Osztályok

A Java nyelvben az osztály az ábrázolás alapegysége. Minden adatot, és minden műveletet osztályokon belül értelmezünk. Mivel a nyelv megalkotásakor törekedtek a tisztán objektumorientált kialakításra, osztályokon kívül nem helyezhetünk el adatokat, utasításokat.

Minden osztály – az elméleti megközelítést követve – két logikailag különváló részre bontható: adatokra és műveletekre. Az adatok (adattagok) az egyes objektumok tulajdonságainak tárolására, a műveletek (metódusok) pedig kizárólag ezeken az adatokon való műveletvégzésre szolgálnak.

Az alábbi példában egy téglalapokat leíró osztályt definiálunk. A téglalapot az *a* és *b* oldalának hosszúságával jellemezzük, és értelmezzük a terület és a kerület műveleteket.

```
public class Teglalap{                          //osztálydefiníció
  private double a;                                    //adattagok
  private double b;

  public Teglalap(double a_, double b_){          //konstruktor
     a=a_; b=b_; }
  public double kerulet(){          //metódusok, tagfüggvények
     return 2*(a+b); }
  public double terulet(){
     return a*b;  }
}
```

A programban a `Teglalap` nevű publikus osztályt definiáljuk, melynek két adattagja van. Az adattagok az adatrejtés elve szerint privát minősítésűek lesznek.

Az *„a”* és *„b”* adattagok deklarációja után a konstruktor műveletét fejtjük ki. A konstruktor arra szolgál, hogy a `Teglalap` osztályból példányokat hozzunk létre a későbbiekben. A konstruktort minden esetben az osztály nevével megegyező nevű metódusként definiáljuk.

Figyeljük meg, hogy a konstruktor definiálásakor nem adtunk meg visszatérési értéket, mivel a konstrukció eredménye egy Teglalap típusú objektum lesz. A konstruktor illetve a metódusok végrehajtható utasításait kapcsos zárójelek közé írjuk. A metódusok definiálásakor először a láthatóságot, a visszatérési típust, a metódus nevét, majd a formális paraméterlistát adjuk meg.

A Java nyelv névadási konvenciója szerint az osztályok neve nagybetűvel, az adattagok, metódusok és egyéb azonosítók pedig minden esetben kisbetűvel kezdődnek. Ennek betartása javítja a programok és a forráskódok olvashatóságát, átláthatóságát.

Az osztályok definiálásához szorosan hozzátartozik, hogy egy osztályt egy, az osztály nevével azonos nevű, **„.java”** kiterjesztésű fájlban adunk meg, kis- és nagybetű helyesen. Tehát a fenti példát a **Teglalap.java** forrásfájlban helyezzük el.

Megjegyzés: Egyes operációs rendszerek nem konvertálják helyesen a unicode karaktereket, így az oszálynevek különbözni fognak a tárolt fájlnevektől. Ekkor az elkészített programok nem lesznek hordozhatók, más rendszereken bizonytalan a futtathatóságuk. Ezért az osztályok névadásánál kerüljük az ékezetes karaktereket, jóllehet a nyelv szabályai ezt megengedik.

A `Teglalap` osztály definiálásával, majd fordításával már egy használható osztályt kaptunk. Mivel azonban az osztály csak egy típust definiál, még szükséges az is, hogy példányosítsuk. A példányosítás során születik meg egy konkrét objektum, amely már rendelkezni fog saját memóriaterülettel, és az üzeneteket is ennek a konkrét objektumnak küldhetjük.

A példányosításhoz és a `Teglalap` osztály kipróbálásához egy új osztályt definiálunk. A `Teszt` osztály forrásfájlját (**Teszt.java**) ugyanabban a könyvtárban helyezzük el, mind a tesztelni kívánt **Teglalap.java** forrásfájlt.

```
public class Teszt{
  // A main metódus, a program belépési pontja
  public static void main(String args []){

     // t1 objektum deklarációja
     Teglalap t1;

     // példányosítás, konstruktor hívása
     t1 = new Teglalap(3,4);

     //ugyanez tömörebben
     Teglalap t2 = new Teglalap(2,5);

     // A program törzse, az egyes objektumok elérése
     System.out.println("A t1 kerülete:" + t1.kerulet() );
     System.out.println("A t1 területe:" + t1.terulet() );
     System.out.println("A t2 kerülete:" + t2.kerulet() );
     System.out.println("A t2 területe:" + t2.terulet() );
  }
}
```

A fenti példában létrehoztuk a `Teszt` osztályt, amelyből a `Teglalap` típusú objektumokat el kívánjuk érni. A futtathatósághoz – konzolos programok esetén – egy main metódust kell definiálnunk, amelynek törzsrészében megadjuk a program utasításait. (A main többi módosítójára még később visszatérünk.) A `Teglalap` típusú `t1` objektumot a `Teglalap` osztály példányosításával keltjük életre.

A this kulcsszó

A metódusok definiálásakor gyakran használt elem a `this` kulcsszó. A Java nyelvben a `this` több jelentést is takarhat.

- A metódusokban szereplő `this` az aktuális objektumra való hivatkozást jelenti. Az egységbezártság és az adatrejtés elvében azt fogalmaztuk meg, hogy egy metódus az objektum saját adatain dolgozik. Ezért

egy adattagra való hivatkozás – minősítés nélkül – alapértelmezés szerint az aktuális példány változóját éri el. Ehhez az adott metódusnak tudnia kell, hogy éppen melyik példány adataival kell dolgoznia.

Ezt úgy oldották meg, hogy minden egyes metódus rendelkezik egy rejtett, referencia típusú paraméterrel, amely éppen az aktuális példányt jelöli. Ezt a paramétert minden egyes híváskor a végrehajtandó metódus megkapja. A this használatával az osztály adattagjai akkor is elérhetőek lesznek, ha a paraméterből érkező, vagy azonos nevű lokális változók elfednék azokat (lásd láthatósági szabályok).

A `Teglalap` osztály konstruktorát így is felírhatjuk:

```
public Teglalap(double a, double b){
  this.a = a;
  this.b = b;
}
```

Ahol „`this.a`” a példány adattagját, az „`a`” pedig az érkező paramétert jelenti.

- A this tagfüggvényként is értelmezhető. Az 4.3. fejezetben részletesen lesz szó a konstruktorok felültöltéséről, most csak annyit jegyzünk meg, hogy a `this()` hivatkozással az osztályban definiált konstruktort érhetjük el.

3.2.5. Kifejezések és operátorok

A kifejezések alapvetően két célt szolgálnak: egyrészt a már ismert értékekből új értékeket állítanak elő, másrészt mellékhatásként változásokat eredményezhetnek. A kifejezések értékének meghatározását kiértékelésnek hívjuk. A kifejezések operandusokból (változók, konstansok), operátorokból (műveleti jelek) és zárójelekből állnak.

Az operátorok elhelyezkedése szerint az egyes műveletek lehetnek:

- prefix – az operátor az operandus előtt áll,
- postfix – az operátor az operandus után áll, vagy
- infix jellegű – az operátor az operandusok között helyezkedik el.

Ezeken kívül a kifejezéseket csoportosíthatjuk operandusainak száma alapján is, így beszélhetünk egyoperandusú (unáris) vagy kétoperandusú (bináris) kifejezésekről. A Java nyelv örökölte a C nyelv egyetlen háromoperandusú műveletét, a feltételes kifejezést is (`?:`).

A Java nyelvben a kiértékelési sorrend egy úgynevezett precedencia táblázat alapján meghatározható. Mivel a nyelv erősen típusos, ezért a kifejezésekhez is egyértelműen rendelhető típus. A típusok közötti átjárást a típuskonverzió teszi lehetővé.

A nyelvben a végrehajtási sorrend (a kifejezés kiértékelése) a műveleti jelek közötti precedencia szabály alapján következik.

Operátor	Megnevezés
`[ ]`	tömb indexelés
`.`	tagelérés pl: `java.lang.Math.PI`
`( )`	zárójeles kifejezés
`kif++ kif--`	postfix operátorok
`++kif --kif +kif -kif`	prefix operátorok
`! ~`	logikai NEM, bitenkénti NEM
`new`	példányosítás
`(típus)kif`	típuskényszerítés
`* / %`	aritmetikai műveletek
`+ -`	
`<<`	eltolás balra, a legkisebb bit 0-t kap
`>>`	eltolás jobbra, és balról a legmagasabb helyiértékű bit értéke kerül be
`>>>`	eltolás jobbra, balról 0 érkezik.
`< > <= >=`	összehasonlítások
`instaceof`	az objektum példánya-e az osztálynak
`== !=`	egyenlőség vizsgálatok
`&`	bitenkénti ÉS
`^`	bitenkénti kizáró VAGY
`|`	bitenkénti VAGY
`&&`	logikai ÉS
`||`	logikai VAGY
`? :`	feltételes kifejezés
`= += -= *= /= %=`	hozzárendelés, értékadás
`>>= <<= >>>=`	bit szintű értékadások
`&= ^= |=`	

3. táblázat: Műveletek precedenciája

3.2.6. Típuskonverzió

A Java fordító a fordítás során minden esetben megvizsgálja, hogy az egyes kifejezések összeegyeztethető típusokból állnak-e, pontosabban automatikus konverzióval módosítva azonos típusú kifejezésekké konvertálhatók-e a vizsgált kifejezések. A nyelv három fajta konverziót nyújt a programozók számára:

- **Automatikus konverzió:** Az automatikus konverzió minden esetben végrehajtásra kerül, amikor hozzárendelés, vagy paraméterátadás történik. Abban az esetben, ha a fogadó oldal tartalmazza az érkező adatot leíró adattartományt, az érkező adatot erre a bővebb adattípusra alakítja a fordító. Ha a konvertálás nem hajtható végre, a fordító hibát jelez. Néhány esetben azonban az automatikus konverzió értékvesztéssel járhat, ezt a fordítóprogram figyelmeztető üzenettel jelzi.

```
byte b=1; short s; int i; long l; float f; double d;
s = b;                  // bővebb értelmezési tartomány
i = s; l = i;           // bővítés, automatikus konverzió
f = l;                  // adatvesztés előfordulhat!
d = f;                  // bővítés, automatikus konverzió
```

  Objektumok referenciái között is létezik automatikus konverzió. Egy adott osztály objektum referenciája felhasználható mindenhol, ahol valamilyen ősosztály típusú azonosítót várunk.

```
Sikidom alakzat = new Teglalap(5,6);
```

- **Explicit konverzió:** Amennyiben az automatikus konverzió nem elégséges, nem egyértelmű, vagy információvesztés léphet fel, akkor lehetőségünk nyílik explicit típuskonverziót alkalmazni.

```
double d = 12.8;
int i = (int)d;         // i értéke 12 lesz!
short s = (short) i;
Őstípus változónév = (Őstípus)leszármazottTípusúObjektum;
```

- **Szövegkonverzió:** Ha egy kifejezésben `String` típusra van szükség, de a kifejezés nem ilyen, akkor a fordító automatikusan szöveges adatokká konvertálja azt. A konverziót az adott objektum `toString()` metódus hívásával éri el. (Mivel a Java nyelvben minden osztály közös őse az `Object` osztály, az abban definiált `toString()` metódus a leszármazottakban elérhető, felüldefiniálható.)

Abban az esetben, ha a kifejezésben nem objektumok, hanem csak elemi típusok szerepelnek, akkor a `java.lang` csomag megfelelő osztályaiban definiált `toString()` műveletekkel történik meg az automatikus konverzió, azaz a kifejezéseket legvégül karakterekké konvertálja, majd a karaktereket összefűzve `String`-ként felhasználja.

```
int a=4;
System.out.println(23);      // eredmény:  23
System.out.println(23+a);    // eredmény: 234
System.out.println((23+a)); // eredmény:  27
```

3.2.7. Hozzáférési kategóriák

Az objektumorientált alapelvek durva megsértése lenne, ha egy osztály adattagjait bármely más osztályból elérhetnénk. Nagy programok esetén pedig tényleg bonyolulttá válna, hogy minden osztályról nyilvántartsuk, mely másik osztályokkal tart fenn adat szintű kapcsolatot (esetleg olyanokkal is, amelyekkel nem is volt a tervező szándékában). Két, vagy három osztály esetén, programozási szempontból természetesen egyszerűbb megoldás lenne az adattagok direkt elérése, de az adatabsztrakciós és az objektumorientált elvek az adatrejtés minden esetben való szigorú használatát várják el a programozótól.

A Java nyelv négy hozzáférési kategóriát definiál. Az adattag-, metódus- és osztályhozzáféréseket a nyelv részét képző módosító kulcsszavakkal használjuk.

- **Nyilvános tagok** (public): Ahhoz, hogy egymástól teljesen független osztályok példányai is ismertségi kapcsolatba kerülhessenek, szükséges a mindenki által hozzáférhető tagok definiálása is. Nyilvános minősítést osztályok és az ezen osztályokon belüli metódusok kaphatnak. (Adattagokat az adatrejtés fent vázolt elmélete szerint nem definiálunk nyilvános minősítéssel, jóllehet megtehetnénk. Egyetlen kivétel van, amikor egy adattagot nyilvánosságra hozunk: a konstans értékek. A Java nyelv a konstans értékeket `static final` minősítéssel defininiálja, ezen adattagok értéke futás során nem változhat. Például a π értékét `java.lang.Math.PI`-vel érhetjük el.)
- **Csomag szintű tagok:** Amennyiben egy osztályt, vagy az osztály egy adattagját, metódusát nem soroljuk egyetlen más kategóriába sem (nem írjuk ki a private, protected, public módosítókat a definiáláskor), akkor az azonos csomagban (azonos könyvtárban található fájlok) elhelyezkedő osztályok ezeket a tagokat szabadon elérhetik. A csomagon

kívül elhelyezkedő osztályokból azonban már elérhetetlenek lesznek. A későbbiekben látni fogjuk, hogy a Java nyelv nagyon sokat használja a csomagokat, és a logikusan felépített osztályhierarchiákat. Egy csomagba általában az egymással szorosan együttműködő osztályok kerülnek, amelyek ezt a hozzáférési kategóriát hatékonyan használják ki.

- **Védett tagok** (protected): Ezt a kategóriát nevezik még a leszármazottakban hozzáférhető tagok csoportjának is. A védett kategória a csomagszintű, félnyilvános kategória kiterjesztése. A védett tagokhoz hozzáférhetnek az azonos csomagban (azonos könyvtár) elhelyezkedő osztályok. Ha egy `A` osztály egy védett tagjához egy másik csomagból származó `B` osztály szeretne hozzáférni, akkor ezt akkor teheti meg, ha az osztályok öröklési kapcsolatban állnak, vagyis a `B` az `A` osztály leszármazottja. Másképp fogalmazva: védett minősítés esetén az egyes tagok az osztályhierarchián belül láthatóak, tehát egy leszármazottból elérhetjük az ős védett adattagjait, még akkor is, ha a leszármazottat egy másik csomagban definiálták. (Egy védett konstruktort egy más csomagba tartozó gyermek csak `super()` hívással hívhat meg!)
- **Privát tagok** (private): Az osztályok minden olyan adattagját és metódusát priváttá tesszük, amelyeket meg szeretnénk védeni más osztályoktól vagy működésük mellékhatásaitól. Az így minősített tagok csak az osztályon belül láthatók, hivatkozhatók. Vagyis ezeket az adattagokat csak ennek az osztálynak a metódusai, illetve konstruktora éri el. A privát metódusok szintén csak az őt definiáló osztály metódusaiból, konstruktorából érhetőek el.

A példányok szintjén a tagelérés a következő szabály szerint zajlik: Egy osztályban definiált metódus elérheti az osztály minden példányának adattagját, még ha az privát minősítést is kapott. Pl. egy példányra meghívott metódus elérheti a paraméterében érkező, ugyanolyan osztályú példány privát adattagjait is, hiszen ekkor nem hagyjuk el az osztály hatáskörét!

3.2.8. Vezérlési szerkezetek

A Java a strukturált programozáshoz hasonlóan az utasításokat sorról-sorra hajtja végre. Azonban az objektumorientált programokat is valahogyan vezérelni kell. Az ismert alapszerkezetek a Java nyelvbe is bekerültek, az alábbiakban ezekkel ismerkedünk meg.

Utasítások

Legelőször az utasításokkal foglalkozunk. Az utasítások alapvetően két csoportba sorolhatók: kifejezés-utasítás, és deklarációs utasítás. Minden utasítást a „**;**” karakterrel zárunk le. Minden utasítás helyén állhat egy blokk, amelyet utasítások sorozataként **{** és **}** jelek közé írva hozunk létre. Egy blokkon belül létrehozhatunk lokális változókat deklarációs utasítással. Minden lokális változó az őt magába foglaló blokk végéig létezik.

Logikai kifejezés

A Java nyelvben a boolean típus `true` és `false` konstanssal reprezentálja a logikai értékeket. A logikai kifejezésről beszélünk akkor, ha a kifejezés logikai értékét a fordító minden esetben egyértelműen meg tudja határozni automatikus konverziókkal. A relációs és a logikai operátorok, illetve zárójelezés segítségével összetett logikai kifejezéseket is felépíthetünk. Számos szerkezetben használunk logikai kifejezéseket.

```
a >= b+1
objektum != null
objektum instanceof Osztaly
```

Feltételes kifejezés

Egy adott elemi változónak, vagy objektumnak sokszor feltehető eldöntendő kérdés, melyre igen/nem választ várunk. A feltételes kifejezés egy, a C nyelvből ismert háromoperandusú kifejezés:

```
feltétel ? kifejezés-ha-igaz : kifejezés-ha-hamis
a > b ? a : b          //feltételes kifejezés
max = (a>=b)? a : b;   //értékadó kifejezésbe ágyazva
```

A feltétel egy logikai kifejezés, amely a feltétel igazsága esetén a `kifejezés-ha-igaz`, különben pedig a `kifejezés-ha-hamis` értéket szolgáltatja a teljes kifejezés értékeként.

Elágazások, szelekciók

Az `if` utasítással a programban két-, vagy többirányú elágazást hozhatunk létre, ahol a megadott feltételtől függően hajtódik végre a program egyik, vagy másik ága. Az if utasítás szerkezete a következő:

```
if (feltétel) utasítás;
```

Az `if` utasítás feltétel kifejezése a fentiekhez hasonló logikai kifejezés. Itt is fordítási időben történik ellenőrzés arra, hogy a feltétel kiértékelhető-e, vagyis a benne szereplő kifejezés automatikus típuskonverzióval `boolean` típussá konvertálható-e. Futás során, ha a feltétel igaz, akkor az utasítást, vagy az utasítás helyén álló blokkot a program végrehajtja. Ezután a program a következő utasítással folytatja futását.

Az `if` utasítás második alakja, amikor a programot feltételtől függően két ágra bontjuk. Ha a feltétel igaz, akkor `utasítás_1`, különben az `esle`-ág, vagyis az `utasítás_2` kerül végrehajtásra.

```
if (feltétel) {
  utasítás_1;
  }
else {
  utasítás_2;
  }
```

Megjegyzés: A Java kódolási konvenciók értelmében (lásd: 206. oldal) az egyértelműség kedvéért a vezérlési szerkezetekben már egyetlen utasítás használata esetén is kerülni kell a blokkzárójelek nélküli formát (a kapcsos zárójeleket ki kell írni)!

Amennyiben a programot úgy szeretnénk vezérelni, hogy ne csak kettő, hanem több irányban legyen elágaztatható, akkor többszörös elágazást használunk. A többszörös elágazás megfogalmazásánál azonban egy-két apróságra oda kell figyelnünk.

```
if (feltétel_1){
  utasítás_1;
  }
else if (feltétel_2) {
  utasítás_2;
  }
else {
  utasítás_3;
  }
```

Egy konkrét példában a vezérlőszerkezet az alábbiak szerint alakul:

```
short a=1;
if (a<0){
  System.out.println("Negatív");
  }
else if (a>0) {
  System.out.println("Pozitív");
  }
else {
  System.out.println("Zérus");
  }
```

A végrehajtás a következő logika szerint működik: Először megvizsgálja a `feltétel_1` kifejezést. Ha ez igaz, akkor az `utasítás_1`-et végrehajtja, majd a vezérlés a következő (az `if` szerkezet utáni) utasításra kerül. Ha `feltétel_1` hamis volt, akkor megvizsgálja `feltétel_2` logikai értékét. Ha ez igaz, akkor az `utasítás_2`, ha hamis, akkor `utasítás_3` kerül végrehajtásra, majd a vezérlés a következő utasításra lép.

A többszörös elágazás egy utasításnak számít, még akkor is, ha definíciója több sorból is áll, vagyis a fordító egy egységként kezeli! A feltételek megfogalmazásakor ügyelni kell arra, hogy azok egymást kizáró eseményeket szimbolizáljanak. Amennyiben olyan ágat definiálunk, ahova a vezérlés sohasem juthat, a fordító hibát jelez. Ha a legutolsó `else` ág hiányzik, és egyik feltétel sem teljesül, akkor az elágazás egyik utasítása sem lesz végrehajtva, a program a következő utasításra lép.

Egy általános, többágú esetszétválasztás (szelekció) modellje az alábbi szerkezettel érhető el:

```
if (feltétel_1) {
  utasítás_1;
  }
else if (feltétel_2) {
  utasítás_2;
  }
else if (feltétel_3) {
  utasítás_3;
  }
…
else {
  utasítás_n;
  }
```

A Java nyelv örökölte a C nyelvből ismert switch utasítást, melyet egy különleges elágazás típusnak, esetszétválasztásnak hívunk.

```
switch (egész-kifejezés) {
  case konstans-kifejezés_1 :
     utasítás_1;
     break;
  case konstans-kifejezés_2 :
  /* továbblépés! */
  case konstans-kifejezés_3 :
     utasítás_2;
     break;
  …
  default:
     utasítás_n;
}
```

Az utasításhoz érve először az ún. egész-kifejezés értékelődik ki. Itt olyan kifejezéseket adhatunk meg, amelyek kiértékelése után megszámlálható típusba tartozó értéket (`char`, `byte`, `short`, `int`) kapunk (azaz lebegőpontos kifejezéseket, karakterláncokat már nem adhatunk meg). A case címkék mellett álló konstans kifejezéseknek az egész kifejezéssel kompatibilis típusúnak kell lennie. Futás közben az egész kifejezés kiértékelése után, ha valamelyik címke értéke megegyezik a kifejezés értékével, akkor az ott definiált utasításokat sorban végrehajtja. Az utasítások ezután sorban végrehajtásra kerülnek, „rácsorognak" a következő ág utasításaira – függetlenül a címkéktől – hacsak egy `break` utasítás a `switch` szerkezet elhagyását ki nem kényszeríti. Ha egyetlen konstans kifejezés sem felel meg az egész kifejezés értékének, akkor a `default` címke utáni utasítások kerülnek végrehajtásra.

Ciklusok

Amennyiben bizonyos utasításokat, vagy utasítás-sorozatokat egymás után többször végre kell hajtani, ciklusokat, iterációkat használunk. A Java nyelv a ciklusszervezési utasításait a C nyelvtől örökölte.

Az utasítások ismétlése általában egy feltétel (kilépési feltétel) teljesülése esetén megszakad. A program a soron következő utasítás végrehajtását kezdi meg. A feltétel nélküli, örökké ismétlődő ciklust végtelen ciklusnak hívjuk. A ciklusokat a feltétel kiértékelése szempontjából két csoportra oszthatjuk: elöl-, illetve hátultesztelő ciklus.

Az elöltesztelő ciklus a ciklusmag lefutása előtt minden esetben kiértékeli a feltételes kifejezést. Mindaddig, amíg ez a feltétel igaz logikai értéket szolgáltat, a ciklusmag újra és újra megismétlődik. Amikor a feltétel már

nem teljesül, a program a ciklust követő utasítással folytatja futását. Előfordulhat az az eset is, hogy a ciklusmag egyszer sem lesz végrehajtva.

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

```
int i=0;
while (i<10){
  System.out.println(i);
  i++;
}
```

Abban az esetben, ha a ciklusmagot legalább egyszer végre kell hajtani, akkor célszerűbb a hátultesztelő ciklust használni. Ekkor a feltétel kiértékelése a ciklusmag első lefutása után értékelődik ki először. Ha a feltétel igaz, akkor a ciklusmag utasításai megismétlődnek. A ciklusból való kilépés itt is a feltétel hamissá válásakor lehetséges.

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

```
int i=0;
do {
  System.out.println(i);
  i++;
} while (i<10);
```

Az elöltesztelő ciklusok egy speciális esetében – amikor megszámolható lépésben kell utasításokat végrehajtanunk – a `while` helyett a `for` ciklusutasítást használjuk. A `for` ciklusban a fulcsszót követő zárójelben, pontosvesszőkkel adjuk meg az inicializáló- és logikai kifejezést, és a léptető utasítást. A szintaktika megengedi, hogy a kezdőérték adást, a feltétel kifejezést, vagy a léptető kifejezést elhagyjuk. Az inicializátorban lehetőségünk van lokális változók deklarálására is. Ezek a változók csak a ciklus utasításblokkján belül értelmezettek, a blokk végén megszűnnek. Használatával világosan jelezhető, hogy ezt a változót csak a ciklusszervezés idejére definiáltuk.

```
for ( <inicializálás> ; <feltételes kif.> ;<léptetés> ) {
  utasítások;
}
```

```
for (int i=0 ; i<10 ; i++){
  System.out.println(i);
}
```

A feltételes kifejezés alapértelmezés szerint mindig igaz értékű, így a teljesen hiányos `for(;;);` utasítás szintaktikailag helyes ugyan, de szematikai szempontból nem, hiszen végtelen ciklust eredményez.

Egy adott blokkból nem csak a feltétel hamissá válásával van lehetőségünk kilépni. A break utasítás – amint a `switch` utasításnál is láthattuk – egy adott blokkból való kilépésre szolgál. Ha a vezérlés egy `switch`, `for`, `while`, vagy `do` utasítás blokkjában egy `break` utasítást talál, akkor az őt tartalmazó blokk utáni utasítással folytatja a végrehajtást.

Amennyiben `címke : utasítás;` alakú ún. utasításcímkét használunk, akkor a `break címke;` alakú hívás hatására a program a címkével jelölt utasítással folytatja a végrehajtást. Azonban metódusból, vagy inicializátor blokkból való kiugrásra a `break` nem használható.

A `continue` utasítás kizárólag egy `while`, `do`, vagy `for` utasításon belül értelmezett. Hatására a ciklusmag hátralévő része átugorható, a ciklus a feltételes kifejezés kiértékelésével folytatja futását.

```
int i=-10;
while (i < 10){
  if (i==0) continue;
  System.out.println(100/i);
  i++;
  }
```

Az utasítás megadható `continue címke;` alakban is, ekkor a `break`-hoz hasonlóan egy címkével jelölt utasítással folytatódik a program futása. Metódusból, vagy inicializátor blokkból való kiugrásra a `continue` sem használható.

Megjegyzés: Amennyiben a struktúrált programozás alapelveivel való kompatibilitásra törekszünk az objektumorientált programokban is, az egyes algoritmusokat és vezérléi szerkezeteket úgy kell felépíteni, hogy azok ugró utasításokat ne használjanak! [Marton02]

A ciklusok, iterációk tetszőleges mélységben egymásba ágyazhatók, azonban egy öt szintnél mélyebb egymásba ágyazásra csak ritkán van szükség. Az ilyen ciklusok csak jó szintaktikus tagolással, illetve megfelelő kommentezéssel láthatóak át.

```
for (int i=2 ; i<100 ; i++){
  System.out.print(i+" osztói: ");
  for (int j=1 ; j<=i ; j++){
     if((i%j)==0) System.out.print(j+ " ");
     }
  System.out.println();
}
```

3.2.9. Tömbök

A hagyományos programozási nyelvekben használt a tömb, mint az azonos típusú elemekből képzett, fix elemszámú összetett adatszerkezet. A tömbelemek indexelhetők. A Java nyelv is definiál tömb adattípust, amelyet az alaptípusokból vagy referenciákból képezhetünk. A tömbök főbb jellemzői:

- Előre megadott, `n` számú elemet tartalmaz,
- az adatok azonos típusba tartoznak,
- az elemeket a `0`. elemtől `n-1`. elemig érhetjük el az indexelés segítségével. Az indexeléshez szögletes zárójelpárt használunk: `[]`.

A tömbök egymásba ágyazhatóak, így többdimenziós tömböket is deklarálhatunk. Azonban a Java a deklarált tömböt még nem engedi felhasználni, az elemek számára a memóriában helyet is kell foglalnunk.

A tömb deklarációja

A tömb a Java rendszerben a referencia típusú változók közé tartozik. (A C és C++ nyelvtől eltérően ez egy valódi dinamikus memóriaterületet használó változó, és nem a mutató típus egy másik alakja!) Egy tömb deklarációt az alábbi két módon tudunk megadni:

```
<elemtípus>[] <tömbazonosító>;
<elemtípus> <tömbazonosító>[];
```

Mindkét jelölésmód megengedett, csak kisebb különbségeket fedezhetünk fel a deklarációs utasításokban:

```
int[] aT, bT;                          // itt aT és bT is tömb
int cT[], d;                 // cT egy tömb, de d elemi adat!
Alkalmazott[] dolgozok;          //referenciatömb deklaráció
```

A tömb deklarálásával megadtuk, hogy az adott referencia (`aT`, `bT`, `cT` és `dolgozok`) egy egyelőre meghatározatlan méretű tömbre fog hivatkozni.

Nagyon fontos észrevennünk, hogy a deklaráció során **nem** adhatjuk meg a tömb méretét, csakis a tömb létrehozásakor (**new** operátorral, vagy az inicializáló blokkal)! Az `"int a[3];"` alakú deklaráció fordítási hibát okoz!

A tömb létrehozása

A tömb deklarációja után – a futás alatt a memóriában – a megadott elemszámra helyet is kell foglalnunk a new operátor segítségével:

```
new <elemtípus> [méret];
```

ahol `méret` egy nemnegatív, maximum `Integer.MAX_VALUE` nagyságú egész érték.

A `new` operátor létrehozza a memóriában a megadott adatstruktúrát és visszadja a tömb referenciáját, amelyet a tömb típusú változó értékül kap. Ezt a referencia értéket értékül adhatjuk egy olyan referencia típusú változónak, amely egy ilyen tömbre képes rámutatni.

Ez a tömb, létrehozása után már „bejárható”, elemei hivatkozhatók, illetve adatokkal feltölthetők. A tömbelemek automatikus kezdőértéket kapnak, amely boolean típus esetén `false`, char esetén `'\u0000'`, egészek esetén `0`, valós típusoknál `0.0`, és referenciatípusok esetén `null` érték.
Az alábbi példákban egyszerű tömb típusú változókat hozunk létre:

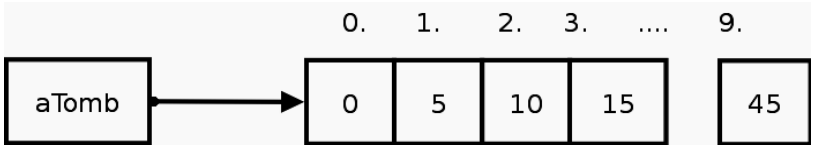
```
int [] iT;
double[] dT;
String [] sT;
…
iT = new int[365];
dT = new double[365/12];     // 30 az egészosztás miatt!
sT = new String[4];
```

A tömblefoglalt méretét futási időben is elérhetjük a következő kifejezéssel: `tömbazonosító.length`. A tömb maximális indexe `length-1` lesz. (Ez a számadat valójában a tömbök Java nyelvbeli definíciójában megadott konstansának értéke.)

A tömb futása során „ismeri” a méretét, amelyet futás során, minden egyes tömbelem-hivatkozáskor ellenőrz is a virtuális gép. Ha ez az index nem érvényes, egy `IndexOutOfBoundsException` típusú kivétel keletke-

zik, és a futás rendszerint megszakad (lásd: Kivételkezelés). Tehát a C nyelvvel ellentétben a tömbön kívüli adatterületre nem hivatkozhatunk!

```
short [] aTomb;         // aTomb tömbreferencia deklaráció
aTomb = new short[10]; // tömb létrehozása, helyfoglalás
…
// tömb bejárása és inicializálása
for(int i=0; i < aTomb.length; i++){
  aTomb[i] = i*5;
  }
```



14. ábra: A tömb elhelyezkedése a memóriában

A tömböket tömörebb formában is létrehozhatjuk. A deklarációs utasítást és a helyfoglalást egy lépésben is megtehetjük:

```
int[] honap = new int[12];
```

Lehetőségünk van a tömböket inicializáló blokkal definiálni. Ilyenkor a deklaráció során automatikusan – az inicializáló blokk kiértékelésével – létrejön a tömb objektum és megtörténik a helyfoglalás. Ezután nem kell és nem is szabad `new` operátort használni! A tömbinicializáló blokkot az alábbi formában adhatjuk meg:

```
<elemtípus> [] <tömbazonosító> = { <érték0>, <érték1>, …};

int[] aTomb = {1, 2, 3, 7, 11};
char[] maganh = { 'a', 'e', 'i', 'o', 'u'};
boolean[] valaszok = { true, true, false};
double[] arak = {1202.1, 33.35};
String[] nevek = {"Eszti", "Reni", "Egó", "Peti", "Sanyi"};
```

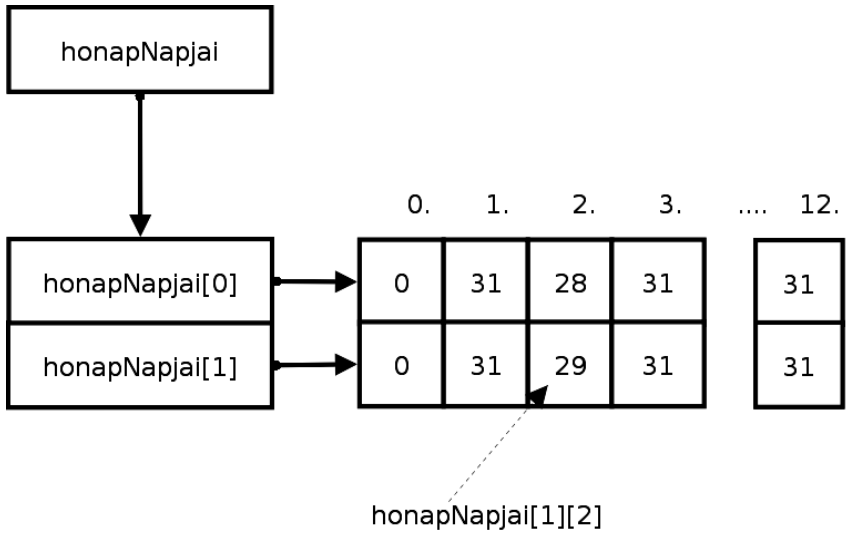
Többdimenziós tömbök

A Java nyelv a C nyelvhez hasonlóan a többdimenziós tömböket tömbökből álló tömbökkel képezi le. A többdimenziós tömböt úgy deklaráljuk, hogy a tömb alaptípusát is tömbként deklaráljuk. A többdimenziós tömb létrehozása hasonló módon működik, mint az egydimenziós esetben.

```
<elemtípus>[][] <tömbazonosító>;

double [][] a;          // 2 dim. tömb deklaráció
a = new double[2][3];   // 2 dim. tömb létrehozása
```

Többdimenziós tömböt inicializáló blokkal is létrehozhatunk. A művelet hasonlít az egydimenziós esethez:

```
<elemtípus> [][] <tömbazonosító> = {
  {<érték0>, <érték1>, …} , {<érték0>, <érték1>, …}, …
  };

int[][] honapNapjai = {
  {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
  {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
  };
```



15. ábra: A kétdimenziós tömb memóriamodellje

A tömbök tárolási módszere egy kicsit összetettebb, mint az egydimenziós esetben. A tömb létrehozása során a tömbazonosító referenciája egy olyan tömböt ér el, amely tömbökre mutató referenciákból áll. A fenti definíció alapján a honapNapjai referencia egy kételemű tömbre mutat. Itt a `honapNapjai[0]` és a `honapNapjai[1]` egy-egy 13 elemű egydimenziós tömbre mutató referencia. (Így ezek a „belső” tömbök akár értékül adhatóak egy `int[]` – típusú egydimenziós tömbnek is.)

A tárolt tömbelemeket egyszerűen elérhetjük a `honapNapjai[x][y]` alakú hivatkozással. Itt valójában a referenciákon keresztül végül elérünk az `int` típusú tárolt adatokhoz. (A módszer ismerős lehet a C nyelv mutató aritmetikája alapján.)

A többdimenziós tömböket több lépésben is létrehozhatjuk. Először egy olyan tömböt hozunk létre, amely majd tartalmazza a többi tömbre mutató referenciát. Ezután az egyes referenciákhoz lefoglalunk egy-egy tömböt. Ezek az újabb tömbök akár különböző méretűek is lehetnek, így a modellezendő problémához legjobban illeszkedő struktúrát is elkészíthetünk!

```
int[][] alsoMatrix = new int[3][];

for (int i=0; i < alsoMatrix.length; i++ ){
  alsoMatrix[i] = new int[i+1];
  for (int j=0; j < alsoMatrix[i].length; j++ ){
     alsoMatrix[i][j] = 1;
}
```

Ebben a példában egy alsó háromszögmátrix elemeit tároló kétdimenziós tömböt készítettünk el. Az első sorban deklaráltunk és lefoglaltunk egy háromelemű tömböket tartalmazó tömböt. Ezután egy ciklus segítségével a három tömbelemhez hozzárendeltünk egy-egy egydimenziós tömböt, amelyek elemszáma 1, 2 illetve 3 volt. Legvégül egy belső ciklussal ezeket a tömbelemeket inicializáltuk 1 kezdőértékkel.

Értékadás tömbök között

Tömbök között akkor beszélhetünk értékadásról, ha típusaik kompatibilisek. Adott `t1` tömb értékadás szerint kompatibilis `t2` tömbbel a következő esetekben:

- Elemi adattípus esetén, ha `t1` és `t2` elemtípusa azonos;
- Referencia típusok esetén, ha `t2` elemtípusa azonos `t1` elemtípusával (osztályával), vagy annak egy leszármazottjával.

A tömbök értékadásakor a tömbreferenciához új értéket rendelünk. Értékadáskor a tömbelemek nem másolódnak, így az értékadás érvényesnek számít akkor is, ha a két tömb nem azonos elemszámú.

```
int [] t1 = new int[12];
int [] t2 = new int[100];
…
t1 = t2;
```

A `t1=t2` utasítás hatására a `t1` tömbreferenciához hozzárendeltük a `t2` referenciáját. Ezután tehát mindkét azonosítóval ugyanazt a 100 elemű tömböt érjük el. Az eredetileg `t1`-hez rendelt 12 elemű tömb már nem lesz soha többé elérhető (hacsak nem jegyeztük fel a referenciáját valahol). A tömbök megszüntetéséért a virtuális gép szemétgyűjtő mechanizmusa felel (lásd: 80. oldal).

3.2.10. Karakterek és szöveges adatok

A Java nyelv char típusa egyetlen, kétbájtos unicode karakter ábrázolására alkalmas alaptípus. A unicode szabvány 65536-féle különböző írásjelet különböztet meg, ezért használhatjuk az azonosítókban, illetve a változókban az ékezetes, római és egyéb karaktereket. Egy karaktert aposztrófok között helyezünk el: `'a'`. A char egész jellegű megszámlálható típus, így könnyen konvertálható egész értékké és vissza.

Az alábbi táblázatban láthatjuk azokat a karakter konstansokat, melyeknek a Java nyelvben különleges jelentése van:

karakterkód	jelentés
\n	soremelés
\r	kocsivissza
\t	vízszintes tab karakter
\f	lapdobás
\\	a \ karakter
\”	idézőjel
\’	aposztróf
\b	backspace
\000	karakter oktálisan (0-377)
\u0000	unicode karakter hexadecimálisan (\u0000 – \uffff)

4. táblázat: A karakteres escape szekvenciák

A karakteres értékekhez hasonlóan ún. szövegliterálokat is megadhatunk idézőjelek között: `"Java programozás"`. Ezek tulajdonképpen konstans karakterláncok, és korlátozott a használhatóságuk.

A szövegek, illetve karakterláncok kezelésére a Java nyelv nem ad meg alaptípust, erre a Java környezetben definiált `String` osztályt használjuk.

A String osztály

A `String` osztály konstans karakterláncok tárolására alkalmas. Az osztály a karakterláncokat karakteres tömbben tárolja két számértékkel (a sztring első karakterének pozícióját, és a szöveg hosszát), és definiálja a hozzá tartozó alapműveleteket (hossz, kis-nagybetűs konverzió, összefűzés, összehasonlítás, részsztring képzés, karakterek keresése stb.). A létrehozás után azonban a `String` típusú objektumot módosítani már nem tudjuk. Az ilyen típusú változót leggyakrabban az alábbi utasításokkal hozunk létre (példák):

```
String szov1 = new String("első szöveg");
String szov2 = "második szöveg";
String szov3 = "Eredmény ="+3 ;
```

A három megadási mód egyenértékű, a második és harmadik esetben nem írtuk ki a `new` kulcsszót, itt automatikusan történik meg a példányosítás (a fordító automatikusan kiegészíti a `new`-val).

A harmadik esetben a 3-as egész értéket a fordító az automatikus konverzió segítségével szöveggé alakítja, majd ezt a szöveget hozzáfűzi az előtte álló karakterlánchoz. Végül a kapott új karakterláncra mutató referenciát kapja meg a `szov3` referencia.

Megjegyzés: A számértékek konverziójakor a virtuális gép a számérték csomagoló osztályának `toString()` metódushívásával alakítja szöveggé az adott értéket.

A sztringek automatikus konverziója és a „+” operátorral történő összefűzése erőforrás-igényes művelet, helyette a `StringBuffer` osztály használata ajánlott!

Sztringet karaktertömbökből, bájttömbökből, vagy annak egy részéből is képezhetünk.

```
char [] karaktertomb = {'P', 'i', 's', 't', 'i', 'k', 'e'};
byte [] bajttomb = {65, 66, 67, 68, 69, 70};
String s1 = new String(karaktertomb);              // Pistike
String s2 = new String(bajttomb);                   // ABCDEF
String s3 = new String(karaktertomb, 1, 4);           // isti
String s4 = new String(bajttomb, 3, 2);                // EF
```

A sztringek kezelése során a pozícionálás 0-tól a szöveghossz-1 értékig terjedhet. A szöveget nem indexelhetjük túl (a tömbökhöz hasonlóan) különben a `StringIndexOutOfBoundsException` kivétel keletkezik.

Sztringekkel az alábbi műveleteket hajthatjuk végre:

- Sztring előállítása karakterlánc-literálból, karakter-, vagy bájttömbből.
- Megkaphatjuk a szöveg hosszát (`length`), megkereshetjük egy sztring adott indexű karakterét (`charAt`), rész-szövegét (`subString`), és a szöveg elejéről, végéről levághatjuk a fehér karaktereket (`trim`).
- Egy sztringet összehasonlíthatunk egy másik sztringgel referencia (`equals`) és karakter szinten (`compareTo`), illetve megállapítható, hogy egy sztring tartalmaz-e egy másikat (`startsWith`, `endsWith`, `regionMatches`).
- A sztringhez hozzáfűzhetünk egy másik sztringet (`concat`, vagy +), a szöveg karaktereit kis- és nagybetűssé alakíthatjuk (`toLowerCase`, `toUpperCase`),
- A szöveg egyes karaktereit kicserélhetjük (`replace`)
- A szövegben karaktereket és rész-sztringeket kereshetünk (`indexOf`)

A sztringekből álló tömbök létrehozására is lehetőségünk nyílik. Legegyszerűbb esetben a tömböt sztring literálokkal inicializálhatjuk. Ekkor a sztringtömb változó egy olyan tömb típusú referencia, amely `String` objektumokra mutató referenciákat tartalmaz.

```
String [] honapok = {"január", "február", "március"};
```

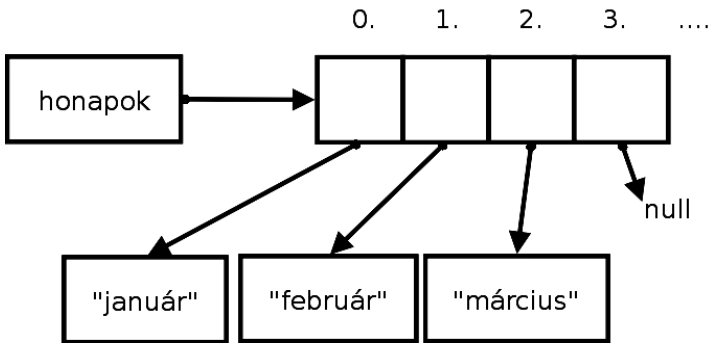
Az így definiált tömb elemeit azután elérhetjük a `honapok[index]` alakú hivatkozásokkal, illetve a tömbelemre, mint `String` objektumra meghívhatjuk a sztringkezelő függvényeket is:

```
System.out.println(honapok[1].charAt(0));      //kimenet: "f"
```

A fenti definíciót természuetesen megadhatjuk a new operátor segítségével is úgy, hogy az egyes tömbelemekhez futás közben rendelünk értéket.

```
String[] honapok = new String[12];
honapok[0] = "január";
honapok[1] = "február";
honapok[2] = "március";
```

Megjegyzés: Az indexeléssel vigyáznunk kell, mert ha olyan tömbelemre próbálunk hivatkozni, amelyikhez még nem rendeltünk egyetlen sztring értéket se (pl: `honapok[3].charAt(0)`), hiszen akkor a futás közben egy `NullPointerException` kivétel keletkezik!



16. ábra: Tömbök képzése karakterláncokból

Ilyen karakterekből álló tömb a `main` osztálymetódusban paraméterként szereplő `String args[]` is. Segítségével a program indításakor megadott parancssori paramétereket érhetjük el szöveges adatként. (Figyelem, az `args[0]` már az első valódi paraméter értékét veszi fel!)

```
/** ArgumentumKiiro.java
* hívása: java ArgumentumKiiro param1 param2 param3 ....
*/
public class ArgumentumKiiro {
  public static void main(String[] args) {
  if (args.length > 0) { // paraméterlista-tömb elemszáma
    for (int i = 0; i < args.length; i++)
      System.out.println(args[i]);
  } else {
    System.err.println("Nincsenek paraméterek!");
  }
 }
}
```

A tömbkezelés további lehetőségeiről a 8.5. fejezetben ejtünk még néhány szót.

A StringBuffer osztály

Mint láttuk, a `String` osztály konstans karakterláncokat kezel. A virtuális gép minden karakterlánc-összefűzéskor egy-egy újabb memóriaterületet foglal az újonnan előálló szöveges adat számára. Az ilyen műveletek nagyon memóriapazarlóak (a virtuális gép a lefoglalt és nem használt memóriaterületeket csak a szemétgyűjtéssel tudja ismét használhatóvá tenni) ezért használatuk kerülendő!

Abban az esetben, ha a futás alatt több szrtingből álló szöveges adatot szeretnénk előállítani, amelynek részeit, hosszát stb. más-más időpontban kapjuk meg, akkor az ilyen szövegműveletekhez a `StringBuffer` osztályt kell használnunk. A `StringBuffer` osztály a String osztályhoz hasonlóan a Java környezet része. A `String` osztályban ismertetett alapműveleteken kívül a következő műveletek elvégzésére használjuk:

- Egy `String` objektumot `StringBuffer`-ré alakíthatunk, és vissza.
- A szöveges adathoz hozzáfűzhetünk, vagy az adatba egy másik szöveget beszúrhatunk (append, insert).
- A szöveg hossza módosítható (rövidítés, bővítés). A `StringBuffer` objektum rendelkezik egy `capacity` adattaggal, amely az objektum tényleges memóriafoglalását mutatja. A szöveg aktuális hosszát a `length()` metódussal kaphatjuk meg. Amennyiben a hozzáfűzés, vagy beszúrás során a szöveg nem férne be az adott memóriaterületre, a `StringBuffer` méretét a virtuális gép megnöveli.
- A szöveg megfordítható (reverse).
- A szövegből karakterek, vagy rész-szövegek törölhetők.

A Java osztálykönyvtárakban gyakran találkozunk `StringBuffer` típusú referenciával visszatérő metódusokkal.

Az alábbi példaprogram-részletben a `StringBuffer` osztály működését tanulmányozhatjuk (a teljesség igénye nélkül):

```
//Konstrukció
StringBuffer szoveg1 = new StringBuffer(2);
//Konstrukció egy már definiált sztring segítségével
StringBuffer szoveg2 = new StringBuffer("Almafa");
//Automatikus tárbővítés:
szoveg1.append("Nagyon hosszú szöveg! …");
//A length() a tartalmazott szöveg hosszát, a capacity() a
//StringBuffer objektumban rendelkezésre álló helyet jelenti
int hossz = szoveg1.length();
int kapacitas = szoveg1.capacity();
```

```
System.out.println(szoveg1);
System.out.println("szoveg1 hossza = " + hossz);
System.out.println("szoveg1 kapacitása = " + kapacitas);
System.out.println("--------------------------------\n");

//A StringBuffer objektum kibővítése
szoveg1.ensureCapacity(20);
szoveg2.ensureCapacity(50);

System.out.println(szoveg2);
//Elemi értékek hozzáfűzése és beszúrása a StringBuffer
//objektumba
double d = 3.54;
int i = 100;
char [] karaktertomb = {'P', 'i', 's', 't', 'i', 'k', 'e'};
szoveg2.append(d).append(i);
szoveg2.insert(0, karaktertomb);

// A StringBuffer objektum módosítható:
char regi = szoveg2.charAt(7);
szoveg2.setCharAt(7, 'a');
// A StringBuffer objektum tartalmának rövidítése:
szoveg2.setLength(2);
// A StringBuffer objektum kapacitásának csökkentése:
szoveg2.trimToSize();
System.out.println("szoveg2  = " + szoveg2);
System.out.println("szoveg2 hossza = " + szoveg2.length());
System.out.println("szoveg2 kapacitása = " +
szoveg2.capacity());
```

3.2.11. A csomagoló osztályok

Minden elemi típushoz léteznek ún. csomagoló vagy burkoló osztályok (wrapper). Ezek olyan osztálydefiníciók (`Boolean`, `Character`, `Integer`, `Long`, `Float`, `Double`), amelyek egy adott értéket egy objektumban helyeznek el, rájuk referenciákkal hivatkozhatunk, és az osztályban definiált műveleteket végrehajthatjuk. Összetett adatszerkezetekben hatékonyan használhatóak, ahol az egyes elemeket referenciával kell elérnünk. Így az olyan konténer objektumok, amelyek csak objektumokat tárolhatnak, alkalmasak elemi adatok közvetett tárolására is. Megjegyezzük, hogy az ilyen jellegű adattárolás nagyon memóriapazarló.

```
Float f = new Float(3.14);
```

Az `f` referenciára ezután értelmezhető számos művelet, amely a tárolt értéket megvizsgálja, konvertálja. Ugyanígy használható a többi csomagoló osztály is, az adott típusra jellemző műveletekkel. A csomagoló osztályok használata előnyös lehet a már említett konténer osztályokban, vagy min-

den olyan helyen, ahol az adott értéket objektumként szeretnénk felhasználni, esetleg konvertálni.

3.2.12. A felsorolt típus (enum)

A Java 5.0-ás kiadásában bevezették a felsorolt típusokat [12]. Ezek a felsorolt típusok (enumerációk) olyan jellegű adatszerkezetek, amelyeket már az ANSI C nyelv is használt ugyanilyen névvel. A működési alapelve hasonló, mégis a megvalósításban számos különbséget találunk. Először azonban tekintsük át, hogy mikor használhatunk felsorolásokat!

Hagyományos megoldások

A régebbi Java verziókban felsorolt szerkezeteket használtunk például egy munkafolyamat állapotainak követésére. Ekkor az egyes státuszokat egy-egy konstans értékkel lehetett definiálni:

```
public class Kapcsolat{
  public static final int RENDBEN = 0;
  public static final int HIBA= 1;
  public static final int KAPCSOLODAS = 2;
  public static final int ADATFELOLTES = 3;
…
}
```

A konstansok megadhatók a használni kívánt osztályban osztályváltozóként (lásd: 4.5. fejezet), vagy egy független interfészben is (lásd: 5. fejezet). Ennek a megadásnak az előnye az, hogy a konstansok csak 4-4 bájtot foglalnak (`int` típus) és értékük még szelekcióban (`switch`) is felhasználhatóak.

Azonban ennek a megoldásnak számos hátránya is van: a fordítás során a konstans értékek már behelyettesítve jelennek meg a kódban, nincs lehetőség típusellenőrzésre, nyomkövetéskor, vagy az értéket kiíratva nem szolgáltat informatív eredményt, és fejlett adatszerkezetekbe sem helyezhető el egyszerűen (hiszen ott csak referenciákkal dolgozunk).

Megadható a konstans szöveges értékként is. Ekkor ugyan javul az olvashatóság, de megnő az adatterület, és a vezérlés bonyodalmasabbá válik a sztringműveletektől.

Második lehetőségként lehetőségünk van egy olyan osztály definiálására is, amely az előzőeken túlmutatva csak osztályszintű szöveges konstansokat tartalmaz, és számos műveletet megvalósít, de nem rendelkezik publikus konstanssal.

```
public class Kapcsolat{
  private String nev;
  private Kapcsolat(String nev){ this.nev = nev;}
  public String toString(){return nev;}
  //statikus példány konstrukció az osztály betöltésekor
  public static final Kapcsolat RENDBEN
     = new Kapcsolat("rendben");
  public static final Kapcsolat HIBA
     = new Kapcsolat("hiba");
  public static final Kapcsolat KAPCSOLODAS
     = new Kapcsolat("kapcsolódás");
  public static final Kapcsolat ADATFELOLTES
     = new Kapcsolat("adatfeltöltés");
}
```

Megjegyzés: A konstruktor azért kap privát minősítést, hogy csak az osztálybetöltés során jöjjenek létre a konstans objektumok (static final minősítés), és a későbbiekben ne lehessen több ilyen objektumot létrehozni!

Az így megadott konstansok egyszerűen felhasználhatók más osztályokból:

```
System.out.println(Kapcsolat.HIBA);
```

Az egyetlen hátránya ennek a megoldásnak az, hogy több erőforrást köt le, és egy kicsit pazarló a tárgazdálkodása. Ezek a típusok nagyon sok esetben használhatók, de mivel referencia típusról van szó, egy `switch`-ben nem használhatjuk fel, hiszen ott csak egész-kifejezések állhatnak.

Enum típus definiálása

A Java fejlesztői a fenti megoldások ötvözéseként a felsorolás jellegű adatszerkezetekhez megalkották az `enum` típust, vagy más szóhasználatban az `enum` osztályt, hiszen a típust valójában a `java.lang.Enum` osztályban definiálták. Ez az osztály nagyban eltér az eddig megismertektől, mint azt látni fogjuk az alábbiakban. A típusban felsorolt értékeket enum konstansnak hívjuk, hiszen azok a futás során nem változtathatják meg értéküket.

Ugyanígy az enum típusú változó definiálása egyszerűsített, eltér a hagyományos példányosítástól, és elmarad a `new` kulcsszó is. A definíció szintaktikájában leginkább az ANSI C-vel való analógia fedezhető fel. Enum típust az alábbi módon definiálunk:

```
enum <enumtípusnév> {<enumkonstans>, [<enumkonstans>, […]]}
```

```
public class FelsorolasMinta{
 public enum Kapcsolat {RENDBEN, HIBA, KAPCSOLODAS,
ADATFELTOLTES};
 public enum Napok {HE, KE, SZE, CSU, PE, SZO, VAS};
…
}
```

Ahol az enum típusnév a `Kapcsolat`, a {}-ben levő felsorolás pedig az enumkonstansokat jelöli. Az enumkonstansokat jellegük miatt csupa nagybetűvel írjuk, a Java névadási konvenciói szerint.

Jellemzői:

- Az enum típusok önálló osztályok, melyek a `java.lang.Enum` osztályból származnak, tehát az osztályokra és interfészekre vonatkozó szabályok alapján működnek.
- Enum típust csak osztály, vagy példányváltozóként definiálhatunk, blokkon belüli lokális változóként nem!
- Az enum típusok nem rendelkeznek publikus konstruktorral, ezért csak a deklarációkor kaphatnak kezdőértéket.
- Az enum típusok egyes konkrét értékei `static final` módosítóval rendelkeznek, tehát futási időben nem változtathatják meg értékeiket.
- Az enum konstansok konkrét értékekkel való összehasonlításakor a == művelet használható, és nem szükséges az `equals()` metódust hívni.
- Az enum típusok implementálják a `Comparable` és a `Serializable` interfészt, amely nagyban megkönnyíti a típus felhasználhatóságát kollekciókban (lásd: 9.1. fejezet).
- Az enum típusokban használt `toString()` metódus az enum konstans címkéjének karaktereit szolgáltatja (`Kapcsolat.RENDBEN.toString() => "RENDBEN"`)
- Az enum típusokban használt `valueOf()` metódus a `toString()` ellentéte, a szöveges reprezentációból konstans értéket szolgáltat (`Kapcsolat.valueOf("HIBA") => Kapcsolat.HIBA`)
- Az enum típus értékeit tömbbe konvertálhatjuk a `values()` metódussal.
- A `java.lang.Enum` osztályt csak a fenti módon lehet példányosítani, a `new` operátor itt nem alkalmazható, illetve leszármaztatni sem lehet (final osztályminősítés).
- Az enum konstans értékek nem egyszerűen egész konstansok, hanem a felsorolás osztály egyes statikus példányai. A fordító érzékeny is ezekre a típusokra, két eltérő enum típus nem hasonlítható össze.

Az enum típus felhasználása

Az enum típusként definiált konstansokra a következő alakban hivatkozunk:

```
Enumtípusnév.ENUMKONSTANSNÉV.
```

Az enum típusból hozhatunk létre változókat, amelyek értékként az enum típus egy-egy konstans értékét vehetik fel. Az enumkonstans szerepelhet értékadás jobb oldalán, műveletekben és metódusok paramétereiben, vagy visszatérési értékként.

```
//enum típusú lokális változó definiálása
Kapcsolat statusz = Kapcsolat.HIBA;
// Ez a változó csak enumtípus konstans értékeit veheti fel
statusz = Kapcsolat.JO;//OK.
statusz = "HIBA";    // fordítási hiba, nem kompatibilis
típusok!
statusz = Napok.SZE; //fordítási hiba, nem kompatibilis
típusok!
// enum változó műveletben:
if (statusz == Kapcsolat.JO)
    System.out.println ("A kapcsolat minősége jó!");
```

Az enum típusú változók felhasználhatóak `switch` szelekcióban is. Az elágazás szerkezete nagyon egyszerű, hiszen a `switch`-ben elég a megadott enumtípus konstans értékeit kell megadni és megvizsgálni.

(Észrevehetjük, hogy a kódolási szintaktika kiegyszerűsödik, hiszen a case ágakban nem kell megadni a típust, mert az a vizsgált változó típusából egyértelműen következik.)

```
// az enum típusú változó felhasználása szelekcióban
switch (statusz){
  case JO:
    System.out.println ("A kapcsolat felépült.");
    break;
  case HIBA:
    System.out.println ("A kapcsolat nem épült fel!");
    break;
  case SZAKADT:
    System.out.println ("A kapcsolat megszakadt!");
    break;
  }
```

Megjegyzés: A Java 6. verziótól kezdődően a fordító szigorúbban kezeli a felsorolásokat. Amennyiben az `enum` típus `switch` szerkezetben szerepel, és nem fejtünk ki minden lehetséges case ágat, illetve elhagyjuk a `default` címkét, a fordító figyelmeztető üzenetet küld. [13]

Az enum típusok kiegészítése, kiterjesztése

Ahogy a fentiekből láttuk, a felsorolt típusok tulajdonképpen egyszerűsített osztálydefiníciók, melyeket egyszerűsített szintaktikával hozhatunk létre és rugalmasan kezelhetünk. Ezeken túl számos egyéb lehetőségünk van a felsorolt típusok kiegészítésére: pl. az egyes enumkonstans azonosítókhoz hozzárendelhetünk tényleges konstans értékeket, illetve az enum típusba metódusokat is építhetünk.
Abban az esetben, ha ilyen kiterjesztett felsorolt típust szeretnénk használni, akkor azt, mint egy enum típusú osztályt kell definiálni az alábbi módon:

```
public class EnumTeszt{

  public enum Honapok {JAN (31), FEB (28), MAR (31);
    private int nap;
    private Honapok(int nap){this.nap=nap;}
    public int napszam(){return nap;}
  };
```

Megjegyzés: A konstans címkék mögött a konstruktornak szánt paraméter áll zárójelben, és a konstansok felsorolása után pontosvessző áll. Ezután következik a privát konstruktor, és a metódusok. A konstruktornak azért kell privát minősítést adni, mert az `enum` típus csak a {}-jelek közötti részben inicializálható, futás közben új értékkel kibővíteni enm lehet.

A kiterjesztett enum típust például az alábbi módon használhatjuk fel:

```
  public muvelet(){
    Honapok elsoHonap=Honapok.JAN;     // kiterjesztett enum
    System.out.println(elsoHonap);           // kimenet: JAN
    System.out.println(elsoHonap.napszam());   //kimenet: 31
  }
}
```

A kiterjesztett enum típusú változól kiírásakor az automatikus konverzió ugyanúgy működik, mint más objektumok esetén. (Bővebben lásd: 3.2.6. fejezet.)

Az enum típusok egyes elemeihez önálló viselkedést is rendelhetünk. Ekkor a felsorolás egyes elemeihez saját, felültöltött metódusokat is rendelhetünk. (A megvalósítást a Java névtelen belső osztályokkal végzi. Lásd: 5.4.3. fejezet.)

```
public enum Honapok {
     JAN (31) {
      public String muvelet(){ return "esemény januárban";}
     },
     FEB(28) {
      public String muvelet(){ return "esemény februárban";}
     },
     MAR(31){
      public String muvelet(){ return "esemény márciusban";}
     };
   public int nap;
   private Honapok(int nap){this.nap=nap;}
   public int napszam(){return nap;}
   public abstract String muvelet();
   };
```

A kiterjesztett enum típusú változót, és a hozzá kapcsolódó metódust a következőképpen használhatjuk fel:

```
Honapok aktualisHonap=Honapok.FEB;
System.out.println(aktualisHonap.muvelet());
```

3.3. Kérdések

- Melyek a Java nyelv alaptípusai?
- Mit jelent a referencia típus?
- Hogyan definiálhatunk egy osztályt?
- Miről ismerjük fel egy osztály konstruktorát?
- Sorolja fel a Java nyelv négy hozzáférési kategóriáját!
- Mit jelent az automatikus típuskonverzió?
- Mit jelent az explicit konverzió?
- Mit jelent az automatikus szövegkonverzió?
- Milyen vezérlési szerkezeteket ismer a Java?
- Mi a különbség az elágazás (`if`) és a szelekció(`switch`) között?
- Milyen ciklusfajtákat definiál a Java?

- Hogyan adhatunk meg egy egydimenziós tömb adatszerkezetet?
- Miért kell helyet foglalni egy tömbnek?
- Miért nem helyes az `int a[4];` utasítás?
- Hogyan értelmezzük a többdimenziós tömböket?
- Mi a különbség a `String` és a `StringBuffer` osztály között?
- Mik azok a csomagoló osztályok?
- Mire használhatjuk a felsorolt típusokat?

3.4. Feladatok

1. Készítsen olyan példaprogramot, melyben egy, illetve kétdimenziós tömbben tárol egész, ill. valós értékeket. A tárolt értékeket járja be és jelenítse meg!
2. Állítsa elő egy `n x m`-es mátrix transzponáltját!
3. Gyakorolja a karakterláncok kezelését az alábbi példák alapján:
   - Felhasználói adatokat egy sorban adunk meg, vesszőkkel elválasztva: (név, lakcím, irányítószám, telefon). A karakterláncból nyerje ki az egyes részadatokat.
   - Egy mondat, vagy hosszabb szövegrész szavainak (fehér karakterek határolta egységek) sorrendjét fordítsa meg!
   - Egy tetszőleges osztály `toString()` metódusát írja meg `StringBuffer` segítségével. (Azaz a metódus ne tartalmazzon `String` objektumokat, és karakterlánc összefűzést "+" sem!)
4. Csomagoló osztályok felhasználásával konvertáljon karakterláncban megadott értékeket egész, illetve valós számokká!
5. Tervezzen egyszerű, illetve kiterjesztett felsorolt típusokat a következő fogalmakhoz:
   - hét napjai,
   - közlekedési eszközök,
   - sportágak.

4. Az osztályok használata

A 3.2.2. fejezetben már megismerkedtünk az osztályok néhány alaptulajdonságával, most pedig egy átfogóbb képet rajzolunk, hogy hogyan is épül fel egy Java program, és hogy milyen eszköztárat ad kezünkbe a Java API (a Java osztálykönyvtárak gyűjteménye).

4.1. Az osztályok, és az objektumok életciklusa

A Java programok legkisebb önálló egységei az osztályok. Egy-egy osztály a valóság egy jól körülhatárolható egységét írja le. Ezek, ahogy azt az objektumorientált modellezés elméleti részében kifejtettük, a konkrét objektumok leírásának absztrakt modelljei. Például lehetnek az osztályok egy vállalatnál dolgozó személyek nyilvántartására szolgáló modellek, vagy éppen egy kétirányú dinamikus listát leíró adathalmaz és szabályrendszer.
Működése során a program példányosítja az osztályokat, vagyis a modellek sémája szerint konkrét objektumokat (példányokat) hoz létre. Minden objektum rendelkezik egy belső állapottal, amely egyrészt befolyásolja a műveletek végrehajtását az objektumon, másrészt a műveletek hatására meg is változhat.

A példányosítással létrejövő objektum belső állapota a kezdőállapot. A műveletek végrehajtását az egyes objektumok kezdeményezhetik önmagukra, vagy más objektumokra.

Megjegyzés: Az objektum, példány vagy egyed kifejezés az objektumorientált terminológiában ugyanazt a fogalmat takarja, az egyes szakirodalmak ezeket felváltva használhatják.

Az osztályokat (class) a Java nyelvben egy logikailag két részre bontható definíció írja le. Az első rész deklarálja azokat a változókat, (adattagok, tulajdonságok), amelyekkel egy objektum állapota leírható, jellemezhető, és amelyeknek konkrét értékeiben az azonos típusú objektumok különbözhetnek egymástól. Az osztály minden példánya saját memóriaterülettel rendelkezik ezekből a változókból. Szokásos még példányváltozóknak is hívni őket.

A második rész az objektumok viselkedését, működését és az objektumnak küldhető üzenetek feldolgozását végző műveleteket (metódusok,

tagfüggvények) tartalmazza. Egy metódus hasonlít a hagyományos programozási nyelvek eljárásaira és függvényeire. Más szóval egy metódus egy olyan utasítássorozat, amely paramétereket fogadhat, az objektum adattagjain műveleteket hajthat végre, és értéket adhat vissza.

A metódushívás nem más, mint az adott objektum felé irányuló üzenetküldés. A metódus definíciója osztály szintű, hiszen minden egyes objektumon ugyanazt az utasítássorozatot hajtja végre, de a metódus futásakor mindig egy konkrét objektum adattagjaival dolgozik.

Az objektumorientált módszertan a típusokat műveleteikkel együtt tekinti értelmes egységnek (egységbe zártság). Ebből a szempontból egy osztály nem más, mint egy teljes típusdefiníció, amely lehetővé teszi az adatok és az adatokat kezelő műveletek együttes kezelését. A 3.2.7. fejezetben tárgyalt hozzáférési kategóriák alkalmazásával megvalósítható az adatok külvilág elől történő elrejtése is.

Az alábbiakban egy egyszerű osztályt definiálunk. A LinearisFuggveny osztály az elemi matematikából megismert egyszerű függvényt ír le.

```
// A lineáris függvény osztály definíciója
public class LinearisFuggveny {

  //saját adattagok, tulajdonságok
  private double a,b;

  //konstruktor, amellyel majd példányosítunk
  public LinearisFüggveny(double a_, double b_) {
     a=a_; b=b_;
     }
  // metódusok
  // a lineáris függvény -> y értéke x helyen, azaz ax+b
  public double y(double x) {
     return a*x+b;
     }
  // egy újabb metódus, amelyik szövegesen is megjeleníti
  // a függvény jellemzőit
  public String toString() {
      return "Lineáris függvény y="+a+"*x+"+b;
  }
}
```

Ez a példa egy önálló osztálydefiníció, és teljes fordítási egységet képez. Ezt az osztályt csak a `LinearisFuggveny.java` forrásfájlban lehet elhe-

lyezni, a Java névadási konvenciói miatt. Ezután az osztály már lefordítható, és sikeres fordítás után egy `LinearisFuggveny.class` tárgykódú állományt kapunk.

Az osztály definícióját a `class` kulcsszó vezeti be, amelyet az osztály neve követ. Az osztályokat **nagybetűvel** kezdődő azonosítóval látjuk el. Összetett osztálynév esetén nem használhatunk szóközöket, a szavakat egybeírjuk, és minden szót nagybetűvel kezdünk, esetlegesen a "_" karakter is használható. A névadásnál célszerű főneveket választani.

Az osztály hozzáférési kategóriába is sorolható, aszerint, hogy honnan engedélyezzük majd ennek az osztálynak a példányosítását. Ezt a `class` kulcsszó elé írt módosító szóval tehetjük meg. Ez a módosító a vele ellátott egység egészét ruházza fel a hozzáférési kategória tulajdonságaival.
Egy osztály definiálásakor a `public`, `final`, vagy `abstract` módosítót használhatjuk.

A `public` módosító azt jelenti, hogy az osztályt a nyilvános hozzáférési kategóriába soroljuk, tehát ez bármely más osztályokból hivatkozható, és példányosítható. Ha az osztály elé nem írjuk ki a `public` módosítót, akkor azt az osztályt csak a saját csomagján belüli (azonos könyvtárban lévő) osztályokból érhető el. Az `abstract` és `final` módosítókról az öröklődés fejezetben lesz szó. Az osztály neve után található kapcsos zárójelek között az osztály tényleges definíciója áll.

Az osztály definíciója során először az adattagokat, majd a metódusokat adjuk meg. Példánkban két privát változót `a, b` és három metódust (`LinearisFuggveny, y, toString`) definiáltunk.

A változókat **kisbetűvel** kezdődő főnévvel, a metódusokat **kisbetűvel** kezdődő igével szokás azonosítani. (Az összetett, több modulból és komponensből álló munkákban az eligazodást nagymértékben könnyíti a Java kódolási konvenciók betartása. Lásd: 12. fejezet) Egy adott érték beállítását általában a set, egy érték lekérdezését a get szócskával kezdünk (`setName`, `getName`). Egy adott tulajdonság fennállását vizsgáló, logikai értéket visszaadó metódust a megfelelő melléknévről célszerű elnevezni, és azt is előtaggal szokás ellátni (`isEmpty`).

Megjegyzés: Sajnos a magyar és az angol nyelv eltérő jellegzetességei miatt ez az elv nem mindig tartható be. A programokban nagyon sokszor keverednek a magyar és angol metódusnevek. Egy programon belül célszerű azonban minél inkább egységes elnevezéseket használni.

Az adattagok megadása után fejtjük ki a `LinearisFüggveny` nevű konstruktort. A konstruktor, mint azt a 3.2.2. fejezetben kifejtettük egy speciális metódus, az osztály példányosításakor kerül végrehajtásra.

Példánkban két olyan metódust definiáltunk, amely értéket ad vissza. Az elsőt y-nal jelöltük – a matematikai analógia miatt – vagyis a függvényértékeket dupla pontosságú lebegőpontos számként meghatározó tagfüggvényt. A tagfüggvény minősítése publikus, tehát más osztályokból elérhető. A másik metódus a `toString`. Ez a metódus a függvény értéket jól olvasható szöveges formában szolgáltatja. (Valójában ez egy felüldefiniált metódus, részletesen az öröklődéss tárgyalásakor elemezzük.)

4.1.1. Osztályok betöltése a virtuális gépbe

Eddig az osztály definiálásával foglalkoztunk. Egy osztály a fordítása után lesz felhasználható, azaz a virtuális gépbe betölthető. A virtuális gép egy adott osztályt akkor tölt be a memóriába, amikor arra az első hivatkozás megtörténik (az interpreter jelleg miatt).

Azt az előző fejezetből láthattuk, hogy egy program futtatását úgy érhetjük el, hogy a virtuális gép indításakor paraméterben megadunk egy olyan osztályt, amelyiknek létezik egy main metódusa (pl: `java Teszt`). Ekkor ezt az osztályt az interpreter betölti és a Java szabványa szerint, a `main` metódussal megkezdi az utasítások végrehajtását.

Abban a pillanatban, amikor a futás „odaér” egy objektum deklarációs utasításához, és ott egy addig még nem használt típusazonosítót talál, azonnal megpróbálja betölteni az objektum típusának megfelelő osztályt a virtuális gép memóriájába.

```
Teglalap t1;
LinearisFuggveny lin1;
```

Amennyiben a Java környezet a `CLASSPATH` környezeti változóban megadott könyvtárakban, vagy csomagokban ilyen nevű osztályt nem talál, a program futási hibával leáll. (Az osztály meglétét természetesen fordításkor is ellenőrzi, és a hibát már itt is jelzi; de előfordulhat extrém esetben, hogy az előzőleg lefordított osztály `class` fájlja elérhetetlen, vagy törölve lett.)

A betöltés során a virtuális gép inicializálja az osztályban szereplő osztályváltozókat és osztályfüggvényeket (statikus adattagok és metódusok) – bővebben lásd az 4.5. Osztályváltozók fejezetben, továbbá beállítja a virtuális metódusok táblázatát (VMT). Amint az osztály betöltése és inicializálása megtörtént, már alkalmas a példányok létrehozására.

4.1.2. A példányosítás

A példányosítás folyamatakor egy

```
t1 = new Teglalap(3,5);
```

alakú híváskor a `t1` referenciaváltozó a `new` utasítás hatására példányt, vagyis egy objektumra mutató referenciát kap. A konstruktorhívás menete a következő:

A `Teglalap t1;` sor a `t1` példány deklarációja volt. A `t1` azonosítót – hasonlóan az alaptípusokhoz – `Teglalap` típusúnak deklaráljuk. Itt még csak az azonosító típushoz rendelését végeztük el, a `t1` objektum még nem létezik. A következő sor a példányosítás művelete: a `t1 = new Teglalap(3,4);` utasítás hatására a rendszer a `Teglalap` osztály két egész paraméterrel rendelkező konstruktorát megkeresi, és meghívja az aktuális (3,4) paraméterekkel. A konstruktorhívás hatására a memóriában megtörténik a helyfoglalás, és az adattagok megkapják alapértelmezett értéküket. Ezután a lefut a konstruktor törzse, és rendre végrehajtja az ott megadott utasításokat. (Esetünkben – lásd 3.2.4. – a téglalap oldalait beállítja a paraméterben átadott értékekre.) A konstruktor törzsének végrehajtása után már rendelkezünk egy objektummal, de ekkor az objektumnak még nincs semmi kapcsolata az azonosítóval. Ekkor a `t1`-hez tehát hozzáreneli az objektum címét, és ettől a ponttól fogva az objektumunk már pontosan definiáltnak tekinthető. Mivel a `t1` azonosító referencia típusú, ezért innentől kezdve a „.” operátor segítségével a metódusai elérhetők, vagyis az objektumnak már küldhető üzenet.

A referencia típusú változók deklarációját közvetlenül követnie kell a konstruktorhívásnak, mert a csupán deklarált változót még nem rendeltük egyetlen objektumhoz sem, így a referenciaazonosító `null` értékű. Ha egy `null` értékű referencia metódusát kívánjuk elérni, akkor futásidejű hibát kapunk (`NullPointerException`). Ennek a hibának az elkerülése végett célszerű a tömörebb definíciós-példányosító utasítást használni:

```
LinearisFuggveny lin1 = new LinearisFuggveny (2,-2);
```

4.1.3. A példányok és az osztályok megszüntetése

Az osztályok és az objektumok mindaddig az operatív tárban maradnak, amíg azokat használjuk. Mivel a Java nyelvben nincsenek destruktor metódusok, a memória felszabadításának terhe lekerült a programozó vállá-

ról. A memória menedzselését a virtuális gép maga végzi a felhasználó elől elrejtve. A memória egyetlen egységet alkotó, ún. szemétgyűjtő mechanizmussal menedzselt dinamikus memóriaterületként értelmezett (garbage collected heap).

A virtuális gép minden betöltött osztályt és minden egyes létrehozott objektumot, a hozzá tartozó referenciával nyilvántart. Amennyiben egy objektumot már nem használunk és egyetlen referencia sem mutat az adott memóriaterületre (kiléptünk az objektumot létrehozó blokkból, vagy a referenciáját `null` értékre állítottuk stb.), onnantól a virtuális gép ezt a területet a szemétgyűjtés végrehajtásáig zárolja. Ugyanígy, ha egy osztályra már nincs szükség (már nincs belőle egyetlen példányosított objektum sem), az osztály által lefoglalt területet is szemétgyűjtés által felszabadíthatónak jelöli a virtuális gép.

A szemétgyűjtést a virtuális gép általában akkor végzi el, ha elfogy a rendelkezésre álló memória, de explicit `System.gc()` hívással a programból is kérhetjük (garbage collect). A szemétgyűjtés az „elérhetetlen” objektumokat „felkeresi”, és az általuk foglalt területeket felszabadítja.

Megjegyzés: A szemétgyűjtő minden felszabadítandó objektumon végrehajtja a `finalize()` metódust (melyet az `Object` osztályból örököl), illetve minden felszabadítandó osztályon a `classFinalize()` osztálymetódust, és csak ezután történik meg a tényleges memóriafelszabadítás.

4.2. Öröklődés

Az objektumorientált szemlélet szerint egy adott osztályt öröklődéssel specializálhatunk. Az öröklődés legegyszerűbb esete, amikor egy osztályt egy már létező osztály kiterjesztéseként definiálunk. A kiterjesztés jelentheti új tulajdonságok (adattagok), és új műveletek (metódusok) megjelenését. Az új, bővített osztály (vagy más néven leszármazott) az ős minden adattagjával és metódusával rendelkezik, a definícióban csak a megjelenő új adattagokat és metódusokat kell definiálni. A Java nyelvben az öröklődést az `extends` kulcsszóval adjuk meg:

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

  public Alkalmazott (String nev_, int fizetes_){
     nev =nev_;
     fizetes = fizetes_;
  }
  public void fizetesEmeles(int mennyivel){
     fizetes += mennyivel;
  }
}

public class Fonok extends Alkalmazott{
  private int potlek;

  public Fonok (String nev_, int fizetes_, int potlek_){
     nev =nev_;
     fizetes = fizetes_;
     potlek = potlek_;
  }
  public int gepkocsiHasznalat(){
     return fizetes/10;
  }
}
```

Az osztálydefinícióban az `extends` kulcsszó mögötti osztálynév jelzi, hogy melyik osztályt kívánjuk a bővíteni.

A Java nyelv csak egyszeres öröklést támogat, tehát az `extends` kulcsszó mögött kizárólag egyetlen, már definiált osztály neve állhat. A létrehozott leszármazott osztály egy önálló új osztály lesz, amely a definícióban megadott adattagok, és metódusokon kívül az ős adatait, és metódusait is tartalmazza. A leszármazott osztályból ugyanúgy hozhatunk létre objektumokat, mint az ősből. Azonban a gyermek csak azokhoz az örökölt tartalmakhoz férhet hozzá, amelyeket a szülő megenged (Lásd 3.2.7. fejezet hozzáférési kategóriák). Ha nagyon szigorú adatrejtést alkalmazunk, akkor az adattagokat csak az ősosztály metódusain keresztül érhetjük el:

- Ha egy ős privát adattagokat definiál, akkor a leszármazott közvetlenül nem férhet hozzá ezekhez az adattagokhoz – jóllehet azok az objektum sajátjai – csakis az ősben definiált, ezeken az adattagokon manipuláló metódusokon keresztül hozzáférhetők. (Egy közérthető példával: gondoljunk egy gyermek szülőjénél lévő zsebpénzére, amely csak a megfelelő kérésre lesz a gyermek számára elérhető.).

- Az ős félnyilvános tagjaihoz a leszármazott hozzáférhet, ha vele egy csomagban van.
- Az ős védett tagjaihoz a leszármazott mindenképpen hozzáférhet, akár más csomagban is definiálhatjuk a leszármazottat.
- Az ős publikus metódusaihoz a leszármazott ugyanúgy hozzáférhet, mint bármilyen másik osztály.

Gyakori programozói hiba az öröklési láncokban publikus adattagokat definiálni, hiszen a programozó ideje „drága”, és nem kíván bajlódni a megfelelő metódusok megírásával. Azonban ilyenkor az objektumorientált alapelvek alaposan sérülnek, tehát az így elkészített program nem tekinthető objektumorientáltnak!

A leszármazott osztály az örökölt adattagok és metódusokon kívül rendelkezik még a saját adattagjaival, metódusaival. Így a példában a `Fonok` osztályban a `nev` és a `fizetes` adattagokon kívül a `potlek`-kal, valamint a `fizetesEmeles` metódus mellett a `gepkocsiHasznalat` metódussal is rendelkezik. A metódusok kezelésében az ős és a leszármazott között kódmegosztás jön létre, hiszen a virtuális gép a két osztály örökölt metódusaihoz csak egyetlen közös memóriaterületet rendel.

A jól megtervezett öröklési hierarchia megtervezésével a forráskódok áttekinthetőbbé, hatékonyabbá és könnyebben módosíthatóvá válnak, hiszen a redundáns kódolás elkerülhető.

4.2.1. A közös ős

A Java nyelvben létezik egy közös ősosztály. Ha egy osztályt definiálunk és nem adunk meg expliciten őst, akkor automatikusan az `Object` osztály leszármazottjaként jön létre. Ez a közös ős a `java.lang` csomag része. Ha a Java osztályait, mint egy fát ábrázoljuk, akkor a fa gyökérpontjában ez az `Object` osztály áll.

Az `Object` osztály metódusait minden osztály örökli. A programozó ezek közül keveset használ (`toString`, `getClass`, `equals`), ám a virtuális gép annál többet. Ezek a metódusok képezik a kapcsolatot az objektumok és a virtuális gép között. Ilyen a már említett `finalize()`is, illetve a párhuzamos programok írásában nagy szerepet kapó `wait()`, `notify()`, és `notifyall()`.

A közös ős használata azért hatékony, mert a minden osztályra és minden példányra értelmezhető alapvető műveleteket a nyelv így csak egyszer definiálja, és azt minden leszármazottjára kiterjeszti.

4.2.2. A konstruktorok szerepe öröklődéskor

A konstruktorok nem öröklődnek. Mint már feljebb is láttuk a konstruktornak az adott osztály példányosításában, és az objektumok inicializálásában van szerepe. Az öröklődést pedig azért használjuk, hogy az adott osztályt specializáljuk, így az esetek nagy részében új adattagokkal bővítjük. Az új adattagokat is illik inicializálni, ezért új konstruktorra van szükség. Ezért a nyelv specifikálásakor a Java fejlesztői úgy jártak el, hogy az öröklésből kihagyták a konstruktorokat.

Vannak olyan esetek, amikor a szülő konstruktorának hívása szükséges. Ezt a Java környezetben a `super()` metódushívással érhetjük el.

```
public Fonok (String nev_, int fizetes_, int potlek_){
   super (nev_, fizetes_);
   potlek = potlek_;
}
```

A szülő konstruktorának meghívása több szempontból is célszerű. Egyrészt csökkenthető a redundáns kódok száma. A példában ugyan csak két adattag szerepelt, de egy valós feladatban egy osztálynak – az összes öröklődést beleszámítva – rengeteg adattagja is lehet. Másrészt ha a szülő adattagjai privát minősítésűek, akkor csak a super hívással tudjuk ezeket az erősen védett adattagokat elérni.

A `super()` hívásnak – működési elve miatt – meg kell előznie a konstruktor többi utasítását! A `super()` hívás folyamata a következő:

- A példányosítás során az adott objektum tárterület kap.
- Majd elkezdi végrehajtani a konstruktorban megadott utasításokat.
- Ha az első utasítás egy `super()` hívás, akkor a példányváltozó inicializálása előtt végrehajtja az ős megfelelő konstruktorát. (A `super()` hívásnak azért kell első helyen állnia, mert először az őstől örökölt változókat kell inicializálni, és csak ezután lehet a leszármazott objektumhoz rendelni.)
- Ezután a vezérlés visszatér a leszármazotthoz, és a fennmaradó utasításokat is végrehajtja.

A paraméterek nélküli konstruktort – `Osztály()` – alakban, alapértelmezett konstruktornak nevezzük. Ha egy osztályhoz nem definiálunk egyet-

len konstruktort sem, akkor a fordítóprogram automatikusan ellátja egy alapértelmezett konstruktorral. Az alapértelmezett konstruktor a példányosításkor meghívja ős alapértelmezett konstruktorát. (Ha a fordító egyetlen ősben sem talál kifejtett alapértelmezett konstruktort akkor a közös ősosztály `Object()` konstruktora fog lefutni.)

Megjegyzés: A `super` kulcsszónak van még egy jelentése (a `this`-hez hasonlóan). Ha egy leszármazott osztályból egy ősosztály adattagját kívánjuk elérni – és a hozzáférés engedélyezett – akkor a `"super.adattag"` hivatkozással megtehetjük azt.

4.3. Többalakúság

A többalakúsággal, mint objektumorientált alapfogalommal az elméleti részben már megismerkedtünk. Most nézzük meg, hogy a Java nyelv milyen megoldást nyújt az alapelv megvalósításához, vagyis ahhoz, hogy objektumaink és műveleteink polimorfikus működéssel bírjanak.

4.3.1. Többalakúság az objektumok között

A Java nyelv szigorúan előírja minden változóhoz a típusosságot. Mind az elemi típusok, mind a referenciák esetén egyértelműen meg kell adnunk, hogy az adott változó/objektum melyik típushoz tartozik. Azonban láttuk, hogy az alaptípusoknál létezik az automatikus, illetve az explicit típuskonverzió, vagyis egy változó értékét egy vele kompatibilis típusba konvertálhatjuk (Lásd: 3.2.6. fejezet.).

Ez a típuskonverzió a referencia típusok között is létezik. Az automatikus konverzió segítségével hívhatjuk meg egy leszármazott objektumhoz az ősben definiált metódusát. Ilyenkor az adott objektumot a rendszer automatikusan az ős típusára konvertálja, majd ez alapján hívja meg a megfelelő metódust.

Ugyanígy egy `Alkalmazott` típusú referenciához egy `Fonok` típusú objektumot is rendelhetünk (lásd: 4.2. fejezet). Az automatikus konverzió itt is végbemegy, mert az objektumhierarchiában a felette álló osztály kompatibilis a leszármazott típussal (az öröklődés tulajdonságai miatt, a leszármazott típus tartalmazza az ősosztályban definiált adattagokat, illetve metódusokat).

Megjegyzés: A kompatibilitás fordítva nem működik, azaz egy `Fonok` típusú referencia automatikusan nem rendelhető egy `Alkalmazott` objektumhoz, a fordító `"incompatible types"` hibaüzenettel leáll.

Ezt a megoldást a Java programok sokszor alkalmazzák, mikor osztályhierarchiákat kell kapcsolatba hozni. Ekkor a hierarchia „tetején" levő osztályt, vagy interfészt kapcsoljuk hozzá egy másik osztályhoz. A hierarchián belüli automatikus konverzió segítségével a példányok műveletei elérhetőek lesznek, tervezésük, nyilvántartásuk is átláthatóbb lesz.

A `java.util` csomag `Vector` osztálya alapértelmezetten `Object` típusú referenciákat tárol egy dinamikus tömbben. Mivel az `Object` osztály minden más osztály őse, így automatikus konverzió segítségével ez az osztály alkalmas bármilyen példány tartalmazására. Ezt, mint konténer osztályt részletesen tárgyaljuk a későbbiekben.

Explicit típuskonverziót, vagy típuskényszerítést is alkalmazhatunk referenciákra. Ha a fenti módon egy ős típusú referenciához egy leszármazott típusú objektumot rendelünk, akkor csak az ősben definiált metódusokat érjük el, hiszen az ős nem tud a leszármazott műveleteiről.

Ekkor két lehetőségünk van. Az első az, hogy típuskényszerítést alkalmazunk az objektum referenciájára. A típuskényszerítést az alaptípusokéhoz hasonlóan megadhatjuk `(LeszármazottOsztály)referencia` alakban.

A másik megoldás az, ha már az ős számára is ismertté tesszük az adott műveletet, melyet a leszármazottakban felüldefiniálunk (esetleg absztrakt osztályok és metódusok, vagy interfészek használatával).

4.3.2. Metódusok felüldefiniálása

Egy osztály kiterjesztése, specializálása általában szükségessé teszi egyes műveletek újrafogalmazását, hiszen a leszármazott nagyon sokszor másképp hajtja végre az adott műveletet. Ezt hívjuk a metódusok felüldefiniálásának (overriding).

Egy osztály felüldefiniál egy metódust akkor, ha az adott műveletet örökölné valamelyik ősétől, de az osztály saját definíciót is ad rá.

A `LinearisFuggveny` osztály "y" metódusa felüldefiniálja a `KonstansFuggveny` osztály azonos szignatúrájú(lásd: 4.3.3. fejezet) metódusát – ahol `y(x)` a függvény x helyen felvett értékét jelöli.

A `KonstansFuggveny` osztályban az „y" metódust úgy definiáljuk, hogy az egyszerűen az objektum adattagjával térjen vissza (függetlenül x értékétől), azonban a lineáris függvények esetén ez a visszatérési érték már az ax+b képlet alapján határozható meg.

A leszármazott osztályban a `super().metódus()` alakú hívással hivatkozhatunk az ősosztályban definiált metódusra.

```
public class KonstansFuggveny {
  protected double a;

  public KonstansFüggveny(double a_) {a=a_;}
  public double y(double x) { return a;}
  public String toString() {
      return "Konstans függvény y="+a ;
  }
}
public class LinearisFuggveny extends KonstansFuggveny {
  protected double b;

  public LinearisFüggveny(double a_, double b_) {
     a=a_; b=b_;
     }
  public double y(double x) {
     return a*x+b;
     }
  public String toString() {
      return "Lineáris függvény y="+a+"*x+"+b ;
  }
}
```

A példában még egy másik metóduson keresztül is megfigyelhető a felüldefiniálás. Az osztályokban definiált `toString()` metódusokon ezt használjuk ki. A `toString()` metódust az `Object` osztály vezeti be. Szerepe az, hogy az objektum referenciához egy szöveges megjegyzést, értelmezést fűzzünk. Ez a metódus az `objektum.toString()` hívással érhető el.

A legtöbb esetben azonban a `toString()`-et a virtuális gép automatikusan hívja. A Java nyelvben ezt a metódust úgy definiálták, hogy amikor egy objektum referenciát olyan helyen használunk, ahol azt Stringgé kell konvertálni, az automatikus konverzió ennek a metódusnak a végrehajtásával fut le.

Az alábbi programrészletben ezt használjuk ki. A `KonstansFuggveny` osztályból példányosítunk `k1` néven. Ezután a kiadott két utasítást a virtuális gép azonos módon hajtja végre. Vagyis eredményként a szöveges kimenetre kétszer kiírja a „Konstans függvény y=4” üzenetet.

```
KonstansFuggveny k1 = new KonstansFuggveny(4);
System.out.println(k1.toString());
System.out.println(k1);
```

A metódusok felüldefiniálásának fontos szabálya, hogy az új metódusnak az ősben definiálttal kompatibilisnek kell lennie. Ez azt jelenti, hogy:

- a metódus neve, paramétereinek típusa és száma (szignatúra), és a visszatérési típusnak meg kell egyeznie;
- az új metódus hozzáférési kategóriája nem lehet szűkebb a felüldefiniálandó metódusnál;
- az új metódus csak olyan kivételeket válthat ki, amilyeneket az eredeti is kiválthat. (A kivételkezelésről a 6.6. fejezetben lesz szó.)

A polimorfizmus sarokpontja a késői kötésnek nevezett objektumorientált alapelv hatékony működése (futás alatti kötés, dinamikus kötés). Ahogy azt az elméleti részben is láttuk, az egyes objektumoknak ismernie kell a saját műveleteit. Ahhoz, hogy a futás alatti kötés hatékonyam működjön, a metódusokat nem szabad direkt módon szorosan az osztályhoz kapcsolni (fordítási időben), hanem valamilyen technikával az egyes példányokhoz kell rendelni. Így az adott objektum műveleteit a virtuális gép minden esetben eléri.

A Java nyelvben – a tisztán objektumorientált jelleg miatt – egy osztály minden metódusát virtuálisnak tekintünk, vagyis minden metódust alapértelmezés szerint dinamikus kötéssel rendelünk az egyes objektumokhoz. Ezt a Java egy rejtett szolgáltatása végzi (dynamic method dispatch), amely minden egyes metódushívásnál megvizsgálja, hogy az egy felüldefiniált vagy felültöltött metódusra érkezett-e. Ha igen, akkor azt a metódust keresi meg és hajtja végre, amely az adott objektumhoz tartozik.

Megjegyzés: Minden esetben az objektum konstrukciójakor megkapott (hozzárendelt) metódusokat hajtja végre, még akkor is, ha az automatikus konverzió kihasználásával, egy ős típusú referenciával hivatkozunk a példányra.

A `final` módosító segítségével lehetőségünk van egy osztályt, vagy metódust véglegesnek tekinteni. A `final` kulcsszó használata után egy osztályt már nem lehet leszármaztatni, illetve a metódust nem lehet felüldefiniálni. Így elérhető, hogy egy metódus a leszármazott osztályokban is garantáltan ugyanúgy hajtódjon végre. (Ilyen például az `Object` osztály `getClass()` metódusa, amely az adott példányt leíró dinamikus osztálytípust határozza meg. Mivel ennek megváltoztatása az egész Java környezetre kihatással lenne, ezért véglegesítették.)

4.3.3. Metódusok felültöltése

Sok programozási nyelv az egyes függvényeket, metódusokat csak a nevük alapján azonosítja, így a hasonló műveletekhez eltérő azonosítókat kell rendelnünk. A Java nyelvben azonban az egyes metódusokat az ún. szignatúra azonosít. A szignatúra a metódus nevéből, paramétereinek típusából és számából áll. Az így specifikált azonosító képzési szabály nagyobb szabadságot ad a programozók kezébe. A hasonló műveleteket azonos névvel, de eltérő paraméterezéssel használhatjuk, és a magasabb szinten megfogalmazott műveleteket könnyebben implementálhatjuk a nyelvben.
Ezt a technikát a metódusok felültöltésének (overloading) nevezzük.

```
public class Fonok {
  private String nev;
  private int fizetes, potlek;

  public Fonok (String nev_, int fizetes_, int potlek_){
     nev= nev_; fizetes = fizetes_;
     potlek = potlek_;
  }
  public void fizetesEmeles(int emeles_){
     fizetes += emeles_;
  }
  public void fizetesEmeles(int emeles_, int potlek_){
     fizetes += emeles_;
     potlek += potlek_
  }
```

A példában a `fizetesEmeles` metódusra adtunk kétfajta értelmezést. Ezután egy példány az azonos üzenetre „a főnök fizetésemelést kap” kétféleképpen reagálhat. A hívás után – a futás idejű kötés miatt – a paraméterezéstől függően a megfelelő utasítássorozat kerül végrehajtásra.

```
Fonok fonok1 = new Fonok("Béla", 100000, 5000);
fonok1.fizetesEmeles(25000);
fonok1.fizetesEmeles(25000, 1000);
```

A felültöltött metódusok kiválasztása – elméleti szinten, ha több felüldefiniált és felültöltött metódus is szerepel – nem teljesen egyértelmű folyamat. A Java minden esetben, a dinamikus metódushívás miatt, a hívás helyén az aktuális paraméterek típusa szerinti „legjobban illeszkedő" metódust hívja meg.

Nagyon gyakran előfordul, hogy egy osztályban felültöltött konstruktorokat használunk. A használat oka a példányosítás folyamatának általános értelmezése. Gyakran kell ugyanis egy objektumot különféle értékekből létrehozni, de előfordulhat olyan eset is, amikor a létrehozás pillanatában még nem ismert minden szükséges adat.

Ha például egy osztályban törtszámokat szeretnénk kezelni, akkor a példányosításnak több felültöltött metódusdefiníciót is adhatunk:

```
public class Tort{
  private int a,b;           //számláló és nevező

  public Tort(){             //alapértelmezett konstruktor
     a = 0; b = 1;
  }
  public Tort(int a_){
     a = a_; b = 1;
  }
  public Tort(int a_, int b_){
     a = a_; b = b_;
     }
  public Tort(Tort t){                  // másoló konstruktor
     a = t.a ; b = t.b;
     }
  …
}
```

A paraméterek nélküli `Tort()` alakú konstruktort alapértelmezett konstruktornak nevezzük (default constructor). Híváskor nem fogad paramétereket, így valamilyen alapértelmezett értékekkel látjuk el a példányt. A második konstruktor segítségével egész számokból képezhetünk törteket, pl. a `Tort(3)` alakú konstruktorhívással. A harmadik esetben egy tört számlálójából és nevezőjéből képezzük az objektumot a `Tort(3,4)` hívással.

Az negyedik konstruktor definíció a létrehozandó osztály egy példányán keresztül hoz létre új objektumot. Ezt más néven másoló konstruktornak is nevezzük (copy constructor).

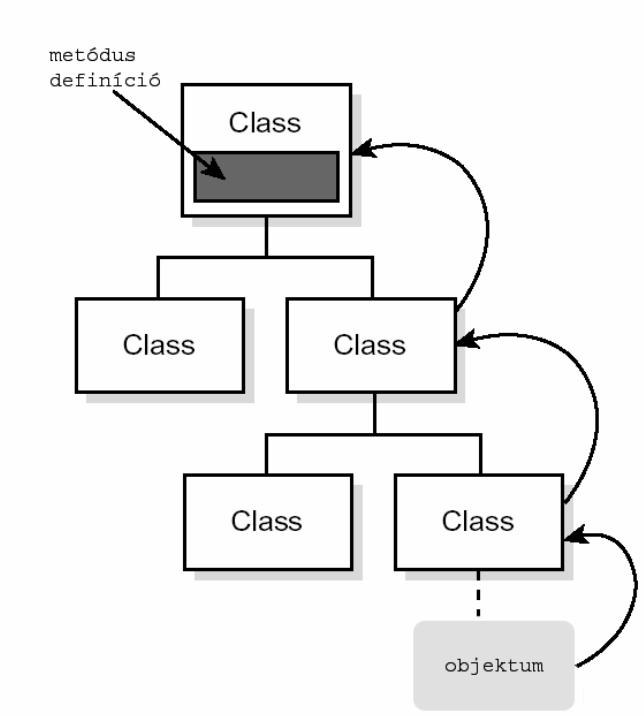
Megjegyzés: Említettük, hogy ha egy osztályban nem definiálunk egyetlen konstruktort sem, akkor a fordító ellátja alapértelmezett konstruktorral. Ha azonban megadunk legalább egy saját konstruktort, akkor nem kerül befordításra az alapértelmezett konstruktor.

A `this` kulcszó tárgyalásakor említettük (lásd: 3.2.4.), hogy fontos szerepe lesz a konstruktoroknál. A konstruktordefiníció első utasításában a `this(…)` hívással hivatkozhatunk egy előzőleg már definiált konstruktorra. A `this()` hívás használatával a definíciót visszavezethetjük egy már definiált műveletre, így csökkenthető a kód redundanciája, hiszen a felültöltött konstruktorok hasonló műveleteket hajtanak végre, kisebb módosításokkal. Az előző példát akár így is definiálhatjuk:

```
public Tort(int a_, int b_){a = a_; b = b_;}
public Tort()         { this(0,1); }
public Tort(int a_)   { this(a_,1); }
public Tort(Tort t)   { this(t.a,t.b); }}
```

4.3.4. A metódushívás folyamata az öröklési láncban

Ha elkészítünk egy osztályhierarchiát, akkor a metódusokat általában a művelet első előfordulási szintjén definiáljuk. Ha egy leszármazott osztályt példányosítunk, akkor az, az öröklődésben tárgyalt szabályok szerint elérheti azokat. A metódushívás helyességét már a fordító ellenőrzi, hiszen egy objektum csak olyan metódusra hivatkozhat, amelyikkel kapcsolatban áll, de ennek eldöntése a kezdő programozó számára nem mindig egyértelmű. A futás során a metódushívást a virtuális gép a következő mechanizmussal hajtja végre: ha az adott osztályban talál olyan szignatúrájú metódust, akkor azt végrehajtja, ha ilyen nincs, akkor megkeresi a legközelebbi ősében.



17. ábra: A metódushívás folyamata az öröklési láncban

Felültöltött, illetve vegyesen felültöltött és felüldefiniált metódusok esetén a hívásnak megfelelő szignatúrájú metódust keresi meg és hajtja végre.

Abban az esetben, ha egy metódust az adott öröklési ágon valahol felüldefiniáltunk, akkor a hívás helyéhez „legközelebb" eső metódus kerül végrehajtásra. Vagyis a vezérlés csak addig lépdel felfelé a hierarchián, amíg a definíciót meg nem találja.

4.3.5. A rekurzió

Egy osztályban definiált metódus természetesen meghívhat más metódusokat. Abban az esetben, ha egy metódus önmagát hívja, rekurzív metódushívásról beszélünk. A rekurzió a hagyományos programnyelvekben használt rekurzív működéshez hasonlóan használható. Egyszerű és könnyen áttekinthető kódot ad, de ugyanúgy lassabb és erőforrás igényes a végrehajtása (a paramétereknek és a lokális változóknak minden egyes híváskor tárhelyet kell foglalni).

Készítsünk egy `egyszerusites()` függvényt, amely két szám hányadosát egy rekurzív legnagyobb közös osztót meghatározó `lnko(a,b)` függvénynyel egyszerűbb alakra hozza:

```
private void egyszerusites(){
   if (b<0){  a=-a; b=-b;}
   if (a==0) b=1;
   else {
        int o= lnko(Math.abs(a), b);
        a/=o;
        b/=o;
   }
}
//rekurzív legnagyobb közös osztó számítás
private int lnko(int x, int y){
   if(x%y!=0) return lnko(y, x%y);
   else return y;
}
```

Az egyszerűsítés a tört számlálóját és nevezőjét a két szám legnagyobb közös osztójával osztja. Ehhez egy olyan rekurzív függvényt használunk (lnko), amelyik két pozitív egész számból a maradékképzés segítségével előállítja a legnagyobb közös osztót.

Készítsünk el egy olyan Törtszámokat kezelő osztályt, amely kihasználja a felültöltés adta lehetőségeket.

```
/**
* Tortszámokat bemutató példaprogram
* overloading, felültöltés, privát és nyilvános metódusokkal
*/
public class Tort{
     //számláló, nevező
     private int a, b;

     //felültöltött konstruktorok
     public Tort(int a_, int b_){
        a=a_ ;
        b=b_;
        egyszerusites();
     }
     public Tort(int a_ ) { this(a_, 1); }
     public Tort() { this(0,1); }
     public Tort(Tort t) { this(t.a, t.b);}
```

```
//saját metódusok def.
private void egyszerusites(){
   if (b<0){  a=-a; b=-b;}
   if (a==0) b=1;
   else {
        int o= lnko(Math.abs(a), b);
        a/=o;
        b/=o;
   }
}

//rekurzív legnagyobb közös osztó számítás
private static int lnko(int x, int y){
   if(x%y!=0) return lnko(y, x%y);
   else return y;
}


//nyilvános metódusok
public String vegyesTort(){
   return (b==1)? " "+a : ((a/b)+" + "+(a%b)+"/"+b);
}
public void szoroz(int i_, int j_) {
    a*=i_;
    b*=j_;
    egyszerusites();
}
public void szoroz(int i_) {
    szoroz(i_, 1);
}

public void szoroz(Tort t_) {
    szoroz(t_.a, t_.b);
}

public void osztas(int i_) {
    szoroz(1, i_);
}

public void osztas(Tort t_) {
    szoroz(t_.b, t_.a);
}

public void hozzaadas(Tort t2) {
    a=a*t2.b+t2.a*b;
    b*=t2.b;
    egyszerusites();
}
```

```
        public void kivonas(Tort t2) {
            a=a*t2.b-t2.a*b;
            b*=t2.b;
            egyszerusites();
        };
        //... stb.
    }
    /* Tesztprogram */
    public class TortTeszt{
      // Tört példányok
      public static void main(String[] args){
        //Tort1 osztályból példányosítunk
        Tort t1= new Tort(1,3);
        Tort t2= new Tort(4,3);
        System.out.println("Az első tört: "+t31.vegyesTort());
        System.out.println("A második tört: "+t32.vegyesTort());
        //műveletvégzés Törtek között,
        //az eredmény t1-ben keletkezik
        t1.osztas(t2);
        System.out.println("Hányadosuk: "+t1.vegyesTort());
        }
      }
```

4.4. Absztrakt osztályok

Az öröklődés és a többalakúság nagyon fontos szempont az objektumorientált modellek tervezésekor. Az ősosztályok „közös nevezőre” hozzák a leszármazott osztályok működését, vagyis definiálják a közös műveleteket és megadják a szabványos üzenetek prototípusait. Ezzel egy egységesen kezelhető osztályhierarchiát adhatnak a programozó kezébe. Az osztályhierarchia legtetején szereplő osztályok szerepe általában az, hogy a leszármazottaknak előírja, hogy milyen üzenetekre válaszoljon. A leszármazott osztályokból létrehozott példányok a „keresd az őst” elvnek megfelelően megkeresik a hozzájuk tartozó üzenet végrehajtásának definícióját.

Sokszor azonban az osztályszerkezet tetején álló osztály – a feladat modellezéséből adódóan – csak egy elméleti fogalom, vagyis csak teljesen általános fogalmakkal írható le, azaz hiányoznak belőle a konkrétumok.

Ha például síkidomok területét szeretnénk meghatározni, akkor a síkidom területe egy általános érvényű fogalom (absztrakt), hiszen képlettel csak egyes konkrét alakzatok (téglalap, kör, háromszög) területét tudjuk kiszámítani.

Az ilyen általános érvényű osztályokhoz absztrakt osztályokat definiálunk. A Java nyelv az absztrakt osztályok definiálását a class kulcsszó elé írt abstract módosítóval oldja meg.

Az absztrakt osztályokban – és csakis itt – megadhatunk absztrakt metódusokat is. Az absztrakt metódusok törzs nélküli metódus deklarációk. A metódusokat a leszármazott osztályokban kötelező felüldefiniálni, megvalósítani.

Az absztrakt metódust úgy deklaráljuk, hogy a metódust szintén ellátjuk az `abstract` módosítóval, és a metódus definícióját üres utasítással jelöljük. (A kapcsoszárójelek helyén pontosvessző áll.)

```
public abstract class Sikidom {

  public abstract double kerulet();
  public abstract double terulet();
}

public class Teglalap extends Sikidom {
  private double a, b;

  public Teglalap (double a, double b) {
     this.a = a;
     this.b = b;
     }
  //az ős absztrakt metódusainak felüldefiniálása
  public double kerulet() {
     return 2*(a+b);
     }
  public double terulet() {
     return a*b;
     }
}

public class Kor extends Sikidom {
  private double r;

  public Kor (double r) {
     this.r = r;
     }
  public double kerulet() {
     return 2*r*Math.PI;
     }
  protected double terulet() {
     return r*r*Math.PI;
     }
}
```

A `Sikidom` osztály egy absztraktként definiált osztály két absztrakt metódussal a kerület és a terület meghatározására. A két leszármazott osztályban (`Kor`, `Teglalap`) ezeket a metódusokat felüldefiniáljuk, és konkrét utasítással látjuk el, hiszen a modell ezen a szintjén már ismerjük a megfelelő képleteket.

Az absztrakt osztályok csak abban különböznek egy átlagos osztálytól, hogy tartalmazhatnak törzs nélküli, absztrakt metódusokat. Mivel az absztrakt metódusok csak a metódusok szignatúráját rögzítik – nem adnak definíciót – ezért az absztrakt osztályok nem példányosíthatók. (Természetesen tartalmazhatnak rendes adattagokat és metódusokat is.)

Ha absztrakt osztályból hoznánk létre objektumokat, akkor azoknak nem lehetne érvényes üzenetet küldeni, futás közbeni hibákat idéznének elő, hiszen az absztrakt metódusok nem tartalmaznak utasításokat. (Az absztrakt metódusok előtt nem használhatjuk a private, static és a final módosítókat.)

Az absztrakt osztályok tartalmazhatnak egyéb adattagokat és definiált metódusokat is.

Az absztrakt osztályok leszármazottjai is lehetnek absztraktak, de az osztályhierarchiát úgy kell kialakítani, hogy legyen legalább egy olyan osztály, amely az absztrakt metódusokat megvalósítja.

Az absztrakt osztályok használata során előfordulhatnak tipikus programozói hibák.

- A fordító nem engedi meg, hogy absztrakt osztályokat példányosítsunk, és a „`Sikidom is abstract; cannot be instantiated`” hibaüzenetet adja.
- A másik gyakori hibalehetőség, hogy a leszármazott osztályban nem definiáljuk az ős absztrakt metódusait. Ekkor a „`Teglalap is not abstract and does not override abstract method kerulet() in Sikidom`” hibaüzenetet kapjuk, vagyis a fordító azt üzeni, hogy vagy a `Teglalap` osztályt kell absztraktnak minősíteni, vagy meg kell adni a `kerulet()` metódus definícióját.

4.5. Osztályváltozók és osztálymetódusok

Az osztályok eddig megismert adattagjait példányváltozóknak is hívhatjuk, hiszen ezeknek a változókhoz minden egyes példánynál külön tárterület tartozik. Néha azonban szükségünk van olyan változókra és műveletekre, amelyek az egész osztályra vonatkoznak. Az alábbiakban megismerkedünk

az osztályváltozókkal (class member) és az osztálymetódusokkal (class method). Az osztályváltozókat és metódusokat még a statikus adattag és statikus metódus névvel is illethetjük. Használatukhoz a `static` kulcsszót használjuk.

4.5.1. Osztályváltozók

Egy osztályváltozó az egész osztályra vonatkozó értéket tárol. Az osztályváltozók az osztály minden példánya számára közösek és elérhetők. Vagyis az osztályváltozók számára pontosan egy tárterületet rendel a virtuális gép.

Az alábbi példában egy cég alkalmazottai mindannyian 62 éves korukban mennek nyugdíjba. Mivel ez a tulajdonság minden egyes példányra közös, osztályváltozóban definiáljuk. Így a példányoknak nem kell külön-külön tárterületet foglalni ehhez az értékhez.

```
public class Alkalmazott{
  private static int nyugdijKorhatár = 62;
  private int eletkor;
…
  public String nyugdijbaVonulas(){
     return "Nyugdíjba vonulás "
        +(nyugdijKorhatar - eletkor)+ " év múlva";
     }
}
```

Az osztályváltozók definiálási sorrendben kapnak kezdőértéket. A kezdőértékadás már az osztály virtuális gépbe való betöltésekor megtörténik. (A példányváltozók természetesen a konstrukció során kapnak kezdőértéket.) Az osztályváltozók kezdőérték adásánál a már definiált osztályváltozókat felhasználhatjuk, de a példányváltozókat nem! Az osztályváltozókat később az objektumok kezdőértékéhez (példányváltozók) már felhasználhatjuk, hiszen addigra az értékük már definiált lesz.

Az osztályváltozókat az osztály definíciója közben ugyanúgy érünk el, mint példányváltozókat. Nagyon gyakori, hogy osztályváltozókban konstans értékeket tárolunk. Ilyenkor a minősítés `static final`, vagyis csak egyszer kaphat kezdőértéket a változó, amelynek értéke a futás során nem változtatható meg. (Ebben az esetben nem sértjük meg az adatrejtés elvét, hiszen a változó végleges, csak olvasható.)

A konstansként definiált végleges osztályváltozókat csupa nagybetűvel írjuk. A Java környezet megadásakor a lang csomagban található Math osztályba gyűjtötték a gyakori matematikai állandókat és függvényeket.
A Java API a π értékét a

```
public static final double PI = 3.14159265358979323;
```

osztályváltozóval definiálja.

A publikus osztályváltozókra `Osztály.osztályváltozó` alakban is hivatkozhatunk, pl.: `Math.PI` . Ilyen statikus adattag pl. a `System` osztály `out` adattagja is, amely alapértelmezés szerint az operációs rendszer karakteres kimenetét jelenti, így ennek az osztályváltozónak a felhasználásával tudunk a képernyőre karaktereket írni. (pl.: `System.out.println()`.)

4.5.2. Osztálymetódusok

Az osztálymetódusok szerepe nagyban hasonlít az osztályváltozókéra. Ahogyan egy „normál” metódus az egyes példányok adattagjain értelmezett műveletet jelent, úgy az osztálymetódusok az osztály egészére vonatkozó műveleteket jelentenek.

Egy osztálymetódus értelemszerűen csak az osztályváltozókhoz férhet hozzá. Az egyes példányváltozók, vagy a `this` használata fordítási hibát okoz. Az osztályműveletek függetlenek az osztály egyes objektumaitól, az osztály virtuális gépbe való betöltése után már azonnal hívhatóak, mielőtt még egyetlen példányt is létrehoztunk volna. Ezért az osztálymetódusok hívása legtöbbször az `Osztály.osztálymetódus()` alakban történik. (Természetesen egy adott példányra is meghívhatjuk a műveletet `objektum.osztálymetódus()` alakban, de szemantikailag az előző megoldás a megfelelőbb.)

A már említett `Math` osztály számos művelete osztálymetódusként lett definiálva, hiszen a gyakran használt számítási módszereket, képleteket így egyszerűen érhetjük el. Például a négyzetgyökvonást az alábbi osztálymetódus definiálja:

```
public static double sqrt(double a){}
```

Ha ezt a műveletet használni szeretnénk, akkor egy alábbihoz hasonló hívást adunk ki a programunkban:

```
double gyokketto = Math.sqrt(2);
```

Ha az Alkalmazott osztályban a nyugdíjkorhatárt meg szeretnénk változtatni, akkor azt egy osztálymetódusban tudjuk megtenni:

```
public class Alkalmazott{
  //osztályváltozó:
  private static int nyugdijKorhatár = 62;
  //osztálymetódusok:
  public static void nyugdijKorhatarEmel(){
     nyugdijKorhatár++;
     }
  public static void nyugdijKorhatarCsokk(){
     nyugdijKorhatár--;
     }
  …
}
```

A main, mint osztálymetódus

A fentiekhez hasonló osztálymetódus a `main` is. Első programunkban nem fejtettük ki, hogy a main metódus definíciója miért ilyen előre rögzített alakú:

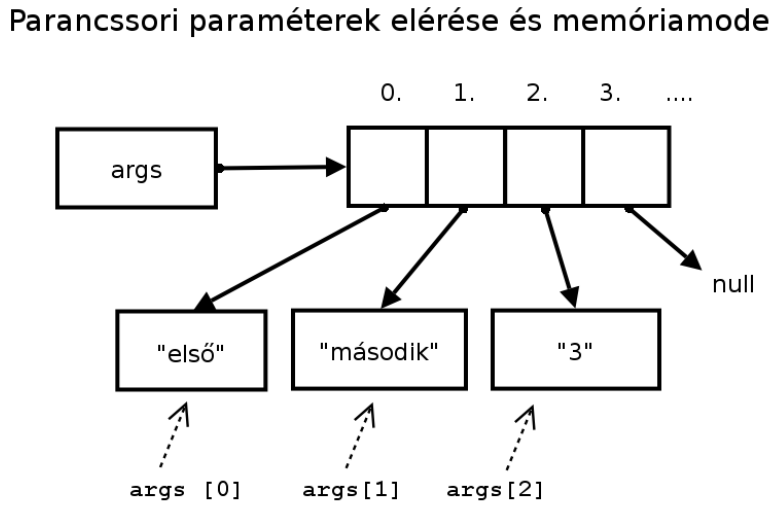
```
public static void main(String args[]) {…}
```

A main osztálymetódus szolgál arra, hogy egy publikus osztály virtuális gépbe való betöltésekor „elindulhasson”, és a programunk *belépési pontja* legyen. Ehhez a metódusnak már ismertnek kell lennie az osztály betöltése után közvetlenül, ezért használnunk kell a `static` módosítót.

A `public` módosító azért kötelező, mert egy külső hívásról van szó, más hozzáférésű metódust nem tudnánk elindítani, a void pedig azt jelzi, hogy a `main` végrehajtása után nem ad vissza értéket a hívónak, vagyis a virtuális gépnek.

A `main` osztálymetódus bemenő paramétere egy sztringekből álló tömb, amely a híváskor a megadott parancssori paramétereket tartalmazza. Azonban a Java nyelv a 0. indexszel már az első „valódi” beérkező paramétert éri el, és nem a programindító parancs nevét, mint a C, vagy a C++ nyelvben.

```
public class Peldaprogram {
  public static void main(String[] args) {
    if (args.length > 0){
      for (int i = 0; i < args.length; i++)
        System.out.println(args[i]);
    } else {
      System.out.println("Nincsenek paraméterek!");
    }
  }
}
```



18. ábra: A parancssori paraméterek használata

4.5.3. Az osztályváltozók és osztálymetódusok elfedése

Egy osztály leszármaztatása során lehetőségünk van az ősökben definiált példány vagy osztályváltozókkal azonos nevű változókat definiálni. Ekkor a leszármazott osztály elfedi az ős azonos nevű adattagjait. Az elfedett változókat a leszármazott örökli, de közvetlenül nem fér hozzá, csak a minősített `super.adattag` hívással, vagy típuskényszerítéssel.

Az 4.3. többalakúság fejezetben kifejtettük, hogy a metódusok felüldefiniálhatók. Az osztálymetódusok tárgyalásakor pontosítanunk kell ezt a kijelentést, hiszen csak a dinamikus kötésű példánymetódusok definiálhatóak felül. Az osztálymetódusok esetén nem beszélünk polimorfizmusról, hiszen ezek a metódusok csak az adott osztályra vonatkoznak.
Az osztálymetódusok esetén elfedésről beszélünk. Egy osztálymetódus elfedi az ősben definiált, vele megegyező szignatúrájú metódusokat.

Az osztálymetódusok fordítási időben történő statikus kötéssel rendeli a fordító az osztályhoz. Az osztálymetódusokat dinamikus kötésű (tehát nem `static` minősítésű) metódusokkal nem szabad elfedni! Az elfedésre a felüldefiniálással azonos korlátozások vonatkoznak (visszatérési érték, kivételek, hozzáférési kategóriák).

4.6. Kérdések

- Milyen szabályok vonatkoznak egy osztály definiálására a Java nyelvben?
- Milyen hozzáférési kategóriákat adhatunk meg az egyes elemeknek?
- Mit tudunk a virtuális gép tárgazdálkodásáról?
- Hogyan zajlik a példányosítás folyamata?
- Milyen kulcsszót használunk az öröklődés jelzésére?
- Öröklődnek-e a konstruktorok?
- Mit jelent a super() hívás?
- Mit jelent a szignatúra szó?
- Mit jelent egy metódus felüldefiniálása?
- Mit jelent egy metódus felültöltése?
- Hogyan találja meg a virtuális gép az objektumhoz tartozó metódust?
- Van-e lehetőségünk a Java nyelvben rekurzív metódust írni?
- Hogyan definiálunk absztrakt osztályt?
- Hogyan deklarálunk absztrakt metódust?
- Mi történik, ha absztrakt osztályból szeretnénk példányosítani?
- Hogyan definiálunk osztályváltozót, és osztálymetódust?
- Mikor kell osztályváltozót, ill. osztálymetódust használnunk?
- Az osztály példányai elérik az osztályváltozókat? Ha igen, hogyan?
- Az osztálymetódusok elérik a példányok adattagjait? Ha igen, hogyan?
- Mit jelent a `static final` minősítés?
- Mikor beszélünk a változók elfedéséről?

4.7. Feladatok

Tervezze meg és készítse el az alábbi Java nyelvű osztályokat! Ügyeljen az adatrejtés betartására!

- `Téglatest` (a téglatestet a, b, c oldalával jellemezzük; definiáljon konstruktort, a további számítandó értékek: felszín, térfogat).
- `Alkalmazott` (név, lakcím, havi fizetés; metódusok: a személyi adatokat megjelenítő, és az éves fizetést kiszámító).
- `Henger` (sugár, magasság, ill. felszín, térfogat) és `Gömb` (sugár, ill. felszín, térfogat).

- Készítse el azt az öröklési szerkezetet, melyben egy `Test` ősosztályból származó `Téglatest`, `Hasáb`, `Henger`, `Gömb` osztályokat használhatunk. A leszármazott osztályokban adja meg a felszín és a térfogat-számítási metódusokat.
- Készítsen egy független tesztelő osztályt, melynek `main` metódusában a fenti osztályokat példányosítja, és részletesen bemutatja (megfelelő szöveges tájékoztatással) az egyes metódusok működését.
- Készítsen egy absztrakt `Állatok` osztályt, amelyben egy `élőhely()` absztrakt metódust definiál. Az osztály leszármazottjai legyenek a `Háziállatok` és a `Vadállatok`. A leszármazott osztályban valósítsa meg felüldefiniálással az absztrakt metódust. (Az osztályok tetszőleges egyéb tartalommal kiegészíthetők.)
- Készítsen egy osztályt a `Tort` osztályhoz hasonló módon a `Komplex` számok kezelésére (adattagok a valós és a képzetes rész) a megvalósítandó műveletek (felültöltéssel) összadás, kivonás, szorzás és osztás valós illetve komlex számmal.
- Az `Alkalmazott` osztályban kezelje osztálytagként a példányosított objektumok számát. Adjon meg osztálymetódusokat, amelyekkel az objektumok számát pontosan karbantarthatjuk. Mutassa be példaprogramban a helyes működést.

5. Interfészek

A Java alapvető építőelemeit az osztályok jelentik. Az objektumorientált tervezési és programozási munka során a legtöbbször osztályokkal dolgozunk. Az előzőekben megismerkedtünk az absztrakt osztályokkal, mint az általánosságban megfogalmazott és konkrét példányokkal, kizárólag örökítési céllal létrehozott osztályok típusával. Ilyen szerepet töltenek be a Java nyelvben az interfészek is.

Az interfész nem más, mint az osztályok egy olyan általánosítása, amely csupán konstans értékeket, és metódusok deklarációját tartalmazhatja. (Az absztrakt osztályok még tartalmazhattak változó jellegű adattagokat, és definiált metódusokat is.)

Az interfész csak egy felületet, kapcsolódási pontot jelent, hiszen itt a műveletek prototípusát adjuk meg, azaz egy szabályt, hogy milyen műveletet akarunk az adott objektumon végrehajtani. A modellezésben ezt a magasabb absztrakciós szintet már elvonatkoztathatjuk minden konkrét adattól, változótól.

Az összehasonlítást például absztrakt módon megadhatjuk úgy, hogy ha két objektumot össze akarok hasonlítani, akkor azt egy logikai függvénnyel végezzük. Az egyes konkrét osztályokban pedig megadjuk az összehasonlítás szabályát, hogy milyen tényleges adattagok, vagy más jellemzők segítségével hasonlítunk össze két objektumot. (Ha például személyeket hasonlítunk össze, akkor név szerint, gépjárműveket pedig mondjuk típusuk szerint különböztetünk meg stb.)

Ha egy adott adatszerkezetben rendezetten szeretnénk objektumokat tárolni, akkor definiálnunk kell a rendezési szabályt (metódust). De ezt a rendezési szabályt ismernie kell majd annak az osztálynak is, amelyiknek a példányait tárolni szeretnénk (hiszen itt tudjuk megadni az összehasonlítás „képletét” az egységbe zártság és adatrejtés miatt). Valahogy tehát szinkronba kell hozni ezt a két osztályt, hogy azonos üzenettel hívják, illetve hajtsák végre a műveleteiket. Erre szolgálnak az interfész osztályok, hiszen itt csak a „protokollt” adjuk meg.

Az interfészek tényleges működése majd az azokat implementáló (megvalósító) osztályokon keresztül történik, így az absztrakt modell konkrét tartalmat kap.

Az interfész referencia típus, de közvetlenül nem példányosítható. Azonban nagyon gyakran használjuk, mint a típusok összekapcsolását jelentő hivatkozást.

Egy tömbben síkidom típusú objektumokat szeretnénk tárolni. A modellezésben a síkidom típusú objektumokat konkrétan a téglalap, kör, háromszög stb. osztályokból példányosítjuk. A tömb Java nyelvi tulajdonsága miatt egyszerre csak egyféle típusú elemeket tartalmazhat, így a tárolást csak külön kör, téglalap, háromszög típusú tömbbel lehetne megoldani. Azonban ha a tömböt az interfész típusával deklaráljuk, akkor – az automatikus típuskonverzió miatt – a tömbelemek az interfészt implementáló osztály objektumai lehetnek.

Deklarálhatunk tehát interfész típusú azonosítót, amely egy olyan osztály egy példányára vonatkozik, amelyik azt az interfészt megvalósítja. Így az interfész típus bármelyik leszármazott osztály példányához felhasználható.

5.1. Interfészek deklarációja

Az interfészeket az `interface` kulcsszó segítségével deklaráljuk, ahol az interfész neve valamilyen melléknév:

```
public interface Interfésznév {
  final int KONSTANS = kezdőérték;
  public boolean művelet();
}
```

A `java.lang` csomag `Comparable` interfésze az egyes objektumok összehasonlítását teszi lehetővé. Deklarációja nagyon egyszerű:

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

A Java nyelv nem engedi meg, hogy egy osztálynak több őse legyen, de megengedi egy, vagy több interfész felhasználását, megvalósítását.

Az interfészek általában két olyan osztályt kötnek össze logikailag, amelyeknek nem sok köze van egymáshoz, mégis szeretnénk egy olyan közösen használható felületet megadni, amelyik mindkét osztályban garantálja, hogy

az adott művelet hasonló módon menjen végbe. A rendezések során például fontos szerepet játszik az egyes objektumok összehasonlítása.

5.2. Interfészek kifejtése, megvalósítása

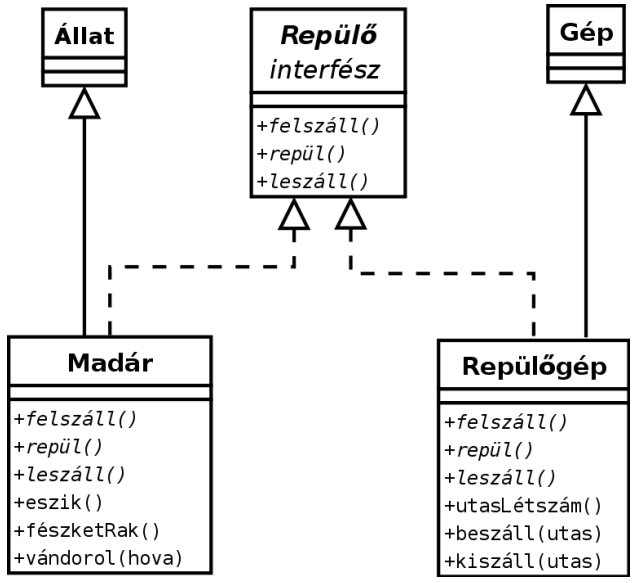
Egy osztály megvalósít egy interfészt, ha az osztálydefinícióban az `implements` kulcsszóval bejelentett interfész metódusaira definíciót ad.

```
public class Osztály implements Interfész;
public class Osztály extends Ős implements Interfész;
public class Osztály extends Ős implements Intf1, Intf2;
```

Ha a fenti `Comparable` interfészt egy konkrét osztályban kell megvalósítani, akkor az alábbi alakban lehet azt megtenni:

```
public class Osztály implements Comparable{
  double d;
  //az interfészben előírt metódus definíciója:
  public int compareTo(Comparable obj){
     return ( this.d - (Osztály)obj.d );
     }
… }
```

Ezzel a definícióval már az osztály felhasználható az olyan osztályokban, ahol a `Comparable` interfésszel valósítják meg a rendezéshez szükséges objektumok összehasonlítását.



19. ábra: Interfész használata

Az ábrán látható feladatban két osztályt szeretnénk ellátni hasonló funkciókkal. Abban az esetben, ha a két osztály egy-egy független leszármazási lánc része, akkor a Java megszorításai miatt egy második őst nem már nem rendelhetünk hozzájuk. A `Madár`, mint az `Állat` osztály, illetve a `Repülőgép`, mint a `Gép` osztály leszármazottja két teljesen független öröklési láncot jelent, mégis mind a két osztályban használni szeretnénk a repülési képességeket meghatározó műveleteket. Ehhez interfészt kell használnunk. Az interfész nevét itt melléknévi értelemben használjuk (milyen?)! Az interfészben deklarált metódusfejeket az interfészt megvalósító osztályban fejtjük ki. Így elmondhatjuk, hogy a `Madár` is és a `Repülőgép` is `Repülő` (repülni képes).

```
/** A repülni képes élőlények/tárgyak közös interfésze*/
public interface Repulo {
  public void felszall();
  public void leszall();
  public void repul();
}

public abstract class Eloleny{
  private String nev;
  public Eloleny(String nev_){ nev = nev_;}
  public String getNeve(){ return nev;}
  public abstract String taplalkozik();
}

public abstract class Gep{
  private String nev;
  public Gep(String nev_){nev = nev_;}
  public String getNeve(){ return nev;}
  public abstract String hasznalatiUtasitas();
}


public class RepuloGep extends Gep implements Repulo{
     public RepuloGep(String nev_){
     super(nev_);
  }
  public String hasznalatiUtasitas(){
     return "A repülőgép légi közlekedési eszköz.";
  }
  // az interfészben deklarált metódusok kifejtése
  public void felszall(){
     System.out.println("A repülőgép a
        kifutópályáról száll fel.");
  }
```

```
    public void leszall(){
       System.out.println("A repülőgép egy hosszú
          kifutópályára száll le, hosszan fékezve.");
    }
    public void repul(){
       System.out.println("A repülőgép 10000m magasban
          óránként 800-900 kilométert tesz meg.");
    }
}
public class Madar extends Eloleny implements Repulo{
    public Madar( String nev_){super (nev_);}
    public String taplalkozik(){
       return "A madarak rovarokkal, magokkal táplálkoznak.";
    }
    // az interfészben deklarált metódusok kifejtése
    public void felszall(){
       System.out.println("A madarak sűrű
          szárnycsapásokkal szállnak fel.");
    }
    public void leszall(){
       System.out.println("A madarak a lehető
          legkisebb helyre is le tudnak szállni");
    }
    public void repul(){
       System.out.println("A madarak óránként
          20-100 kilométert tesznek meg.");
    }
}
public class Teszt{
   public static void main(String args[]){
     //példányosítás
     Madar galamb1 = new Madar("Galamb 12");
     RepuloGep airbus1 = new RepuloGep("Airbus 340");
     // majd a metódusok elérése
     System.out.println("madar1 neve: "+ galamb1.getNeve());
     galamb1.taplalkozik();
     galamb1.felszall();
     galamb1.repul();
     galamb1.leszall();
```

```
        System.out.println("airbus1 neve: "+ airbus1.getNeve());
        airbus1.hasznalatiUtasitas();
        airbus1.felszall();
        airbus1.repul();
        airbus1.leszall();
    }
}
```

Az interfészt implementáló osztálynak definiálnia, vagy ősén keresztül „ismernie” kell az adott metódust, így egy interfész egy osztályhierarchiában többször is implementálható, nem okoz fordítási hibát. (Olyan eset is előfordulhat, hogy az ősosztályban már szerepel egy, az interfészben deklarált szignatúrájú metódus, ekkor az – öröklés miatt – ismert metódust nem kell újra definiálnunk, de akár az osztályban az ős metódusát felül is definiálhatjuk.)

Egy példányról az `instanceof` operátorral tudjuk eldönteni, hogy az adott osztálynak vagy ősosztálynak példánya-e, illetve, hogy az adott interfészt megvalósítja-e.

```
objektum instanceof Osztály;
objektum instanceof Interfész;
```

Interfészeket gyakran fogunk használni a Java nyelvben (a `java.util` csomagban) definiált konténer osztályokban, illetve számos absztrakt modellel leírt feladat megoldása során.

5.3. Interfészek öröklődése

Az interfészek öröklődési kapcsolatba is állhatnak. Ebben az esetben kiterjesztésről beszélünk.

```
interface KiterjesztettInterfész extends ŐsInterfész{…}
```

Ekkor a kiterjesztett interfész örökli az ősének konstans tagjait, és metódusdeklarációit, illetve tartalmazza a saját törzsében deklaráltakat. Az ilyen típusú interfészek megvalósításakor mind az ős, mind a kiterjesztett interfész metódusaira definíciót kell adnunk, különben a fordító a `"Osztály is not abstract and does not override abstract method metódus(int) in Interfésznév"` hibaüzenettel jelzi, hogy az implementáló osztálynak ki kell fejtenie a metódust (vagy az osztályt absztraktnak kell minősíteni).

5.4. Belső osztályok

A belső vagy beágyazott osztályon egy osztályon, vagy interfészen belül definiált másik osztályt, vagy interfészt értünk. A többi osztályhoz hasonlóan egy belső osztálynak is lehet láthatósága (publikus, védett, csomagszintű, vagy privát). A belső osztályok elhelyezésének kétféle megvalósítását szoktuk megkülönböztetni: statikus, vagy dinamikus.

Ha egy osztályt utasításblokkon belül hozunk létre, akkor lokális osztályról beszélünk. Névtelen osztályok pedig akkor keletkeznek, amikor egy már létező osztályt a példányosítás során új műveletekkel látunk el. (A belső osztályok mélyebb és átfogóbb tanulmányozásához [Nyéky01] megfelelő fejezeteit ajánlom.)

5.4.1. Tagosztályok

A tagosztályok olyan osztályok definícióját jelenti, amelyeket egy osztály adattagjai és metódusai kifejtése között, mint az osztály egy újabb tagját definiáljuk.

```
class KülsőOsztály …{
  …
  <minősítés> class BelsőOsztály {
  …
  }
}
```

Statikus tagosztályok

A statikus és dinamikus deklaráció élesen elkülönül egymástól. Statikus esetben azt használjuk ki, hogy egy beágyazott, statikus minősítésű osztály elrejthető a külvilág elől, annak létezéséről csak a tartalmazó tud. A statikus belső osztályok szoros kapcsolatban vannak a tartalmazó osztállyal. A külső osztály metódusai hozzáférnek a statikus tagosztályok privát tagjaihoz, így a belső osztály tagjainál szereplő minősítések csak a „külvilágnak” szólnak. Magának a belső osztálynak a minősítése (`public`, `protected`, `private`) megengedett, láthatósága így szabályozható. A statikus tagosztály definiálása esetén maga a definíció független a tartalmazó osztálytól.

A `Lista` osztályt egy kétirányú láncolt lista megvalósítására szeretnénk definiálni. A lista minden elemét a privát `Elem` belső statikus osztály példányaként definiáljuk. Az `Elem` osztály önállóan létezik a `Listá`-tól függetlenül, de a külvilág elől rejtve marad. Kezdetben üres listából indulunk ki, majd a `beszur` metódussal tudunk újabb elemeket a listába felvenni.

```
class Lista{
  private Elem elso;                  // Belső osztályú adattag
  private static class Elem {         // statikus belső osztály
     private Object adat;
     private Elem elozo, kovetkezo;

     Elem (Object adat, Elem elozo, Elem kovetkezo){
        this.adat = adat;
        this.elozo = elozo;
        this.kovetkezo = kovetkezo;
        }
  }// belső osztály vége
  public void beszur(Object adat){     // beszúrás a listába
     elso = new Elem(adat, null, elso);
     if(elso.kovetkezo != null){
        elso.kovetkezo.elozo = elso;
     }
  }
  public Lista(){…}
  //…A további listakarbantartó műveletek
}

public class ListaTeszt{
   public static void main(String args[]){
     // A listába szánt objektumok
     Integer szam1 = new Integer(-4);
     Double szam2 = new Double(3.15);

     // A lista objektum létrehozása
     Lista dinamikusLista = new Lista();
     // Lista feltöltése,
     dinamikusLista.beszur("Indul a lista");
     dinamikusLista.beszur(szam1);
     dinamikusLista.beszur(szam2);
     dinamikusLista.beszur(new Boolean (true));
     // Lista kiírása
     System.out.println(dinamikusLista);
   }
}
```

A belső `Elem` osztály privát adattagjaihoz hozzáférhetnek a tartalmazó osztály metódusai.

Megjegyzés: Belső osztályként interfészeket is deklarálhatunk.

Nem statikus tagosztályok

Ha egy belső osztályt a fenti példa alapján nem statikusnak adunk meg (hiányzik a `static` kulcsszó), akkor a belső osztály elérheti a tartalmazó osztály privát adattagjait is. Ezt a `KülsőOsztálynév.this.adattag` minősített hívással tehetjük meg. Itt már érvényesülnek a beágyazás dinamikus jellegzetességei is.

Megjegyzés: A nem statikus tagosztályok példányosításakor előálló belső objektum(ok) a befoglaló osztály adott példányához (befoglaló példány) tartoznak. Így a belső osztály metódusaiban két aktuális példány is értelmezve van. Az eredeti értelemben vett elsődleges példány mellett ott van az aktuális példány befoglaló példánya, mint másodlagos aktuális példány. A másodlagos aktuális példány változóira ugyanúgy hivatkozhatunk, mint az elsődleges példányaira, és a befoglaló osztály metódusait minősítés nélkül is hívhatjuk a másodlagos aktuális példányra. Vagyis a nem statikus tagosztály látja és elérheti a befoglaló példány változóit.

Egy belső osztályt akkor használunk nem statikus értelemben, ha a külső és a nem statikus tagosztály között 1:N kapcsolat áll fenn.
Egy nem statikus belső osztály egy példányát kívülről a

```
belsoObjektum = kulsoObjektum.new BelsoOsztaly()
```

alakú konstruktorhívással hozzuk létre.

5.4.2. Lokális osztályok

A lokális osztályok egy ún. osztálydefiniáló utasítással definiált osztályok. Érvényességük csak az őket tartalmazó blokkra terjednek ki, így hozzáférési minősítést nem kaphatnak. A lokális osztály látja a definíciós pontjában elérhető változókat, ám törzsében csak olyan `final` változókra hivatkozhatunk, amelyek már kaptak kezdőértéket. Ezt a korlátozást azért vezették be, mert ezek az osztályok a lokális változókról másolatot készítenek és az osztályt tartalmazó blokkból való kilépés után is megőrzik értéküket, zárványok keletkeznek.

A lokális osztály ugyanazon szabályok szerint érheti el a tartalmazó osztály tagjait, mint az előző részben tárgyalt statikus és nem statikus tagosztályok.

```
…
public void createLocalObject(final String e){
 // konstans deklaráció
 final char f = 'f';
 // az i változó nem lesz használható a lokális osztályban!
 int i = 0;
  // Lokális osztály - egy blokkon belül definiált –
  // csak ebben a blokkban értelmezett
   class Local {
    private char c = 'c';
    public void printVars() {
      System.out.println(c);  // OK
      System.out.println(e);  // OK
      System.out.println(f);  // OK
      System.out.println(i);  // fordítási hiba!
    }
   } //class Local vége
   // Lokális osztály példányosítása
   Local lokalispeldany = new Local();
   // üzenetküldés a lokális példánynak
   lokalispeldany.printVars();
  }
…
```

5.4.3. Névtelen osztályok

Sokszor szükségünk van olyan osztályokra, amelyet csak egyszer használunk, példányosítunk. Ezeknek az osztályoknak a definícióját a példányosítás helyén is megadhatjuk. Az így keletkező osztályt névtelen osztálynak nevezzük. Megadása úgy történik, hogy a tartalmazó osztályból egy új ideiglenes leszármaztatást készítünk, új metódusokkal úgy hogy a tartalamzó osztály példányosítása során heyben megadjuk az új osztály definícióját.

```
new Osztály(paraméterek) { osztálytörzs }
```

Adjon meg `Osztaly`-ból leszármazott névtelen osztályt, amelyben definiál egy új metódust is!

```
…//tetszőleges blokkan:
Osztaly objektum = new Osztaly(){
     // A névtelen osztály törzse
     public nevtelenEgyMetodusa() {
        return szamitottErtek;
     }
  }
…
```

5.5. Kérdések

- Milyen célra használunk interfészeket?
- Interfész törzsében milyen jellegű adatokat helyezhetünk el?
- Minek tekinthetők az interfész metódusai?
- Lehet-e használni az abstract módosítót interfészekben?
- Mit nevezünk az interfész kifejtésének (megvalósításának)?
- Interfészek között létezik-e öröklődés?
- Mire használhatjuk a belső osztályokat?
- Mi jellemzi a `Comparable` interfészt, hol használható fel?

5.6. Feladatok

- Definiáljon egy `Sikidom` interfészt, amelyben deklarálja a kerület és területszámításhoz szükséges metódusokat, majd definiáljon benne jellemző konstansokat. Alakítsa át a `Kör`, a `Téglalap` és a `Háromszög` osztályokat úgy, hogy azok megvalósítsák a `Sikidom` interfészt.
- Készítsen interfészt `Hasonlithato` névvel (a `Comparable` mintájára), és deklaráljon egy műveletet, amely két síkidomot a területük alapján összehasonlít. Alakítsa át úgy az előző feladatot, hogy a `Kör`, `Téglalap` és `Háromszög` osztályok ezt az interfészt is megvalósítják!
- Definiáljon verem adatszerkezetet belső osztályok segítségével, majd lássa el karbantartó műveletekkel.

6. Csomagok, komponensek

Egy nagyobb program elkészítésekor törekednünk kell az áttekinthető, könnyen módosítható, és továbbfejleszthető kód előállítására. Ennek eléréséhez a programokat megfelelően tagolni kell. A tagolásnak már az objektumorientált modellezés és a programtervezés szintjén meg kell nyilvánulnia. Az összetartozó osztályokat ezért – a modell és a működés szerint – csomagokba szervezzük. A Java osztálykönyvtárait is csomagokban, jól átlátható szerkezetben definiálták. A csomagokból képezhető egy magas szintű zárt egység, más szóhasználattal élve komponens.

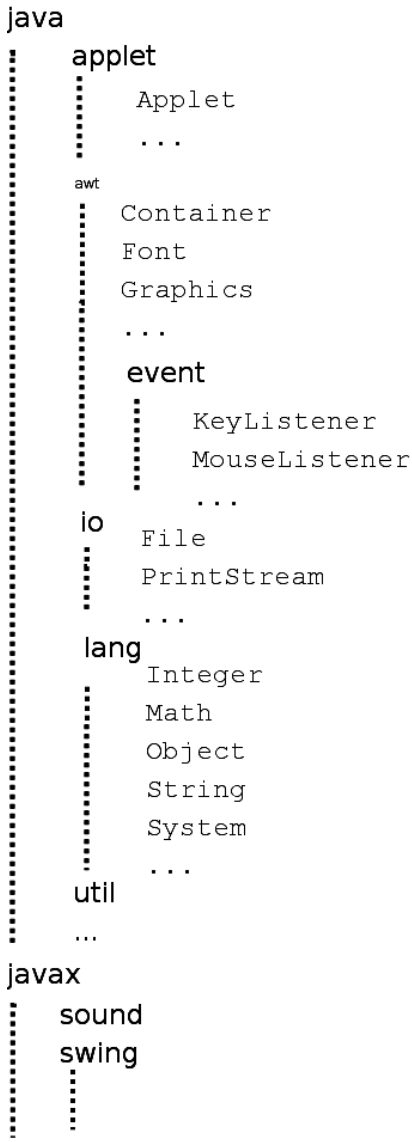
Minden csomag egy önálló **névteret** (namespace) vezet be, amelyben egyedivé teszi a benne található osztályok szerkezetét. A csomag alkalmas a hozzáférési kategóriák kezelésére, és a csomagszintű hozzáférési kategóriával definiált osztályok, interfészek, metódusok csak a csomagon belül lesznek elérhetőek.

A csomagoknak nyelvi szinten a fordítási egységek(osztályok, intrfészek) adnak konkrét tartalmat. Egy csomaghoz tartozó fordítási egységben található a csomag forráskódja. Ez a csomagszerkezet hierarchikus – hasonlóan az állományok könyvtárszerkezetben való elhelyezéséhez – így a csomagok között logikai alá-fölérendeltségi viszony is előfordulhat. Ekkor alcsomagnak nevezzük a csomagban található másik csomagot. Minden csomag tetszőleges számú alcsomagot tartalmazhat.

6.1. A Java nyelv alapcsomagjai

A JDK csomagstruktúrája követi a fájlrendszerek könyvtárszerkezetét, így a főcsomagok egy-egy különálló könyvtárban helyezkednek el, amelynek alkönyvtárai lesznek a főcsomagok alcsomagjai stb. A legfelső szinten helyezkedik el többek között a `java` csomag. Az egyes alcsomagokba további csomagokat, vagy osztályokat és interfészeket találunk. Így a `java.lang` csomag – mint a nyelv alapcsomagja – a nyelv alaposztályait, a működéshez elengedhetelen osztályok definícióit tartalmazza, a `java.io` csomag a fájlkezeléshez szükséges osztálydefiníciókat, a `java.applet` csomag tartalmazza a hagyományos html oldalakba épülő alkalmazások (appletek) leírását tartalmazó osztályokat, a `java.awt` a grafikus alkalmazások alaposztályait tartalmazza stb.

A Java2 megjelenésekor 1999-ben, a nyelvet is bővíteni kellett [1]. Erre a `javax` csomag szolgált, amelynek alcsomagjai a megjelenő új kódokat tartalmazták.



20. ábra: A Java osztálystruktúrája (részlet)

A Java csomagnevek tetszőlegesek lehetnek, azonban a `java`, `javax`, vagy `sun` kezdetű csomagokba kizárólag a Sun szakemberei helyezhetnek csomagokat. A csomagnevek megkötése továbbra az is, hogy a létrehozott új csomagok nem ütközhetnek a Java API-ban megadottakkal.

A csomagok és osztályok használatához az `import` kulcsszó után megadott, „.” operátorral elválasztott minősítések szolgálnak. Ha például az `io` csomag `File` osztályát szeretnénk elérni, akkor azt a névtérbe importálhatjuk a következő utasítással:

```
import java.io.File;
```

Ha az `io` csomag minden osztályára szükségünk van, akkor azt így jelöljük:

```
import java.io.*
```

A további bővítési lehetőségeket, az interneten használt URL hivatkozásokhoz hasonlóan az egyes gyártók által kínált osztályokat is elérhetüjük:

```
import com.sun.security.auth.*;
```

A csomagokat hagyományosan fájlrendszerben, egyes esetekben adatbázisokban is tárolhatjuk. A fájlrendszerben tárolt csomagok egy kiinduló könyvtárból érhetőek el. Ebből a könyvtárból, mint gyökérből kiindulva adhatjuk meg az egyes alcsomagok könyvtárát, és ezzel párhuzamosan a forráskódban a csomag elérését.

A Java források vizsgálatával – a Java környezet alapkönyvtárában található `src.zip` fájlból – láthatjuk, hogy a Math osztály hivatkozása `java.lang.Math` alakú, és forrása `Math.java`, amely a `"java/lang"` könyvtárban található. (A lefordított osztályok is ugyanezt a hierarchiát követik az `rt.jar` fájlban.)

A kialakított osztályszerkezet eléréséhez az operációs rendszerben szükségünk van egy CLASSPATH nevű környezeti változóra, ahol a csomagok gyökérpontjait rögzítjük.

A Java környezet a CLASSPATH-ban megadott könyvtárak alapján a csomaghivatkozásokat eléri. A CLASSPATH környezeti változóhoz nem csak könyvtárnevek, hanem `"csomag.zip"` és `"csomag.jar"` alakban megadott fájlok is hozzáfűzhetők. Mivel a Java rendszert felkészítették a tömörített állományok feldolgozására is, ezért hozzáadható a `.zip`, vagy az ugyanilyen tömörítési algoritmust használó `.jar` fájl is. Sokszor megadjuk az aktuális könyvtárat jelentő hivatkozást is: `"."`.

Az elérési utat az alábbi parancsokkal módosíthatjuk (Linux-Windows):

```
export CLASSPATH=
  $CLASSPATH:/home/java/munka: /home/java/csomag.jar: .
SET CLASSPATH=
  %CLASSPATH%;c:\java\munka;c:\java\csomag.jar;.
```

Egy Java csomag típusdefiníciókat (osztályok, interfészek forrásfájljait) és további alcsomagokat tartalmazhat. Egy alcsomag nem kapcsolódik szorosan a hierarchiában felette állóhoz, mivel a Java specifikáció szerint a csomagok függetlenek, legyenek azok egymás melletti, illetve egymás alatti könyvtárban definiálva. A hierarchikus szerkezet csak a modellezés folyamatát és a csomagok közötti logikus eligazodást segíti.

6.2. A fordítási egység

A Java környezetben a fordítási egység az a hely, ahol a csomagok forráskódja megtalálható. Fájlrendszerben tárolt csomagok esetén ez maga a forrásfájl.

Egy fordítási egység a csomagdeklarációval kezdődik, ezután következnek a más csomagokra vonatkozó importdeklarációk, majd a típusdeklarációk.

```
package sajatcsomag;
import java.net.*;
…
public class Osztaly{
…
}
```

6.2.1. Névtelen csomagok

Abban az esetben, ha a fordítási egység elejéről elhagyjuk a csomag deklarációt (`package`), akkor az egy ún. névtelen csomagba tartozik. Ez a névtelen csomag azoknak a forrásfájloknak az összessége, amelyek egy alkönyvtárban helyezkednek el (jellemzően az operációs rendszer alapértelmezett munkakönyvtárában). A névtelen csomag csak nagyon egyszerű programokhoz, általában ideiglenes, vagy tesztosztályok fejlesztéséhez használható. A névtelen csomaban nem deklarálhatunk alcsomagokat. Egy összetett alkalmazás esetén, a névtelen csomagokkal elvesztjük a modularitás adta előnyöket és a program átláthatósága is erősen romlik, hiszem minden osztályt fizikailag egy könyvtárban kell tárolnunk.

6.3. Csomagok deklarációja

Minden fordítási egység elején deklarálhatjuk, hogy az melyik csomaghoz tartozik. A csomagdeklarációt a `package` kulcsszó vezeti be és a csomag neve követi, (pl: `package sikidom;`).

A `package` kulcsszó csak a fordítási egység legelején állhat, és egy fordítási egységben csak egyetlen ilyen deklaráció lehet! A csomagnév – a Java kódolási konvenciók ajánlása alapján [6] – egy csupa kisbetűkből álló azonosító lehet.

Egy csomag szabadon bővíthető újabb elemekkel, így egy fordítási egység, (osztály, vagy interfész) egyszerűen azáltal, hogy egy csomag nevét deklarálja, automatikusan a csomag részévé válik.

Ha a CLASSPATH környezeti változóban meghatározott könyvtárban vagyunk akkor a fenti csomagdeklarációval ellátott fordítási egységeket az ugyanolyan nevű könyvtárban kell elhelyeznünk. (A fenti példa forrásállománya a `sikidom` könyvtárba kerül.)

Ha a csomagnév összetett, pl. `package csomag1.csomag2.csomag3;` alakú, akkor értelemszerűen a `/csomag1/csomag2/csomag3` könyvtárba kell elhelyezni, különben a virtuális gép a hivatkozott osztályokat nem találja meg.

6.4. Csomagok importálása

A fordítási egységek csomagdeklarációja után következhetnek az import deklarációk. Minden egyes Java osztály a tartalmazó csomag nevével és az osztály nevével magadott ún. minősített névvel hivatkozható, pl: `java.util.Vector`. Azonban a fejlesztés hetékonyabbá tételéhez, és a minősített nevek sorozatos kiírása helyett az osztályokra rövid nevekkel célszerű hivatkozni. (Ez nagymértékben javítja a forráskód áttekinthetőségét is.)

> Az import deklaráció valójában nem más, mint a más csomagokban deklarált publikus típusok egyszerű névvel történő elérésének lehetősége a minősített nevek helyett.

```
import sikidom.Teglalap;
import sikidom.*;
import java.util.*;
import java.util.zip.Zipfile
import com.sun.security.auth.login.ConfigFile
```

Az import deklaráció két alakban használható. Az első esetben a `sikidom` csomag egyetlen osztályát importáltuk, a második esetben pedig mindegyiket.

A harmadik esetben a `java` csomag `util` alcsomagjának összes típusát (osztályok, interfészek) importáltuk. Ez az importálás nem jelenti egy adott csomag alcsomagjainak automatikus importálását (a művelet nem rekurzív), mert mint az előző fejezetben láttuk, az alcsomagok nem ismerik egymást, és nem ismerik a felettük, vagy alattuk elhelyezkedő csomagokat sem, így a negyedik sorban levő `zip` alcsomag `Zipfile` osztályára való hivatkozást külön meg kell adnunk.

Az import deklaráció a csomagon belül csak az adott fordítási egységben értelmezett, vagyis ha másik egységben is használni szeretnénk azt a típust, akkor ott újra importálni kell.

Megjegyzés: Az importálás nem jelenti a fordítási egységek megnyitását és a forráskód bemásolását, (tehát lényegesen különbözik a C nyelv #include direktívájától) csak a típusok rövid névvel való elérésére szolgál.

A fenti példában található importálások után az osztálynevek megadása a következő minősített nevek megadását eredményezik:

```
Teglalap     →   sikidom.Teglalap
Vector       →   java.util.Vector
Zipfile      →   java.util.Zipfile
ConfigFile   →   com.sun.security.auth.login.ConfigFile
```

6.5. A jar fájlok használata

Egy csomagot a Java keretrendszer részét képező `jar` „Java Archive” segédalkalmazás segítségével (/bin/jar, jar.exe) tömörítve tárolhatjuk, későbbi fejlesztésekben komponensként felhasználhatjuk. A `jar` fájlok tömörítési technikája megegyezik a `zip` állományok tömörítési algoritmusával.

Egy `jar` fájlban az egymásba ágyazott csomagok egy egységként, (modulként, komponensként) kezelhetőek. A komponenst a CLASSPATH környezeti változóba bejelentve a többi csomag számára elérhető és hivatkozható lesz.

Ezen felül a `jar` fájl számára megadható – az ún. manifest fájllal, ami a `jar` fájl leírója – egy olyan osztálya is, amelyik belépési pontként szolgál, így egy modul a virtuális gépbe betöltve önállóan is futtatható (amennyiben van egy `public static void main()` osztályfüggvénye).

Egy `jar` fájl leírását a `Manifest.mf` fájlban találjuk:

```
-Manifest-Version: 1.0
-Created-By: 1.5.0 (Széchenyi István Egyetem)
-Main-Class: Teszt
```

A `Main-Class` bejegyzésben adható meg a belépési pont,azaz a fenti példa szerint a `jar` fájl betöltésekor a publikus `"Teszt"` osztály `main` osztálymetódusával elkezdődik a program végrehajtása.

6.5.1. Jar fájl készítése

A `.jar` fájlok elkészítését a `jar` segédprogram megfelelően paraméterezett hívásával kezdeményezhetjük. Az aktuális könyvtárban (ahol a csomagot készíteni szeretnénk) az alábbi paranccsal indíthatjuk a modulépítést:

```
jar cvfm sajatcsomag.jar Manifest.mf .
```

ahol az alábbi kapcsolókat használtuk:

```
c - create, új fájl készítése
v - verbose, azaz beszédes kimenet
f - az előálló .jar fájl neve
m - a Manifest fájl neve
. - a csomagolni kívánt könyvtár neve
```

Ezután a `sajatcsomag.jar` néven előálló modult futtathatjuk, vagy felhasználhatjuk a további fejlesztéshez. (A `jar` segédprogramnak még számos, itt nem tárgyalt funkciója és lehetősége van.)

Megjegyzés: A manifest fájl szintaktikája előírja, hogy minden sora, így az utolsó után is legyen soremelés, különben a jar csomagoló ezt nem veszi figyelembe!

6.5.2. Jar fájl futtatása

Egy `jar` fájl, amely tartalmaz egy olyan publikus osztályt, amelynek van `main` osztálymetódusa és ezt a `manifest` fájlban is megadtuk, az alábbi paranccsal futtatható:

```
java -jar sajatcsomag.jar
```

Említettük már, hogy a CLASSPATH környezeti változóban nemcsak könyvtárakra, hanem .jar, vagy .zip állományokra is hivatkozhatunk.

Tehát, ha egy kész modult (komponenst) újra fel akarunk használni, akkor a CLASSPATH környezeti változóban megadjuk a komponens `jar` fájljának abszolút elérési útját:

```
export CLASSPATH=$CLASSPATH:/home/munka/sajatcsomag.jar
SET CLASSPATH=%CLASSPATH%;c:\munka\sajatcsomag.jar
```

Ezután az egyes fordítási egységekben (forrásprogramokban) a modul csomagjai elérhetők, a definiált osztályok és interfészek importálhatók, hivatkozhatók.

Egyedi esetben a virtuális gép indításakor explicit módon is megadhatjuk a CLASSPATH értékét a `-cp` kapcsolóval:

```
java -jar -cp /home/munka/egyikcsomag.jar masikcsomag.jar
```

6.6. Kérdések

- Milyen célra használjuk a csomagokat?
- Hogyan kell egy csomagot bejelenteni a forráskódban?
- Mire használjuk a CLASSPATH rendszerváltozót?
- Mire használhatóak a jar fájlok?

6.7. Feladatok

- Egy tetszőleges, előzőekben kidolgozott feladathoz (lásd: 3.4, 4.7, 5.6. fejezet) készítsen egy csomagot, majd a csomag osztályaiból és a bemutató programból állítson elő futtatható `.jar` állományt.
  - a logikailag összetartozó osztályok egy könyvtárban legyenek,
  - a bemutató program egy szinttel ezen könyvtár fölött helyezkedjenek el.

7. Kivételkezelés

A programozási munka során általában a helyes használatnak megfelelően készítjük fel a programokat. A működés során azonban előfordulhatnak olyan, esetleg kivételes állapotok is (a helytelen használatból, valami külső okból, vagy programozási figyelmetlenségből eredően), amikor a program hibásan működik.

A hagyományos programozási technikákban a szubrutinok hibakódokkal jelezték a végrehajtás helyességét, illetve hibáit. A Java nyelv más szemléletmóddal kezeli a hibák előfordulását. A programkód kivételes állapotba kerülésekor a virtuális gép egy kivételt (`exception`) „vált ki". A kivételek olyan objektumok, amelyek egyes kivétel-osztályokba sorolhatók. A kivétel osztályokat a `Throwable` osztály alosztályaiként definiálták. A kivétel jellegű osztályok két csoportba sorolhatók:

- **Error** (hiba): Az ilyen típusú kivételek olyan alapvető hibákat jelölnek, amelyeknek azonnali programfutás leállás az eredménye, és nem kezelhetőek le (pl: elfogy a memória).
- **Exception** (kivétel): Ezek a fajta kivételes események a futás során lekezelhetők. Ekkor a program normális menete megszakad, és a vezérlést az előírt vezérlési szerkezetekkel az aktuális hibát lekezelni hivatott programrészre irányíthatjuk.

A kivételkezelés erősen épít a Java tipikusan objektumorientált elemeire: A futási időben előálló hiba hatására egy objektum keletkezik, amely a hiba fellépésének körülményeit (a hiba típusát és a program aktuális állapotát) írja le, ezt a virtuális gép észleli és átadja egy, a kivétel feldolgozására szánt programrésznek, amely a kivételobjektumba „csomagolt" információknak megfelelően gondoskodik a folytatásról.

A kivételek feldolgozása szempontjából megkülönböztetünk ellenőrzött és nem ellenőrzött kivételeket. Azokat a kivételeket, amelyeket feltétlenül fel kell dolgozni (azaz kötelező vagy specifikálni, vagy elkapni), ellenőrzött kivételnek nevezik.

A Java fordító minden esetben hibát jelez, ha úgy észleli, hogy egy metódus valamilyen ellenőrzött kivételt válthat ki, de az nem tartalmazza az adott kivételt elkapó programblokkot, illetve a kivételobjektum típusát sem specifikálja. Ilyen esetek például a fájl- és hálózatkezeléssel, speciális sztringkonverzióval stb. foglalkozó programrészek.

Kivételes esemény a következő futás alatti esetekben keletkezhet:

- A program futása során valamilyen rendellenesség történt (tömb túlindexelése, osztás nullával, nemlétező fájlra hivatkozás, stb).
- A program egy `throw` utasítása váltotta ki a kivételt. Ez megtörténhet a Java osztálykönyvtárakban, vagy a programozó által létrehozott kódban is.
- Egy aszinkron kivétel érkezett. Ez a többszálú programokban fordul elő, amikor az egyik szál valamiért leáll.

A leggyakrabban az elsőként említett ok miatt keletkeznek kivételek.

Ha egy kivételes esemény keletkezik a kód végrehajtása során, a programkódtól a virtuális gép átveszi a vezérlést. Ezután egy olyan helyet keres a programkódban, ahol a kiváltott kivétel lekezelése megtörténhet. A kivétel lekezelése egy speciálisan arra definiált kódblokkban történik. A kivételkezelő blokkok egymásba ágyazhatóak, így egyszerre több alkalmas ilyen kódrészlet is létezhet, ekkor a virtuális gép az egymásba ágyazott blokkokból „kifelé” haladva az elsőként megtalált, hibát lekezelő blokkot hajtja végre.

A kivételkezelő blokkjának lefutása után a program a blokk utáni utasítással folytatja futását.

7.1. Kivételosztály definiálása

A kivételek kezelésére szolgáló `Exception` osztálynak számos alosztálya van, és ezek együtt egy összetett osztályhierarchiát alkotnak.
A leggyakrabban előforduló kivételek osztálya az `Exception`, vagy ennek `RunTimeException` leszármazottja, illetve ennek további leszármazottjai.

A `java.lang` csomagban a kivételeknek készen adott osztályait találjuk. A saját kivételosztályt a `Throwable`, az `Exception` vagy a `RuntimeException` osztályok kiterjesztéseként definiálhatjuk. A saját kivételosztályok többszintű hierarchiát is alkothatnak.

```
Exception
   |  ClassNotFoundException
   |  CloneNotSupportedException
   |  IllegalAccessException
   |  InstantiationException
   |  InterruptedException
   |  NoSuchFieldException
   |  NoSuchMethodException
   |  RuntimeException
         |  ArithmeticException
         |  ArrayStoreException
         |  ClassCastException
         |  EnumConstantNotPresentException
         |  IllegalArgumentException
               |  IllegalThreadStateException
               |  NumberFormatException
         |  IllegalMonitorStateException
         |  IllegalStateException
         |  IndexOutOfBoundsException
               |  ArrayIndexOutOfBoundsException
               |  StringIndexOutOfBoundsException
         |  NegativeArraySizeException
         |  NullPointerException
         |  SecurityException
         |  TypeNotPresentException
         |  UnsupportedOperationException
```

21. ábra : A kivételosztályok szerkezete (részlet)

Az `Exception` osztály közvetlen kiterjesztései ellenőrzött kivételnek számítanak, míg a `RuntimeException` kiterjesztései nem ellenőrzöttnek. Egy adott típusú kivételt kiváltó osztály programozójának kényelmes, ha a kivételt nem ellenőrzött típusúnak definiálja, azonban az indokolatlanul ellenőrizetlennek nyilvánított kivételt kiváltó osztályt utóbb felhasználó másik programozónak, vagy az alkalmazás felhasználójának annál több kellemetlensége származhat. Ezért csak olyan kivételeket célszerű nem ellenőrzöttnek venni, amelyek elvileg bárhol előfordulhatnak.

A kivétel osztálya fontos szerepet játszik annak meghatározásában, hogy futási időben melyik kivételfeldolgozó blokk fogja elkapni (`catch`) az aktuálisan előállt kivételt. Egy kivételfeldolgozó blokk olyan kivételobjektumot tud elkapni, amelynek osztálya a feldolgozó blokk paraméterének osztályával azonos vagy a paraméter osztályának leszármazottja. A kivétel osztályának definíciójában a kivételt leíró attribútumokat (a hiba fellépésé-

nek körülményeit), az ezeknek értéket adó konstruktort és az olvasó metódusokat lehet megadni, illetve felüldefiniálni.

```
class MyException extends Exception {
  public String toString() {
     return "Hiba történt!";
     }
}
class MyException2 extends Exception {
  public MyException2(String s) {
     super(s);
     System.err.println("MyException2 hiba lépett fel!");
     }
}
```

7.2. A kivétel keletkezése, kiváltása

Legegyszerűbb esetben egy feltételes utasítással vizsgálunk egy helyzetet, és ha a kivételes állapot előáll, akkor a `throw` utasítással egy kivételt válthatunk ki (dobhatunk). Ekkor a vezérlés az objektumot tartalmazó osztályban megadott kivételkezelési utasításokat hajtja végre.

```
int a, b, c;
…
//a változók futelőzőleg értéket kaptak
if (b==0){ throw new MyException();}
else c = a/b;
```

Az előre megadott, vagy a saját osztállyal definiált kivételt az egyes metódusokhoz is hozzárendelhetjük. Ehhez a `throws` kulcsszóval a metódusfej specifikációjába illesztjük a kivételes eseményt lekezelő osztályt, vagy osztályokat.

A `RuntimeException` osztályba és ennek alosztályaiba tartozó kivételeket a java környezet automatikusan a metódusokhoz rendeli. Így az ezekbe az osztályokba tartozó és gyakran előforduló hibákat nem kell a metódusoknál megjelenítenünk, a virtuális gép a `RuntimeException` jellegű kivételeket szükség esetén automatikusan dobja.

```
<típus> metódusnév(<paraméterek>) throws KivételOsztály { }

…
public void metodus1() throws MyException {
  …
  if (feltétel) throw new MyException("Metodus1 hiba!");
  …
  }
```

7.3. A kivétel feldolgozása

Eddig még csak kiváltottuk a hibát, de szeretnénk is kezdeni valamit ezekkel a kivételekkel. A feldolgozáshoz a Java környezet szabványos vezérlési szerkezetet definiál.

Ha a `throw` utasítással előálló, vagy a környezet által automatikusan kiváltott kivételeket el szeretnénk kapni, akkor ahhoz a levédendő utasításokat, metódushívásokat egy `try {}` blokkba kell helyezni.

Az ilyen blokkban végrehajtott utasításokat a virtuális gép megpróbálja végrehajtani. Ha nem sikerül, akkor a közvetlenül a `try` blokk után álló `catch` szerkezet (vagy szerkezetek) segítségével megpróbálhatjuk elkapni a kivételkor keletkező objektumot.

A `try-catch` szerkezet legegyszerűbb váza a következő:

```
try {
   // Kivételt kiváltó blokk.
}
catch(KivételOsztály1 kivételobjektum){
  // A kivételt elkapó/feldolgozó blokk.
}
```

A `catch` blokkokból többet is definiálhatunk, ha a hibát többféleképpen szeretnénk kezelni. Utolsó blokként egy `finally` blokkot is definiálhatunk

```
try {
  // Kivételt kiváltó blokk.
}
catch(KivételOsztály1 kivételobjektum){
  // 1. típusú kivétel feldolgozása.
}
catch(KivételOsztály2 kivételobjektum){){
  // 2. típusú kivétel feldolgozása.
}
…
finally {
  // Záró blokk
}
```

A kivételt elkapó/feldogozó programrészt a megfelelő `catch` blokkban kell megírni. Egy `catch` blokk a paramétere típusának megfelelő osztályú kivételobjektumot tud elkapni, ami úgy értendő, hogy a kivételobjektum típusa azonos a paraméter típusával vagy leszármazottjával.

Ha a try blokk által kiváltott `k1` kivétel osztálya (`KivételOsztály1`) közvetlen vagy közvetett leszármazottja a `k2` osztályának

(KivételOsztály2), és a két kivételt különbözőképpen kell feldolgozni, akkor lényeges a catch blokkok sorrendje. Ilyenkor a k2 kivételt elkapó catch(KivételOsztály2 ...) blokk nem előzheti meg a k1 elkapására szánt catch(KivételOsztály1 ...) blokkot, mert akkor az a k1-et is elkapná, azaz a k1 kivétel nem jutna el oda, ahol szándékunk szerint fel kellene dolgozni.

Ha ugyanis valamelyik catch blokk megkapta a vezérlést, akkor azt már sem a try blokk nem kapja vissza, sem az adott try-catch-finally kivételkezelő szerkezeten belüli másik catch blokk sem kaphatja meg.

A kivételt elkapó catch blokk lefutása után finally blokk hiányában a futás kilép a kivételkezelő szerkezetből, különben a szerkezetből kilépést még megelőzi a finally blokk lefutása is.

Ha a try blokkban kiváltott ellenőrzött kivételek valamely típusára nem írtunk catch blokkot, akkor az ilyen típusú kivételt specifikálni a kell a metódus fejrészében!

A finally blokkban a try blokk olyan befejező tevékenységét kell programozni, amelyet végül mindenképpen végre kell hajtani, akár megszakította a try blokk futását a valamelyik kivétel, akár nem.

```
try {
  // Kivételt kiváltó blokk.
}
catch(ArrayIndexOutOfBoundsException e){
  // Speciális kivételt feldolgozó blokk.
}
catch(RuntimeException e){
  // Általános kivételt feldolgozó blokk.
}
catch(Exception e){
  // Minden kivételt feldolgozó blokk.
}
```

A kivételosztályok közötti leszármazási lánc a következő: Exception – RuntimeException – ArrayIndexOutOfBoundsException. A try-catch blokkokat a speciális kivételtől a minden kivételt feldolgozó blokkig adtuk meg. Ha fordítva definiálnánk a catch blokkokat, akkor sem az ArrayIndexOutOfBoundsException, sem a RuntimeException kivételeket nem tudnánk külön-külön lekezelni, hiszen az összes ilyen típusú kivételt a catch(Exception e) blokk kezelné le.

7.4. Kérdések

- Mit jelent a kivétel, kivételes állapot fogalma?
- Mi a különbség a hiba (error), és a kivétel (exception) között?
- Mi válthat ki kivételeket?
- Mi történik a throw utasítás után?
- Mi a try-catch-finally szerkezet koncepciója?
- Több catch ág definiálásakor milyen szempontok szerint kell az egyes feldolgozó ágakat megtervezni?

7.5. Feladatok

- A Sikidomok osztályokhoz készítsen saját kivételeket leíró osztályokat, amelyek az alábbi hibák esetén váltanak ki kivételt:
  - negatív, vagy zérus oldalhossz, vagy sugár megadása,
  - megszerkeszthetetlen háromszögek megadása,
  - olyan adatok megadása, amelyeknél a számított értékek már túlcsordulnak.

8. A Java platform hasznos lehetőségei

8.1. Szöveges adatok formázása

A Java platform 5. kiadásában számos olyan szöveges formázó utasítást definiáltak, amelyek az előző verziókból hiányoztak. A hagyományos Java szintaktikával egyszerű sztringműveletekkel, vagy összefűzéssel lehet a szöveges kimenetet formázni. A fejlesztői közösség unszolására bevezették az ANSI C nyelvben jól ismert formázott szövegkezelést (`printf`), amelyben egy műveletben ötvözték a megjelenítést és a formázást. A metódust a `java.io.PrintWriter` és `java.io.PrintStream` osztályokban helyezték el.

A metódus változó számú paramétert fogadhat. Az első paraméter a kiíratni kívánt szövegadat, amelyben különleges formázó karaktereket helyezhetünk el. A formázó karakterek (formátum specifikátorok) meghatározzák, hogy a további paraméterek milyen módon jelenjenek meg szöveges adatként. A formázó karakterek %-jellel kezdődnek, és valamilyen típust jelölő karakterrel végződnek. Általános formában:

```
%<szélesség><.pontosság>típuskarakter
```

A típuskarakterek jelölésére részben átvették a C nyelv szintaktikáját, részben kiegészítették a Java-specifikus formázási lehetőségekkel. A típuskaraktereket (nagy számuk miatt) öt fő kategóriába sorolták: általános, karakter, numerikus (egész vagy valós), dátum/idő, illetve egyéb típusok.
A szöveges formázó karakterek lehetséges értékeit a következő táblázat tartalmazza. (További részletekért érdemes a `java.util.Formatter` osztály specifikációját áttanulmányozni.)

Formázó karakter	Típus	Jelentés
%b, %B	általános	Ha a paraméter `null` értékű, akkor az eredmény "`false`" lesz. Ha elemi `boolean`, vagy `Boolean` objektum típusú, akkor az eretmény "`true`", vagy "`false`" értékű lesz.
%h, %H	általános	Ha a paraméter `null` értékű, akkor az eredmény "`null`" lesz. Különben az eredmény az hexadecimális értékű lesz, az `Integer.toHexString(arg.hashCode())` hívás eredményeként.
%s, %S	általános	Ha a paraméter `null` értékű, akkor az eredmény "`null`" lesz. Ha a paraméter megvalósítja a `Formatter` interfészt, akkor annak `formatTo()` metódusának kimenete lesz, különben az eredmény a paraméter `toString()` értéke lesz.
%c, %C	karakter	Az eredmény Unicode formátumú karakter lesz.
%d	egész	Az eredmény tízes számrendszerű, egész formátumú.
%o	egész	Az eredmény nyolcas számrendszerű, egész formátumú.
%x, %X	egész	Az eredmény hexadecimális egész formátumú.
%e, %E	valós	Az eredmény tízes számrendszerű, normálalakú, valós formátumú.
%f	valós	Az eredmény tízes számrendszerű, valós formátumú.
%g, %G	valós	Az eredmény tízes számrendszerű, rögzített pontosságú, normálalakú és valós formátumú.
%t, %T	dátum/idő	A dátum/idő formázások jelölésénél használt formázókarakter. Bővebben lásd: 0. fejezet.
%%	általános	A százalékjel '%' ('\u0025')
%n	általános	Platformfüggő sorvégjel. (CR, CR-LF)

5. táblázat: Szövegformázás

Nézzünk egy konkrét példát:

```
System.out.printf("%s felhasználó %d próbálkozás után
bejelentkezett. Utolsó látogatás ideje: %tc %n",
felhNev, belepesekSzama, utolsoLatogatasDatuma);
```

Itt %s a jól ismert szöveges helyettesítést, %d egy egész érték helyettesítését jelöli, a %tc egy `Date`, `Calendar`, vagy milliszekundumokban megadott időadatot helyettesít a lokális dátumformátummal, és legvégül a %n egy platformfüggő sorvégjelet jelenít meg.

A `PrintWriter` és a `String` osztályok ezeken felül rendelkeznek egy `format()` nevű osztálymetódussal. Míg a `printf()` egy megadott kimenetre küldi a formázott karakterláncot, addig a `format()` a megformázott sztringet visszaadja. Így a fenti szöveg inicializálható új szöveges változóként az alábbi módon:

```
String uzenet = String.format("%s felhasználó %d próbálkozás
után bejelentkezett. Utolsó látogatás ideje: %tc %n",
felhNev, belepesekSzama, utolsoLatogatasDatuma);
```

8.2. Naplófájlok használata (logging)

Az egyszerű konzol alkalmazásoknál, amikor a program csak a standard kimenetre dolgozik, egyszerű a figyelmeztetések és a hibák jelzése. A fejlesztés, majd a működés szakaszában a `System.out.print`, vagy a `System.err.print` hívásokkal a programok könnyen jelezhetik állapotaikat, a felmerülő hibákat a fejlesztő, vagy a felhasználó felé.

Ha azonban ha egy kliens-szerver alkalmazást, vagy egy összetett projektet írunk, akkor ez már nem használható ki ilyen könnyen. A szerverprogramok jellemzően egy távoli gépen futnak, amelyhez esetlen nincs is karakteres terminál kapcsolva, így például a karakteres konzolüzeneteknek nem igazán használhtóak. Ezekben az esetekben egy más megoldást kell választanunk.

A Java 1.4-es kiadása [11] óta lehetőségünk van a `java.util.logging` csomag használatára, amely a Java programok szabványos naplógenerálását teszi lehetővé. A naplóképzés általánosan a következő elven működik: A fejlesztő egy `Logger` objektumot hoz létre, amelyet összekapcsol a figyelni kívánt osztállyal, vagy csomaggal. Ezután ez az osztály képes az alábbiakban tárgyalt beépített, hétszintű üzenetkezelő mechanizmust hívni, abban figyelmeztető, vagy információs bejegyzéseket elhelyezni az alkalmazás belső állapotairól. Ezek az üzenetek számos szabványos formában megje-

leníthetők: a memória egy elkülönített részén, a konzolon, egyszerű szöveges, vagy XML formázott fájlban, hálózati socketen keresztül, vagy ezek valamely kombinációjaként.

A naplózási mechanizmus segít a programok fejlesztési, telepítési és felhasználói szakaszában a részletes hibadetektálásban, és a nyomkövetésben.

A `java.util.logging` csomag főbb osztályai:

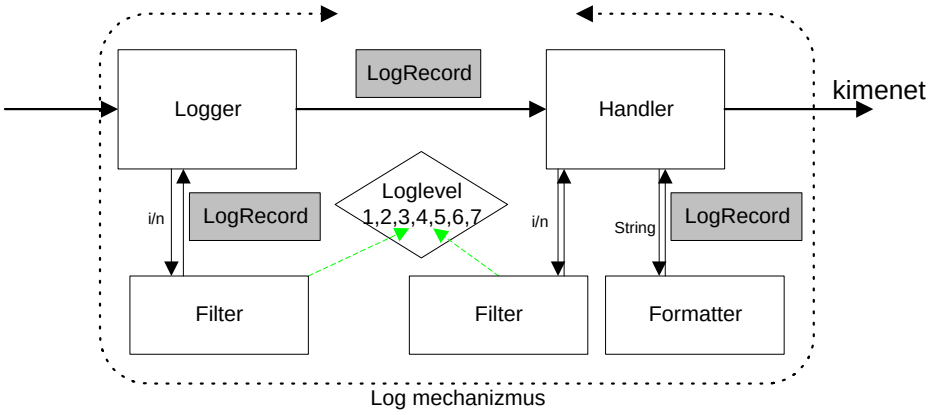
- **Logger:** A naplókezelés főosztálya. Az alkalmazások a bejegyzéseket ezen osztály példányain keresztül kezdeményezik. Számos metódussal rendelkezik, melyek a megfelelő szinteken a naplóba az adott üzeneteket bejegyzik.
- **LogRecord:** A LogRecord objektumok maguk az egyes logbejegyzések vivőobjektumai, a naplózó mechanizmus ezeket küldi a Handler felé.
- **Handler:** Feladata a logbejegyzések valamilyen kimenetre való elküldése (memória, fájl, socket stb.). Számos leszármazottja ismert. (`MemoryHandler`, `ConsoleHandler`, `FileHandler`, `SocketHandler`, `treamHandler`.)
- **Level:** Az általános naplóbejegyzésekhez használt jelzésszintek. A programban beállítható, hogy a `Logger` és a `Handler` milyen szintű és a szint feletti logrekordokat kezeljen.
- **Formatter:** A logrekord objektumok a kimenetre küldés előtt a `Formatter` osztályok segítségével áformázhatók, átalakíthatók. Jelenleg két implementációja adott, a SimpleFormatter és az XMLFormatter.
- **LogManager:** A virtuális gép globális objektuma. Felelős a `Logger` objektumok kezeléséért. Így egy nagyobb program osztályai (akár más csomagból is) ugyanazokat a loggereket használhatják.

A naplókezelés a következő modell szerint értelmezhető:

Az alkalmazás indulásakor a `LogManager` inicializálja a `Logger` objektumok névterét. Amennyiben egy konfigurációs fájlban van megadott logmenedzser bejegyzés, akkor az azokban megadott értékekkel inicializálja.

A programból csak a `Logger` objektumhoz érkezhetnek üzenetek. Minden egyes `Logger` objektum rendelkezik egy névvel, és egy naplózási küszöbbel. A naplózási küszöb és afeletti szintű üzenetekből a `Logger` objektum egy `LogRecord` objektumot állít elő és elküldi a megadott `Handlernek`. A `Handler` ezt a `Logrecord`-ot a megadott `Formatter`

segítségével megformázza, majd elküldi az osztályának megfelelő kimeneti csatornára. (A `Handler` rendelkezhet a `Logger`-től különböző naplózási küszöbbel, ez a `setLevel()` metódussal módosítható, amelyet a `Filter` osztályban definiál a Java környezet.)



22. ábra: A Java naplózási mechanizmusa

A naplózási szintekből a Java API hetet rögzít előre. Ezeket a `Level` osztály definiálja konstansként: `SEVERE` – a legmagasabb szint a rendszerszintű hibák jelzésére,

`WARNING` – a figyelmeztető üzenetek szintje,
`INFO`,
`CONFIG` ,
`FINE`,
`FINER`,
`FINEST` – a legalacsonyabb prioritású szint, nyomkövetési üzenetekre.

Azokban a forrásokban, ahol a loggolást használni szeretnénk, természetesen importálnunk kell a `java.util.logging` csomagot. Ezután egy `Logger` és egy `Handler` objektumot kell létrehoznunk. A programban a logger és handler objektumot össze kell kapcsolnunk, illetve beállíthatjuk a naplózás szintjét, és a handlerhez hozzárendelhetünk egy szűrőt is. Nagy előnye ennek a mechanizmusnak, hogy a `Logger` objektumhoz több handlert is rendelhetünk, így a naplózás több ágon, több módszerrel is folyhat egyidőben.

Az alábbi mintapéldában egy fájl-alapú naplózást láthatunk működés közben.

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

public class LogTeszt{
  // Logger objektum inicializálása
  private Logger naplo = Logger.getLogger("sajatnaplo");
  //a naplo bejegyzéseket megjelenítő handler objektum
  private Handler kezelo;

  public void init() throws Exception{
    //A naplo bejegyzéseket egy fájlba szeretnénk kiírni
    kezelo = new FileHandler("valami.log");
    // a logger érzékenységének beállítása
    // (SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST),
    naplo.setLevel(Level.FINEST);
    // amely szöveges formában jeleníti meg a bejegyzéseket
    kezelo.setFormatter(new SimpleFormatter());
    // és végül a loggert összekapcsoljuk a handlerrel
    naplo.addHandler(kezelo);

    // logbejegyzések generálása
    naplo.info("Elindult a loggolás!");
  }

  public void muvelet(int a, int b) {
    naplo.fine("Algoritmus indul a="+a+", b="+b );
    if (a>b){
       naplo.warning("Az a nagyobb, mint b!");
    }else if (a==b){
       naplo.warning("Az a egyenlő b-vel!");
    } else {
      naplo.warning("A b nagyobb, vagy egyenlő mint a!");
    }
    naplo.fine("Itt vége a hasonlító algoritmusnak!");
  }
```

```
    public int kivetel(int a, int b )throws
        ArithmeticException{
      // Belépési üzenet FINEST szinten
      naplo.entering("LogTeszt","kivetel");
      // kivételkezeléshez használt loghívás FINEST szinten!
      if (b==0) naplo.throwing("LogTeszt", "kivetel",
        new ArithmeticException());
      // Kilépési üzenet FINEST szinten
      naplo.exiting("LogTeszt", "kivetel");
      return a/b;
    }

    public static void main(String[] args) throws Exception{
       LogTeszt pelda = new LogTeszt();
       pelda.init();
       pelda.muvelet(12, 13);
       try {
         pelda.kivetel(13, 0);
       } catch (Exception e){
         e.printStackTrace();
       }
    }
}
```

8.3. Dátum és időkezelés

A dátum és időadatok ábrázolásához a Java keretrendszer a kezdetektől definiált típusokat. Az időadatokat alapvetően long típusú egészként képzelhetjük el, melyek a UNIX konvenció szerint az 1970 óta eltelt időt szimbolizálják milliszekundumban. Mára az alaptípusokat kiegészítették, így a legteljesebb igényeket is kielégítik. A típus alapját a `java.util.Date` osztály jelenti, melyben konstans időadatokat tárolhatunk. A `java.util.Calendar` osztály és leszármazottai segítségével pedig a dátum és időadatokon végezhetünk el különböző konverziós műveleteket, számításokat.

Az időadatok kezelése egyrészt a csillagászati megfigyeléseken, másrészt a nagyon pontos atomórákon alapul. A csillagászati alapú időszámítást Európában a GMT (Greenwich Mean Time) rendszer adja, melyben egy nap $24 \times 60 \times 60 = 86400$ s. Azonban megfigyelték, hogy a Föld forgása nem egyenletes, így a GMT tudományos szempontokból nem a legpontosabb. Az atomórák korára sikerült pontosítani ezt az időszámítási alapot, így az UTC (Universal Time Converter) számításban egy évben kétszer (jún. 30.

és dec. 31.) a nap egy-egy extra másodperccel hosszabb. A Java virtuális gépek alapértelmezésként ezt az UTC formátumot használják.

8.3.1. A dátumkezelő osztályok

Date

Az időadatokat az 1970.00.00 00:00:00:000 időponttól számítva milliszekundumokban tárolja. Csak konstans időadatok kezelésére szolgál. Számos metódusa már elavult, mára már csak a `toString()`, az `equals()`, a `before()`, az `after()` használhatóak. A paraméterek nélküli `Date()` konstruktorhívás az aktuális gépidővel inicializált dátumadatot tartalmazó objektumot szolgáltatja.

Calendar

A `Date` osztály kiegészítéseként hozták létre, amely az időadat tárolásán kívül egyes helyfüggő szolgáltatásokat is képes kezelni (pl. időzónák). Az osztály példányaiból lekérdezhető, beállítható a kívánt dátumérték, és különböző konverziók végezhetők el rajtuk.

A számos előre definiált konstans érték segítségével egy konkrét naptáradat egyes összetevői lekérdezhetők. Legfontosabb metódusok a `get()`, a `set()` és az `add()`. Az add metódussal lehet például az egyes dátumértékhez valamilyen értéket hozzáadni, melyet értékhelyesen, a szökőéveket figyelembe véve végez el. Pl: 2006.12.11 +3 hónap = 2007.3.11 lesz.

TimeZone

A `TimeZone` osztály a hordozható programok, más dátum és időformátumokra való inicializálásra és konvertálásra használható. A `TimeZone` absztrakt osztály, nem példányosítható. Használatához a `TimeZone.getDefault(zóna)` hívás szükséges, ahol a zóna egy szöveges adatként megadott időzóna érték. Az inicializáláshoz számos előre rögzített érték tartozik „Europe/Budapest”, „America/Los_Angeles” formában. Azonban a „GMT+01”, vagy a „GMT+0010” formátum is használható, melyekben a greenwichi időzónát 1 órával, illetve 10 perccel toltuk el.

Az európai időszámítási rendszerhez speciálisan használható még a `SimpleTimeZone` osztály is, amely a Gergely-naptárakhoz olyan könnyen használó funkciókat nyújt, mint például a téli/nyári időszámítás pontos kezelése.

Dátumadatok formázott megjelenítése

A `Date` és `Calandar` objektumok rendelkeznek már előre definiált `toString()` metódusokkal, így a dátumadatok szöveges reprezentációi adottak. Azonban a megjelenítéshez UTC formátumot használnak, amely sokszor túlzottan részletes, és nehezen olvasható: `"Fri Sep 08 13:31:05 CEST 2006"`. A szöveges adatok formázásánál láttuk, hogy a `printf()` és a `format()` metódusok alkalmasak dátum/idő alakú formázásra is a `%t...` formázó karakter segítségével. Az alábbi táblázatban ezeket a formátum specifikátorokat tekintjük át.

Formázó karakter	**Jelentés**
(időadatok)	
`%tH`	Óraadat 24 órás formátumban, vezető nullákkal feltöltve, 00 - 23.
`%tI`	Óraadat 12 órás formátumban, vezető nullákkal feltöltve, 01 - 12.
`%tk`	24 órás óraadat, vezető nullák nélkül, 0 - 23.
`%tl`	12 órás óraadat, vezető nullák nélkül, 1 - 12.
`%tM`	Percadat, vezető nullákkal feltöltve, 00 - 59.
`%tS`	Másodpercadat, vezető nullákkalfeltöltve, 00 - 60 (A "60" olyan speciális érték, amely az UTC korrekciós másodperceket jelzi).
`%tL`	Három karakteres milliszekundum adat, vezető nullákkal feltöltve, 000 - 999.
`%tN`	Kilenc karakteres nanoszekundum adat, vezető nullákkal feltöltve 000000000 - 999999999.
`%tZ`	Az aktuális időzóna szöveges értéke.
`%ts`	Az UTC kezdőórája óta eltelt másodpercek (1970. 01. 01 00:00:00 UTC) Értékei `Long.MIN_VALUE/1000` és `Long.MAX_VALUE/1000` közöttiek lehetnek.
`%tQ`	Az UTC kezdőórája óta eltelt idő milliszekundumban. Értékei `Long.MIN_VALUE` és `Long.MAX_VALUE` közöttiek lehetnek.

6. táblázat: Dátum- és időformázás

Formázó karakter	Jelentés
(Dátumadatok)	
`%tB`	Helyspecifikus, teljes hónapnév pl. "January", vagy „január”. (A formátum a helyspecifikus beállításokat követi, bővebben lásd `Locale` osztály)
`%tb`	Helyspecifikus, rövidített hónapnév pl. "Jan", vagy „jan.”.
`%tA`	Helyspecifikus, teljes alakú napnév pl. "Sunday", "Monday" , vagy „vasárnap”
`%ta`	Helyspecifikus, rövid formátumú napnév. "Sun", "Mon" , vagy „V”, „H”
`%tC`	A négyjegyű évszámok első két jegyét jelölő számadat, vezető nullákkal feltöltve, 00 - 99
`%tY`	Négyjegyű évszámadat, vezető nullákkal feltöltve, 0000 – 9999.
`%ty`	Kétjegyű évszámadat, vezető nullákkal feltöltve, 00 - 99.
`%tj`	Az év napjának sorszáma, vezető nullákkal feltöltve, 001 – 366.
`%tm`	Kétjegyű hónapadat, vezető nullákkal feltöltve, 01 - 13.
`%td`	Kétjegyű napszám adat, vezető nullákkal feltöltve, 01 - 31
`%te`	Kétjegyű napszám adat, vezető nullák nélkül, 1 - 31.
(Vegyes dátum/idő formázások)	
`%tR`	24 órás időadat, megegyezik a `"%tH:%tM"`-vel
`%tT`	24 órás időadat, megegyezik a `"%tH:%tM:%tS"`–vel.
`%tr`	12 órás időadat, megegyezik a `"%tI:%tM:%tS %Tp"`-vel.
`%tD`	Dátumadat, amely megegyezik a`"%tm/%td/%ty"`-vel.
`%tF`	ISO 8601 formájú dátumadat, megegyezik a `"%tY-%tm-%td"`-vel.
`%tc`	UTC időformátum, megegyezik a `"%ta %tb %td %tT %tZ %tY"`-vel.

7. táblázat: Dátum- és időformázás (folytatás)

Az alábbi példa bemutatja a nyelv dátum- és időkezelésének legfontosabb elemeit:

```
import java.util.Date;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.TimeZone;
import java.util.SimpleTimeZone;

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

// Melyik időzónák megfelelőek nekünk, magyaroknak? (1 óra
Greenwich-től)
  String[] megfelelo=
     TimeZone.getAvailableIDs(1*60*60*1000);
  System.out.println("Greenwich-től keletre 1 órányira
     levő időzónák:");
  for(int i=0;i<megfelelo.length;i++){
     System.out.println("\t" + megfelelo[i]);
  }
  System.out.println("--------------------------");


  //Vesszük azt az időzónát, amit hivatalosan használunk
  //(Central European Time = GMT+1):
  SimpleTimeZone cet = new SimpleTimeZone(1,"CET");
  //Beállítjuk a nyári időszámítás kezdetét április
  //első vasárnapjára, hajnali 2 órára
  cet.setStartRule(Calendar.APRIL, 1, Calendar.SUNDAY,
    2*60*60*1000);
  // ... és a végét október utolsó vasárnapjára, 2 órára.
  cet.setEndRule(Calendar.OCTOBER, -1, Calendar.SUNDAY,
    2*60*60*1000);

  //Ezzel az időzónával létrehozunk egy
  //Gergely-féle időszámítást használó naptárat:
  Calendar calendar = new GregorianCalendar(cet);
  //Az aktuális időpillanatot lekérdezzük,
  Date most = new Date();
  // majd a naptárat beállítjuk erre az időpontra.
  calendar.setTime(most);

  // ezután jön a kiíratás a Calendar osztály konstansaival
  System.out.println("Az aktuális idő:");
  System.out.print(calendar.get(Calendar.ERA)==0 ? "i.e. "
      : "i.sz. ");
  System.out.print(calendar.get(Calendar.YEAR) + ".");
  System.out.print((calendar.get(Calendar.MONTH)+1) + ".");
  System.out.print(calendar.get(Calendar.DATE) + " ");
```

```
        System.out.print(calendar.get(Calendar.AM_PM)==1?"du "
            :"de ");
        System.out.print((calendar.get(Calendar.HOUR)+1) + ":");
        System.out.print(calendar.get(Calendar.MINUTE) + ":");
        System.out.print(calendar.get(Calendar.SECOND) + " mp. ");
        System.out.println(calendar.get(Calendar.MILLISECOND) +
           " ezredmp.");
        System.out.println("Ez a "+
        calendar.get(Calendar.DAY_OF_YEAR)+
            ". nap ebben az évben.");

        System.out.println("A hónap" +
           calendar.get(Calendar.WEEK_OF_MONTH) +
           "., a teljes év " +calendar.get(Calendar.WEEK_OF_YEAR)+
           ". hetében járunk. ");


        // három hónap múlva milyen nap lesz?
        System.out.println("Három hónap múlva ilyenkor: ");
        calendar.add(Calendar.MONTH, 3);
        System.out.println(calendar.get(Calendar.YEAR)+ "."
           +(calendar.get(Calendar.MONTH)+1) + "."
           +calendar.get(Calendar.DATE));
    }
}
```

8.4. Időmérés nanoszekundumokban

Az alapértelmezett legkisebb időmérték, melyet a dátumosztályok használnak, a milliszekundum. Azonban korlátozásokkal, a futásidejű műveletek mérésére nanoszekundumos felbontást is használhatunk. A java 5. verziójától a `System` osztály rendelkezik egy `currentTimeMillis()` és egy `nanoTime()` metódussal. Ezek a metódusok egy-egy long típusú értékkel szolgálnak, melyek a relatív futásidőt szimbolizálják. Algoritmusok számításigényének mérésére hasznos segítséget nyújthatnak.

```
long start = System.nanoTime();
muvelet();
long end = System.nanoTime();
long elteltIdo = end-start;
```

8.5. Tömbműveletek

A 3.2.9. fejezetben megismerkedtünk a Java tömbkezelési módszereivel. A sztringkezelés során megismertük azokat a metódusokat, amelyek a szöveges adatokat bájt-, vagy karaktertömbbe konvertálja, illetve ilyen tömbök-

ből állít elő sztringadatokat (3.2.10.). Sok esetben azonban a tömbökkel más műveleteket is szeretnénk elvégezni.

A sokszor használt `java.lang.System` osztály rendelkezik egy statikus `arraycopy()` metódussal, amely azonos típusú tömbök közötti adatmásolást tesz lehetővé. Az adatmásolás a tömb egészét, vagy egy tartományát érintheti. Ha nem azonos típusokkal próbálkozunk, akkor ezt a fordító „`incompatible types`" hibaüzenettel jelzi.

Másoljuk át egy karakteres tömb egy résztartományát egy másik tömbbe.

```
char [] szoveg = "Java Standard Edition 5.0".toCharArray();
char [] masolat = new char[100];
char [] masolat2 = new char[100];
// A szöveg 5. karakterétől kezdődően a másolatba,
  a 0. karaktertől kezdve 16 karaktert másolunk.
System.arraycopy(szoveg, 5, masolat, 0, 16);
// A masolat tömb 9. karakterétől  másolat2-be,
 az 5. elemtől kezdve a 7 karaktert átmásolunk.
System.arraycopy(masolat, 9, masolat2, 5, 7);
//A tömbök tartalma:
for (int i=0; i<szoveg.length; i++)
  System.out.print(szoveg[i]);
  // Kimenet: Java Standard Edition 5.0
System.out.println();
for (int i=0; i<masolat.length; i++)
  System.out.print(masolat[i]);
  // Kimenet: Standard EditionNULL,NULL...
System.out.println();
for (int i=0; i<masolat2.length; i++)
System.out.print(masolat2[i]);
  // Kimenet: NULL,NULL,NULL,NULL,Edition,NULL,NULL...
System.out.println();
```

Megfigyelhető, hogy a másolat tömb azon részei, ahova nem került értelmes karakter, a 0 ASCII kódú karakterrel inicializált.

A `java.util` csomag tartalmaz egy `Arrays` nevű osztályt. Ez az osztály azokat a könnyen és egyszerűen használható tömbműveleteket foglalja magába, mint a tömbök inicializálása megadott kezdőértékekkel, a bináris keresés, a rendezés, és az értékek szerinti összehasonlítás.

A műveletek osztálymetódusokként lettek definiálva, így használatuk a következő alakú: `Arrays.muvelet(paraméterek)`. Az `Arrays` osztály használatához természetesen a forrásprogram elején importálni kell:

```
import java.util.Arrays;
```

A tömbmanipulációs műveletek alapja a tömb elemeinek összehasonlíthatósága. A Jva alaptípusait használó tömbök esetén (`boolean`, `int`, `double`, `char` stb.) a rendszer gondoskodik a típusnak megfelelő, ún. alapértelmezett sorrendről, így ezeknél általában nem ütközünk akadályokba. Ugyanígy sztringek esetén is létezik egy alapértelmezett sorrend, hiszen a `String` osztályt az API eleve összahasonlíthatónak definiálja.

Objektumok használata esetén azonban már a programozó felelőssége az objektumok összehasonlíthatóságának biztosítása. Ehhez az Arrays osztály megköveteli a `Comparable`, illetve a `Compare` interfész implementálását, és annak `compateTo()`, vagy `compare()` metódusának kifejtését. (Bővebben lásd a 0. fejezetben.)

Az `Arrays` osztály leggyakrabban használt művelete a rendezés. A `sort()` metódus segítségével rendezhetjük a tömb típusa szerinti természetes sorrendbe. A `sort` műveletet meghívhatjuk az egész tömbre, illetve a tömb egy intervallumára.

A másik gyakori művelet egy elem kikeresése. Az `Arrays` osztályban a bináris keresés algoritmusát implementálták a fejlesztők.

Nagyon fontos azonban tudni, hogy a bináris kereső algoritmus csak rendezett tömb esetén működik, rendezetlen tömb esetén meghatározhatatlan a végeredmény!

A bináris keresés ugyanúgy, mint a rendezés feltételezi, hogy a tömb elemi összehasonlíthatók. A fordító itt is megköveteli a megfelelő interfészek implementálását. A `binarySearch()` metódus két paramétert fogad: magát a rendezett tömböt, és a keresett értéket. A keresett érték típusának meg kell egyezni a tömb elemeinek típusával, így sok esetben explicit típuskonverziót kell használnunk.

Rendezzünk egy egész számokból álló tömböt, majd keressük meg egyes konkrét értékek helyét (indexét)!

```
short [] egesztomb = new short[] {12, 3, -5, -1, 15, 0};
Arrays.sort(egesztomb);      // {-5, -1, 0, 3, 12, 15}
for (int i=0; i<egesztomb.length; i++)
  System.out.print(egesztomb[i]+ " ");

int index = Arrays.binarySearch(egesztomb,(short)3);  // 3.
System.out.println("\nA 3 pozíciója: "+ index);
index = Arrays.binarySearch(egesztomb,(short)100);
// nem található, negatív visszatérési érték
System.out.println("A 100 pozíciója:" +index);
```

A fenti példában a sort metódust az egész tömbre alkalmaztuk. Abban az esetben, ha a tömbnek csak egy részintervallumát rendezzük, álljon például a következő forráskód-részlet:

```
Arrays.sort(egesztomb, 2,4); //{12, 3, -5, -1, 15, 0}
```

A paraméterként megadott két egész szám a rendezendő tömbtartomány indexeit jelzi.

Referenciatömbök esetén a fenti műveletek ugyanígy használhatók.

Az alábbi példában bemutatjuk az `Arrays.equals()` metódus működését tömbök esetében. Az összehasonlítás során akkor lesz egyenlő a két tömb, ha annak minden elempárja megegyezik. Ehhez szükséges, hogy az objektumok rendelkezzenek felüldefiniált `equals()` metódussal. (Az `Arrays.equals()` metódus az objektumok saját `equals()` metódusait fogja meghívni.)

Példánkban a `String` objektumok rendelkeznek `equals()` metódussal, amely akkor tér vissza igaz értékkel, ha a két karakterlánc minden karaktere megegyezik.

```
// Referenciatömbök tömbműveletei
String [] szovegtomb = new String[] {"itt", "az", "idő"};
//másolatkészítés az Object.clone() metódussal
String [] masolata = (String [])szovegtomb.clone();
Arrays.sort(szovegtomb);  //{"az", "idő", "itt"}
for (int i=0; i<szovegtomb.length; i++)
   System.out.print(szovegtomb[i]+" ");
// megvizsgáljuk a két tömböt
boolean egyezik = Arrays.equals(szovegtomb, masolata);
System.out.println("\nA két tömb "+
   (egyezik?"megegyezik!":"nem egyezik meg!"));
```

Az `Arrays` osztály használható többek között még adatfeltöltésre, és kezdőértékadásra, amelyet a `fill()` metódus segítségével az lábbi módon adhatunk meg:

```
byte[] data = new byte[100];  // üres tömb, minden elem 0
kezdőértékű
Arrays.fill(data, (byte)-1); // minden elem -1 értékű

Arrays.fill(data, 3, 7, (byte)-5); // tartomány feltöltése:
//{-1, -1, -1, -5, -5, -5, -5, -5, -1, -1,.....}!
```

8.6. Kérdések

- Melyek a hasonlóságok és a különbségek az ANSI C és a Java nyelv szövegformázó lehetőségei között?
- Milyen célra használjuk a naplózó fájlokat?
- A naplózásnak milyen kimeneti csatornái lehetnek?
- Mi a különbség a Logger és a Handler osztály között?
- Mi jellemzi a Java dátum- és időkezelését?
- Melyik osztály segítségével tudunk különleges tömbműleteket végezni, és hogyan?

8.7. Feladatok

- Az Alkalmazott osztályt egészítse ki egy születési dátumot tartalmazó adattaggal, és módosítsa az érintett metódusokat.
- Készítsen egy tetszőleges feladathoz egy komplett naplózó szolgáltatást.

9. Fejlett adatszerkezetek

A programtervezési munkában gyakori feladat a rögzített objektumok valamilyen szempont szerinti csoportosítása, rendezése, adott tulajdonságú elem megkeresése, illetve a tárolt elemek karbantartása (új elemek felvétele, módosítás, törlés stb.). Az elemek karbantartásához és az elemek kereséséhez az elemeket valahogyan azonosítanunk kell, vagyis meg kell tudnunk állapítani egy elemről, hogy azt keressük-e (referenciája, vagy valamely tulajdonsága alapján). Továbbá a rendezéshez ezeket az elemeket össze kell tudni hasonlítani, vagyis meg kell állapítanunk, hogy melyik objektum szerepel majd a sorban előbb, illetve utóbb.

A programozási gyakorlatban számos adattárolási módszer és rendezési algoritmus létezik, ezeket többé-kevésbé a Java környezet is támogatja. A klasszikus adatmodellek és a rajtuk értelmezett eljárásmodellek az objektumorientált rendszerekre is kiterjeszthetők. A magasabb fokú adatszerkezetek – amelyek majd az egyes objektumokat tárolják – kiegészíthetőek az adatszerkezetet manipuláló műveletekkel. Zárt egységbe rendezhetőek, vagyis megadhatóak rájuk interfész-, illetve osztály definíció.

A magasabb fokú adatszerkezeteket az alábbi modell szerint rendszerezhetjük:

- **Asszociatív adatszerkezetek**: az elemek közötti laza kapcsolatokat az elemek azonos tulajdonságai létesítik.
  - Tömb, dinamikus tömb,
  - Indextábla,
  - Halmaz.
- **Szekvenciális adatszerkezetek:** az egyes elemek szigorúan egymást követve helyezkednek el. Mindig van egy kezdőelem, és minden elemet egy másik követ. A kapcsolat 1:1 jellegű.
  - Verem (LIFO),
  - Sor (FIFO),
  - Kétirányú és cirkuláris listák.
- **Hierarchikus adatszerkezetek:** az elemek egymással alá-fölérendeltségi viszonyban vannak. A kapcsolatok 1:N alakúak.
  - Fa (bináris-, többágú-, feszítőfa, kupac)
  - Összetett listák

- **Hálós adatszerkezetek:** bármelyik csomópont bármelyik csomóponttal kapcsolatban állhat. A kapcsolatok N:M alakúak.
  - Irányított gráf
  - Hálózat

9.1. A gyűjtemény keretrendszer

A belső osztályok tárgyalásakor már láttunk egy példát az egyirányú lista egy lehetséges megvalósítására (lásd: 5.4. fejezet). A Java osztálykönyvtárakban a `java.util` csomagban megtalálható a „Gyűjtemény keretrendszer”(Collection Framework), amely a legfontosabb és gyakran használt adatszerkezetekre kész modelleket ad. A Java osztálykönyvtárak 1.5-ös kiadásakor ezt a csomagot alaposan kibővítették.

A legjellemzőbb gyűjteményfajták a tömbökre, dinamikus tömbökre és az objektumok referenciákon keresztül történő összekapcsolásán alapulnak.

A gyűjtemény keretrendszer felépítési elve a következő:

Az *interfészek* hierarchiája a megfelelő absztrakt adattípusokat specifikálja. Az őket megvalósító *osztályok* pedig tartalmazzák a konkrét adat- és műveletreprezentációkat. Az interfészek és a megvalósító osztályok szétválasztására azért van szükség, hogy a magas szintű objektumorientált modellezést (OOA, OOD) a tartalmazott adatoktól függetlenül tegyék lehetővé.

Az elkészült típusok hosszas tervezési és tesztelési folyamat eredményei, így használatuk növelheti a fejlesztendő programok megbízhatóságát. Az alábbiakban megismerkedünk a magasabb szintű összetett adatszerkezetek alapfogalmaival, majd áttekintjük a Java csomag egyes interfészeit, osztályait.

9.1.1. A konténer

Konténer objektumoknak nevezzük az 1:N adatszerkezeteket megvalósító összetett osztálypéldányokat. Egy konténerosztály az adatstruktúra tárolásán kívül a keresési, bejárási és karbantartási funkciókat is megvalósítja. (Első közelítésben egy egyszerű egydimenziós tömb is konténernek számíthatana, de több szempont szerint sem beszélhetünk konténer viselkedésről, hiszen a tömb nem tekinthető osztálynak, és a karbantartó műveleteket is csak az adatszerkezettől elválasztva lehetne megadni.)

A Java gyűjtemény keretrendszerében a konténer osztályok magas szintű modelljének vázát és működésének alapelemeit a `Collection` interfész tartalmazza.

A konténer olyan objektum, amely objektumokat tárol, és alkalmas különböző karbantartási, keresési és bejárási funkciók megvalósítására.

Modellezési és programfejlesztési szempontból a fejlesztő feladata azt a konténerosztályt kiválasztani, ami az adott feladat megoldására a legalkalmasabb. Ha nincs a feladatra alkalmas konténer, akkor vagy egy már meglévő osztály leszármaztatásával az osztályt továbbfejlesztjük, vagy nagyon ritkán egy teljesen újat írunk.

A Java osztálykönyvtárai általánosak és bármilyen osztályból származó objektumot tartalmazhatnak. A konténerbe felvett objektumok ezen általánosság miatt `Object` osztályúak lesznek. Így egy konténer elemeinek metódusait csak típuskényszerítéssel érhetjük majd el:

```
((Osztálytípus)elemelérés).metódus();
```

A típuskényszerítés során a zárójelezés azért szükséges mert a `"."` operátor magasabb prioritású, mint a típuskényszerítés.

Megjegyzés: A generikus típusok alkalmazásával elhagyhatók lesznek az explicit típuskonverziók, így a sokszor olvashatatlanná váló kód kiegyszerűsíthető (lásd: 9.4. fejezet).

9.2. A gyűjtemény keretrendszer interfészei

Először az absztrakt modellel, vagyis az interfészekkel kell megismerkednünk. Itt adható meg a tárolt adatoktól független működés, és itt írható elő a számtalan leszármazott osztály számára – az azonos üzenetküldés. A polimorfizmus segítségével az egyes leszármazott osztályokban megadhatóak lesznek a speciális kereső, rendező és bejáró algoritmusok. A keretrendszer az alábbi interfészeket definiálja:

```
Collection
   └ List
   └ Queue
   └ Set
      └ SortedSet
Map
   └ SortedMap
```

A konténer osztályokat csoportosíthatjuk abból a szempontból, hogy az az adatokat milyen adatstruktúra szerint tárolja:

- Kollekció-jellegű: A kollekció jellegű konténer egyszerű soros adattárolást valósít meg. Ilyen osztályok a `Vector`, `HashSet`, `TreeSet`, `ArrayList`, `LinkedList` stb.
- Leképezés-jellegű: Egy leképezés-jellegű konténerben a nyilvántartás kulcsobjektumok szerint történik. A kulcshoz információhordozó objektumok rendelhetők. A kulcsok alapján a keresési és karbantartási munkák hatékonyabbak. Ide tartoznak a `HashMap`, `HashTable`, `TreeMap` stb.

Tekintsük át, hogy a Java osztálykönyvtárakban definiált interfészek milyen lehetőségeket kínálnak a konténer jellegű adattároláshoz:

- A **Collection** interfész a kollekció jellegű osztályok őse, gyűjtőmodellje. A belőle származó interfészek, majd osztályok az adattárolást más-más módon valósítják meg. Hol engedélyezett az elemek többszörös felvétele, hol nem. Valahol a tárolás rendezett, míg máshol rendezetlen.
- A **List** interfész a dinamikus listákat tartalmazó osztályok közös interfésze. Ezekben a konténerekben az elemek sorrendje szigorúan soros, az elemek indexelhetőek, az adatok nem rendezettek.
- A **Queue** interfész az adatsorok, elsősorban a FIFO jellegű tárolók használatához nyújt modellt. Egyes implementáló osztályok a sorokat prioritásos sorként kezelik, vagyis értelmezik a sorok rendezését.
- A **Set** interfész a halmaz-jellegű adattárolás magas szintű megvalósítása. Egy halmazba csak egyedi adatok tehetők, két azonos objektum nem tárolható.
- A **SortedSet** interfész a rendezett halmaz-jellegű konténerosztályok működését írja le. Mint a Set interfész leszármazottja, az őt implementáló osztályokban is az adattárolás egyedi.

- Az kollekcióktól független leképezés jellegű interfészek kezelése. A **Map** (leképezés) interfészt implementáló osztályok kulcs-érték párok alapján tárolja a bejegyzéseket. A kulcsokat valamilyen kollekcióban tárolják, segítségükkel érhetőek el a tartalmazott objektumok. (Mind a kulcsok, mind az értékek lehetnek elemi, vagy objektum típusúak.)
- A **SortedMap** a Map interfész leszármazottja, az őt implementáló osztályokban lehetőségünk van a kulcsok rendezésére.

9.3. A gyűjtemény keretrendszer osztályai

A gyűjtemény keretrendszer konténer osztályai az előző pontban tárgyalt interfészek megvalósításával egy-egy konkrét adattárolási osztályt határoznak meg. Ezeknek az osztályoknak példányai a programokban egy-egy osztály adattagjaiként, vagy akár lokális változóként is felhasználhatók ugyanúgy, mint más objektumpéldányok.

9.3.1. Halmazok

A halmazok valamilyen közös jellemzővel bíró adatok „laza" kapcsolata. Az elemek egyénileg elérhetőek, de nem ismerik egymást.

HashSet

A `Set` interfészt valósítja meg, így elemei egyediek, és rendezetlenek. A konténer osztályban az alábbi halmazműveletek értelmezettek: elem hozzáadása a halmazhoz (`add`), elem tartalmazás vizsgálata (`contains`), elemek törlése (`clear`). Az elemeket az ún. hasítási technika segítségével (hashing) tarthatjuk karban. A hasítás lényege az, hogy az elemeket valamilyen szempont szerint csoportosítjuk, és ezekhez a csoportokhoz egy-egy hasító kódot rendelünk (pl: számok tízes maradéka, nevek kezdőbetűje stb.). Az egyes elemeket ezután az így előálló csoportokhoz egyértelműen hozzárendeljük. A hasító kódokkal egy rendkívül gyors indexelési eljárást kapunk, így az elemek visszakeresése és manipulálása nagyon hatékony lehet, igaz az adattárolás sorrendje kaotikus, előre nem meghatározható. (A `hashCode` metódus felüldefiniálásával lehetőségünk van saját hasító függvényeket is definiálni.)

Egy programrészletben mutassuk be a halmaz működését. Egy szövegből keressük ki a benne szereplő betűket.

```
…
String szoveg = ”Micimackó”;
Set betuk = new HashSet();
for (int i=0; i<szoveg.length(); i++){
  betuk.add(new Character(szoveg.charAt(i)));
 …}
System.out.print(”A ”+szoveg+” -ben található betűk:”);
System.out.println(betuk);
…

Lehetséges kimenete: [i, k, M, a, c, ó]
```

A példában a `betuk` egy `HashSet` típusú konténer objektum. A szöveg bejárásakor a karakterek csomagoló osztályának segítségével (`Character`) egy-egy új objektumot hoztunk létre, amely az egyes karaktereket referenciával elért objektumokként kezeli. Ezeket a referenciákat a halmazhoz adjuk. A `Character` osztály `equals()` metódusával a háttérben, a behelyezendő karaktereket összehasonlítja a már tároltakkal, és ha ilyen elem még nem szerepelt, akkor azt a halmazba felveszi.

Kiíratáskor az osztály `toString` metódusát érjük el automatikus konverzióval.

Figyeljük meg, hogy a `betuk` deklarálásakor az interfésznek megfelelő típust (`Set`) rendeltünk az azonosítóhoz. Ez a megadás nem hibás, hiszen a halmazzal az interfészben megadott műveletekkel kommunikálunk. Segítségükkel lehet a redszertervek absztrakt tervezésű modelljeit lépésenként konkrét tároló osztályokká alakítani. (Ez a módszer a következő fejezetekben többször is előfordul.)

TreeSet

Ez is egy halmazt valósít meg, de mivel a `SortedSet` interfészt implementálja, ezért csak olyan objektumokat tartalmazhat, amelyek egymással összehasonlíthatók. Az összehasonlításhoz az osztályoknak a `Comparable` interfészt implementálniuk kell. Azelőző példában a `TreeSet` osztályt is használhatjuk a betűk vizsgálatára, hiszen a `Character` csomagoló osztályban definiálták az összehasonlítás műveletét (`compareTo`). A példában a `betuk` deklarációját adjuk meg az alábbi alakban:

```
Set betuk = new TreeSet();
```

Ekkor a program az objektumokat már rendezetten tárolja. Az osztály nevéből következtethetünk, hogy a halmaz tárolási módja egy fával történik. Ezt a belső, kiegyensúlyozott bináris fával történő tárolást a konténer osztály elrejti előlünk.

Ha saját objektumokat szeretnénk `TreeSet`-ben tárolni, akkor a típust definiáló osztálynak implementálnia kell a `Comparable` interfészt, ebből következően ki kell fejteni a `compareTo` metódust, ahol definiálhatjuk az osztályba tartozó példányok összehasonlító műveletét valamely jellemző alapján.

Tároljunk egy rendezett halmazban téglalapokat.

A rendezéshez a már régebben definiált `Teglalap` osztályt fel kell készíteni a konténerben való tárolásra az alábbi módon:

```
public class Teglalap implements Comparable {
  private double a, b;

  public Teglalap (double a, double b) {
     this.a = a;
     this.b = b;
     }
  public double kerulet() {return 2*(a+b); }
  public double terulet() {return a*b; }
  public String toString(){
     return ("Téglalap ["+a+","+b+"]");
     }
  // A Comparable interfészben deklarált metódus:
  public int compareTo(Object obj){
     return
     (int) ( this.terulet() - ((Teglalap)obj).terulet() );
     }
}
```

Az alábbi programrészlet segítségével adhatunk meg rendezett halmazt:

```
Teglalap t1 =new Teglalap(5,6);
Teglalap t2 =new Teglalap(4,4);

…
Set alakzat= new TreeSet();
alakzat.add(t1);
alakzat.add(new Teglalap(12,4));
alakzat.add(t2);
alakzat.add(t1); //már egyszer szerepel, így kimarad!
```

```
System.out.println(alakzat);
…

Kimenet:
[Téglalap [4.0,4.0], Téglalap [5.0,6.0], Téglalap
[12.0,4.0]]
```

A `Comparable` interfész megvalósítását a fordítónak az `implements` kulcsszóval jelezzük. Ekkor meg kell adni az `int compareTo(Object o)` metódus definícióját, amely az aktuális példány és a paraméterben érkező referencia összehasonlítását teszi lehetővé. Visszatérési értéke egy egész szám. A definícióban a téglalapok „nagyságát” területükkel mérjük. Természetesen megadható más szabály is.

Ha a `compareTo` metódust nem definiáljuk, akkor „does not override abstract method” fordítási hibaüzenetet kapunk, ha pedig a `Comparable` interfészt felejtjük el bejelenteni, akkor a futás során a futtató környezet nem tudja garantálni, hogy ugyanazt a `comapareTo` metódust látja, amit az interfész megkövetel, és `ClassCastException` kivételt vált ki az összehasonlítás művelete.

LinkedHashSet

A `HashSet` osztály bővítése, ahol az adattárolás kétirányú láncolt listában történik. Ez egy olyan halmazt eredményez, amely az elemeket beviteli sorrendben tartalmazza. (A `HashSet` előre megjósolhatatlan sorrendben adja vissza az adatokat.) Ez az osztály jó kompromisszum abban az esetben, ha az elemek sorrendje számít, és gyorsabb adatmanipulációt szeretnénk, mint amit a `TreeSet` nyújt.

BitSet

Segítségével egy dinamikus bitsorozatot tartalmazó objektumot kezelhetünk. A `BitSet` osztály objektumain könnyen végezhetünk logikai műveleteket (ÉS, VAGY, kizáró VAGY), vehetjük két halmaz metszetét, kiszámíthatjuk kardinalitását (a benne szereplő `true` értékek száma). A megvalósításban egy olyan logikai elemekből álló halmazról van szó, amely indexelhető, elemei elérhetőek. Ez az osztály önálló, megvalósítása az előzőektől független (nem implementálja a `Set` interfészt)!

Ábrázoljuk bitsorozatokkal a számok kettővel, hárommal és hattal való oszthatóságát.

```
…
BitSet paros = new BitSet(10);
BitSet harommalOszthato = new BitSet(); //üres halmaz
BitSet hattalOszthato = new BitSet();

for (int  i=0; i< 20; i+=2){
  paros.set(i);
  hattalOszthato.set(i);
  }
for (int  i=0; i< 20; i+=3){
  harommalOszthato.set(i);
  }
// BitSet művelet: ÉS
hattalOszthato.and(harommalOszthato);

System.out.println("Páros számok: "+paros);
System.out.println("Hárommal osztható számok: "
   +harommalOszthato);
System.out.println("Hattal osztható számok: "
   +hattalOszthato);
…
//Kimenet:
//Páros számok: {0, 2, 4, 6, 8, 10, 12, 14, 16, 18}
//Hárommal osztható számok: {0, 3, 6, 9, 12, 15, 18}
//Hattal osztható számok: {0, 6, 12, 18}
```

EnumSet

Mint azt a 3.2.12. fejezetben láthattuk, a Java 1.5-ös verziójától lehetőségünk van felsorolt típusú adatok használatára (enum) [12]. A kollekció keretrendszerben helyet kapó típus a `java.util.EnumSet`. Ebben az adatszerkezetben a felsorolt típusú adatok halmazát készíthetjük el. A működés elve eltér a többi halmazétól, mert itt az új típust csak közvetlenül a speciális halmazműveletek segítségével hozhatjuk létre, az alábbi módokon:

```
…
public enum Napok {HE, KE, SZE, CSU, PE, SZO, VAS};
…

EnumSet foglaltNapok = EnumSet.of(Napok.HE, Napok.SZE,
Napok.PE);
EnumSet hetNapjai = EnumSet.allOf(Napok.class);
EnumSet ures = EnumSet.noneOf(Napok.class);
EnumSet szabadNapok = EnumSet.complementOf(foglaltNapok);
EnumSet munkaNapok = EnumSet.range(Napok.HE, Napok.PE);
```

```
System.out.println( foglaltNapok );          // [HE, SZE, PE]
System.out.println( hetNapjai );   // [HE, KE, SZE, CSU, PE,
SZO, VAS]
System.out.println( ures ); // []
System.out.println( szabadNapok );    // [KE, CSU, SZO, VAS]
System.out.println( munkaNapok );  // [HE, KE, SZE, CSU, PE]
```

Természetesen az EnumSet osztály is támogatja a generikus típusokat (bővebben lásd: 9.4.), így az alábbi definíció is használható:

```
EnumSet <Napok> foglaltNapok = EnumSet.<Napok>of(Napok.HE,
Napok.SZE, Napok.PE);
```

9.3.2. Listák

A lista adatszerkezet szekvenciális adattárolást tesz lehetővé. A listákban az elemsorrend rögzített, valamilyen bejáró algoritmussal végigjárhatjuk az elemeket. A listák belső adattárolása történhet referenciatömbökkel, egy-és kétirányú listákkal.

ArrayList

A listát egy olyan dinamikus tömbbel reprezentálja, amely futási időben újraméretezhető. A megvalósításban a konténer egy referenciatömböt takar. A lista memóriában foglalt méretét a kapacitása (`capacity`), a benne található elemek méretét a size adattagban tárolja. Amennyiben a méret eléri a kapacitást, akkor a virtuális gép a listának új és nagyobb helyet foglal a háttérben.

A lista karbantartó műveletei: elem hozzáadása (`add`), elem törlése (`remove`), teljes lista törlése (`clear`), tartalmazás vizsgálat (`contains`), keresés (`indexof`), elem elérése (`get`), elem cseréje (`set`) stb.

Vector

A `Vector` osztály olyan kollekció, amely objektumok rendezetlen tárolására képes, az `ArrayList`-hez hasonló módon (dinamikus tömb). A `Vector` abban tér el az `ArrayList`-től, hogy az adatokat szinkronizáltan kezeli, ennek nagy szerepe lesz a párhuzamos, többszálú programok fejlesztésekor, hiszen ott szükség van a minden egyes szálból elérhető konténer objektumokra.

Helyezzünk el téglalap objektumokat egy vektorban!

Az előzőekben definiált `Teglalap` osztály felhasználásával, pl. az alábbi programrészlet segítségével elkészítünk egy vektort:

```
…
Teglalap t1 = new Teglalap(5,6);
Teglalap t2 = new Teglalap(4,4);

List v1= new Vector();
v1.add(t1);
v1.add(t2);
v1.add(t2);

// Egy konstrukcióval létrehozott objektum közvetlen
// (referenciaváltozó nélküli) hozzáadása a konténerhez.
v1.add(new Teglalap(12,4));

// A vektor 3. elemét eltávolítjuk.
v1.remove(2);

System.out.println("A vektor elemei:\n"+v1);
…
//Kimenet:
//A vektor elemei:
//[Téglalap [5.0,6.0], Téglalap [4.0,4.0], Téglalap
//[12.0,4.0]]
```

Az előző példában kifejtett elv szerint `List` típusúnak deklarált `v1` vektorba az elemek referenciájának megadásával tehetünk elemeket. Azonban közvetlenül is adhatunk elemet a vektornak, egy konstruktor megadásával. Ekkor a referenciát csak a vektor műveletein keresztül érhetjük el. A programrészlet végén a vektor tartalmát egyszerűen kiíratjuk a vektor `toString` metódusán keresztül (automatikus konverzió), amely sorra bejárja a vektor elemeit, és azok `toString` metódusaival állítja elő az eredménysztringet.

Stack

A `Vector` osztály leszármaztatásával előálló `Stack` osztály egy verem rendszerű adattárolót (LIFO – last in first out) reprezentál. A kiterjesztés során öt új metódust vezet be az adatmanipuláláshoz. A `push` és `pop` metódusok segítségével lehet a verem tetejére egy elemet elhelyezni, illetve levenni. A `peek` metódus segítségével a verem tetején levő elemet felhasználhatjuk, anélkül, hogy a veremből kivennénk. A veremben kikereshetünk egy értéket (`search`) és megvizsgálhatjuk, hogy üres-e (`empty`).

Adjunk meg a fenti példára verem reprezentációt. Az elemeket fordított sorrendben adjuk vissza!

```
…
Stack verem= new Stack();
verem.push("A verem első eleme");
verem.push(t1);
verem.push(t2);
verem.push(t2);
verem.push(new Teglalap(12,4));
verem.push("A verem utolsó eleme");

System.out.println("A verem tartalma: ");
while (!verem.empty()){
  System.out.println((verem.pop()));
  }
…

//Kimenet:
//A verem tartalma:
//A verem utolsó eleme
//Téglalap [12.0,4.0]
//Téglalap [4.0,4.0]
//Téglalap [4.0,4.0]
//Téglalap [5.0,6.0]
//A verem első eleme
```

A példából láthatjuk a verem működési elvét: az utoljára betett elemet kapjuk meg elsőként. Az is látszik a megadáson, hogy a konténerek különböző típusú elemeket is tartalmazhatnak. (Itt nem használtuk ki a magasszintű modellekből kapott lehetőségt, azaz azt, hogy a konténert interfész segítségével definiáljuk, hiszen a verem speciális metódusait is el kellett érnünk típuskényszerítés nélkül.)

LinkedList

A `List` interfész dinamikus, kétirányú láncolt listával implementált osztálya. Az egyes objektumok össze vannak kapcsolva, azaz minden elem tartalmazza az őt megelőző, illetve követő elem referenciáját.

A lista első és utolsó eleme kitüntetett szereppel bír, hiszen mindkettőtől elkezdhetjük a lista bejárását. Adatkarbantartó műveletei az általános listaműveleteken kívül: `addFirst`, `addLast`, `getFirst`, `getLast`, `removeFirst`, `removeLast`. Ezek a műveletek kiemelten kezelik a lista

első és utolsó elemét. (A `LinkedList` megvalósítja a `Queue` interfész metódusait is, kifejtését lásd alább.)

Tároljunk szöveges adatokat egy kétirányú listában!

```
List list = new LinkedList();
list.add("a nyári");
list.add("nap");
list.add(0,"le");
list.add(0,"Tüzesen");
list.add(1,"süt ");

// Típuskésnyszerítéssel elérhetőek a LinkedList saját
metódusai is:
((LinkedList)list).addLast("sugára.");
// Lista bejárása egy speciális interfész segítségével:
ListIterator it = list.listIterator();
while (it.hasNext()) {
  System.out.println(it.next());
  }
…
//Kimenet:
//Tüzesen
//süt
//le
//a nyári
//nap
//sugára.
```

9.3.3. Sorok

Adatsorok (Queue) kezeléséhez használt sorok (FIFO tároló – first in first out) annyiban térnek el a hagyományos listáktól, hogy az adatkarbantartó műveleteik (beszúr, kivétel, keresés) speciálisan működnek. Új elemeket általában a sor végére helyezhetünk (`add`). A sor elejéről egy elem levehető (`poll`), vagy a sor első eleme elérhető (`peek`) stb. A sorokat tulajdonképpen egy-egy speciális listának is felfoghatjuk.

LinkedList

Az előzőkben ismertetett `LinkedList` osztály implementálja a `Queue` interfész metódusait is, tehát felfogható egy sor-jellegű adattárolónak is (FIFO). Ilyenkor a sorokra jellemző adatkezelő műveleteket is használhatjuk (`add`, `poll`, `peek`).

PriorityQueue

Egy olyan sor kezelésére van lehetőségünk, amelyet valamilyen szempont szerint rendeznünk kell.

A rendezés feltétele, hogy a sorba illesztett elemeknek összehasonlíthatóak legyenek, vagyis a `TreeeSet` és `TreeMap` osztályoknál is használt `Comparable` interfészt implementáláják. Egy adott elem a hozzáadásakor egyből a rendezési szempont szerinti megfelelő helyére kerül. A sor hossza tetszőleges lehet.

(Az egyidejű konkurens programozási feladatokban használnak még speciális osztályokat, amelyek skálázhatók, hibatűrők: `ArrayBlockingQueue`, `DelayQueue`, `PriorityBlockingQueue`, `LinkedBlockingQueue` és `SynchronousQueue`, de ezekkel ebben a jegyzetben nem foglaklozunk.)

9.3.4. Leképezések

A leképezések, vagy hozzárendelések olyan speciális típusú konténerek, amelyekben kulcs-érték párok segítségével tárolhatunk elemeket. Minden kulcshoz egy és csakis egy érték tartozhat. A leképezések alapműveletei a listákhoz, sorokhoz és a halmazokhoz hasonlók (`get`, `put`, `clear`, stb). A `Map` nterfész a Java régebbi verzióiban meglevő `Hashtable` osztályt váltotta fel.

A tárolás módja a következő: a leképezés az egyedi kulcsértékekhez egy-egy tárolandó objektum referenciáját (értékek) rendeli hozzá. A kulcsértékek tárolási módja ezen a szinten még nincs meghatározva, de a kulcsok egyediségét a megvalósító osztályoknak biztosítani kell. A hozzárendelés egyirányú leképezés, így csak kulcs-érték irányban lekérdezhető. Amennyiben egy kulcshoz egy újabb put metódussal egy új értéket rendelünk hozzá, a régebbi hozzárendelés elveszik.

HashMap

A `Map` interfész megvalósítása hasítótábla segítségével. Az adatok elhelyezéséhez a `put` metódust használhatjuk, amelyben az adott kulcshoz rendelhetünk értékeket (pl. nevekhez életkor adatot) és a `get` metódus segítségével érhetünk el egy kulcshoz tartozó bejegyzést.

Az egyes adatmanipuláló műveletek a kulcsértékek alapján érik el az adatokat. (Az értékek elérését nem szabad összekeverni az indexeléssel, hiszen az adatszerkezet nem szekvenciális. A hasítási technika a kulcsértéket tulajdonképpen egy függvénnyel előálló értéke szerint azonosítja. A hasítófüggvények működését bővbben lásd: 0. fejezet.)

TreeMap

A `Map` interfész implementációja rendezett fa adatszerkezet segítségével úgy, hogy a kulcsoknak összehasonlíthatóaknak kell lenniük. Az összehasonlítás hasonló a `TreeSet`-ben alkalmazotthoz. A kulcsként szereplő objektumoknak implementálniuk kell a `Comparable` interfészt. A kulcs-érték párok ebben az osztályban is a `put` és a `get` metódussal kezelhetők. A kulcsok a `keySet`, a hozzárendelt értékek pedig az `entrySet` metódussal halmazként külön-külön is lekérdezhető.

Tároljunk leképezések segítségével neveket a hozzájuk tartozó életkorral együtt, névsorba rendezetten.

```
Map szemelyek = new TreeMap();
szemelyek.put(”Pisti”, 17);
szemelyek.put(”Béla”, 43);
szemelyek.put(”Dezső”, 60);
szemelyek.put(”Dezső”, 71);//a régi hozzárendelés elveszik!
// kiírás kulcsok szerint
System.out.println (”Kulcsok: ”+szemelyek.keySet());
// kiírás értékek szerint
System.out.println (”Értékek: ”+szemelyek.values());
// kiírás együtt
System.out.println (”A teljes leképezés: ”+szemelyek);
// vizsgálatok:
System.out.print("Van Pisti nevű kulcs? ");
  if (szemelyek.containsKey("Pisti")){
    System.out.println("Igen.");
    System.out.println("Értéke:"+szemelyek.get("Pisti"));
  }
  else System.out.println("Nincs!");
…
//Kimenete:
//Kulcsok: [Béla, Dezső, Pisti]
//Értékek: [43, 71, 17]
//A teljes leképezés: {Béla=43, Dezső=71, Pisti=17}
//Van Pisti nevű kulcs? Igen.
//Értéke:17
```

IdentityHashMap

A `HashMap` osztály egy alternatívája. Ebben a ritkán használt osztályban a kulcsértékek referenciái szerinti összehasonlítás végezhető el.

LinkedHashMap

A leképezést egy kétirányban láncolt listában tartja. Általában leképezések másolatához használják, mert a leképezés a fizikai felvétel sorrendjét megtartja.

EnumMap

Enum típusú felsorolt adatokhoz használható leképezés. Az adatokat speciális tömbökben helyezi el. Tömör adattárolást tesz lehetővé.

Az `EnumMap` osztály a speciálisan felsorolt típusok hozzárendeléséhez használható. Segítségével nagyon könnyen tudunk a felsorolt típus alapú hozzárendeléseket, vagy indexeléseket megadni.

A típus definiálásához fel fogjuk használni a felsorolt típus osztályát, mint inicializátort, továbbá a generikus típusokat(a generikus típusokkal bővebben a 9.4. fejezetben találkozunk), a fordítási idejű típusellenőrzések miatt az alábbi formában:

```
EnumMap<Kapcsolat, String> kapcsolatLeiro =
   new EnumMap<Kapcsolat, String>(Kapcsolat.class);
```

Vagyis a `kapcsolatLeiro` egy olyan objektum lesz, amelyik a felsorolt típusú konstans értékekhez egy-egy szöveges adatot rendel. A leképezés kezdetben üres és a feltöltést a már leképezéseknél tárgyalt `put` metódussal végezhetjük el:

```
kapcsolatLeiro.put(Kapcsolat.HIBA, "A kapcsolat nem épült
fel!");
```

Az `EnuMap` típus használata ezután analóg a többi `Map` típussal.

9.3.5. Az Iterator és az Enumeration

Az `Iterator` interfész segítségével egy olyan objektumot nyerhetünk, amelyek az egyes konténerekben tartalmazott elemsorozat másolatát tartalmazza. Az iterátor segítségével az adatszerkezetek referenciáiról a memóriában egy új, ideiglenes sorozat készíthető, segítségével a konténerek (tulajdonképpen a benne tárolt elemek) bejárhatóak, függetlenül attól, hogy a konténer milyen tárolási módszer szerint tárolja az adatokat.

Egy iterátor az `Enumeration` interfésszel, vagy pedig az újabb fejlesztésű `Iterator` interfésszel adható meg. Minden kollekció jellegű osztály implementálja az `iterator()` metódust, és segítségével egy kollekció

helyesen bejárható. Az iterátor alkalmazásával egy olyan objektum jön létre, amely gyakorlatilag egy bejárható referenciasorozat.

Megjegyzés: Az interfész típusú visszatérési típusú metódusok olyan névtelen (leszármazott) osztályokat hoznak létre, amelyek az adott interfészeket megvalósítják.

Az iterátorban három metódust használhatunk: a `hasNext()` megadja, hogy az iterátornak van-e még következő eleme, a `next()` visszaadja a következő elem referenciáját, a `remove()` kiveszi a referenciát az iterátorból – anélkül, hogy az eredeti kollekciót módosítaná.

(A Map interfészt megvalósító leképezések közvetlenül nem iterálhatóak, csak a kulcs-érték adataik külön-külön.)

Hozzunk létre egy listát, melyben egész értékeket tárolunk, majd egy iterátor segítségével listázzuk ki.

```
ArrayList lista = new ArrayList();
for (int i=0; i<20; i++){
  lista.add(i);
  }
Iterator it = lista.iterator();
System.out.println("A tárolt értékek:");
while (it.hasNext()){
  System.out.print( it.next());
  }
```

Az előzőekben láthattunk egy példát a `ListIterator` interfészre is, amely a dinamikus listák bejárását teszi lehetővé (lásd 0. fejezet).

A Java régebbi kiadásaiban szereplő `Enumeration` interfész működése hasonló elven működik, mint az `Iterator`, azonban nem tartalmaz `remove` metódust, ezért használata a fejlesztők szerint már nem ajánlott, bár kompatibilitási szempontok miatt máig a környezet része maradt.

```
for (Enumeration e = v.elements() ; e.hasMoreElements() ;){
     System.out.println(e.nextElement());
     }
```

9.4. Általánosított típusok (generics)

A Java fejlesztői környezet kiadásának 5. verziójában [12] számos új elem jelent meg, amelyek közül a leglátványosabb az általánosított típusok és metódusok megjelenése (generic type, generic method).

A fejlesztők már Java környezet tervezésekor azt a célt tűzték ki, hogy a hibalehetőségek a lehető legkorábban, lehetőleg még fordítási időben derüljenek ki. A generikus típusok bevezetése is ezen filozófia része.

Más megközelítés szerint [Nyéky03, 694.p] a generikus típusok használata az adatszerkezetek és algoritmusok absztrakciójával, az absztrakt módon megfogalmazott feladatkiírás egy hatékony kódolását tükrözi, melyben elvonatkoztathatunk a lényegtelen részletektől.

A generikus típusok használatához a nyelv egyes elemeit kiterjesztették. Így definiálhatók olyan osztályok, melyek adattagjai ill. metódusainak paraméterei vagy visszaadott értékei általánosított (helyettesíthető) típusokkal láthatók el.

9.4.1. Generikus típusok használata

Az általánosított típusok legtöbbet használt, és legtöbbször bemutatott példája a láncolt listák osztálya. A `List` interfészről a következő bejegyzést találjuk az API-ban: `java.util.List<E>`. Az `<E>` formula jelöli, hogy a `List` interfész – és minden megvalósítása – generikus típusokat kezel. Itt maga az `E` jelöli azt a konkrét típust, amelyet majd a listában tárolni kívánunk. A leírásban többször is találkozunk az `<E>` szimbólummal, pl. a `void add(<E> o)`, vagy az `<E> get()` metódusnál. A jelölésben itt az `E`, mint helyettesítendő típus áll.

Vagyis lehetőség nyílik olyan absztrakt, konténer jellegű adatszerkezetek megfogalmazására, melynek definiálásakor még nem ismerjük a tárolni kívánt osztályok típusát. A típust elegendő csak a konkrét listaobjektum példányosításakor rögzíteni. Ekkor az `<E>` szimbólumok a kívánt típussal helyettesítődnek.

Hozzunk létre egy olyan kétirányban láncolt listát, amely csak `String` típusú elemeket tartalmaznak!

```
public static void main(String[] args(){
//Generikus típusú változó definiálása
LinkedList<String> szavak = new LinkedList<String>();
// adatfeltöltés
szavak.add("első");
szavak.add("második");
szavak.add(1234); // fordítási hiba!
szavak.add(new Teglalap(3,4)); //fordítási hiba!
System.out.println(szavak); //kimenet: [első, második]
System.out.println(szavak.get(1)); //kimenet: második
//Nincs szükség típuskényszerítésre!
String s = szavak.get(0);
}
```

A generikus típusok használatának több előnye is van:

- A fordító már fordítási időben ellenőrzi a generikus típusokban használt helyettesítendő típust (a példában a `String` típust), és az objektumok pontos metódushívását is. Így sok futásidejű `ClassCastException` típusú kivétel előfordulása eleve elkerülhető. Az egyes paraméterek és a visszatérési értékek típusa rögzített és ellenőrizhető.
- A generikus típusok használatakor a fordító már ismeri a konkrét típust, ezért a sorozatos típuskényszerítés (ami sokszor az átláthatóságot nehezíti) is elhagyható. Továbbá a használt típusnak a leszármazottjai is felhasználhatóak lesznek.

A Java keretrendszer 1.4-es kiadása még nem támogatta a generikus típusokat, így pl. egy láncolt listát csak a legáltalánosabb referenciatípusokkal tölthettünk fel (`Object`), és a felhasználás során az egyes tárolt objektumok metódusait csak típuskényszerítéssel érhettük el. Még bonyolultabb volt a helyzet abban az esetben, ha az osztály kellően összetett adatszerkezettel bírt. Ilyenkor egy sorban nemegyszer két-három, vagy több típuskényszerítés is előfordulhatott, ami gyakorlatilag olvashatatlanná tette a kódot. A generikus típusok használatával ez kiküszöbölhető.

Készítsünk leképezést, melyben egész értékekhez szöveges adatokat rendelhetünk! (Oldjuk meg a feladatot valós számokkal is!)

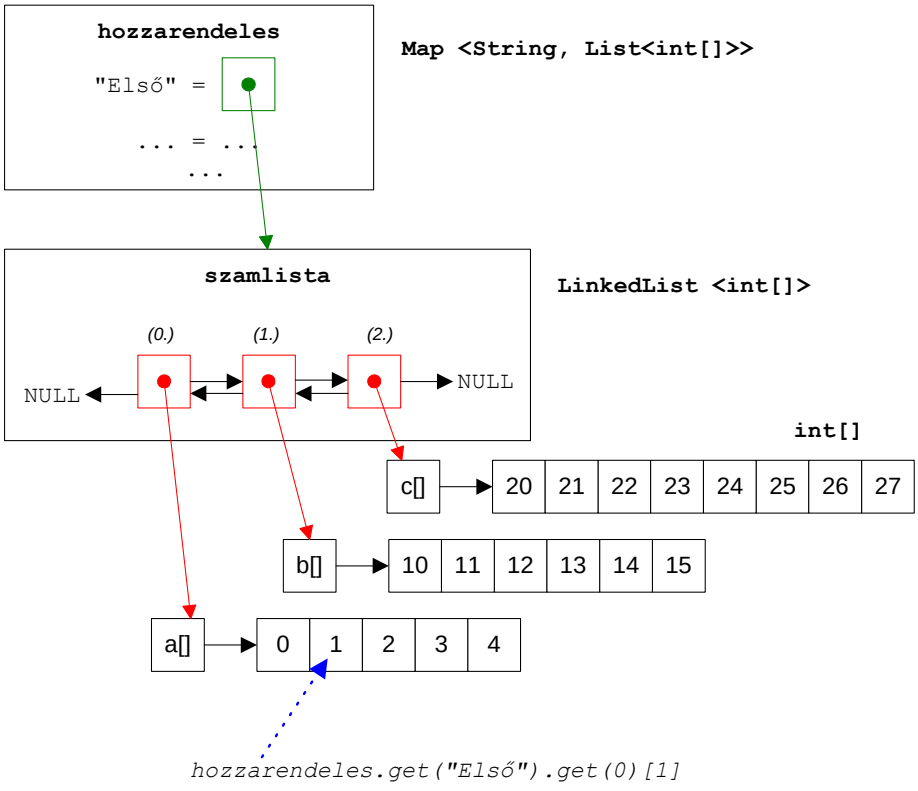
```
//Egész számokkal
Map <Integer, String> hozzarendeles1 =
     new HashMap <Integer, String>();
hozzarendeles1.put(1, "első");
hozzarendeles1.put(2, "második");
hozzarendeles.add(3.14, "Pi"); //fordítási hiba!
System.out.println(hozzarendeles1);

//Általános számformátummal
Map <Number, String> hozzarendeles2 =
     new HashMap <Number, String>();
hozzarendeles2.put(1, "első");
hozzarendeles2.put(2, "második");
hozzarendeles2.put(3.14, "Pi"); // OK.
System.out.println(hozzarendeles2);
```

A generikus típusok egymásba ágyazhatók, így egy kellően összetett adatszerkezet is könnyen kezelhető lesz.

Az alábbi példában egy olyan összetett adatszerkezetet definiálunk, amelyben egész számokból álló tömböket láncolt listában helyezünk el, és ilyen listából kulcs-érték párokkal leképezéseket készítünk. (Hasonló módon más adatszerkezetek is felépíthetők.)

```
// Egészekből álló tömbökből egy kétirányban láncolt listát
készítünk,
// majd a listát behelyezzük egy leképezésbe
int [] a = {0,1,2,3,4};
int [] b = {10,11,12,13,14,15 };
int [] c = {20,21,22,23,24,25,26,27};

// lista elkészítése
List <int[]> szamlista = new LinkedList<int[]>();
szamlista.add(a);
szamlista.add(b);
szamlista.add(c);

// a lista tartalma, egyszerűsített hozzáférés:
System.out.println("A lista tartalma:");
for (int i=0; i < szamlista.size(); i++){
for (int j=0; j < szamlista.get(i).length; j++) {
    System.out.print(szamlista.get(i)[j]+", ");
}
System.out.println();
}
```

```
//A lista elhelyezése a megfelelően előkészített
hozzárendelésbe/leképezésbe
//itt jól megfigyelhető a generikus típusok egymásba
ágyazódása

Map <String, List<int[]>> hozzarendeles =
     new HashMap <String, List<int[]>>();
//Az "Első" kulcshoz hozzárendeljük a szamlista
// láncolt listát (amely maga is tömbökből áll)
hozzarendeles.put("Első", szamlista);

//Az "Első" kulcshoz hozzárendelt lista első tagjának
// második eleme a következőképpen érhető el:
// nem szükséges típust kényszeríteni,hiszen
// a definícióból a típusok egyértelműen adódnak!)
System.out.println(hozzarendeles.get("Első").get(0)[1]); //
kimenet: 1
```



23. ábra: Generikus típusok használata összetett adatszerkezetekben

A kollekció keretrendszer használatával egy fordítási időben ellenőrzött, a futásidejű hibákra kevésbé érzékeny kód állítható elő. A típuskonverziót az adatszerkezetek definiálásakor megadott típusok szerint a fordító automatikusan elvégzi. A fordító azonnal figyelmeztető üzenettel, vagy hibával jelzi a nem megfelelő típushasználatot, míg a generikus típusok nélkül esetleg egy távolabbi helyen kapnánk futásidejű kivételt.

Természetesen a fenti példa még nem „tökéletes", csak a típusok egymásba ágyazhatóságát mutatja be, ugyanis a tömbök és a lista nem megfelelő indexelése kiválthat még futásidejű kivételt. Ezért célszerűbb lenne egy megfelelő osztályt megfelelően ellenőrzött metódusokkal definiálni, de terjedelmi korlátok miatt itt erre most nem térünk ki.

9.4.2. Generikus osztályok definiálása

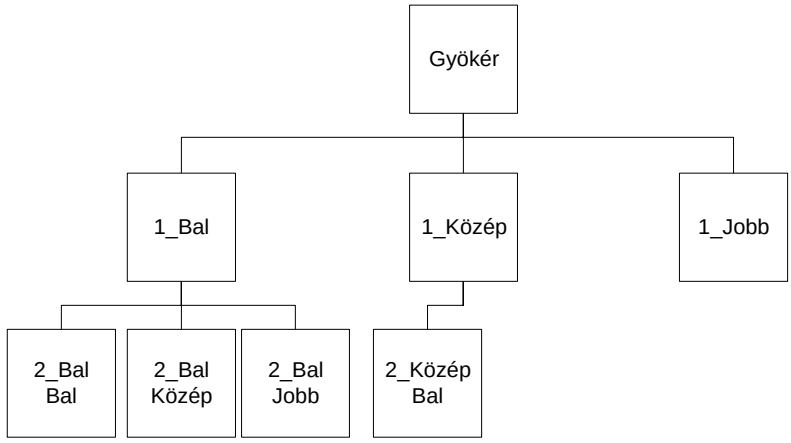
A fentiekben láttuk, hogy egy generikus típust hogyan kell felhasználni. Az alábbiakban megismerkedünk a generikus típus definiciókkal, azaz azzal, hogy hogyan készíthetünk olyan általánosított osztályokat, melyek majd csak a példányosításkor kapnak konkrét típusokat.

Az alapelv nagyon egyszerű. Az alábbi példákban a <V> és a V szimbólummal jelöljük a helyettesítendő általános típust. Az ABC nagybetűivel jelzett típus az osztály példányosításakor kapja majd meg azt a konkrét típust, amelyet helyettesíteni fog. Ekkor a fordító a szimbólumok helyére a konkrét típust rendeli és ezzel a valódi típussal hozza létre a bájtkódot (`.class`).

Definiáljunk egy N-ágú fát leíró típust generikus típusokkal!

```
import java.util.*;
public class TobbaguFa<V>{
  //A fa egy csomópontjának értéke
  private V ertek;
  //A fa ágai, mint egy dinamikus referenciatömb
  private List <TobbaguFa<V>> agak =
      new ArrayList<TobbaguFa<V>>();
  //Konstruktor
  public TobbaguFa(V ertek){
      this.ertek=ertek;
  }
```

```
    //Metódusok
    V getErtek(){ return this.ertek;}
    void setErtek(V ertek){ this.ertek=ertek;}
    int getAgakSzama(){ return agak.size();}
    TobbaguFa<V> getAg(int n){ return agak.get(n);}
    void addAg(TobbaguFa<V> ujag){ agak.add(ujag);}
  }
public class NFaTeszt{
  public static void main(String[] args){
  TobbaguFa <String> gyoker =
      new TobbaguFa<String>("Gyökér");
  //Adatfeltöltés
  gyoker.addAg(new TobbaguFa<String>("1_Bal"));
  gyoker.addAg(new TobbaguFa<String>("1_Kozep"));
  gyoker.addAg(new TobbaguFa<String>("1_Jobb"));
  gyoker.getAg(0).addAg(new TobbaguFa<String>("2_Bal Bal"));
  gyoker.getAg(0).addAg(new TobbaguFa<String>("2_Bal Közép"));
  gyoker.getAg(0).addAg(new TobbaguFa<String>("2_Bal Jobb"));
  gyoker.getAg(1).addAg(new TobbaguFa<String>("2_Közép Bal"));
  ...
  }
}
```



24. ábra: N-ágú fa szerkezeti modellje

Amennyiben a generikus osztályok példányosításából elhagyjuk a generikus típus pontosítását (`<Típus>`), akkor a fordító ezt "`Note: ....java uses unchecked or unsafe operations. Note: Recompile with -Xlint:unchecked for details.`" figyelmeztető üzenettel jelzi. A

figyelmeztetés kikapcsolható, (például korábbi Java verzióra írt forrásokhoz), ha a fordítást az `-Xlint:unchecked` opcióval végezzük.

9.4.3. Generikus típusok korlátozásai

Abban az esetben, ha valamennyire konkretizálni szeretnénk a típust, vagy a definíció „túl általános”, akkor megadhatunk korlátozásokat is. Erre az `extends` kulcsszó használható, korlátként pedig megadható egy osztály, illetve tetszőleges számú interfész (& jelekkel összefűzve). Ekkor a generikus típusból csak a megadott osztály vagy annak leszármazottjai szerint példányosíthatunk.

Abban az esetben, ha a generikus típus korlátozásakor interfészeket is megadtunk, akkor a példányosításkor a fordító megköveteli az interfészek megvalósítását is. Ekkor a típust leíró osztályt az alábbi módon definiálhatjuk:

```
public class TobbaguFa<V extends Number>{...}
public class TobbaguFa<V extends Number & Comparable>{...}
```

Azaz a `TobbaguFa` osztályban olyan típusokat használhatunk, amelyek a `Number` csomagoló osztály leszármazottai és megvalósítják a `Comparable` interfészt. Ekkor az osztályban megadhatunk pl. rendezési metódusokat is. A korlátozásokat természetesen az adott feladathoz kell igazítani.

9.4.4. Generikus metódusok

Abban az esetben, ha egy metódusnak olyan paramétert kívánunk adni amely a fenti generikus típusokat kezeli, akkor ezt korlátozás nélkül megtehetjük az alábbi szintaktika használatával: `public void listakezeles(List <E> lista){}`, mint generikus típusú paraméter, vagy `public List<E> getLista(){}`, mint visszatérési típus, és ahol `E` helyén bármely ismert interfész, vagy osztályazonosító állhat.

A `TobbaguFa` osztályban ilyen metódus volt maga a konstruktor is, illetve néhány metódus is (pl: `getErek()`, `addAg()`).

Az egyik előző példában használt egész tömbökből álló listához írjunk megjelenítő metódust!

```
public void listaTartalma(List <int[]> lista){
  for (int i=0; i < lista.size(); i++){
    for (int j=0; j < lista.get(i).length; j++) {
      System.out.print(lista.get(i)[j]+", ");
    }
    System.out.println();
  }
}
```

9.4.5. Határozatlan típusparaméterek

A generikus típusok definiálásánál azonban sok esetben nem tudjuk előre megadni a korlátozásokat, vagy a típust eleve absztraktnak tervezzük. Ekkor a <?> joker típust használhatjuk. Az előző példákban az <E> vagy <V> használatakor az egyes felhasználási helyeken kötelező volt a típusegyezés (a betűk helyére a fordító helyettesíti a konkrét típust). Sokszor azonban ennél lazább definícióra van szükségünk. Például egy osztályhierarchia osztályaiból szeretnénk valamilyen fejlett adatszerkezetet megadni.

```
public class TobbaguFa<V>{
private V ertek;
private List TobbaguFa<? extends V>> agak =
     new ArrayList<TobbaguFa<? extends V>>();
…
}
```

Ekkor a fa egyes csomópontjaiban nemcsak a `V` objektumok, hanem annak leszármazottai is lehetnek.

9.5. Kérdések

- Hogy csoportosíthatjuk az adatszerkezeteket?
- Hol használhatjuk a konténer osztályokat?
- Melyik csomagban található a gyűjtemény keretrendszer?
- Melyek a halmaz adatszerkezet főbb jellemzői?
- Melyek a dinamikus tömb adatszerkezet főbb jellemzői?
- Melyek a lista adatszerkezet főbb jellemzői?
- Melyek a leképezés (hozzárendelés) adatszerkezet főbb jellemzői?
- Mi a különbség egy sor és egy verem tároló között?
- Mire szolgál az `Iterator` osztály?
- Mire használhatóak a generikus típusok?
- Mi a különbség a generikus típus, a generikus metódus között?

9.6. Feladatok

- Készítsen példaprogramot, melyben egy `Vállalat` (cégnév, vezető, alkalmazottak, tevékenységi kör) személyi nyilvántartását vezetjük. A vállalatnál egy cégvezető és számos `Alkalmazott` (név, személyi ig. szám, életkor) adatait tartjuk nyilván. A `Vállalat` megalakulásakor a cégvezető már ismert. Az alkalmazottak tárolását valamilyen konténer osztály segítségével oldja meg! A program legyen képes:
  - ismertetni a vállalat összes dolgozójának adatait név szerinti abc sorrendben,
  - életkor szerinti csökkenő sorrendben,
  - az összes dolgozó életkorát egy évvel megnövelni,
  - felvenni és elbocsátani alkalmazottakat.
- Készítsen példaprogramot, amelyben halmaz adatszerkezetekben tárolja emberek csoportjait. Az első csoport legyen a `Lakók`, tagjai a következők: Béla, Tóni, Géza, Józsi, Andi. A `Bolt` dolgozói a következő személyek: Andi, Dezső, Géza. A közeli `Szervíz` dolgozói: Andi, Józsi, Tóni.

  Jelenítse meg azokat a személyeket, akikre igazak az alábbi megállapítások:
  - aktív dolgozók,
  - munkanélküliek,
  - több helyen is dolgoznak,
  - azok a dolgozók, akik nem laknak a környéken.
- Készítsen olyan programot, amely rendezetlen leképezésben a következő adatokat tartja nyilván:
  - hallgatók neve és pontszám,
  - becenevek és személyi adatokat tartalmazó objektunmok.

10. Az I/O és hálózatkezelés

A legegyszerűbb programok is kommunikálnak a külvilággal. Egy karakter megjelenítése a képernyőn, vagy beolvasása a billentyűzetről; a fájl és hálózatkezelés, illetve az adatbázisok elérése a Java nyelvben hasonló koncepció szerint zajlik. Az adatok valahonnan érkeztetése, illetve valamilyen célba eljuttatása általánosan az adatfolyamok (stream) segítségével zajlik. Az adatfolyamok egy adott adathalmazt (pl: karakter, bájt, objektum) képesek szabványosan fogadni, illetve az adott célba eljuttatni. Ezen az elven működik az eddig használt `System.out.println()` hívás is.

A Java keretrendszerben a `java.io` csomagban definiálták a folyamkezelést és a fájlkezelést megvalósító osztályokat. A csomag használatához be kell jelentenünk a fordítási egységek import deklarációs részében:

```
import java.io.*;
```

(A `java.nio` csomagban definiált pufferelt, skálázható ki és bemenettel itt nem foglalkozunk.) A Java környezet gazdag osztály- és interfészkínálata lefedi a legtöbb olvasási és írási feladatot. Az alábbiakban ezekkel ismerkedünk meg.

A fájlok elérése és kezelése a `File` osztály segítségével lehetséges. Minden egyes `File` típusú objektum egy-egy fájlrendszerbeli elemet reprezentál. Az objektumokat útvonalukkal azonosítjuk. A File osztály segítségével a fájlrendszer bejárható, a fájlok átnevezhetőek, törölhetőek, attribútumaik módosíthatók és üres állományok is létrehozhatók. A fájlok eléréséhez relatív, vagy abszolút útvonal megadásával juthatunk el. Az alábbi megadással az aktuális könyvtárhoz viszonyított relatív útvonalat adtunk meg:

```
File f = new File("ikonok/javalogo.gif");
```

A File.separator statikus adattagban az adott operációs rendszerhez tartozó fájl elválasztó karaktert tartja. Ennek segítségével az előbbi útvonalat `"ikonok"+File.separator+"javalogi.gif"`-ként megadva hordozhatóvá tehetjük a kódot.

Listázzuk ki az aktuális könyvtár összes `txt` kiterjesztésű fájlját!

```
import java.io.*;

public class listazas {
  public static void main( String[] args )
     throws IOException {
    File könyvtár = new File(".");

    // névtelen belső osztály alapján szűrünk
    File [] fájlok = könyvtár.listFiles(new FileFilter(){
       public boolean accept( File útvonal ){
          return (útvonal.isFile()
               && útvonal.getName().endsWith(".txt"));
          }
       } ); //listFiles vége

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

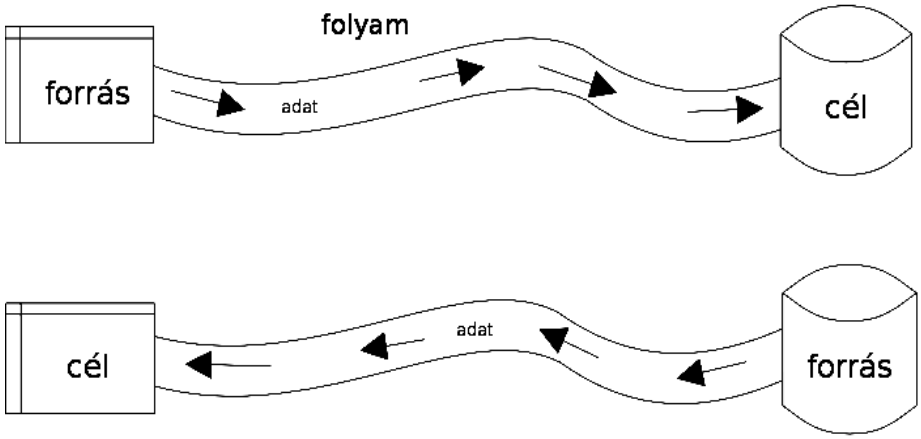
Töröljünk egy adott fájlt a fájlrendszerből!

```
import java.io.*;

class Torles {
 public static void main(String args[]) throws IOException{

   File törlendő = new File("torlendo.txt");
   if( törlendő.canWrite() ) {
       törlendő.delete();
       }
 }
}
```

10.1. Adatfolyamok

A folyam (stream) egy olyan objektum, amely adatokat képes írni, illetve olvasni egy adott célhelyre, illetve célhelyről. A folyamok használatakor a felhasználó nem észleli a mögöttes hardver elemek sajátosságait, így az egy hatékony, magas szintű és rugalmas ki- és bemenetet biztosít programozási szempontból. Az adatfolyamok segítségével a kimenet célja, vagy a bemenet forrása könnyen változtatható, a program sokoldalúvá tehető.

A folyamkezelés során az adatokat sorosan dolgozza fel a Java környezet (FIFO tárolók). A folyamokat a bennük haladó adatok szerint bájt-, karakter-, (elemi típusú) adat-, illetve objektumfolyamnak is nevezhetjük. Az adatáramlás iránya alapján pedig beviteli (input) és kimeneti (output) folyamokat szokás megkülönböztetni.



25. ábra: Adatfolyamok

A folyamok egymásba láncolhatóak, így egy-egy folyam egy másikba alakítható, illetve szűrőfolyamok (filter stream) segítségével az átáramló adatok pufferezhetőek, szűrhetőek, vagy akár egy másik folyam számára átkonvertálhatóak.

A folyamkezelés lépései a következők:

- Az adatfolyam megnyitása (`open`).
- Amíg információ érkezik:
  - olvasás az adatfolyamból, (ill. írás az adatfolyamba: `read`, `write`).
- Adatfolyam lezárása (`close`).

A folyamok megnyitását a megfelelő osztály példányosításával érhetjük el. Az előálló objektumon ezután a kívánt műveleteket elvégezhetjük, beleértve a lezárást is.

10.2. Kapcsolat a fájlrendszerrel

A Java keretrendszer io csomagjában az alábbi osztályhierarchiát alakították ki. (Itt csak a legfontosabb és leggyakrabban használt osztályokat emeltük ki az átláthatóság kedvéért.)

```
Object
 └ InputStream
     └ ByteArrayInputStream
     └ FileInputStream
     └ FilterInputStream
         └ DataInputStream
         └ BufferedInputStream
           …
     └ObjectInputStream
   …
 └ OutputStream
     └ ByteArrayOutputStream
     └ FileOutputStream
     └ FilterOutputStream
         └ DataOutputStream
         └ BufferedOutputstream
           …
     └ ObjectOutputStream
   …
 └ Reader
     └ InputStreamReader
         └ FileReader
     └ BufferedReader
         └ LineNumberReader
     └ CharArrayReader
   …
 └ Writer
     └ OutputStreamWriter
         └ FileWriter
     └ BufferedWriter
     └ CharArrayWriter
   …
```

Az osztályhierarchia tetején az `InputStream`, az `OutputStream`, a `Reader` és a `Writer` absztrakt osztályok állnak. Az egyes speciális feladatokra szolgáló osztályok ezek valamelyikének a leszármazottjai.

Az `InputStream`, `OutputStream` osztályok a bájtalapú ki- és bemeneti folyamok létrehozásához készült. Itt az input/output (a továbbiakban I/O) alapegysége a bájt, vagyis minden jel 8 bites (0-255 értékű) lehet.

A `Reader`, `Writer` osztályok a karakteres I/O kezelésének ősosztályai. Az adatokat 16 bites unicode karakterként értelmezik. Tipikusan ilyen folyamok kezelik a billentyűzetről érkező és a képernyőre szánt adatokat. Az osztályok feladata a konverzió végrehajtása, amikor a unicode karaktereket átalakítja az adott fájlrendszer, (ill. az explicit kódkészlet megadásával) 8 bites ábrázolási tartományúra, illetve vissza.

Az osztályok nevei beszédesek, hiszen a fenti négy osztály minden leszármazottja nevében viseli az ősosztály alaptulajdonságait. A folyamkezelő osztályok (a megvalósított interfészeknek köszönhetően) hasonló nevű metódusokkal vezérelhetők pl: `read`, `write`, `flush`, `close` (azonos üzenetküldés).

A hierarchiában lefelé haladva találjuk a négy absztrakt osztályból származó osztályokat. A bájt és karakter alapú osztályok között nagyfokú szimmetriát fedezhetünk fel.

Az írás és olvasás művelete számos hibalehetőséget hordoz, hiszen itt az adott hardverrel, vagy az operációs rendszer egyes részeivel történik kommunikáció. Az `IOException` kivételosztály, illetve ennek 20-25 alosztálya segítségével a kivételes események pontosan lekezelhetők (Az egyes metódusokban végrehajtott i/o műveletek előtt biztosítani kell a kivételek kiváltását a `throws IOException` utasítással, vagy egy `try-catch` szerkezettel.)

10.2.1. Adatfolyamok írása, olvasása

Mint említettük, a bájtfolyamok írásához és olvasásához az InputStream és `OutputStream` osztály leszármazottjait használhatjuk. Az osztályok definiálnak egy-egy absztrakt `read()`, ill. `write()` metódust. Ezeken a polimorf műveleteken keresztül specializálták a leszármazott osztályok működését.

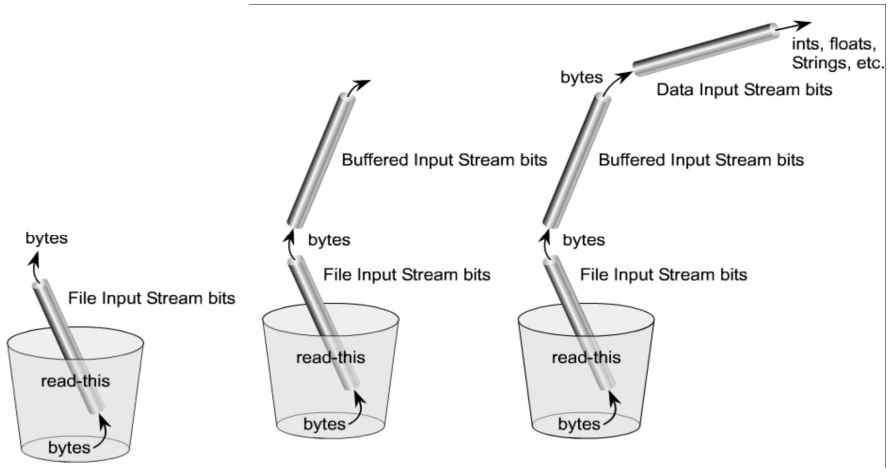
- `FileInputStream`: fájlból olvasást lehetővé tevő bájtfolyam. A `read` metódus segítségével egy fájl tartalmát bájtonként kiolvashatjuk a fájl elejéről folyamatosan. Ha a hivatkozott fájl nem létezik akkor a virtuális gép `FileNotFoundException` kivételt hoz létre.
- `FileOutputStream`: egy adott fájlba író bájtfolyam. A `write` metódus segítségével egy fájlba folyamatosan bájtokat írhatunk. Ha a fájlt nem lehet létrehozni, akkor az `FileNotFoundException` kivételt vált ki.

- `FilterInputStream`, `FilterOutputStream`: Ez a két osztály a szűrő folyamok ősosztálya. Az adattovábbításban, mint közbenső folyam, a bejövő bájtsorozatot egy adott típusú bájtsorozattá, vagy speciális működésű folyammá alakítja. A szűrőfolyamok csak egy már létező folyamból állíthatók elő.
  - `DataInputStream`, `DataOutputStream`: Segítségükkel egy folyam valamilyen alaptípusú, vagy sztring adattá konvertálható. Az adatok bájtfolyammá alakításával egy másik folyamba írhatóak lesznek. Nagyon gyakran használt az egyes konverziós műveleteknél, vagy amikor a gépi adatábrázolást szeretnénk a kimeneten megjeleníteni, vagy beolvasni a `readUTF`, `readBoolean`, `readChar`, `readInt` stb., illetve a `writeBoolean`, writeChar, `writeInt` stb. metódusok segítségével.
  - `BufferedInputStream`, `BufferedOutputStream`: ezen szűrőfolyamok beiktatásakor a memóriában tárolt adatokat a kiírás, vagy olvasás során egy átmeneti tárolóban összegyűjti (puffereli) lecsökkentve az operációs rendszer szintű író/olvasó műveletek számát. A puffer mérete megadható.
  - `PushbackInputStream`: használatával lehetőségünk van egy adott folyamra visszatenni, majd újra olvasni az adatokat (unread).
  - `PrintStream`: az adatfolyamra közvetlenül küldhetünk karaktereket, azok a rendszer alapkódolása alapján automatikusan bájtsorozattá konvertálódnak. A nyelv valamennyi adattípusával használható. Automatikus transzlációt végez, azaz a sorvégjeleket az operációs rendszeren használatossá konvertálja.
- `ByteArrayInputStream`, `ByteArrayOutputStream`: az érkező adatok bájttömbökbe olvasására, ill. bájttömbök kiírására alkalmas.
- `PipedInputStream`, `PipedOutputStream`: alkalmas ún. csövek kialakítására. Segítségükkel megszervezhetjük a párhuzamos programrészek közötti kommunikációt.
- `SequenceInputStream`: a bemeneti csatornák összefűzésére és egymás utáni feldolgozására alkalmas.
- `ObjectInputStream`, `ObjectOutputStream`: Ezen osztályok különlegessége, hogy az objektumokat és a referenciákkal elért adatokat (tömbök, sztringek) is egy adott folyamba foglalhatjuk. Ennek a megvalósításnak a neve a szerializáció (lásd: 10.2.3. fejezet).

A következő példában az alábbi egyszerű író-olvasó feladat megoldását láthatjuk:

Másoljuk át a standard bemenetről érkező karaktereket a kimenetre sorvégjelig! Oldjuk meg a feladatot pufferezetten is!

```
import java.io.*;

public class Olvasas {

  public static void main( String[] args ){
  try {
     int b;
     // Első eset: a bemenet átmásolása a kimenetre:
     System.out.println(
        "A bemenet átmásolása a kimenetre enter végjelig:");
     while (( b = System.in.read())!='\n')
        System.out.write(b);
     System.out.flush();

     // Második eset:
     // A standard bemenetről pufferes folyamot nyitunk  ->
     // a bemenetet átirányítjuk a folyamba -> és visszaírjuk
     System.out.println("\nA bemenet átmásolása pufferesen
        a kimenetre enter végjelig:");
     BufferedInputStream in =
         new BufferedInputStream(System.in);

     while (( b = in.read())!='\n')
        System.out.write(b);
     in.close();
     System.out.flush();

  } catch (IOException e){
     System.err.println(e);
     }
 }
}
```



26. ábra: Adatfolyamok egymásba ágyazhatósága

Most jutottunk el odáig, hogy az eddig gyakran használt `System.out` objektumot megértsük. Eddig a `println` metódussal üzenetet küldtük, hogy valamilyen szöveges adatot megjelenítsünk.

A `System.out` egy előre definiált kimeneti adatfolyam. A Java környezet a `System` végleges osztályban definiálta az operációs rendszer standard input, output és hibacsatornájának elérését.

Ezek olyan osztályszintű adattagok, amelyek maguk is egy-egy folyamként léteznek, így a folyamok standard üzeneteit ismerik és használják. Először egy `int` típusú változóba bájtonként beolvassuk a billentyűzetről érkező adatokat (`System.in`), majd ezeket egyesével kiírjuk (`System.out`).

A második esetben a bemeneti folyamot átalakítjuk, és a bemeneti folyam segítségével előállítunk egy szűrt adatfolyamot, amelyet a Java környezet pufferezetten kezel. Ezután a kimeneti folyamba irányítjuk a bájsorozatot. Hiba esetén a standard hibacsatornára írjuk ki az üzeneteket.

Az operációs rendszerben a standard bemeneti, kimeneti és hibacsatornák át is irányíthatóak. A következő parancs hatására:

```
java Olvasas <input.txt >output.txt 2>error.txt
```

a bemenetet az `input.txt` fájlból veszi és a kimenetet az `output.txt`, a hibákat az `error.txt` fájlban kapjuk.

10.2.2. Karakteres írás-olvasás

Szöveges állományokkal – amelyben az információt karakterek sorozata hordozza – a `Reader` és a `Writer` absztrakt osztályok leszármazottjai segítségével dolgozhatunk, melyek az alapműveleteket (`read`, `write`, `close`) megvalósítják.

A szöveges állományok soronként olvasható ill. írható információkat tartalmaznak. A sorok elválasztását a sorvégjelek (`EOL`) segítségével történik, melyeket transzlációval kell feldolgozni. (A Unix (`CR -'\n'`) és a Windows (`CR+LF -"\r\n"`) operációs rendszerek eltérően kezelik a sorvégjeleket, ezen jelek kölcsönös átalakítása a transzláció).

Az állományokban értelmezett egy fájlmutató jellegű index, amely lehetővé teszi a fájlban történő pozícionálást.

A leszármazott osztályok a bájt szintű adatfolyamokhoz hasonló osztályokat definiálnak:

- `BufferedReader`, `BufferedWriter`: ez a szűrőfolyam pufferezett olvasást és írást támogat, segítségével csökkenthetőek az i/o rendszerhívások, gyorsabb működést eredményez.
  - `LineNumberReader`: az adatállományt úgy alakítja át, hogy a többszörös soremeléseket elhagyja, majd a sorokat megszámozza.
- `CharArrayReader`, `CharArrayWriter`, `StringReader`, `StringWriter`: az adatfájlból közvetlenül karaktertömb illetve sztring típusú folyamba olvashatjuk az adatokat.
- `FilterReader`, `FilterWriter`: a szűrőfolyamok absztrakt osztályai.
  - `PushBackReader`: ezzel a szűrőfolyammal egy folyamból kiolvasott karaktert szükség esetén visszatehetünk az adatfolyamba, majd újra kiolvashatjuk.
- `InputStreamReader`, `OutputStreamWriter`: ez a két osztály jelenti a hidat a karakteres és a bájt szintű folyamok között. Egy adott bájtfolyam olvasásakor egy adott karakterkódolás szerinti konverzióval az érkező adatokat karakterekké konvertálja, és visszafelé egy karaktersorozatot bájtsorozattá konvertál. A karakterkódolás (ISO-8859-1, ISO-8859-2, CP-852, CP-1250, UTF-8, US-ASCII stb.) karakterlánc konstansok valamelyike lehet. Nagyon gyakori a pufferelt folyamokkal együttes használata. Nagyon fontos, hogy a programok író és olvasó részletei ugyanazt a kódolást használják.
  - `FileReader`, `FileWriter`: leszármazottként ez a két osztály is bájt-karakterfolyamok közötti konverzióra használható, azonban

minden esetben az alapértelmezés szerinti kódolást és alapértelmezés szerinti pufferméretet használ.

- `PipedReader`, `PipedWriter`: a bájtfolyamokban tárgyaltakhoz hasonlóan csövekhez, a szálak közötti információcseréhez használt adatfolyam.
- `PrintWriter`: ez a karakterfolyam rengeteg felültöltött print metódust tartalmaz, így hatékonyan formázott kimenetet érhetünk el ennek a folyamnak a felhasználásával. Az alaptípusok mellett lehetőségünk van referencia típusú változók szöveges formában való megjelenítésére. Minden egyes újsor karakter után automatikusan végrehajtja a `flush` metódusát, és kicseréli a soremelés karaktereket az adott környezet sorzáró karaktereire (automatikus transzláció). Az adatfolyam műveletei sohasem váltanak ki `IOException` jellegű kivételt.

Az alábbi példaprogram a karakteres kimenet kezelését mutatja be:

```
import java.io.*;
public class Szovegki {

  public static void main( String[] args )
     throws IOException {

  //Írás a kimenet.txt fájlba FileWriter segítségével,
  //automatikus szövegkonverzió
  FileWriter f = new FileWriter("kimenet.txt") ;
  //Az adott fájlhoz egy folyamot rendelünk
  PrintWriter out = new PrintWriter(f);

  out.write("A fájlba elkezdek szöveget írni.");
  out.print("Folytatom, \n és sort emeltem!\n");
  out.println("Így is írhatok!");
  //A JDK 1.5-től már a C-nyelvben megismert printf függvény
  //is használható:
  out.printf("%20s %5d","De \"C\" szerint így is:" ,300 );

  //A folyam lezárása
  out.close();
  }
}
```

A szöveges adatok feldolgozásához hatékonyan használható a `StreamTokenizer` osztály. Nagyon hasonlít a szűrő jellegű folyamokra: a bemeneti adatfolyam feldolgozásakor az adatfolyamot adategységekre bontja. Ezek az egységek (token) lehetnek azonosítók, számok és sztring

literálok. Használata egyes nyelvi feldolgozó alkalmazások gyors fejlesztését teszi lehetővé. Az osztályban konstansként definiálták az egyes tokenek értékeit:

- `TT_EOF` (-1) – állomány vége,
- `TT_EOL` ('\n') – sorvégjel,
- `TT_NUMBER` (-2) – szám jellegű adat,
- `TT_WORD` (-3) – szöveg vagy azonosító jellegű adat.

Egy szövegfájlból olvassunk be sorokat és bontsuk fel különálló szavakra. Jelenítsük meg a standard kimeneten a szöveg jellegű adatokat!

```
import java.io.*;

// a fájlból érkező szöveget szavakra bontjuk...
class Szavakra {
  public static void main( String args[] )
     throws IOException {

   FileInputStream fajl = new FileInputStream("vers.txt");
   InputStreamReader input = new InputStreamReader(fajl);

   // a folyam tagolásához használt osztály
   StreamTokenizer st = new StreamTokenizer(input);

   while( st.nextToken() != StreamTokenizer.TT_EOF )
       if( st.ttype == StreamTokenizer.TT_WORD )
         System.out.println(st.sval);
   }
}
```

10.2.3. Szerializáció

Amennyiben objektumokkal szeretnénk folyamműveletet végezni, akor ehhez az `ObjectInputStream` és az `ObjectOutputStream` adatfolyamokat használhatjuk. Ez a két osztály definiálja a `readObject()` ill. a `writeObject()` metódusokat. Ezek segítségével az adatfolyamból betölthetünk, illetve kimenthetünk tömb, vagy sztring típusú adatokat, illetve olyan objektumokat, amelyek megvalósítják a `Serializable`, vagy az abból származtatott `Externelizable` interfészeket.

A `Serializable` interfész egyetlen új metódust sem követel meg az osztály definíciójában, csak azt jelezzük, hogy az osztály privát adattagjai részt vehetnek az adatfolyam I/O műveleteiben. (Gondoljuk meg, hogy az

objektum írás- olvasás jellegű műveletek tulajdonképpen az adatok nyilvánosságra hozatalát jelentik, és az adatrejtés alapelve miatt ezt „csak úgy" nem tehetnénk meg.)

Az `Externalizable` interfész megvalósításakor a `readExternal()`, és a `writeExternal()` metódusokat kötelező definiálni, amelyben kifejtjük, hogy az adatok elmentése és visszaolvasása hogyan történjen.

Egy objektum szerializációjakor nemcsak egy objektum adatait, hanem az őt leíró osztály azonosítóját is elmentésre kerül, és a beolvasáskor ezen osztályazonosító szerint történik a beolvasás. Amennyiben az írás és az olvasáa külön modulban van akkor a mentéskor használt osztály teljes definíciójának a fogadó (beolvasó) oldalon is ismertnek kell lennie. A `static` és `tranzient` módosítóval ellátott adattagokat a szerializáció nem menti, beolvasáskor ezek az adattagok alapértelmezett értéket kapnak.

10.2.4. Közvetlen elérésű állományok

Az eddigi adatkezeléssel csak a soros (szekvenciális) adatfeldolgozást tették lehetővé. Azaz minden olvasás és írási folyamat csak meghatározott sorrendben hajtható végre.

A tömbök indexeléses eléréséhez hasonlóan néha szükségünk van a forrásfájlok véletlen sorrendű elérésére, a fájlon belüli pozicionálásra. A `RandomAccessFile` osztály segítségével ezt is megtehetjük. Az osztály által elért fájl esetén meghatározható a hossz (`length`), az aktuális fájlmutató (`getFilePointer`), és szabadon pozicionálható (`seek`). Egy fájl megnyitható csak olvasásra `"r"` illetve írásra és olvasásra `"rw"`. A közvetlen elérésű fájlok is támogatják a korábban bemutatott adatfolyamok kezelését, de ezzel részletesen itt nem foglalkozunk.

10.3. Hálózatkezelés

A Java hálózati könyvtára (`java.net`) jelenleg a TCP/IP protokollcsaládra épül, ami a jelenlegi Internet hálózat alapja. Ennek részleteit csaknem teljesen elfedi a programozó elől. A kommunikációt egyrészt a TCP (Transfer Control Protocol) kapcsolatorientált protokoll, másrészt a datagrammokra (adatcsomag) épülő UDP (User Datagram Protocol) protokoll segítségével valósítja meg. Ezeken túl támogatja a HTTP protokoll közvetlen használatát, de lehetőséget ad egyéb kezelő eljárások (protocol handler) fejlesztésére.

A TCP/IP hálózat szállítási rétege kétfajta, az ún. összeköttetés alapú és összeköttetés-mentes szolgáltatást ismer.

Az összeköttetés alapú szolgáltatást megvalósító TCP protokoll tulajdonképpen a kommunikáló programok között egy kétirányú adatfolyamot valósít meg, amelyen az információ bájtsorozatként áramlik. A kapcsolatorientált szolgáltatást megbízhatónak is nevezik, mert a TCP biztosítja a bájtfolyam torzításmentes, sorrendhelyes átvitelét, kiküszöbölve az elveszett, megsérült, megduplázódott, vagy késve érkezett információcsomagok hatását. Ha pedig semmi nem segít, akkor legalább a küldő, de néha a vevő is értesül arról, hogy valami helyrehozhatatlan hiba történt.

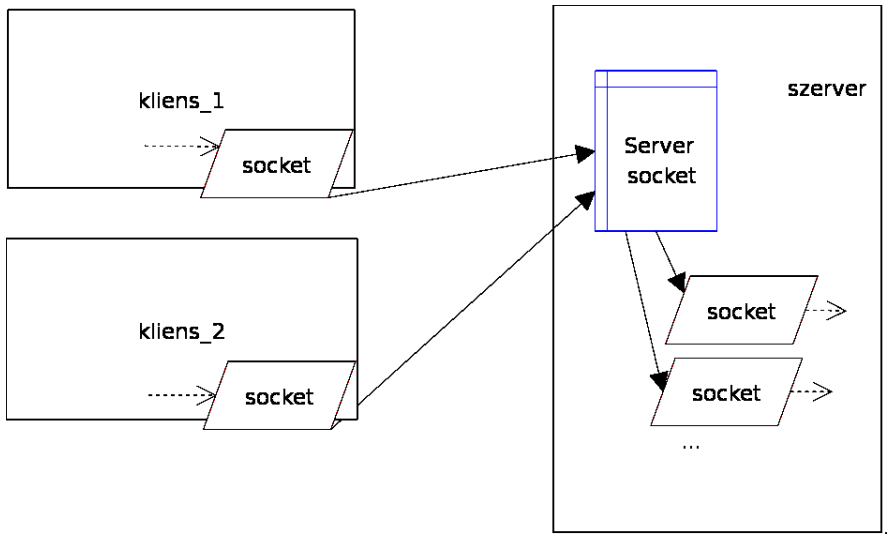
Az alkalmazások egy részének nincs szüksége ekkora megbízhatóságra. Ekkor beérjük az egyszerűbben kezelhető UDP protokoll összeköttetésmentes szolgáltatásával. Ebben az esetben a könyvtárak és a hálózati architektúra mindössze annyit ígér, hogy a csomagba összepakolt bájtokat (maximálisan 1500 bájt) egyben elküldi a címzett felé, és mindent megtesz azért, hogy az odataláljon, de az előforduló hibák lekezelését nem oldja meg.

Mivel mindkét esetben konkrét programok kommunikálnak, ezek egymásra találásához a számítógéphez IP címét ki kell egészítenünk a kommunikációra szolgáló kapcsolódási pontokkal (kapuk, portok). Az első 1024 port használatát általában korlátozzák – csak rendszergazda jogosultságú programok olvashatják – a többi rendelkezésére áll. A kapuk egy része, az ún. jól ismert kapuk (well-known ports) minden számítógépen szabványos funkciókat betöltő programokhoz tartoznak.

A Java környezet hálózatkezelést támogató osztályait a `java.net` csomagban definiálták, használatukhoz a programok elején elérhetővé kell tenni az I/O -kezelést megvalósító osztályokkal együtt:

```
import java.net.*;
import java.io.*;
```

A hálózati kommunikációhoz a foglalatok (socket) nyújtanak alacsony szintű programozási interfészt. Az egyes hálózati végpontok közötti összeköttetést és programok közti adatcserét socket-típusú folyamokon keresztül oldhatjuk meg. A foglalatokról érkező adatfolyamokat egyszerűen más folyamok, szűrőfolyamok bemenetéhez kapcsolhatjuk.



27. ábra: Kliens-szerver kommunikáció felépülése

A Java a fenti hálózati protokollokra különböző típusú socketeket biztosít:

- összeköttetés alaú (kapcsolatorientált) protokoll (`Socket`)
- összeköttetésmentes (kapcsolat nélküli) protokoll (`DatagramSocket`)

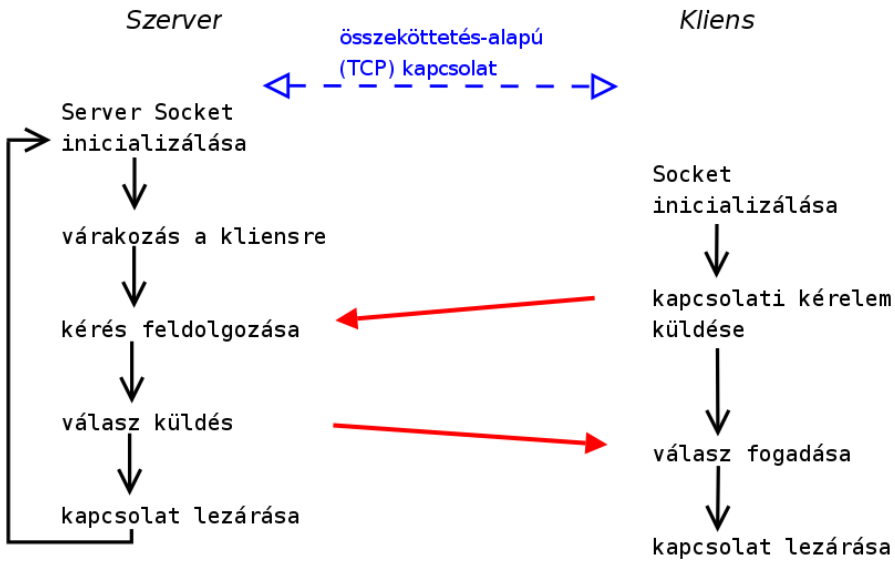
Az alábbiakban a teljesség igénye nélkül látunk néhány példát az egyes hálózatkezelési műveletekre. A témakör mélyebb elsajátításához az irodalomjegyzékben felsorolt műveket ajánlom.

10.3.1. Összeköttetés-alapú kommunikáció

Az összeköttetés-alapú kommunikációt a TCP protokoll `Socket` és `ServerSocket` osztályai valósítják meg, mivel a hálózatos adatforgalomban részt vevő alkalmazásokat működési elvük szerint két csoportba sorolhatjuk: szerverek és kliensek. A kliensek kapcsolatokat kezdeményeznek, a szerverek pedig ezeket a kéréseket kiszolgálják.

A `Socket` osztály egy-egy példánya teszi lehetővé a kommunikációt mind a kliens, mind a szerveroldalon. A szerveralkalmazás egyetlen `ServerSocket` típusú objektumot használ a különböző kliensekkel való kapcsolatfelvételre, de ezt követően minden egyes klienskapcsolatot már külön `Socket` típusú objektum jelképez. A kliensek csatlakozásakor ez a `ServerSocket` objektum kezeli és szolgálja ki a klienseket – egy-egy

`Socket` objektumot rendel az érkező kapcsolati kérelemhez – illetve konkurrens kérés esetén feldolgozza a várakozási sorokat.



28. ábra: Az összeköttetés-alapú kapcsolat kommunikációs idődiagramja

A kliens oldal

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

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

Kapcsolati kérelmet a kliensoldalon, az alábbi a példához hasonlóan tehetjük meg:

```
try {
  Socket s = new Socket("www.sze.hu", 80);
} catch (UnknownHostException e) {
  System.out.println("A szerver nem válaszol!");
}catch (IOException e) {
  System.err.println("Sikertelen kapcsolatfelvétel!");
}
```

Ha egyszer egy kapcsolatot sikerült felépíteni, akkor kinyerhetjük belőle a kommunikációhoz szükséges folyamot – igaz sokszor csak a folyamok többszörös egymásba ágyazásával:

```
try {
  //Kommunikációs végpont felállítása
  Socket s = new Socket("localhost", 12345);

  //Egy kimenő folyamot ráirányítunk a socketre
  OutputStream out = s.getOutputStream();

  // A kimenő folyamot egy karakteres folyam végéhez adjuk
  PrintWriter output = new PrintWriter(out);
  output.println("Helló én egy kliens vagyok!");

  //Adat olvasó folyam a kommunikációs végponton
  BufferedReader input = new BufferedReader(
     new InputStreamReader(s.getInputStream()));
  //Sor(ok) kiolvasása
  String st = input.readLine();
  ...
  //Kommunikáió lezárása
  s.close();
  } catch (UnknownHostException e) {
    System.err.println("A szerver nem válaszol!");
  }
…
```

A szerver oldal

A szerver alkalmazások a `ServerSocket` osztály egy példányának accept() metódusát használják arra, hogy a kliensek kapcsolatkéréseire várjanak. Ez a metódus hozza létre a klienssel való adatcseréhez szükséges `Socket` típusú objektumot.

Miután egy kapcsolat felépült, a szerver már ugyanolyan típusú `Socket` objektumokat használ, mint a kliensalkalmazás. A `ServerSocket` objektum csak a kapcsolat felépítéséhez szükséges.

```
try {
  Boolean vege = false;
  ServerSocket ss = new ServerSocket(12345);
  while (!vege) {
     //Felvesszük a kapcsolatot a következő klienssel
     //A szerver itt várakozik egy új kliensre, ha kell
     Socket s = ss.accept();

     //A kommunikációs végpontról adatot olvasó folyam
     BufferedReader input = new BufferedReader(
        new InputStreamReader(s.getInputStream()));

     //A kommunikációs végpontra adatot író folyam
     PrintWriter output = new PrintWriter(
        s.getOutputStream(), true); // AutoFlush

     //Adatfeldolgozás, adatküldés
     String st = input.readLine();
     System.out.println("Klienstől érkezett: " + st);
     output.println("Viszlát, és kösz!");

     //Kapcsolat lebontása
     s.close();
     if (leállási feltétel) vege =true;
     }
  //A teljes kommunikáció lezárása
  ss.close();
} catch (IOException io) {
  System.err.println("Szerver hiba! "+io);
}
```

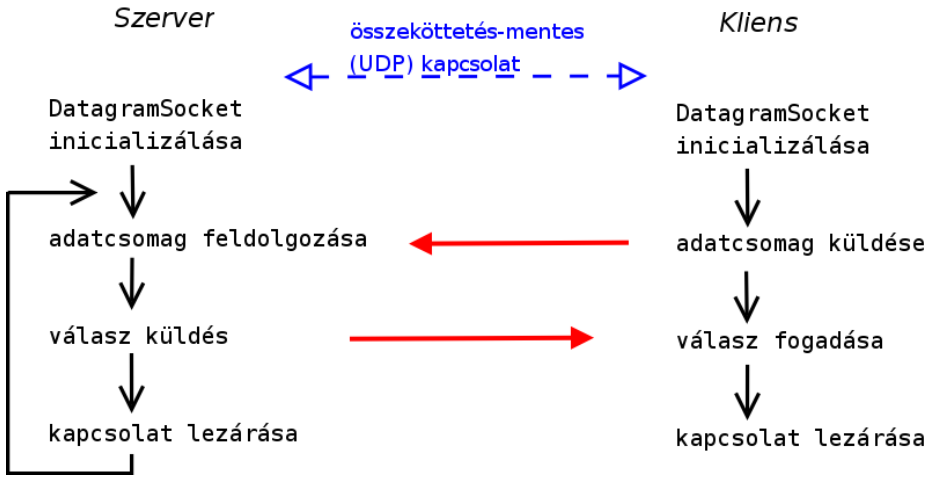
10.3.2. Összeköttetés-mentes kommunikáció

Az összeköttetés-mentes hálózati kommunikáció is a kliens-szerver modell alapján építhető fel. A kommunikációs csatornát az UDP protokoll szerinti csomagkapcsolt összeköttetés testesíti meg. Az adatáramlást megbízhatatlannak kell tekintenünk.

A megvalósításhoz a Java környezet a `DatagramSocket` és a `DatagramPacket` osztályokat nyújtja. A `DatagramSocket` osztály a kommunikációs végpontok folyamkezelését, a `DatagramPacket` pedig az elküldendő adatcsomagokat kezeli. Ez utóbbi tartalmazza a kiinduló és a célállomás címét is.

Egy adatcsomag elküldéséhez először egy adatcsomag `(DatagramPacket)` objektumot hozunk létre, amely tartalmazza:

- az elküldendő adatot,
- az elküldendő adat hosszát,
- a célszámítógép Internet címét,
- a célszámítógép UDP portjának azonosítóját.



29. ábra: Az összeköttetés-mentes kommunikáció időDiagramja

Az adatcsomag objektumot ezután átadjuk egy `DatagramSocket` objektumnak, amelyik ezt a csomagot útjára bocsátja a `send()` metódus segítségével.

```
String szoveg = "Ez itt az átküldendő üzenet!";
int server_port = 123456;
try {
   //Kommunikációs végpont
   DatagramSocket s = new DatagramSocket();

   try {
     //A szerver címe
     InetAddress ip_cim =
        InetAddress.getByName("valami.valahol.hu");
     //Az üzenet hossza és konvertálása
     int uzenet_hossz = szoveg.length();
     byte[] uzenet = szoveg.getBytes();
     //Az UDP csomag összeállítása
     DatagramPacket p =
      new DatagramPacket(uzenet, uzenet_hossz,
                         ip_cim, server_port);
```

```
        //Elküldjük a csomagot a kommunikációs végpontról
        s.send(p);

        //Hibakezelés
        } catch (UnknownHostException e) {
           System.err.println("Ismeretlen hoszt!");
        } catch (IOException e){
             System.err.println("Nem sikerült a küldés!");
        } finally {
           // megszüntetjük a kommunikációs végpontot
           s.close();
        }
    } catch (SocketException e) {
      System.err.println("UDP port foglalása nem sikerült!");
}}
```

Adatcsomag fogadásakor küldéshez képest fordított utat járunk be. Egy `DatagramSocket` objektummal – amely egy adott UDP portot figyel – várakozunk az érkező csomagra. Az érkező adatok fogadására létre kell hozni egy `DatagramPacket` objektumot akkora adatterülettel, ahova az érkező csomag belefér. Ezután a `DatagramSocket` objektum `receive()` metódusával az első érkező adatcsomagot – ha még nem érkezett semmi, akkor megvárja – visszaadja a `DatagramPacket` objektumnak.

```
int server_port = 123456;
try {
  //Helyet foglalunk az üzenetnek
  byte[] uzenet = new byte[1500];
  String szoveg;
  //A csomag, amely a beérkező adatokat fogadja
  DatagramPacket p =
     new DatagramPacket(uzenet, uzenet.length);
  //A szerver kommunikációs végpontja
  DatagramSocket s = new DatagramSocket(server_port);
  //várunk egy csomagot, és a tartalmát beolvassuk p-be
  s.receive(p);

  //A kapott üzenet tartalmát Stringgé konvertáljuk,
  szoveg = new String(uzenet, 0, p.getLength());
  // majd kiírjuk
  System.out.println("Csomag érkezett a " +
     p.getAddress().getHostName() + " számítógéptől, a " +
     p.getPort() + ". UDP portról.\n Az üzenet: " + szoveg);
```

```
//A kommunikációs végpont lezárása
  s.close();
} catch (UnknownHostException e) {
  System.err.println("Ismeretlen hoszt!");
} catch (IOException e) {
  System.err.println("Nem sikerült az adatfogadás!");
}
```

10.3.3. URL-ek használata

Az URL (Uniform Resource Locator) az Interneten elhelyezkedő erőforrások (bármilyen típusú adatforrást) egységes forrásazonosítója. Azonkívül, hogy meghatározzák egy adattömeg helyét az Interneten, információt tartalmaznak az adatok kinyeréséhez használható protokollról is. Az URL-ek többnyire szöveges adatként jelennek meg.

```
protokoll://kiszolgáló/hely/objektum
```

A Java nyelvben az URL-eket az `URL` osztály egy példánya reprezentálja. Egy `URL` objektum alkalmas valamely URL-lel kapcsolatos összes információ kezelésére, valamint segítséget nyújt az általa meghatározott objektum kinyeréséhez. `URL` objektum kétféleképpen hozható létre:

```
URL honlap = new URL("http://www.sze.hu/index.html" ) ;
URL honlap = new URL("http", "www.sze.hu", "index.html" );
```

Az `URL` objektum létrejön függetlenül attól, hogy az általa reprezentált objektum létezik-e vagy sem. A létrehozás során az URL objektum nem teremt hálózati kapcsolatot a kiszolgáló géppel.
A létrehozás alatt a Java elemzi az URL specifikációját és kiemeli belőle a használni kívánt protokollt, majd megvizsgálja, hogy ismeri-e. Abban az esetben ha nem ismeri MalformedURLException kivétel keletkezik.

Jelenítsük meg a kimeneten egy URL-lel azonosított honlap forrását.

```
import java.net.*;
import java.io.*;
public class URLLista {
  public static void main(String [] args) {
    try {
     URL honlap = new URL("http://szt1.sze.hu/index.html");
     //Az URL-lel azonosított erőforrás kiolvasása folyammal
     BufferedReader input = new BufferedReader(
        new InputStreamReader(honlap.openStream()));
     String st;
```

```
        do {
           st = input.readLine();
           System.out.println(st);
        } while (st != null);
        input.close();
        } catch(Exception e) {
            System.out.println(e);
    }}
}
```

A programban az adott URL-lel azonosított adatot egy pufferezett karakteres folyamként megnyitjuk, majd soronként kiíratjuk a kimenetre.

10.4. Kérdések

- Melyik csomagban találjuk a fájl, és melyikben a hálózatkezelést támogató osztályokat?
- Mit nevezünk adatfolyamnak?
- Melyik az a négy fő adatfolyam típus, amelyet a Java nyelv használ?
- Összekapcsolhatók-e a Java adatfolyamai? És ha igen, hogyan?
- Mit jelent az objektumok szerializációja?
- Mi az a socket erőforrás?
- Mi a különbség az összeköttetés-alapú, és az összeköttetés-mentes kommunikáció között?
- Mi a kliens és a szerver jellegű programok főbb szerepei?
- Mi jellemzi az URL osztályt?

10.5. Feladatok

- Egy tetszőleges szöveges fájlt olvasson be az aktuális könyvtárból, majd készítsen az előforduló karakterekből betűstatisztikát.
- Az előző feladat kimenetét állítsa elő folyam segítségével karakteres fájlban (.txt). Az előállú szövegfájl legyen megfelelően tagolt és olvasható.
- Egy tetszőleges feladatban szereplő osztályokhoz készítsen olyan metódusokat, melyek az objektumok adatait egy-egy szövegfájlban tárolják.
- Egy "`matrix.bin`" fájlban kb. 500 db `long` típusú számot szeretnénk letárolni. Készítsen olyan `Matrix` osztályt amely ezzel a fájllal tud dolgozni.

a) Készítse el azt a `general(int n)` osztálymetódust, amely a fenti fájlba n db long értéket ír ki.

b) Készítsen olyan `Matrix(int n, int m)` konstruktort, amely az inicializálás során a fenti fájlból veszi a kedőértékét (ha nincs ilyen fájl, akkor először generáljon). Ha a forrásfájlban nincs elegendő érték a mátrix feltöltéséhez, akkor a többi elem nulla kezdőértéket kapjon.

Egy különálló tesztprogramból tesztelje az alkalmazást.

- Készítsen egy osztályt a XIX. és XX. századi időjárási adatokra (éves max. középhőmérséklet, éves min. középhőmérséklet, éves átlaghőmérséklet, az év legmelegebb, ill. leghidegebb napja stb.) Tárolja a tetszőleges módszerrel felvett adatokat szerializáció segítségével a „idojaras.dat” állományban. A program legyen képes kiolvasni, illetve írni az állományt.
- Készítsen a 8×8-as pályán játszott torpedó játékhoz olyan kliens-szerver programot, melyben egy szerverhez egy kliens csatlakozhat. A hálózati kommunikációban átküldendő az adott koordináta, illetve válaszként a találat jelzése. A szerver generáljon bizonyos koordinátapontokra hajókat, a kliens ezeket a hajókat célozhatja.
- Alakítsa át úgy a programot, hogy egy szerverhez két kliens csatlakozhasson, amelyek ekkor egymás ellen játszhatnak, és a szerver csak közvetít!

11. Fordítási hibaüzenetek

A Java nyelv alapkoncepciója szerint a leggyakoribb programozási hibáknak már a fordítás során ki kell derülnie. Ebből a szempontból a fordító a C, C++ nyelvnél szigorúbb szintaktikai és szemantikai ellenőrzést végez. Az alábbiakban a fordítási időben leggyakrabban előforduló hibaüzenetek főbb csoportjait találhatjuk:

Hibaüzenet	**A hiba oka**
`; missing`	Az utasítás végéről hiányzó pontosvessző: if, if-else szerkezeteknél gyakori.
`already defined`	Egy változót vagy metódust másodszor is definiáltunk.
`ambiguous class`	Kétértelmű osztályhivatkozás. Amikor két importált csomagban van azonos nevű osztály.
`array not initialised`	Egy tömb csak deklarálva lett, de elfelejtettünk hozzá tárterületet rendelni a `new` operátorral.
`can't access class`	Egy osztály nem elérhető, valószínű nem publikus.
`can't be instantiated`	Absztrakt osztályból, vagy interfészből próbáltunk példányt létrehozni.
`does not override abstract method`	Egy absztrakt osztályt leszármaztattunk, vagy egy interfészt bejelentettünk az osztályban, de a benne deklarált műveletet nem fejtettük ki.
`cannot override`	A leszármazott osztály nem tudja felüldefiniálni az ős műveletét, ha az `final`, vagy ha a jogosultságok nem megfelelőek.
`cannot resolve symbol`	Nem található adott nevű csomag, osztály, adat, metódus. Gyakori elírási hiba.
`class has wrong version`	A különböző verziójú fordítóprogramokkal fordított kódok esetén lép fel.
`class not found`	Hiányzó osztálydefiníció, a CLASSPATH környezeti változó nincs beállítva, vagy a csomagelérést elrontottuk.
`class or interface`	{} párok számossági hibája esetén a fordító nem

`expected`	tudja azonosítani az osztályokat, vagy elfelejtettünk osztályt definiálni.
`duplicate methods, method already defined`	Két azonos szignatúrájú metódust próbáltunk definiálni.
`Exception never thrown`	Az I/O, vagy más kivételeket nem kezeltünk le a metódusban.
`identifier expected`	{} hibák esetén a fordító nem tudja értelmezni az azonosítókat, vagy azonosító nélküli típust adtunk meg.
`incompatible type`	Értékadáskor a fordító nem tud automatikusan konvertálni, explicit konverzió szükséges.
`main must be static void`	Egy futtatható osztálynak rendelkeznie kell a `public static void main (String[] args) {}` metódussal.
`method matches constructor name`	Egy osztály konstruktorának nem szabad megadni visszatérési értéket, az alapértelmezetten az osztály típusát kapja.
`method not found`	Hiányzó, vagy nem megfelelő paraméterekkel hívott metódus.
`not initialised`	Lokális változó használata kezdőérték adás nélkül.
`possible loss of precision`	Automatikus konverziókor értékvesztés történhet.
`undefined variable`	Egy változót azelőtt használnánk, mielőtt deklaráltuk volna.
`unclosed character literal`	Lezáratlan karakterláncok, hiányzó aposztróf jelek: '.
`unreported exception`	Az adott kódrészlet valószínűleg kivételeket válthat ki. Az utasításokat `try-catch` blokkba kell foglalni, vagy a befoglaló metódusnak dobnia kell az adott típusú kivételt (`throws`).
`unsorted switch`	Két azonos értékű elágazás egy `switch`-ben.
`type expected`	Adattagok, lokális változók típus megadása nélkül.

12. Kódolási szabványok

A felmérések szerint a szoftverek fejlesztési idejének 80%-a karbantartással és a programok elemzésével telik. Napjainkban a szoftvereket ritkán készíti el egyetlen személy, így az átláthatóságra és az egyértelmű értelmezhetőségre oda kell figyelni. Amennyiben valaki valamely munka forrásait közzéteszi, annak jól felépítettnek és átláthatónak kell lennie.

A kódolási előírások/megállapodások nagyban elősegítik a források átláthatóságát, és lehetővé teszik a programozóknak egy ismeretlen kód gyors és mélyreható értelmezését. A tiszta kódolás segít elkerülni a szemantikai szempontból eredő hibákat.

Az alábbiakban az eredeti kódolási szabványok magyar fordítását közöljük, amelynek eredetije az Interneten elérhető [6].

Megjegyezzük, hogy a fejezetben szereplő „szabályok” nagyrészt csak ajánlások, amelyek a források egységes értelmezését segítik, amitől a források olvashatók és áttekinthetőbbé válnak.

12.1. Fájlnevek

Egy Java nyelvű forrás fájlnak `.java` kiterjesztéssel kell rendelkeznie, míg a lefordított bájtkód a `.class` kiterjesztést kapja.

12.1.1. Általánosan használt nevek

Általánosan használt neveket használjunk makefile esetén (GNUmakefile), ha a gnumake alkalmazást használjuk az alkalmazás fordítására, és az adott könyvtár tartamának szöveges leírását tartalmazó fájl esetén (README).

12.2. Fájlok felépítése

Egy fájl szakaszokból áll, melyeket üres sorokkal kell elválasztani, illetve ajánlott megjegyzéseket tenni, azonosításképpen. A 2000 programsornál hosszabb fájlokat lehetőleg kerüljük. A dokumentum végén található egy példa a megfelelően formázott Java forrásfájlra.

12.2.1. Java forrásfájlok

Minden Java forrásfájl tartalmaz egy publikus osztályt vagy interfészt. Ha ehhez tartoznak privát osztályok vagy interfészek, akkor azokat nyugodtan

elhelyezhetjük ebben a fájlban. A publikus osztálynak vagy interfésznek kell az elsőnek lennie a forrásban.

A Java forrásnak követnie kell a következő sorrendet:

- Kezdő megjegyzések.
- Csomag- illetve importdeklarációk.
- Osztály- és interfészdeklarációk.

12.2.2. Kezdő megjegyzések.

Minden forrás elején egy C-stílusú megjegyzésnek kell lennie, amely tartalmazza az osztály nevét, verzióval kapcsolatos információkat, dátumot és copyright megjegyzéseket:

```
/*
* Osztálynév
*
* Verzióval kapcsolatos információk
*
* Dátum
*
* Copyright megjegyzések
*/
```

12.2.3. Csomag- és import deklarációk

Az első sor a Java forrásban, mely nem megjegyzés, az a csomagdeklaráció. Ezután az importdeklarációk következhetnek. Például:

```
package java.awt;
import java.awt.peer.CanvasPeer;
```

Az egyedi csomag azonosító első tagja mindig kisbetűkből álljon, és top-level domain azonosító legyen, azaz `com`, `edu`, `gov`, `mil`, `net`, `org` vagy a kétbetűs ország azonosító, melyeket az ISO Standard 3166, 1981definiál.

A Java API-ban definiált csomagok is bejelenthetők, de megjegyezzük, hogy a `java`, `javax` és `sun` azonosítók a Sun Microsystems számára fenntartottak.

12.2.4. Osztály és interfész deklarációk

A következő táblázat leírja az osztály- vagy interfészdeklaráció elemeit olyan sorrendben, ahogy a forrásban szerepelniük kell.

1. Osztály/interfész
   dokumentációs megjegyzés
   (/** */)

2. Osztály/interfész fejrésze.
3. Osztály/interfész implementációs megjegyzések, ha szükségesek. Ez a megjegyzés tartalmazhat osztály vagy interfész szintű kiegészítő információkat, melyek nem kerültek be a dokumentációs megjegyzésbe.
4. Osztályváltozók. Először a nyilvános, majd a védett (protected), majd a félnyilvános tagok, végül a privát tagok.
5. Példányváltozók. Követve az előbbi sorrendet.
6. Konstruktorok
7. Metódusok. Ezeket a metódusokat inkább a működés alapján kell csoportosítani, mint a hozzáférhetőség vagy láthatóság alapján. Például egy privát metódus szerepelhet két publikus metódus között. A cél, hogy a kód olvasása és megértése egyszerűbb legyen.

12.2.5. Behúzás

A behúzás alapegysége négy szóköz karakter. A négy szóköz elhelyezésének pontos módja nincs specifikálva (lehet használni szóközt vagy tab-ot is). A tab-nak pontosan nyolc szóközből kell állnia, nem négyből.

Sorhossz

A 80 karakternél hosszabb sorok kerülendők, mivel több terminál és fejlesztőeszköz nem tudja megfelelően kezelni.

Megjegyzés: A példák a dokumentációkban általában rövidebbek, a sorhosszak nem nagyobbak, mint 70 karakter.

Sortörések

Ha egy kifejezés nem fér el egy sorban, akkor a sortörést a következő alapelvek szerint kell végezni:

- Sortörés egy vessző után.
- Sortörés egy operátor előtt.
- Magas szintű törések preferálása az alacsonyszintűekkel szemben.
- Az új sor elhelyezése az előző sorban lévő azonos szintű kifejezés kezdetéhez igazítva.
- Ha a fenti szabályok a kód összezavarásához vezetnének, vagy a kód a jobb margónál sűrűsödne, akkor használjunk nyolc szóközt a többszörös behúzás helyett.

Álljon itt két példa a metódushívásokon belüli sortörésekre:

```
someMethod(longExpression1, longExpression2,
    longExpression3,longExpression4, longExpression5);
var = someMethod1(longExpression1,
        someMethod2(longExpression2,
                longExpression3));
```

A következő két példa az aritmetikai kifejezéseken belüli sortörésekre hoz példát. Az első az ajánlott, mivel a sortörés a magasabb helyen lévő zárójelezett kifejezésen kívül van.

```
longName1 = longName2 * (longName3 + longName4 - longName5)
          + 4 * longName6; // Ajánlott
longName1 = longName2 * (longName3 + longName4
- longName5) + 4 * longName6; // Elkerülendő
```

Következik két példa a metódusdeklaráció behúzására. Az első a hagyományos eset. A második esetben a második és a harmadik sort nyolc szóközzel húztuk be, ugyanis a sorok a jobb oldalon sűrűsödnének a konvencionális megoldás használata esetén.

```
// Konvencionális behúzás
someMethod(int anArg, Object anotherArg,
           String yetAnotherArg,Object andStilAnother) {
...
}
// Többszintű behúzás esetén nyolc szóköz használata
private static synchronized horkingLongMethodName(
        int anArg,
        Object anotherArg,
        String yetAnotherArg,
        Object andStilAnother) {
...
}
```

Sortörésnél if kifejezés esetén a nyolc szóközös szabályt alkalmazzuk, ha a konvencionális (4 szóközös) szabály a törzset túl bonyolulttá tenné. Példa:

```
//Ne használjuk ezt a behúzást
if ((condition1 && condition2)
 || (condition3 && condition4)
 || (condition5 && condition6)) { //A rossz sortörések miatt
 doSomethingAboutIt(); //ez a sor könnyen
} //eltéveszthető
//Helyette használjuk ezt
if ((condition1 && condition2)
        || (condition3 && condition4)
        || (condition5 && condition6)) {
    doSomethingAboutIt();
}
//Vagy ezt
if ((condition1 && condition2) || (condition3 && condition4)
        || (condition5 && condition6)) {
    doSomethingAboutIt();
}
```

Példák a feltételes kifejezés használatára:

```
alpha = (aLongBooleanExpression) ? beta : gamma;
alpha = (aLongBooleanExpression) ? beta
                                 : gamma;
alpha = (aLongBooleanExpression)
      ? beta
      : gamma;
```

12.3. Megjegyzések

A Java programokban kétfajta megjegyzések lehetnek: implementációs és dokumentációs megjegyzések. Az implementációs megjegyzések vannak C++ nyelvű forrásszövegben is, melyeket a `/* */`, vagy a `//` szimbólumok jelölnek. A dokumentációs megjegyzések csak Java nyelvben léteznek, ezeket a `/** */` zárójelpárokkal kell jelölni. A `javadoc` programmal ezek a megjegyzések bekerülnek egy HTML fájlba sok egyéb információval együtt. Az implementációs megjegyzésekkel a forráskód szövege magyarázható. A dokumentációs megjegyzések ezzel szemben a specifikációt magyarázzák, mellőzve az implementációs kérdéseket, és azokhoz a fejlesztőkhöz szólnak, melyeknek nincs szükségük az adott rész forráskódjára.

A megjegyzések szükség esetén nyújtsanak áttekintést, illetve olyan plusz információkat tartalmazzanak, amelyek nem szembetűnők a kód egyszeri elolvasásakor. A megjegyzéseknek csak olyan információkat kell tartalmaznia, melyek segítik a program olvashatóságát, illetve megértését.

Nem triviális tervezői döntések magyarázata odaillő, viszont kerülni kell az olyan információk ismétlését, melyek adottak (és világosak) a kódban. A redundáns adatok módosítása többletmunkát eredményez. A megjegyzések önnyen elévülhetnek, ezért kerülni kell minden olyan megjegyzést, mely a program változása során érvényét vesztheti.

Megjegyzés: A megjegyzések aránytalanul nagy mennyisége takargathatja a kód gyenge minőségét. Ezért új megjegyzés beszúrása helyett érdemes elgondolkozni a kód érthetőbbé tételén. A megjegyzéseket nem szabad körülrajzolni karakterekkel. A megjegyzésekben kerüljük a speciális karaktereket használni.

12.3.1. Implementációs megjegyzések formátumai

Az implementációs megjegyzéseknek négy alakja lehet: blokk, egysoros, utó- és sorvégi megjegyzések.

Blokk megjegyzések

A blokk megjegyzések használhatóak fájlok, metódusok, adatstruktúrák és algoritmusok leírására. Ezen kívül máshol is elhelyezkedhetnek, például metódusokon belül. A blokk megjegyzéseknek egy metóduson belül azonos szinten kell lennie a kóddal, amelyet magyaráz.

A blokk megjegyzésnek egy üres sorral kell kezdődnie, mely elválasztja a megjegyzést a kódtól.

```
/*
* Ez egy blokk megjegyzés.
*/
```

A blokk megjegyzések `/*-` szimbólumsorozattal kezdődnek, mely jelzi a behúzást végző programnak (indent), hogy a megjegyzést nem lehet átformázni.

```
/* -
* Ez egy blokk megjegyzés speciális formátumozással,
* melynél az újraformázás kerülendő.
*
* egy
* kettő
* három
*/
```

Megjegyzés: Ha nem használjuk az behúzást végző programot, akkor nem kell alkalmazni a /*- karaktersorozatot, de érdemes, abban az esetben, ha más akarja használni a mi kódunkon.

Egysoros megjegyzések

Rövid megjegyzések egy sorban alkalmazhatók, azon a szinten, melyen az előző sor is van. Ha a megjegyzés nem írható egy sorba, akkor a blokk megjegyzést kell alkalmazni. Az egysoros megjegyzés előtt egy üres sornak kell állnia. Példa:

```
if (feltetel) {

/* Ha a feltétel teljesül. */
...
}
```

Utómegjegyzések

Nagyon rövid megjegyzések kerülhetnek egy sorba a kódrészlettel, melyet magyaráznak, viszont megfelelő távolságra kell attól tenni. Ha több mint egy megjegyzés tartozik egy köteg programsorhoz, akkor azokat ugyanolyan távolságra kell tenni.

```
if (a == 2) {
return TRUE;            /* speciális eset */
} else {
return isPrime(a);      /* csak páratlan számok esetén */
}
```

Sorvégi megjegyzések

A // karakterekkel megjegyzésbe lehet tenni egy teljes sort, illetve egy sor végét. Nem szabad használni többsoros megjegyzésekhez, abban az esetben viszont lehet, ha a forráskódból több egymás utáni sort akarunk megjegyzésbe tenni.

Példa a három használati módra:

```
if (foo > 1) {
    // Dupla dobás.
    ...
}
else {
return false; // Miért itt?
}
```

```
//if (foo > 1) {
//
//   // Dupla dobás.
// ...
//}
//else {
// return false;
//}
```

Dokumentációs megjegyzések

A dokumentációs megjegyzések jellemzik a Java osztályokat, interfészeket, konstruktorokat, metódusokat és adattagokat [14]. Minden dokumentációs megjegyzést a `/**` és `*/` karaktersorozatok határolnak, osztályonként, interfészenként és tagonként egyszer. Ennek a megjegyzésnek a deklaráció előtt kell szerepelnie:

```
/**
 * Az Example osztály ...
 */
public class Example { ...
```

Megjegyezzük, hogy a legfelső szintű osztályok és interfészek nincsenek behúzva, míg azok tagjai igen. Az osztályokhoz és interfészekhez tartozó dokumentációs megjegyzések első sora (`/**`) nincs behúzva, a következő sorok be vannak húzva egy karakterrel (hogy a csillagok egymás alá essenek). A tagoknál, beleértve a konstruktort is, a behúzás az első sorban négykarakternyi, míg öt a rákövetkezőkben. Ha több információt kell megadni az osztályhoz, interfészhez, változóhoz vagy metódushoz, de nem akarjuk, hogy a dokumentációban megjelenjen, implementációs blokkot, vagy egysoros megjegyzést kell használni azonnal a deklaráció után.

Dokumentációs megjegyzéseket nem lehet egy metóduson vagy konstruktoron belül hagyni, ugyanis a Java a megjegyzés utáni első deklarációra vonatkoztatja a megjegyzést.

12.4. Deklarációk

Deklarációk soronkénti száma

Soronként egy deklaráció ajánlott, a megjegyzések miatt is. Azaz

```
int level; // szint
int size; // tábla mérete
```

ajánlott az

```
int level,size;
```

helyett.

Egy sorba semmiképpen ne tegyünk különböző típusú deklarációkat. Például:

```
int foo, fooarray[]; //HIBÁS!
```

Megjegyzés: A fenti példák egy szóköz helyet hagytak a típus és a név között. Másik elfogadható választási lehetőség a tabok használata, azaz:

```
int     level; // szint
int     size;  // tábla merete
Object  currentEntry; // kiválasztott táblabejegyzés
```

Inicializálás

Érdemes a lokális változókat ott inicializálni, ahol deklarálva lettek. Az egyetlen eset, amikor ez nem használható, ha a kezdőérték valamilyen számítás eredménye.

Elhelyezés

A deklarációkat lehetőleg a blokk elejére tegyük. Blokk a kapcsos zárójelek { } által határolt rész. Ne várjunk a deklarációval addig, míg a változót először használatba vesszük.

```
void myMethod() {
    int int1 = 0;       // metódus törzsének kezdete
    if (condition) {
        int int2 = 0;   // if törzsének kezdete
     ...
    }
}
```

A szabály alóli kivétel a `for` ciklus fejében deklarált változó:

```
for (int i = 0; i < maxLoops, i++) { ... }
```

Azon deklarációk, melyek egy magasabb szinten deklarált nevet elfednek, kerülendők! Például nem ajánlott ugyanolyan névvel egy másik változót deklarálni:

```
int count;
...
myMethod() {
  if (condition) {
    int count; // KERÜLENDŐ!
    ...
  }
  ...
}
```

12.4.1. Osztály és interfész deklarációk

Java osztályok és interfészek deklarációjánál a következő szabályokra kell ügyelni:

- Nincs szóköz a metódus név és a formális paraméterlistát kezdő nyitó zárójel között.
- A metódus fejével egy sorban kell a nyitó kapcsos zárójelnek (``{``) lennie.
- A metódusokat üres sorral kell egymástól elválasztani.
- A záró kapcsos zárójelnek } egymagában kell a sorban állnia, egy oszlopban a fej első karakterével, kivéve, mikor a metódus törzse üres, ekkor egyből a nyitó zárójel után kell állnia.

```
class Sample extends Object {
    int ivar1;
    int ivar2;
    Sample(int i, int j) {
      ivar1 = i;
      ivar2 = j;
    }

    int emptyMethod() {}
...
}
```

12.4.2. Utasítások

Egyszerű utasítások

Minden sornak csak egy utasítást szabad tartalmaznia, például:

```
argv++; // Helyes
argc++; // Helyes
argv++; argcc++; // KERÜLENDŐ!
```

Összetett utasítások

Az összetett utasítások olyan utasítások listája, melyek kapcsos zárójelek között vannak: { utasítások }.

- A kapcsos zárójelek közötti utasításokat egy szinttel beljebb kell húzni.
- A nyitó zárójelnek azon sor végén kell szerepelnie, mely megkezdi az
- összetett utasítást; a csukó zárójelnek sor elején kell szerepelnie, annyival behúzva, mint a kezdő utasítás.
- Zárójeleket akkor is használni kell, ha egy vezérlési szerkezet törzseként csak egy utasítást szeretnénk szerepeltetni. Ez megkönnyíti a törzs bővítését, anélkül, hogy valami hibát vétenénk a zárójelek elhagyása miatt.

A return utasítás

A `return` utasításnál nem szükséges használni a zárójeleket, ha a visszatérési érték így is nyilvánvaló. Példák:

```
return;
return myDisk.size();
returt (size ? size : defaultSize);
```

Az if, if-else, if else-if else utasítások

Az `if-else` szerkezeteket a következő alakban kell használni:

```
if (feltetel) {
    utasitasok;
}
if (feltetel) {
   utasitasok;
} else {
    utasitasok;
}
if (feltetel) {
    utasitasok;
} else if (feltetel) {
    utasitasok;
} else {
    utasitasok;
}
```

Megjegyzés: Az if szerkezeteknél is mindig ajánlott a blokkzárójelek használata akkor is, ha az ág csak egyetlen utasítást tartalmaz. Kerülendő a következő forma:

```
if (feltetel) // KERÜLENDŐ! HIÁNYZÓ BLOKKZÁRÓJELEK!
utasitas;
```

A for utasítás

A `for` utasításnál a következő formát ajánlott követni:

```
for (inicializalas; feltetel; leptetes) {
    utasitasok;
}
```

Egy üres törzzsel rendelkező for utasításnak a következőképpen kell kinéznie:

```
for (inicializalas; feltetel; leptetes);
```

Az inicializálás vagy léptetés során használhatunk vessző operátort, azaz több változót is deklarálunk, de kerülendő a háromnál több változó használata. Ha szükséges, akkor ezt tegyük meg a `for` ciklus előtt, vagy a ciklus törzsének végén.

A while utasítás

A `while` utasítás használata a következő formában ajánlott:

```
while (feltetel) {
    utasitasok;
}
```

Az üres `while` utasítás alakja:

```
while (feltetel);
```

A do-while utasítás

Formája:

```
do {
    utasitasok;
} while (feltetel);
```

A switch utasítás

A `switch` utasítást a következő alakban kell használni:

```
switch (feltetel) {
case ABC:
    utasitasok;
    /* továbblép */
case DEF:
    utasitasok;
    break;
case XYZ:
    utasitasok;
    break;
default:
    utasitasok;
    break;
}
```

Minden esetben, ha a vezérlés a következő eseten folytatódik (azaz hiányzik a break utasítás), használni kell egy megjegyzést a `break` helyén. Jelen esetben a `/*továbblép */` megjegyzést használtuk.

A try-catch utasítás

A `try-catch` utasítást a következő alakban használjuk:

```
try {
    utasitasok;
} catch (ExceptionClass e) {
    utasitasok;
}
```

A `try-catch` utasítást követheti `finally` utasítás, a vezérlés mindenképp eljut erre az ágra, függetlenül attól, hogy a `try`, vagy a `catch` blokk sikeresen végrehajtódott-e, vagy sem. Használata:

```
try {
    utasitasok;
} catch (ExceptionClass e) {
    utasitasok;
} finally {
    utasitasok;
}
```

12.4.3. Fehér karakterek (whitespace)

Üres sorok

Az üres sorokkal a logikailag összetartozó kódrészeket emelhetjük ki, ami növeli az olvashatóságot.
Két üres sort használunk a következő esetekben:

- a forrás fájl két bekezdése között;
- osztály- és interfészdefiníciók között.

Egy üres sort használunk a következő esetekben:

- metódusok között;
- egy adott metódus lokális változói és az első utasítás között;
- blokk vagy egysoros megjegyzés előtt;
- az egy metóduson belüli logikai szakaszok között az olvashatóság növelése érdekében.

Szóközök

Szóközöket kell használni a következő esetekben:

- Ha egy kulcsszót kerek zárójel követi, akkor el kell választani őket egy szóközzel.

```
while (true) {
...
}
```

Megjegyzendő, hogy szóköz nem használható a metódus neve és nyitó kerek zárójele közt. Ez segít elkülöníteni a kulcsszavakat a metódushívásoktól.

- Az argumentum-listában szóköznek kell követnie minden vesszőt.
- Minden kétoperandusú operátort, a . kivételével elválasztunk egy szóközzel az operandusaitól. Tilos szóközt használni az unáris operátor és operandusa között, mint például az előjelváltás (-), a növelő (++) és a csökkentő (--) operátorok esetén. Példák:

```
a += c + d;
a = (a + b) / (c * d);
while (d++ = s++) {
    n++;
}
printSize("size is " + foo + "\n");
```

- A kifejezéseket egy for utasításban el kell választani szóközökkel. Példa:

```
for (expr1; expr2; expr3)
```

- A típuskényszerítéseket szóköznek kell követnie. Például:

```
myMethod((byte) aNum, (Object) x);
myMethod((int) (cp + 5), ((int) (i + 3)) + 1);
```

12.5. Elnevezési konvenciók

Az elnevezési konvenciók könnyebben olvashatóvá és így jobban érthetőbbé teszik a forráskódot. A jól megválasztott azonosítók információkat adhatnak az azonosító funkciójáról, ami javítja a kód érthetőségét.

12.5.1. Azonosító típus

Csomagok

Az egyedi csomagnevek prefixét mindig kis ASCII karakterekkel kell írni és a top-level domain nevek egyikének kell lennie. Jelenleg lehet `com`, `edu`, `gov`, `mil`, `net`, `org` vagy az országok angol kétbetűs azonosítókódjának egyike, melyeket az ISO 3166 és 1981 szabványok definiálnak.

A csomag nevének többi komponensét a szervezet saját belső elnevezési konvencióinak megfelelően változtathatjuk. Például a csomagnevek tartalmazhatnak divízió, osztály, hivatal, projekt, gép és felhasználó neveket.

```
com.sun.eng
com.apple.quicktime.v2
edu.cmu.cs.bovik.cheese
```

Osztályok

Az osztályneveknek főneveknek kell lenniük és minden belső szó első betűjét nagybetűvel kell írni. Törekedni kell, hogy az osztálynevek egyszerűek és kifejezőek legyenek. Kerülendőek a rövidítések, kivéve nagyon nyilvánvaló esetben, mint az URL vagy HTTP.

```
class Raster;
class ImageSprite;
```

Interfészek

Az interfésznevekben a főneveket az osztálynevekhez hasonlóan nagybetűvel kell kezdeni.

```
interface RasterDelegate;
interface Storing;
```

Metódusok

A metódus neveknek kisbetűvel kezdődő igéknek kell lennie, de a belső szavakat nagybetűvel kell kezdeni.

```
run();
runFast();
getBackground();
```

Változók

A `for` változókat kivéve minden példány-, osztály- és osztályhoz tartozó konstans kezdőbetűjének kisbetűnek kell lennie, a belső szavak nagybetűvel kezdődjenek. Például az

```
int i;
char c;
float myWidth;
```

változó nevek nem kezdődhetnek aláhúzás (_) vagy dollárjellel ($), habár mindkettő szintaktikailag megengedett.

A változónevek lehetőleg rövidek és kifejezőek legyenek. A változónév megválasztásakor ügyeljünk arra, hogy a név tükrözze a változók használatának szerepét. Az egy-karakteres változó neveket kerüljük kivéve az ideiglenes, „eldobható” változókat. Az ideiglenes változóknak általános nevei például az `i`, `j`, `k`, `m` és `n` az egész típusúak; `c`, `d` és `e` a karakter típusúak számára.

Konstansok

Az osztályszintű konstansoknak deklarált változókat és a konstansokat csupa ANSI nagybetűvel kell írni és a szavakat aláhúzás jellel (_) kell elválasztani.

```
static final int MIN_WIDTH = 4;
static final int MAX_WIDTH = 999;
static final int GET_THE_CPU = 1;
```

12.6. Programozási tanácsok

12.6.1. Példány vagy osztályváltozóhoz való hozzáférés szabályozása

Egyetlen példány vagy osztályváltozót se tegyünk ok nélkül publikussá. Gyakran a példányváltozónak nem szükséges explicit értéket adni, mert ez egy metódus hívás mellékhatásaként fog bekövetkezni.

A publikus példányváltozó indokolt használatára példa az olyan osztály, ami valójában csak adatstruktúra és nem tárolja az adatok viselkedését. Más szavakkal ha `struct` deklarációt használnánk (ha létezne a Java nyelvben) osztálydeklaráció helyett.

Hivatkozás osztályváltozókra és metódusokra

Kerüljük az objektumok használatát statikus változó vagy metódus eléréséhez. Ilyenkor csak az osztálynév használata a megengedett. Példák:

```
classMethod(); //OK
AClass.classMethod(); //OK
anObject.classMethod(); // KERÜLENDŐ!
```

Konstansok

Numerikus konstansokat nem szabad közvetlenül a kódban szerepeltetni a `-1`, `0` és `1` kivételével, melyek például a `for` ciklus fejbén szerepelhetnek.

Értékadás

Kerülendő több változónak az egy utasításban való értékadás, mert nehezen olvasható. Például:

```
fooBar.fChar = barFoo.lchar = 'c'; // KERÜLENDŐ!
```

Ne használjuk az értékadás operátort olyan helyeken, ahol könnyen összetéveszthető az összehasonlító operátorral. Példa:

```
if (c++ = d++) { //HIBÁS!
...
}
```

Helyette így használhatjuk:

```
if ((c++ = d++) != 0) {
...
}
```

Ne használjunk egymásba ágyazott értékadásokat a futásidő csökkentésére. Ez a fordító dolga. Példa:

```
d = (a = b + c) + r; // KERÜLENDŐ!
```

Helyette:

```
a = b + c;
d = a + r;
```

Zárójelek

Általában véve jó programozói gyakorlat a zárójelek bő alkalmazása a kifejezésekben, hogy elkerüljük a precedencia problémákat. Így azok számára is egyértelmű lesz, akik nem látják át elsőre a kifejezést vagy nincsenek tisztában az operátorok precedenciájával.

```
if (a == b && c == d) // KERÜLENDŐ!
if ((a == b) && (c == d)) // Helyes
```

Visszatérési értékek

Próbáljuk a program struktúráját úgy kialakítani, hogy tükrözze a szándékot. Például az alábbi sorok helyett:

```
if (booleanExpression) {
    return true;
} else {
    return false;
}
```

Ez az egyszerűbb kifejezés álljon:

```
return booleanExpression;
```

Hasonlóan az előzőhöz:

```
if (condition) {
return x;
}
return y;
```

Helyette:

```
return (condition ? x : y);
```

Kifejezés a "?" előtt a feltételes operátorban

Ha egy kifejezés bináris operátort tartalmaz a `"?"` előtt egy háromoperandusú `? :` operátorban, akkor zárójelezni kell. Példa:

```
(x >= 0) ? x : -x;
```

Speciális megjegyzések

Használjuk az `XXX` szót a megjegyzésben annak jelzésére, ha valami még hibás, de működik. Használjuk a `FIXME`, vagy a `TODO` szót pedig annak a jelölésére, hogy valami hibás és nem működik jól.

12.7. Kód példák

Java forrásfájl példák

A következő példa bemutatja hogyan formázzuk az egyszerű publikus osztályt tartalmazó Java forrás fájlt. Az interfészeket hasonlóan kell formázni.

```
/*
* @(#)Blah.java 1.82 99/03/18
*
* Copyright (c) 1994-1999 Sun Microsystems, Inc.
* 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
* All rights reserved.
*
* This software is the confidential and proprietary
* information of Sun Microsystems, Inc. ("Confidential
* Information"). You shall not disclose such Confidential
* Information and shall use it only in accordance with the
* terms of the license agreement you entered into with Sun.
*/

package java.blah;
import java.blah.blahdy.BlahBlah;
/**
 * Az osztály leírása ide jön.
 *
 * @version 1.82 18 Mar 1999
 * @author Firstname Lastname
 */

public class Blah extends SomeClass {
/* Itt következhetnek az osztály implementációs megjegyzései
*/
/** classVar1 dokumentációs megjegyzés */
public static int classVar1;

/**
 * classVar2 dokumentációs megjegyzés,
 * egy sornál hosszabb eset
 */
```

```
private static Object classVar2;

/** instanceVar1 dokumentációs megjegyzés */
public Object instanceVar1;

/** instanceVar2 dokumentációs megjegyzés */
protected int instanceVar2;

/** instanceVar3 dokumentációs megjegyzés */
private Object[] instanceVar3;

/**
 * ...constructor Blah dokumentációs megjegyzés...
 */
public Blah() {
// ...Itt jön az implementáció...
}

/**
 * ...method doSomething dokumentációs megjegyzés...
 */

public void doSomething() {
// ...Itt jön az implementáció...
}

/**
 * ...method doSomethingElse dokumentációs megjegyzés...
 * @param someParam leírás
 */
public void doSomethingElse(Object someParam) {
// ...Itt jön az implementáció...
}
}
```

13. Mellékletek

Az alábbiakban a jegyzet fő fejezeteibe be nem sorolható, illetve néhány gyakran felmerülő kérdést tisztázó fejezetet találunk. A jegyzetben található példaprogramokat terjedelmi okokból nem közöljük, de a forráskódok a `http://szt1.sze.hu/java/` címről szabadon letölthetőek.

13.1. A Java keretrendszer telepítése és a környezet beállítása

Ahhoz, hogy Java programokat fejleszthessünk, be kell szereznünk a Java forrásokat (legegyszerűbben a `http://java.sun.com/j2se` címről).

- Először a Java fejlesztéshez alkalmas szoftvert kell telepíteni. Mivel a rövidítések között gyakori a keveredés, fontos a JRE és a JDK megkülönböztetése:
  - JRE (Java Runtime Environment) ~10Mb /, tartalmazza a java alkalmazásokhoz használható erőforrásokat, mint pl: Java Virtuális Gép (JVM), a szabványos osztálykönyvtárak (class library), Java alkalmazásindító stb. Azonban ez még nem tartalmaz fordítót, hibakeresőt, és fejlesztő környezetet sem, így a fejlesztésekhez nem alkalmas
  - JDK (Java Development Kit) ~40Mb – 60Mb, a java fejlesztésekhez használatos környezet. Több parancssori eszközt tartalmaz (javac, javadoc, appletviewer, HtmlConverter, jar fájlcsomagoló stb.). Tartalmazza a JRE-t is.
  - JDK + NetBeans ~120Mb – 130 Mb, Java fejlesztőkörnyezet, amely már a NetBeans integrált fejlesztőkörnyezetet is tartalmazza.
- Másodszor be kell állítani a megfelelő elérési utakat és környezeti változókat. A PATH bejegyzésben fel kell venni a telepített JDK bin könyvtárát. (pl: `c:\jdk1.5.0\bin\`). Ugyanitt létre kell hozni egy CLASSPATH nevű és `". ; c:\munkakvt "` értékű környezeti változót.

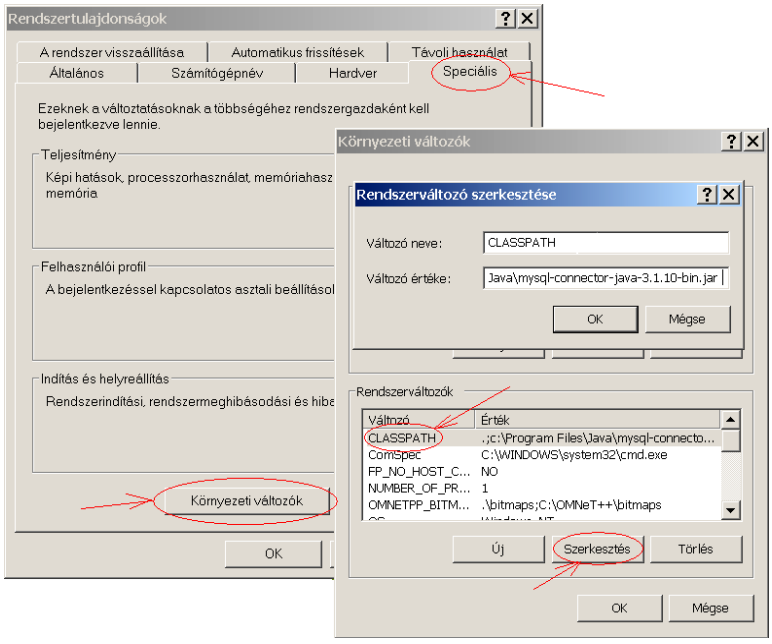
  Linux operációs rendszeren a következő módon:

```
export PATH=$SPATH:/opt/java/bin: .
export CLASSPATH=$CLASSPATH:/home/munka: .
```

  Windows operációs rendszeren parancssorból, vagy `bat` állományból:

```
SET PATH=%PATH%;c:\Program Files\java\jdk1.5.0\bin\ ; .
SET CLASSPATH=%CLASSPATH%;c:\munka\ ; .
```

Windows operációs rendszer esetén a `Sajátgép / Tulajdonságok / Speciális / Környezeti változók` menüpont alatt jegyezhetjük be a változtatásokat.



30. ábra: Környezeti változók beállítása

Amennyiben a környezeti változók beállítása sikerült, egy konzol ablakban (`cmd.exe`) a `java`, ill. `javac` parancsok futtathatóak és bármilyen könyvtárból elérhetőek! (Ellenőrzéshez gépeljük be a `javac` illetve a `java` parancsokat paraméterek nélkül. Amennyiben mindkét parancs egy kb. egy képernyőoldalas üzenetet küld a helyes használatáról, akkor a környezeti változók beállítása sikerült!)

- Ezután telepíthetünk egyes fejlesztő programokat:
  - WsCite:

    Egy nagyon kis helyigényű, egyszerű fejlesztőeszköz (IDE), melynek szövegszerkesztője nagyon felhasználóbarát, nyelvi szintakszis kiemelési, fordítási és futtatási lehetőséggel. (Freeware.)
  - JCreator Light Edition:
    Egy ingyenes és hasznos IDE eszköz. Aki járatos a Visual C++ keretrendszerben annak könnyen eligazodhat projectek és csomagok készítéséhez, fordításához.

– A NetBeans és Java Studio:
  A Sun Microsystems, illetve a Netbeans saját IDE rendszere. Elsősorban grafikus és kliens-szerver alkalmazások fejlesztésére. Rengeteg kiegészítőt integráltak benne, melyekkel professzionális és hatékony alkalmazásokat lehet fejleszteni. (Freeware.)
– Eclipse:
  Az Eclipse (IBM) egy nagyon elterjedt, csomagalapú fejlesztőeszköze. (Freeware.)
– JBuider:
  A Borland Co. terméke. Hasonlóan összetett eszköz, nagy projektek , grafikus felületű (awt, swing) alkalmazások hatékony fejlesztéséhez használható. Főleg a Delphit ismerő programozók között terjedt el.
  stb…

A grafikus keretrendszerek átgondolt és kész környezetet nyújtanak, ám az elkészült programok keretrendszer nélkül is – parancssorból – fordíthatóak, illetve telepített JVM esetén futtathatóak.

13.2. Dokumentációs segédprogramok

A tiszta és átlátható programkódolás mellett nagyon fontos a forrásfájlok dokumentálása. A dokumentációs típusú megjegyzéseket a `/**  */` jelek között kell elhelyeznünk, az adott csomag, osztály, illetve ezen belül az adattagok vagy metódusok definíciója előtt.

A szövegkiemelést támogató szerkesztők használatával a megjegyzések jól elkülönülnek a forráskód többi részétől. Az objektumorientált modellek leprogramozásakor az általános interfészektől, az absztrakt osztályokon át jutunk el a megvalósítást hordozó konkrét osztályok definiálásáig. A felüldefiniált és felültöltött metódusok tehát megjelenhetnek általános és konkrét értelemben is. Ezért mindig meg kell adnunk, hogy a definiálás alatt álló programelem milyen funkciókat tölt be, milyen paramétereket fogad, milyen visszatérési értéket szolgáltat, illetve a definíció során milyen műveletet, algoritmust szeretnénk az adott programrésszel végrehajtani.

Egy forrás későbbi módosítása, vagy a más fejlesztők munkájának elősegítése ezen kommentekkel biztosítható és a hatékony csoportmunka elengedhetetlen része.

Az elkészült és jól dokumentált forrásfájlokat a dokumentációs segédprogramokkal önthetjük szerkesztett formába. Ezek a segédprogramok a

fordítók által kihagyott megjegyzések feldolgozását és valamilyen kimeneti formátumba való konvertálást végzik el. Ilyen ingyenes segédprogramok a Java keretrendszerhez fejlesztett javadoc és az általános (több nyelvhez használható) doxygen [2],[7].

13.2.1. javadoc

A javadoc segédprogramot parancssoros környezetből indíthatjuk a

```
javadoc Osztaly.java
javadoc *.java
javadoc *.java -d dokumentacio -version -author -protected
```

parancs segítségével. A megadott osztályok forráskódjaiból a csomag, osztály és az adattagok, metódusok szerkezetét megvizsgálja, majd a dokumentációs megjegyzéseket a megadott elemekhez kapcsolja. Kimenetként az adott könyvtárban egy jól formázott html szerkezetet készít, melynek kiindulópontja az `index.html` fájl lesz. A `javadoc` kapcsolóival a kimenet tartalmi formátumait adhatjuk meg:

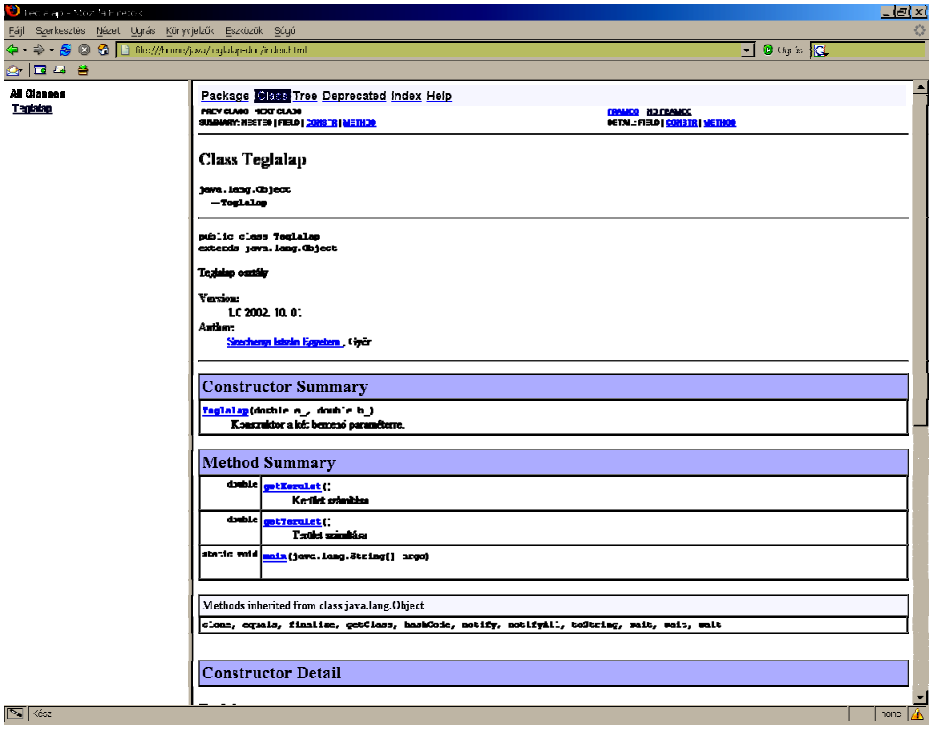
Opció	Jelentés
`-d`	kimeneti könyvtár neve
`-version`	bekapcsolhatók a @version tag-ek
`-author`	bekapcsolhatók a @author tag-ek
`-use`	bekapcsolhatók a dokumentáció keresztreferenciái (más osztályok is importálhtóak)
`-public`	csak a publikus hozzáférésű tagokat jeleníti meg
`-protected`	a protected és public tagokat mutatja
`-package`	a csoport hozzáférésű és protected és public tagokat mutatja
`-private`	minden tagot megmutat a dokumentációban
`-overview <file>`	egy áttekintő oldalt fűz a dokumentáció elejére a megadott html fájlból
`-subpackages`	rekurzív módon a megadott alcsomagok osztályai is kifejtésre kerülnek
`stb.`	

A forrásfájlokban az alábbi javadoc kapcsolókat (tag-ek) helyezhetjük el az egyes leírásokban. Egyes tag-ek az osztály, míg mások az adattagok vagy metódusok leírását segítik.

javadoc tag	Leírás
`@author`	Az -author kapcsoló használatával megadható a forrás szerzője/ tulajdonosa.
`@param`	Konstruktorok és metódusok paramétereinek leírására használható.
`@return`	Megadható, hogy a metódus milyen visszatérési értéket szolgáltat.
`@see`	Hiperlink hivatkozás fűzhető a dokumentumba, amely egy másik osztályra mutat.
`@throws`	Megadható, hogy az adott metódus milyen kivételeket válthat ki.
`@exception`	Megegyezik a @throws tag-gel
`@deprecated`	Egy Deprecated bejegyzést helyez el a leírásba. Ez a fejlesztőknek azt jelzi, hogy az adott művelet elavult, és csak kompatibilitási szempontok szerint maradt a definícióban. Általában az új és hatékonyabb műveletek kiváltják az elavult metódusokat.
`@link`	Egy külső hiperlink hivatkozás fűzhető a dokumentumba, amely egy másik html oldalra mutat.
`@since`	Egy Since bejegyzést helyez el a dokumentumban. A csomagok és az osztályok fejlesztésénél megadható, hogy az adott tulajdonság melyik verzió óta része a definíciónak.
`@version`	Egy version bejegyzést helyez el a dokumentumban, ha a -version opció aktív. Ezzel a szoftver életciklusa, főbb fejlesztési lépései jelezhetőek.

8. táblázat: a javadoc vezérlőelemei

A sikeres dokumentáció elkészítése után az osztálykönyvtárak dokumentációihoz hasonló oldalakat kapunk, amelyet tetszőleges böngészővel megtekinthetünk:



31. ábra: javadoc programmal generált html dokumentáció

13.2.2. doxygen

A javadoc-hoz hasonlóan más szoftverek is képesek egy megfelelően kommnetezett forráskódból dokumentációt készíteni.
A doxygen egy keresztplatformos dokumentáció-készítő rendszer, többféle programozási nyelvhez alkalmazható (C, C++, Java, Objective-C, Python, IDL , Corba, PHP, C#).
A programmal generálható dokumentáció kimeneti formátuma a következők valamelyike lehet:

- HTML
- LATEX
  - PostScript (a LATEX fájlból konvertálható)
  - Pdf (a LATEX fájlból konvertálható)
- RTF
- CHM (Microsoft súgó formátum)
- XML

A doxygen számos nyelven – így magyarul is képes – dokumentáció sablonokat generálni. Az elkészült és megfelelően dokumentált forrásfájlokat a doxygen segítségével parancssorból, vagy grafikus felületről indíthatjuk. (Részletes használati útmutató a program súgójában található.)

Irodalomjegyzék

[Angster01] Angster Erzsébet: Objektumorientált tervezés és programozás. Budapest : 4Kör Bt., 2001. 1020p. ISBN: 963 00 62623

[Cormen01] T. H. Cormen – Ch. E. Leiserson – R. L. Rivest : Algoritmusok. Budapest : Műszaki Könyvkiadó, 2001. 884p.
ISBN: 963 16 3029 3

[Flanagan05] David Flanagan: JAVA in a nutshell, fifth edition. O'Reilly Media Inc., 2005. ISBN: 0-596-007733-6

[Kondorosi04] Kondorosi – László – Szirmay – Kalos: Objektumorientált szoftverfejlesztés. Budapest : Computerbooks, 2004.
ISBN: 963 618 108 X

[Lemay02] Lemay, Laura – Cadenhead, Rogers: Teach yourself in Java 2 in 21 days. Elektronikus jegyzet, 2002.
http://www.cadenhead.org/book/21java/

[Marton02] Marton László - Fehérvári Arnold : Algoritmusok és adatstruktúrák. Győr : Novadat Bt., 2002. 344 p. ISBN: 963 9056 332

[Nyéky01] Java2 útikalauz programozóknak : Nyékyné Gaizler Judit (szerk.) et al. Budapest : ELTE TTK Hallgatói alapítvány, 2001. 1400p. ISBN: 963 463 485 0

[Nyéky03] Programozási nyelvek: Nyékyné Gaizler Judit (szerk.) et al. Budapest, Kiskapu Kft., 2003. 760p. ISBN: 963 9301 477

[Schildt03] Schildt, Herbert : Java 2: A beginner's guide 2nd edition : McGraw-Hill Osborne Media, 2003. Elektronikus jegyzet 544p.
http://www.osborne.com/downloads/downloads.shtml

[Vég99] Vég Csaba – Juhász István: Java start! Debrecen : Logos 2000, 1999. 228p. ISBN 963 03 9005 1

Online hivatkozások:

[1] A Java nyelv története:
http://java.com/en/javahistory/
[2] Doxygen dokumentációs segédprogram:
http://www.doxygen.org
[3] Az UML 2.0 szabvány:
http://www.uml.org/
[4] Java editorok és IDE eszközök összefoglaló leírása:
http://java.about.com/od/idesandeditors/ ,
http://www.javaworld.com/javaworld/tools/jw-tools-ide.html
[5] Java just in time (JIT) interpreter és környezet
http://www.shudo.net/jit/
[6] Java kódolási konvenciók:
http://java.sun.com/docs/codeconv/, illetve magyarul:
http://dragon.unideb.hu/~vicziani/pdf/jkk.pdf
[7] Javadoc segédprogram:
http://java.sun.com/j2se/javadoc/
[8] Java Tutorial:
http://java.sun.com/docs/books/tutorial/index.html
[9] Thinking in Java 3rd edition (Bruce Eckels):
http://www.mindview.net/Books/TIJ/
[10] A Java nyelv hivatalos weboldala:
http://java.sun.com
[11] J2SE 1.4 Merlin Release Contents JSR 59. 2002.05.09:
http://jcp.org/en/jsr/detail?id=59
[12] J2SE 5.0 “Tiger” Feature List JSR 176. 2004.09.30:
http://jcp.org/en/jsr/detail?id=176
[13] Java SE 6 "Mustang" Release Contents JSR 270 (beta version 2005.12.21): http://jcp.org/en/jsr/detail?id=270
[14] How to Write Doc Comments for Javadoc
http://java.sun.com/products/jdk/javadoc/writingdoccomments.html

Tárgymutató