

1

ANGSTER ERZSÉBET

OBJEKTUMORIENTÁLT TERVEZÉS ÉS PROGRAMOZÁS



Bevézetés a programozásba + Objektumorientált paradigmá + Java környezet

+ Java programozási alapok + Osztályok használata, készítése + Konténerek

Java

© Angster Erzsébet, 2001

Harmadik, átdolgozott kiadás, 2003

Minden jog fenntartva.

A könyv ezen online (PDF) változata szabadon terjeszthető változatlan formában.

Kiadja: 4KÖR Bt.

Szakmai lektor: Zsembery Ágoston

Nyelvi lektor: Honffy Pál

Szerkesztő: Angster Erzsébet

Borítóterv: Bérczi Zsófia

ISBN 963 00 6262 3 Ö

ISBN 963 00 6263 1

Akadémiai Nyomda, Martonvásár
Felelős vezető: Reisenleitner Lajos

Előszó

A Java nyelv igazán korszerű, objektumorientált programozási nyelv. Szinte minden informatikához közel álló cég keres Java fejlesztőket.

Jó szoftverfejlesztő képzése rövid idő alatt

Ha a programozás oktatásának célja a mai elvárásoknak eleget tevő szoftverfejlesztők kinevelése, akkor be kell látnunk: mindenekelőtt az objektumorientált (OO) szoftverfejlesztési késziséget és gondolkodásmódot kell kialakítani a programozást tanulókban. A feladat nem egyszerű: a rendelkezésre álló idő bizonyos képzési formákban mindössze két év, s ez idő alatt a mindenkorai piaci igényeknek megfelelő programozókat kell képezni. Egy programozó ma általában OO fejlesztőeszközzel dolgozik, és egyre inkább elvárják az OO CASE eszköz ismeretét is. A hallgatót a nulla programozói tudásról odáig kell eljuttatnunk, hogy objektumorientált módszert alkalmazva tervezze, kódolja, tesztelje és dokumentálja rendszerét. Az a tanár, aki már megpróbálta mindezen ismereteket elsajátítatni a hallgatókkal, tudja, hogy ez milyen nehéz feladat. Egészen bizonyos, hogy nincs vesztegetni való időnk: nem tehetjük már meg, hogy az idő harmadát-felét strukturált alapozással töltök el. Nem tehetjük ezt meg azért sem, mert a hallgatónak az „anyanyelvi” szemlélete ily módon a strukturált lesz, és ez rengeteg felesleges energiát emészti fel. Biztos vagyok benne, hogy azok a hallgatók, akik az objektumorientált paradigmán (szemléletmódon) „nőnek fel”, sokkal otthonosabban fognak mozogni a mai elvárásoknak megfelelő szoftverfejlesztési projektekben.

Előzetes tapasztalatok

Ez a könyv 20 éves oktatói és programozói munkám fontos lépcsőfoka. Az utóbbi öt évben sok konferencián voltam jelen a programozás oktatásának témajában; 1999-ben magam is szerveztem egy munkamegbeszélést Lisszabonban az ECOOP konferencián (European Conference on OO Programming) „Az OO tervezés és programozás bevezetése” címmel.

1999 őszén egy kísérletet kezdtünk el a SZÁMALK-ban a számítástechnikai programozó képzés hallgatóinak egy csoportjával. A programozást tiszta objektumorientált módszerrel kezdtük el tanítani. A hatodik héten a hallgatók már képesek voltak egyszerű feladatok OO tervét (együttműködési és osztálydiagramját) megérteni és módosítani. Az osztálydiagram társítási kapcsolatokat és örökléseket is tartalmazott. Bebizonyosodott, hogy az OO paradiigma nem könnyű ugyan, de az emberhez közel áll, és elsajátítható.

A könyv célja

A könyv első kötetének célja, hogy az Olvasó elsajátítsa az objektumorientált szemléletet, megismérje a Java nyelv és osztálykönyvtár alapelemeit, és biztonsággal használjon egy fejlesztői környezetet. Az I. és II. rész elméleti bevezető; a III-VI. részekben Java programokat írunk konzolos környezetben. A két rész párhuzamosan is elsajátítható. A könyv nem feltételez semmiféle programozási előismeretet, mindenki a számítógép használatában szükséges kis jártasság. **Kérem, tanulmányozza át a Tanulási és jelölési útmutatót!**

A programok készítéséhez a JBuilder 8.0 Personal integrált fejlesztői környezetet, benne a JDK1.4 fejlesztői készletet használjuk.

A könyv második kötetében majd különböző objektumorientált technikákkal bővíjtük ismereteinket, ekkor már igazán elvezetés, eseményvezérelt programokat fogunk írni grafikus környezetben, és adatainkat is el tudjuk már menteni a lemezre.

Köszönnetnyilvánítások

Elsősorban köszönöm férjemnek és lányaimnak, hogy türelemmel és szeretettel vettek körül. Köszönöm szüleimnek, akik egy-egy hétre igazi alkotói lékgört biztosítottak számomra Pécssett, az ősi házban. Zsófi lányomnak külön köszönöm azt a figyelmet és lelkesedést, amellyel élete első könyvborítóját elkészítette.

Zsembéry Ágoston több volt egyszerű lektornál: sokat vitáztunk, hasznos elképzéléseit örömmel építettem könyvembe. Köszönöm kollégáimnak, elsősorban Andor Gergőnek és Seres Ivánnak, akik folyamatosan mellettem álltak, és tanácsaikkal hozzájárultak könyvem csiszolásához; valamint Keszthelyi Zsoltnak és Vörös Beának, hogy elsőként csatlakoztak a Java programhoz.

Hallgatóim közül öten időt és fáradtságot nem kímélve olvasták készülő könyvemet annak tökéletesítése céljából, név szerint: Zétényi Emese, Mórocz Tamás, Auer Noémi, Nagy Tibor és Szkiva Zsolt. Köszönöm nekik is.

Kiadások története

- ◆ 2. kiadás: kisebb javítások; a 20 fejezet néhány új feladattal bővült.
- ◆ 3. kiadás: kisebb javítások; a III. részbe bekerült a JBuilder környezet.

Kedves Olvasó! A könyvvel kapcsolatos észrevételeit szívesen fogadom a következő címen:
angster.erzsebet@gmail.com

2003. január 5.



Tartalomjegyzék

| | |
|---|-----------|
| I. RÉSZ. BEVEZETÉS A PROGRAMOZÁSBA | 1 |
| 1. A számítógép és a szoftver..... | 3 |
| 1.1. Objektum, adat, program | 3 |
| 1.2. Programozási nyelv | 5 |
| 1.3. A program szerkesztése, fordítása, futtatása | 11 |
| 1.4. A szoftverek osztályozása..... | 14 |
| 1.5. Szoftverkrízis | 15 |
| 1.6. A szoftver minőségének jellemzői..... | 17 |
| 1.7. Moduláris programozás | 18 |
| 1.8. Módszertanok | 20 |
| Tesztkérdések | 21 |
| 2. Adat, algoritmus..... | 23 |
| 2.1. Az algoritmus fogalma..... | 23 |
| 2.2. Változó, típus..... | 25 |
| 2.3. Tevékenységsdiagram | 27 |
| 2.4. Pszeudokód | 33 |
| 2.5. Az algoritmus tulajdonságai | 39 |
| Tesztkérdések | 40 |
| Feladatok | 41 |
| 3. A szoftver fejlesztése | 43 |
| 3.1. A szoftverfejlesztés alkotómunka | 43 |
| 3.2. Az Egységesített Eljárás | 44 |
| 3.3. Követelményfeltárás | 50 |
| 3.4. Analízis..... | 50 |
| 3.5. Tervezés..... | 51 |
| 3.6. Implementálás (kódolás)..... | 51 |
| 3.7. Tesztelés | 52 |
| 3.8. Dokumentálás | 53 |
| Tesztkérdések | 54 |
| II. RÉSZ. OBJEKTUMORIENTÁLT PARADIGMA | 55 |
| 4. Mitől objektumorientált egy program? | 57 |
| 4.1. A valós világ modellezése | 57 |
| 4.2. Az objektumorientált program főbb jellemzői..... | 59 |
| Tesztkérdések | 66 |

| | | |
|-----------|---|------------|
| 5. | Objektum, osztály | 67 |
| 5.1. | Az objektum..... | 67 |
| 5.2. | Az objektum állapota | 69 |
| 5.3. | Az objektum azonossága..... | 70 |
| 5.4. | Osztály, példány..... | 70 |
| 5.5. | Kliens üzen a szervernek..... | 73 |
| 5.6. | Objektum létrehozása, inicializálása | 74 |
| 5.7. | Példányváltozó, példánymetódus | 76 |
| 5.8. | Osztályváltozó, osztálymetódus | 79 |
| 5.9. | Bezárás, az információ elrejtése..... | 81 |
| 5.10. | A kód újrafelhasználása | 82 |
| 5.11. | Objektumok, osztályok sztereotípusai..... | 83 |
| | Tesztkérdések..... | 84 |
| | Feladatok..... | 84 |
| 6. | Társítási kapcsolatok | 85 |
| 6.1. | Objektumok közötti társítási kapcsolatok | 85 |
| 6.2. | Osztályok közötti társítási kapcsolatok | 89 |
| 6.3. | A társítási kapcsolat megvalósítása..... | 94 |
| | Tesztkérdések..... | 98 |
| | Feladatok..... | 99 |
| 7. | Öröklődés | 101 |
| 7.1. | Az öröklődés fogalma, szabályai | 101 |
| 7.2. | Az utód osztály példányának adatai és a küldhető üzenetek | 105 |
| 7.3. | Egyszeres, többszörös öröklés | 106 |
| 7.4. | Az interfész fogalma | 106 |
| 7.5. | Láthatóság (hozzáférési mód, védelem) | 108 |
| | Tesztkérdések..... | 110 |
| 8. | Egyeszerű OO terv – Esettanulmány | 111 |
| 8.1. | A fogalmak tisztázása | 111 |
| 8.2. | Gyuszi játéka – fejlesztési dokumentáció | 120 |
| | Tesztkérdések..... | 129 |
| | III. RÉSZ. JAVA KÖRNYEZET | 131 |
| 9. | Fejlesztési környezet – Első programunk..... | 133 |
| 9.1. | A JBuilder letöltése, indítása..... | 134 |
| 9.2. | A könyv melléklete | 134 |
| 9.3. | A JBuilder alkalmazásböngészője..... | 136 |
| 9.4. | JBuilder-projekt fordítása és futtatása | 139 |
| 9.5. | Önálló program fordítása, futtatása | 140 |
| 9.6. | A javalib könyvtár konfigurálása | 144 |
| 9.7. | A javaprogram projekt létrehozása..... | 146 |
| 9.8. | Mintaprogram – Hurrá | 149 |
| 9.9. | A JBuilder szövegszerkesztője | 151 |
| 9.10. | JDK – Java Fejlesztői Készlet | 153 |
| 9.11. | Az API csomagstruktúrája | 159 |
| 9.12. | Fordítás és futtatás több osztály esetén | 161 |
| 9.13. | Integrált fejlesztői környezetek | 163 |

| | |
|---|------------|
| Tesztkérdések | 165 |
| Feladatok | 167 |
| 10. A Java nyelvről..... | 169 |
| 10.1. Az OO programozási nyelvek térhódítása | 169 |
| 10.2. A Java nyelv története..... | 172 |
| 10.3. Az Internet, a World Wide Web és a Java | 173 |
| 10.4. A Java nyelv jellemzői..... | 176 |
| Tesztkérdések | 177 |
| IV. RÉSZ. JAVA PROGRAMOZÁSI ALAPOK | 179 |
| 11. Alapfogalmak | 181 |
| 11.1. Mintaprogram – Krumpli | 181 |
| 11.2. ASCII és unikód karakterek..... | 183 |
| 11.3. A program alkotóelemei | 185 |
| 11.4. Változó, típus..... | 192 |
| 11.5. Primitív típusok | 195 |
| 11.6. A forrásprogram szerkezete | 197 |
| 11.7. Metódushívás (üzenet)..... | 201 |
| 11.8. Értékadó utasítás..... | 203 |
| 11.9. Adatok bevitelle a konzolról..... | 204 |
| 11.10. Megjelenítés a konzolon | 206 |
| Tesztkérdések | 209 |
| Feladatok | 211 |
| 12. Kifejezések, értékadás | 213 |
| 12.1. A kifejezés alkotóelemei..... | 213 |
| 12.2. Operátorok | 215 |
| 12.3. Típuskonverziók | 221 |
| 12.4. Értékadás, értékadási kompatibilitás..... | 223 |
| 12.5. Kifejezések kiértékelése - példák..... | 226 |
| 12.6. Feltétel | 227 |
| 12.7. Paraméterátadás, túlterhelt metódusok | 229 |
| 12.8. java.lang.Math osztály | 231 |
| Tesztkérdések | 233 |
| Feladatok | 235 |
| 13. Szelekciók | 237 |
| 13.1. Egyágú szelekció – if | 237 |
| 13.2. Kétágú szelekció – if..else | 240 |
| 13.3. Egymásba ágyazott szelekciók | 241 |
| 13.4. Többágú szelekciók – else if és switch | 243 |
| 13.5. Független feltételek vizsgálata | 247 |
| Tesztkérdések | 248 |
| Feladatok | 250 |
| 14. Iterációk..... | 251 |
| 14.1. Elöltesztelő ciklus – while | 251 |
| 14.2. Hátultesztelő ciklus – do while | 253 |
| 14.3. Léptető ciklus – for | 256 |

| | | |
|------------|--|------------|
| 14.4. | Ciklusok egymásba ágyazása, kiugrás a ciklusból | 260 |
| 14.5. | Adatok feldolgozása végjelig | 264 |
| 14.6. | Megszámlálás..... | 266 |
| 14.7. | Összegzés, átlagszámítás | 267 |
| 14.8. | Minimum- és maximumkiválasztás | 268 |
| 14.9. | Menükészítés..... | 270 |
| | Tesztkérdések..... | 271 |
| | Feladatok..... | 272 |
| 15. | Metódusok írása | 277 |
| 15.1. | A metódus fogalma, szintaktikája | 277 |
| 15.2. | Paraméterátadás | 283 |
| 15.3. | Visszatérés a metódusból | 285 |
| 15.4. | Metódusok túlterhelése | 286 |
| 15.5. | Lokális változók..... | 289 |
| 15.6. | Néhány példa | 290 |
| 15.7. | Hogyan tervezzük meg metódusainkat?..... | 292 |
| | Tesztkérdések..... | 294 |
| | Feladatok..... | 296 |
| V. | RÉSZ. OSZTÁLYOK HASZNÁLATA, KÉSZÍTÉSE | 299 |
| 16. | Objektumok, karakterláncok, csomagolók..... | 301 |
| 16.1. | A fejezet osztályainak rendszerezése | 301 |
| 16.2. | Primitív típus – referenciátipus | 304 |
| 16.3. | Objektum létrehozása, deklarálása | 305 |
| 16.4. | Értékadás az objektumok körében..... | 307 |
| 16.5. | Az objektum élete, automatikus szemétygyűjtés | 308 |
| 16.6. | String osztály | 309 |
| 16.7. | StringBuffer osztály | 316 |
| 16.8. | Objektum átadása paraméterként | 322 |
| 16.9. | Primitív típusok csomagolása | 325 |
| 16.10. | StringTokenizer osztály | 329 |
| | Tesztkérdések..... | 332 |
| | Feladatok..... | 333 |
| 17. | Osztály készítése | 337 |
| 17.1. | OO paradigma – Emlékeztető | 337 |
| 17.2. | Első mintaprogram – Raktárprogram..... | 339 |
| 17.3. | Második mintaprogram – Bank és a „Jószerencse” | 345 |
| 17.4. | Az osztály felépítése, az osztály deklarációi | 351 |
| 17.5. | Osztálytag, példánytag | 353 |
| 17.6. | Azonosító, hivatkozási kör, takarás | 355 |
| 17.7. | Változók alapértelmezés szerinti kezdeti értékei | 357 |
| 17.8. | A this objektumreferencia | 357 |
| 17.9. | Konstruktörök | 358 |
| 17.10. | Inicializálók..... | 360 |
| | Tesztkérdések..... | 363 |
| | Feladatok..... | 364 |

| | |
|---|------------|
| VI. RÉSZ. KONTÉNEREK | 367 |
| 18. Tömbök | 369 |
| 18.1. A tömb általános fogalma | 369 |
| 18.2. Egydimenziós tömb | 372 |
| 18.3. A tömb szekvenciális feldolgozása | 377 |
| 18.4. Gyűjtés | 379 |
| 18.5. Kétdimenziós tömb | 385 |
| 18.6. Többdimenziós tömb | 390 |
| 18.7. A tömb átadása paraméterként | 392 |
| 18.8. A program paraméterei | 393 |
| 18.9. Feladat – Szavazatkiértékelés | 395 |
| Tesztkérdések | 399 |
| Feladatok | 401 |
| 19. Rendezés, keresés, karbantartás..... | 405 |
| 19.1. Rendezés | 406 |
| 19.2. Keresés | 408 |
| 19.3. Karbantartás | 411 |
| 19.4. Primitív elemek rendezése, keresése | 415 |
| 19.5. String objektumok rendezése, keresése | 420 |
| 19.6. Saját osztályú objektumok rendezése, keresése | 423 |
| 19.7. Szövegek rendezett karbantartása | 429 |
| Tesztkérdések | 433 |
| Feladatok | 434 |
| 20. A Vector és a Collections osztály | 437 |
| 20.1. A konténer funkciói általában | 437 |
| 20.2. Vector osztály | 439 |
| 20.3. Az equals metódus szerepe | 446 |
| 20.4. A konténer elhagyása az UML diagramról | 452 |
| 20.5. Interfészek – Collection, List, Comparable | 452 |
| 20.6. Collections osztály | 456 |
| 20.7. Feladat – Nobel díjasok | 463 |
| 20.8. Feladat – Városok | 467 |
| 20.9. Feladat – Autóeladás | 470 |
| Tesztkérdések | 475 |
| Feladatok | 476 |
| FÜGGELÉK | 479 |
| A tesztkérdések megoldásai | 481 |
| Irodalomjegyzék | 483 |
| Tárgymutató | 485 |

Tanulási és jelölési útmutató

Tanulási útmutató

A tankönyv teljesen kezdők számára készült, **nem feltételez semmiféle programozási előismeretet**; csak a számítógép és az operációs rendszer használatában szükséges egy kis jártasság – elsajátításához kb. fél év szükséges. A könyv távoktatási célra is használható, vagyis a tananyagot **önállóan is meg lehet tanulni**. Természetesen egy tanfolyam mindenkorban felgyorsítja a tanulási időt.

A könyvon kívül Önnek szüksége lesz a következőkre:

- ◆ **JBuilder 8.0 Personal fejlesztői környezet:** Internetről letölthető és ingyenesen használható. A környezet tartalmazza a JDK1.4 fejlesztői készletet.
- ◆ **A könyv elektronikus melléklete (javaprog.zip):** Itt vannak a két kötetben tárgyalt feladatok forráskódjai, a fejezetek végén található feladatok megoldásai, valamint a programok futtatásához szükséges egyéb mappák és fájlok. A melléklet letölthető az Internetről, címét a könyv hátán megtalálja.

Lapozzon a 9. fejezetre, és telepítse e két szoftvert az ott leírtak szerint!

Ajánlott tanulási sorrend – átugorható részek

Kezdje a 9. fejezettel! Alakítsa ki azt a szoftverkörnyezetet, amelyben dolgozni fog, és ismerkedésképpen futtassa a könyv két kötetének programjait! Jobban érthető az elmélet, ha van némi elképzelése a programról és a programkészítés módjáról.

A könyvnek vannak olyan részei, melyek elsajátítása elengedhetetlen, és vannak olyan részei, melyek átugorhatók, illetve halaszthatók. Részletesebben:

- ◆ **I. és II. rész** (Bevezetés a programozásba és Objektumorientált paradigmába): E két rész (1-8. fejezetek) **elméleti bevezető**. Nem baj, ha az elmélet nem tisztul le teljesen, az itt tárgyalt fogalmak ismételten visszatérnek majd. A 8. fejezet csak „hab a tortán” – ha ezt is tökéletesen érti, akkor nagyon eredményes volt eddigi munkája (vagy már voltak némi előismeretei). Ne keseredjen el, ha Gyusziának megoldását nem érti krisztályisztán – a III. résztől majdnem előlről kezdődik minden... Fontos azonban, hogy

majd a gyakorlati tudás megszerzésével együtt vissza-visszalapozzon ezekre a fejezetekre.

- ♦ **III. rész** (Java környezet): A 9. fejezet ismerete **elengedhetetlen a Java programok futtatásához**. A 10. fejezet a Java nyelv ismertetője, erre a későbbiekben nem épülnek közvetlenül anyagrészek.
- ♦ **IV-VI. részek** (Java programozási alapok, Osztályok használata és készítése, Konténelek): **Itt kezdődik a programozás „java”**, a 11. fejezettől konkrét, hús-vér Java programokat írunk – igaz, még csak konzolos környezetben. Az eseményvezérelt, grafikus környezetre majd a 2. kötetben kerül sor, de azt hiszem, itt tipikusan igaz a mondás: „Lassan járj, tovább érsz!”.

Részletesebben az egyes fejezetekről:

11. Alapfogalmak: Alapvető ismeretanyag.
12. Kifejezések, értékkadás: Alapvető ismeretanyag, de a konverziós szabályok alapos ismerete nélkül is folytatható a tanulás.
- 13.-15. Szelekciók, Iterációk, Metódusok írása. Alapvető ismeretanyagok.
16. Objektumok, karakterláncok, csomagolók: A fejezet alapvetően fontos, de a 16.7. `StringBuffer` és a 16.9., Primitív típusok csomagolása kevésbé hangsúlyos téma. A 16.10. StringTokenizer osztály teljesen átugorható.
17. Osztályok készítése: Alapvető ismeretanyag.
18. Tömbök: A két- és többdimenziós tömböknek elegendő csak az elméletét elsajátítani, vagyis a 18.5., és 18.6 pontok átugorhatók. A 18.9., Szavazatkiértékelés egy haladóbb szintű programozási feladat, ennek tanulmányozása is elhagyható.
19. Rendezés, keresés, karbantartás: Az 19.1., 19.2. és 19.3. elméleti pontokat nézze át. A 19.4–19.7. pontok átugorhatók.
20. A Vector és a Collections osztály: Alapvető ismeretanyag. Ez a fejezet mutat rá leginkább a programozás lényegére, innen kezdve oldhatók meg a körülményekhez képest érdekesebb, összetettebb feladatok.

Ellenőrizze tudását!

Minden egyes fejezet végén **tesztkérdéseket** és **feladatokat** talál, melyekkel ellenőrizni tudja tudását. A tesztkérdések megoldásait a könyv függelékében, a gyakorlati feladatok megoldásait a könyv mellékletében találja meg. Ha a teszteket meg tudja oldani, akkor mélyítse el tudását a feladatok megoldásával! A feladatokat **nehézségi szintjük szerint osztályoztuk**:

- ♦ **(A)** Rutinfeladat. Ha ezt nem tudja megoldani, ne menjen tovább!
- ♦ **(B)** Könnyű feladat. Néhány alapfogást kombinálni kell, de nem okozhat sok fejtörést.
- ♦ **(C)** Nehezebb feladat. Komolyabb gondolkodást igénylő vagy összetettebb feladat.

A feladatok többsége Java program készítése. Ha a feladatot megoldotta, vagyis megtervezte (egyszerűbb programok esetén fejben) és lekódolta, vesse össze azt a mellékletben szereplő megoldással!

Jelölési útmutató

A tankönyv a következő jelöléseket használja:

Ilyen dobozokban találhatók az **elméleti ismeretanyagok, tömör formában**. Ha a dobozok tartalmával már tökéletesen tisztában van, és az elmélet alapján a megfelelő Java programokat is meg tudja írni, akkor tegye fel a könyvet a polcára, és vegye le a könyv második kötetét!

A normál szövegben magyarázatok vannak. Erre azért van szükség, mert a doboz túlságosan tömény.

Megjegyzés: Ez csak megjegyzés, sokkal kevésbé fontos, mint a többi információ. De azért ezt sem árt elolvasni.

❖ Ha ilyen jelölést talál, akkor itt valami „bomba” van elrejtve, érdemes résen lenni!

Szintaktika

A Java deklarációk, utasítások szintaktikai szabályait így adjuk meg:

<típus> <változó1>[=<kifejezés1>] [,<változó2>=<kifejezés2>...] ;

A <> „kacsacsörben” kitöltendő dolgok vannak; a [] nagy zárójelben lévő részeket nem kötelező megadni. Ahol három pontot lát, ott folytatható a felsorolás. A fenti szintaktika egy előfordulása például:

int a=b=12*Math.PI, c=1, d;

Java metódusok

A Java szintaktika szerinti metódusokat (például a Java osztálykönyvtár metódusait) úgy adjuk meg, hogy a metódusok fejét egy kis háromszöggel jelöljük meg, ezt követi a metódus leírása:

► void metodus(int par) // ez egy metódusfej

Itt található a metódus magyarázata.

Feladatok és megoldások

Feladat

Az itt megfogalmazott feladatot meg is oldjuk...

Ha szükséges, akkor megtervezzük a programot a feladatspecifikáció alapján. A megoldás teljes forráskódját mindenképpen megadjuk:

Forráskód

```
// Ez egy Java program első sora:  
import java.util.*; // 1  
//...
```

Ha szükséges, akkor a programot elemezzük. A magyarázatban a forráskód megjegyzéseiben megadott számokra hivatkozunk. A feladathoz sok esetben megadjuk a futási eredményt is (amit a konzolon láthatunk):

A program egy lehetséges futása

```
| Adja meg az életkorát: 69  
| Ön egész fiatal!
```

Jó tanulást, örömteli programozást!

I.



I. BEVEZETÉS A PROGRAMOZÁSBA

1. A számítógép és a szoftver
2. Adat, algoritmus
3. A szoftver fejlesztése

II. OBJEKTUMORIENTÁLT PARADIGMA

4. Mitől objektumorientált egy program?
5. Objektum, osztály
6. Társítási kapcsolatok
7. Öröklődés
8. Egyszerű OO terv – Esettanulmány

III. JAVA KÖRNYEZET

9. Fejlesztési környezet – Első programunk
10. A Java nyelvről

IV. JAVA PROGRAMOZÁSI ALAPOK

11. Alapfogalmak
12. Kifejezések, értékadás
13. Szelekciók
14. Iterációk
15. Metódusok írása

V. OSZTÁLYOK HASZNÁLATA, KÉSZÍTÉSE

16. Objektumok, karakterláncok, csomagolók
17. Osztály készítése

VI. KONTÉNEREK

18. Tömbök
19. Rendezés, keresés, karbantartás
20. A Vector és a Collections osztály

FÜGGELÉK

- A tesztkérdések megoldásai
Irodalomjegyzék
Tárgymutató

1. A számítógép és a szoftver

A fejezet pontjai:

1. Objektum, adat, program
 2. Programozási nyelv
 3. A program szerkesztése, fordítása, futtatása
 4. A szoftverek osztályozása
 5. Szoftverkrízis
 6. A szoftver minőségének jellemzői
 7. Moduláris programozás
 8. Módszertanok
-

A programozónak tudnia, de legalábbis sejtenie kell, mi történik „odabent” a számítógépben, amikor a program fut, és mi történik addig, amíg a program futtatható állapotba kerül. Ismernie kell továbbá a minőségi szoftver kritériumait, és tudnia kell, hogy milyen eszközök állnak a programozó, illetve a szoftverfejlesztő rendelkezésére a szoftver elkészítéséhez. E fejezet célja a szoftverrel kapcsolatos alapvető ismeretek bemutatása programozói szemszögből.

1.1. Objektum, adat, program

Objektum

A valós világban előforduló dolgokat **objektumoknak** nevezzük. Egy objektumnak rengeteg olyan mérhető és nem mérhető jellemzője van, amelyek alapján az objektum felismerhető. Az objektumnak vannak **adatai** (tulajdonságai), és van valamilyen, algoritmusokkal leírható **viselkedésmódja**. Az autónak például van márkája és rendszáma, és képes haladni. Az autó az objektum, a márka és a rendszám az autó adatai, a „halad” pedig az autó viselkedésmódja.

Gépi ábrázolás

A számítógép csodálatos, de buta anyaghalmaz, melyet az ember talált ki. A gép nem tud önállóan gondolkodni. A számítógép csak azokat az adatokat tudja megjegyezni, és azokat az

utasításokat tudja végrehajtani, melyeket az ember beléplált. A valós világ objektumainak adatait az ember betáplálja a számítógépbe. Mivel a számítógép elemi szinten minden össze két állapotot képes megjegyezni (0 vagy 1), az adatokat és viselkedésmódokat valamelyen módon ábrázolni kell. Az adatok és algoritmusok tárolásához egyezményes jelek szükségesek. Ismert adatábrázolási szabványok a következők:

- ◆ Karakterek kódolására alapvetően kétféle szabvány ismeretes:
 - ASCII karakterkészlet (**American Standard Code for Information Interchange**; az információcsere amerikai szabványos kódjai), amely egy bájton határozza meg minden egyes karakter kódját. A 9-es számjegy kódja például 57, a @ kódja 64, az A betű kódja 65. A 256 darab ASCII karakter ma már nem elegendő a különböző nemzetek betűinek, számjegyeinek és egyéb speciális karaktereinek ábrázolására.
 - Unikód (unicode) karakterkészlet, mely már 2 bájton ábrázolja a lehetséges karaktereket. A lehetőségek száma így 256-ról 65536-ra emelkedik, s ez bőven elegendő az angol, magyar, orosz, kínai, szanszkrit vagy hottentotta betűk megkülönböztetésére is. (Pillanatnyilag mintegy 35000 karaktert használnak a világ számítógépein.) Az unikód karakterábrázolásról a 11. fejezetben lesz szó.
- ◆ Egész számok kódolására a fixpontos számábrázolást szokás alkalmazni. Negatív szám esetén a szám 2-es komplementét tárolják.
- ◆ Valós számok kódolására a lebegőpontos számábrázolást szokás alkalmazni az IEEE 754 szabvány szerint.
- ◆ Logikai értékek kódolása: általában a hamis érték gépi reprezentánsa a 0, az igazé pedig az 1.

Az algoritmus gépi ábrázolása a gépi kód, vagyis a program. A számítógépen tárolt programnak tudnia kell, hogy egy adatnak pontosan mi az ábrázolási módja, hiszen ugyanaz a bitkombináció mászt jelent akkor, ha a tárolt adat karakter, és megint mászt, ha az valós szám, logikai érték, szöveg vagy kép.

Program, szoftver

Egy számítógépes rendszer hardverből és szoftverből áll (**hardware, software**). Hardvernek nevezünk a számítógép kézzelfogható fizikai komponenseit. Szoftveren általában a számítógépes rendszer meg nem fogható, nem fizikai összetevőit értjük. A szoftver fogalma gyakorlatilag egybeesik a számítógépen működő programok és az azokhoz tartozó adatok fogalmával.

A **program** a számítógép számára érhető instrukciók sorozata, mely az adatok megfelelő számításaival és mozgatásaival egy feladat megoldását célozza. A számítógépes program arra képes, hogy adatokat fogadjon (például a felhasználótól vagy egy számítógéphez csatolt másik géptől), azokat tárolja, pakolgassa, műveletekkel megváltoztassa, továbbadja, illetve a felhasználó számára a kért formában megjelenítse. A programkészítés célja azonban nem pusztta adatszolgáltatás, hanem a felhasználó **informálása**, azaz olyan ismeretek eljuttatása a felhasználó-

hoz, amelyekre neki az adott pillanatban szüksége van. A felhasználó nem érti a gépi adatpakolgatásokat, őt a géptől elrugaszkodva, emberi módon kell informálni. A program feldáta, hogy a gép és az ember közötti hidat felépítse. S mivel a programozó is ember, nemcsak a programok, hanem a különböző programkészítési módszerek is megpróbálnak elrugaszkodni a gépközeli valóságtól, hogy a számítógépet az emberi gondolkodáshoz hasonló utasításokkal vezérelhessék. A programozó az adatokat és az instrukciókat különböző módszerekkel csoportosítja, és a valósághoz hasonlóan megpróbálja ezeket a programmodulokat „életre lehelní”. Az ember igyekszik a számítógépes objektumokat az élőlények mintájára "megszemélyesíteni", hogy közelebb kerülhessen hozzájuk, és könnyebben beprogramozhassa őket. A már kész, beprogramozott objektumokat aztán, ha lehet, újra felhasználja, beépíti más programokba. Minél bonyolultabbá válik a számítógép működése, ez a tendencia annál erősebbé válik.

Objektum: A valós világban előforduló dolog, melynek vannak adatai és van viselkedés-módja.

Program: A számítógép számára érthető instrukciók sorozata, mely az adatok megfelelő számításaival és mozgatásaival egy feladat megoldását célozza.

Szoftver: Egy számítógépes rendszer meg nem fogható, nem fizikai összetevői.

1.2. Programozási nyelv

Gépi kód, natív kód

A számítógép memóriája olyan címezhető tár, amely memóriarekeszkből, bájtokból (byte) áll. A memória tartalma a program működése (futása) közben állandóan változhat. Futtatáskor a **gépi kódú program** betöltdök a számítógép memóriájába. A gépi kódú (gépi nyelvű) program a gép számára végrehajtható utasítások sorozata, ezért ezt futtatható programnak (executable program) is nevezik. A gépi kód más elnevezése még: **natív kód** vagy **tárgykód** – ez a név a kód "természetes" helyére (célhelyére) utal. Egy gépi kódú program adatokat és gépi kódú utasításokat egyaránt tartalmaz. Az 1.1. ábrán egy képzeletbeli, primitív számítógép memóriájának a modellje látható egy gépi kódú programmal. A címzés 8 bites (ily módon csak egy 256 bájtos tár címezhető). A memóriarekeszek szintén egybájtosak, ezek címét és tartalmát hexadecimális számokkal adtuk meg. Az egyes utasítások mellé ún. mnemonikokat (megjegyzést elősegítő szavakat, mint HLT, JMP, LDA stb.) írtunk – ezek azonban csak a mi tájékozódásunkat segítik, a memóriában levő program igazából egy bitsorozat (az alsó sor a bitsorozat hexadecimális formája, a sorozatot bájtonként tagoltuk):

| | | | | | | | | | | |
|----------|----------|----------|----------|----------|-----|---|---|---|---|-----|
| 00000010 | 00010010 | 00000111 | 00000000 | 00100000 | ... | | | | | |
| 0 | 2 | 1 | 2 | 0 | 7 | 0 | 0 | 2 | 0 | ... |

Egy gépi kódú utasítás (művelet, operátor) az utasítás kódjával kezdődik, melyet az utasítás paraméterei (argumentum, operandus) követnek. Hogy egy utasításkód hány bájtból áll, az a processzorra jellemző adat (példánkban 1 bájtos utasítások vannak). Egy utasítás egyértelműen meghatározza az öt követő paraméterek számát, valamint azt, hogy azok egyenként pontosan hány bájtot foglalnak le, és mit jelentenek. Van olyan utasítás, amelynek nincs paramétere; ilyen például az, amelyik a program azonnali leállására utasít (HLT). A paraméter lehet például egy érték vagy egy olyan memóriacím, amelyben egy érték vagy egy további cím van eltárolva.

Fiktív utasításkészletünk a következő utasításokból áll (az első két oszlop a gépi kódú utasítás 1. és 2. bájta):

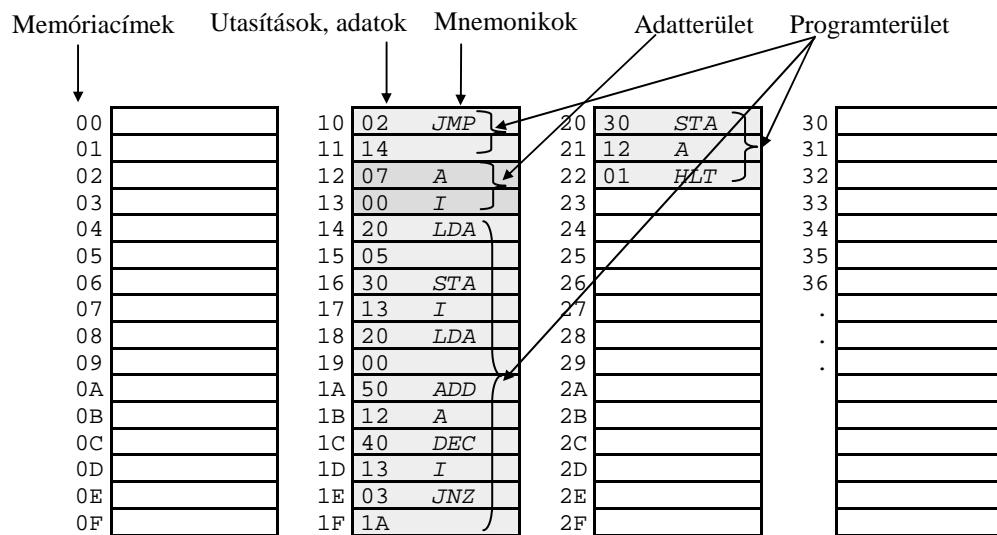
| | | utasítás kódja | utasítás paramétere (ha van) | |
|----|-------|----------------|------------------------------|---|
| 1. | 2. | Mnem. | Angolul | Magyarul |
| 01 | - | HLT | Halt | Állj meg! |
| 02 | cím | JMP | Jump | Ugorj a címrre! |
| 03 | cím | JNZ | Jump on Not Zero | Ugorj a címrre, ha nem nulla volt az érték! |
| 20 | érték | LDA | Load accumulator | Töltsd az akkumulátorba az értéket! |
| 30 | cím | STA | Store accumulator | Tárol le a címrre az akkum. tartalmát! |
| 40 | cím | DEC | Decrement | Csökkentsd 1-gyel a címen levő értéket! |
| 50 | cím | ADD | Add | Add az akk.-hoz a címen levő értéket! |

Feladat

Írunk egy gépi kódú programrészletet, mely a 0012 címen levő egész értéket (a 7-et) megszorozza 5-tel! Az eredmény a 0012 címen keletkezzék!

A megoldás az 1.1. ábrán látható. A program az 10 (hexadecimális szám, ejtsd: egy nulla) címen kezdődik egy ugró utasítással (02 14), ahol a 02 az utasítás (JMP), az 14 pedig az utasítás paramétere (erre a címrre ugorj). Az utasítás hatására a program vezérlése az 14 címen folytatódik. Ezzel a program az adatterületet átugorja. Ha ezt nem tenné, akkor a processzor az adatterületet programnak nézné, és biztosan furcsa dolgok történnének. A következő, 20 05 gépi kódú utasításban a 20 az utasítás (LDA), a 05 pedig az utasítás paramétere, ezt az értéket fogja betölteni az akkumulátorba. A 30 13 utasítás letárolja az akkumulátor tartalmát az 13 címrre (STA) stb. A program működésének lényege, hogy az 13 címrre leteszi az 5 értéket, és annak minden egyes csökkentésekor az akkumulátorhoz adja az 12 címen levő 7-es értéket. Így az akkumulátorban végül ott lesz az $5 \cdot 7 = 35$ érték, amelyet a program végül letesz a 12 memória-címrre.

Megjegyzés: Nem szükséges, hogy a programot lépésről lépésre megérte! Elegendő, ha kap egy elképzelést a gépi kódú programról.



1.1. ábra. Gépi kód egy fiktív számítógép memóriájában

A gépi kód nem emberi olvasásra való. Az előbbi program csak egyjegyű számokat szoroz, és még nem is jeleníti meg az eredményt a felhasználó számára. Két darab 3 jegyű szám összszorzása esetén a program ennek sokszorosára bővülne, melyben az átvitelt és a túlcsordulást is kezelni kellene.

Assembly nyelv

A utasítás kódja egy szám, melyet az ember nehezen jegyez meg. Ezért minden utasításnak van egy emlékeztetője (mnemonik), mégpedig az utasításra jellemző rövidítés.

A processzor csak gépi kódú utasításokat képes végrehajtani, az emlékeztetőket nem érti. Ahhoz, hogy emlékeztetőket használhassunk, meg kell tanítani a számítógépet arra, hogy azokat felismerje. A programozó az utasításokat tehát nem számokkal, hanem emlékeztetőkkel írja meg egy szöveges állományban – ez az **assembly nyelvű forrásprogram**. Van egy program, amely elemzi és gépi kód dátum alakítja, **lefordítja** ezt a forrásprogramot. A fordítást végző programot **assemblernek** nevezzük.

Az assemblert szintén programozók írták meg, vagy gépi kódban, vagy egy másik assembler segítségével. Az assemblernek természetesen az is a feladata, hogy információt adjon a forrásprogramban előforduló esetleges hibákról. Hiba esetén a fordítás sikertelen, a programon javítani kell. Az assemblyben írt program hátránya, hogy már messzebb kerültünk a géptől, az

assembler esetleges hibáinak akár áldozatai is lehetünk. De ha jó assemblert választunk, akkor a megírt program ettől olvashatóbb, javíthatóbb és dokumentálhatóbb lesz. Ez óriási előny.

Nézzük az előző, gépi kódú programunkat assembly nyelven:

Feladat

Írunk egy assembly programrészletet, mely az A címen levő egész értéket (a 7-et) megszorozza 5-tel! Az eredmény az A címen keletkezzék!

Egy assembly forrásprogramban az utasításokon kívül a címeknek is nevet adhatunk (A, BELEP stb.). Ezzel a forráskód érthetőbbé válik, és sokkal könnyebben javítható. A // utáni megjegyzésben az adott sor rövid magyarázata található:

```

        JMP  BELEP    // Ugorj a BELEP címre!
A:      BYTE 7      // 1 byte-os adatterület, A kezdeti értéke 7.
I:      BYTE 0      // Adatterület, I kezdeti értéke 0.
BELEP:   LDA   5      // Töltsd az akkumulátorba az 5-öt!
          STA   I      // Tárol el az I címre!
          LDA   0      // Töltsd az akkumulátorba a 0-t!
UJRA:   ADD   A      // Add az akkumulátorhoz az A címen levő értéket!
          DEC   I      // Csökkentsd az I-t eggel!
          JNZ   UJRA    // Ha ez nem 0, akkor ugorj az UJRA címre!
          STA   A      // Tárol el az akkumulátor tartalmát az A címre!
          HLT           // Állj! Vége a programnak.

```

A számítógéphez közel álló programnyelveket – mint amilyen az assembly nyelv, – **alacsony szintű nyelveknek** nevezzük. Az alacsony szintű nyelveknek vannak előnyei és hátrányai:

- ◆ Előnyök: 1. Sokkal gyorsabb és kisebb helyfoglalású program írható vele; 2. Vannak olyan – erősen hardverhez kötődő – feladatok, amelyeket csak így lehet megoldani.
- ◆ Hátrányok: 1. A programozónak sokat kell dolgoznia; 2. Nehezen érthető és módosítható; 3. Gépenként külön meg kell írni.

Alacsony szintű nyelven különböző processzorokra általában különböző programokat kell írni. Ez természetes, hiszen a számítógépeket és a processzorokat is tervezik – elképzelhető, hogy ugyanazt a problémát az egyik processzorral egészen másképp lehet megoldani, mint a másikkal. Az ilyen programok nem hordozhatók, vagyis az egyik processzoron megírt programot nem tudjuk egy másik típusún működtetni.

Magas szintű nyelv

A **magas szintű nyelvek** közelebb állnak az emberi gondolkodáshoz, mint az alacsony szintűek. Egy magas szintű forrásprogramból már ránézésre is sokat ki lehet olvasni, sejteni lehet, hogy mit fog csinálni, ha azt gépi kóddá alakítjuk át. Nézzük, hogyan oldjuk meg az előbbi, assemblyben megadott feladatot Javában:

Feladat

Írunk egy Java programrészletet, mely az a címen levő egész értéket (a 7-et) meg-szorozza 5-tel! Az eredmény az a címen keletkezzék!

```
{  
    int a = 7; // az "a" változó egy 4 byte-os egész (integer)  
    a = a * 5; // legyen "a" értéke az eredeti ötszöröse  
}
```

Az előbbiekhez hasonlóan ez sem teljes program, hiszen nem sok értelme van egy tárolt értéket öttel megszorozni, ha azt utána nem használjuk semmire. Az azonban látható, hogy ezt a részfeladatot mennyivel egyszerűbben, elegánsabban oldjuk meg magas szintű nyelven. Az a-nak itt is le kell foglalnunk egy tárhelyet, de a szorzást egyetlen utasítással elvégezzhetjük. Ha 4 bájtosnál nagyobb számot akarunk szorozni, akkor nincs más teendőnk, mint az int típust például egy long típusra kicserélni, ami egy 8 bájtos egész szám manipulálására alkalmas. Egy nagyobb számot szorzó assembly forrásprogram terjedelme sokszorosa az ugyanezt a feladatot végrehajtó magas szintű forrásprogramnak, hiszen darabonként kell pakolgatni az eredményt, és közben figyelni a helyi értékeket, a túlcordulást (elfér-e a kiszámolt érték a megadott regiszterben) stb. A magas szintű nyelv erre fel van készítve, ezt előre kidolgozták azok a programozók, akik a nyelv értelmezőjét, fordítóját megírták. A magas szintű nyelven programozóknak „csak” használniuk kell a felkínált magas szintű utasításokat, a fordító behelyettesíti azokat gépi kódú utasításokkal.

A magas szintű nyelvek többé-kevésbé hasonlítanak az emberi nyelvre. Ezek a nyelvek a programozó munkáját nagymértékben megkönnyítik, hiszen a részletek tálalva vannak, csak be kell őket helyezni programunkba. Persze ahhoz, hogy ilyen könnyedén, emberközeli tudjunk programot írni, ahhoz meg kellett egyszer írni azt a programot, amely ezeket az emberközeli utasításokat értelmezi, és gépi kódra lefordítja. Az ilyen programot **fordítónak (compiler)** nevezik. Egy magas szintű utasítás több száz vagy több ezer gépi kódú utasítást is jelenthet egyszerre. Egy magas szintű forrásprogram már hordozható, vagyis könnyen átvihető más platformra (más hardverrel és operációs rendszerrel rendelkező gépekre) is. A fordítóprogram persze továbbra sem hordozható. A forrásprogramot az adott számítógépre megírt fordítóprogrammal a számítógép gépi kódjává kell alakítani.

Sok magas szintű nyelv létezik a világon. A különböző nyelvek más és más célra készülnek: az egyik a logikai feladatok megoldásában erős, a másik adatfeldolgozásra vagy távoli (világhálós) adatelerésre van „kihegyezve”. A nyelvek erősen eltérhetnek egymástól akár koncepcionálisan, akár gyorsaságban vagy megbízhatóságban. A legismertebb magas szintű programozási nyelvek ma a **Java, C#, C++, C, Pascal, Basic, Smalltalk, Eiffel, Modula, Ada, PL/I, Cobol, Fortran, Prolog, CLOS, APL, Clipper**, és még sorolhatnánk. Mindegyik nyelvnek megvannak a különböző szempontok szerinti sajátosságai, erősségei és gyengéi. Hogy melyik nyelv terjed el egy adott helyen és időben, az sok mindenben múlik – használhatóságán kívül például azon, hogy melyiknek van nagyobb irodalma, reklámja, pressziója.

Bájtkód

Vannak fordítóprogramok (ilyen a Java compiler), melyek nem fordítják le teljesen gépi kódra a magas szintű forrásprogramot, hanem egy közbenső, ún. **bájtkódot** (bytecode) állítanak elő. A bájtkód az ember számára olvashatatlan utasításkódokat és hivatkozásokat tartalmazó gépközelí kód, mely mindenkorra rugaszkodik el a géptől, hogy platformfüggetlen (gép- és operációs rendszer független) lehessen. Ezt a bájtkódot aztán közvetlenül a futás előtt egy másik fordító vagy értelmező natív kóddá alakítja. **A bájtkód tehát a magas szintű nyelv és a gépi kód közötti platformfüggetlen közbenső kód.** Azért találták ki, hogy így a már majdnem lefordított kód hordozható (portable) legyen, vagyis könnyűszerrel át lehessen azt vinni különböző platformokra (számítógépes környezetekre).

A bájtkód előnyei:

- ◆ Hordozható, azaz platformfüggetlen, elvileg átvihető bármilyen platformra (gépre).
- ◆ A bájtkódot sokkal könnyebb natív kóddá alakítani. Így a platformspecifikus fordító, illetve futtató rendszer már sokkal egyszerűbb program lehet.

A Java fordító is bájtkódot állít elő. A Java fordító szabványos, azaz bármely környezetben ugyanabból a szabványos forrásprogramból ugyanazt a bájtkódot állítja elő (sajnos csak elvileg, mert van olyan szoftverház, amelyik nem tartja be a szabványt). Ezt a bájtkódot az adott számítógépen a JVM (Java Virtual Machine, Java virtuális gép, futtatórendszer) futtatja úgy, hogy azt utasításonként értelmezi és natív kóddá alakítja.

Gépi kód (natív kód, tárgykód): A gép számára végrehajtható utasítások sorozata. A gépi kódú program csak a megadott célgépen futtatható.

Programozási nyelv: Számítógépes programok pontos leírására használt jelölésrendszer. A jelölésrendszer szabályait betartva a programozó egy szöveges állományt készít, melyet a programozási nyelv fordítóprogramjával fordít le a számítógéphez közelebb álló kódra.

Alacsony szintű nyelv: Olyan programozási nyelv, melyben a vezérlés és az adatstruktúrák közvetlenül visszatükrözik a gépi architektúrát.

Magas szintű nyelv: Olyan programozási nyelv, melyben a vezérlés és az adatstruktúrák inkább a programozó szempontjaihoz igazodnak, semmint a hardver által nyújtott lehetőségekhez.

Assembly nyelv, assembler: Az assembly nyelv a géphez közel álló, alacsony szintű programozási nyelv, amelyben a programozó a gépi utasításokat mnemonikokkal adja meg. Az assembly nyelvű forrásprogramot az assembler fordítja gépi kódra.

Bájtkód: A magas szintű forráskód és a gépi kód közötti platformfüggetlen közbenső kód.

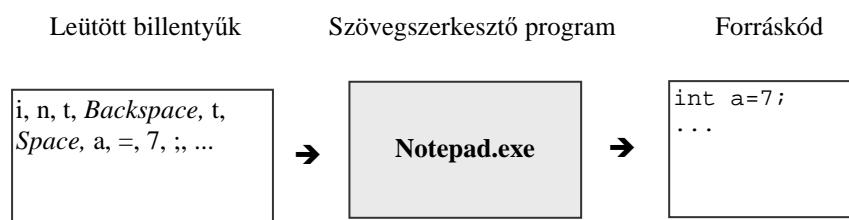
1.3. A program szerkesztése, fordítása, futtatása

A program tehát gépi kódú utasítások sorozata, melyet „betáplálhatunk” a számítógépbe. Amikor egy program fut, adatokat olvas be a beviteli eszközökről, manipulálja az adatokat, majd kiírja őket valamilyen kiviteli eszközre. Ezek közül persze bármelyik elhagyható, a lényeg az, hogy a program értelmes legyen. Nem fontos manipulálni az adatokat, elképzelhető, hogy a billentyűzetről beolvasott karaktereket változtatás nélkül nyomtatóra küldjük. Az is elképzelhető, hogy egy program kiszámol bizonyos adatokat, de azokat semmilyen kiviteli eszközre nem küldi ki. Ekkor azonban felmerül a kérdés: miért írtuk a programot?

A számítógép bekapcsolásakor automatikusan elindul egy program, a számítógép operációs rendszere. Ez a rendszerprogram teremti meg a gép és a felhasználó közti alapvető kommunikációs lehetőségeket, és addig fut, amíg a gépet ki nem kapcsoljuk. Az operációs rendszer egy megfelelő utasításra a megadott programot a memóriába tölti és elindítja. Egy adott programnak lehetnek bemenő (input) adatai, amelyeket a program felhasznál, és kimenő (output) adatai, amelyeket a program produkál.

Szövegszerkesztő program

A fejlesztés alatt álló programot – annak megtervezése után – egy **szövegszerkesztő program** (editor) segítségével meg kell írni (1.2. ábra). A szövegszerkesztő inputja a beütött megjeleníthető karakterek (pl. i, n, =, ;) és vezérlőkarakterek (pl. F1, Ctrl+le vagy Backspace), outputja a kész **forráskód**. A magas szintű forrásprogram egy ASCII kódokat tartalmazó szöveg (az operációs rendszerek legtöbbje még nincs felkészülve az unikód karakterekre). Ezt a szöveget egy olyan szövegszerkesztővel kell szerkesztenünk, amely nem alkalmaz formázó karaktereket. A Word által készített szöveg például tele van tűzdelve a fordítóprogram számára értelmezhetetlen vezérlőkarakterekkel (a stílus, karaktertípus, nyelv, sortávolság, lábjegyzet stb. jelölése miatt). MS-Windows környezetben a forráskód szerkesztésére például a Notepad vagy az Edit program megfelel. Az 1.2. ábrán az `int a=7;` programsort visszük be inputként úgy, hogy közben a t betűt töröljük, majd újra bevisszük.

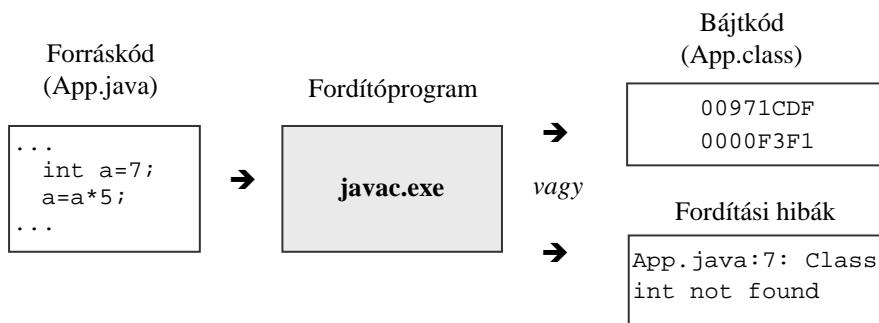


1.2. ábra. A forráskód szerkesztése

Fordítóprogram

A forráskódot egy **fordítóprogrammal** (compiler) le kell fordítanunk. minden magas szintű nyelvnek megvan a maga fordítóprogramja, melyet futtatunk: a fordítóprogram inputja az általunk megírt forrásprogram (source code), outputja pedig az adott nyelvtől, illetve az ahhoz tartozó fordítóprogram típusától függően bájtkód vagy tárgykód (object code). Ha a fordító a forrásprogramban számára értelmetlen szöveget talál, akkor a lefordított kód helyett outputként a fordítási (szintaktikai) hibák listáját kapjuk. Ez utóbbi eset természetesen valamelyen programozói tévedés eredménye.

Windows környezetben a Java nyelv fordítóprogramja a `javac.exe` (java compiler). A Java fordító a `java` kiterjesztésű forráskóból bájtkót generál. A bájtkód állományának kiterjesztése `class` (1.3. ábra).



1.3. ábra. Java program fordítása

A fordítóprogram tehát a forrásnyelven megírt programot a géphez közeli kódra alakítja. A fordító legtöbb esetben képtelen arra, hogy minden információt rögtön „megértsen”, ilyenkor a fordítás több menetben készül. Először elemzi a szöveget, és kiemeli belőle a felesleges információkat, mint például a megjegyzéseket, szóközöket stb. Aztán ellenőrzi az utasítások helyességét, hiba esetén figyelmeztet. Van olyan fordító, mely az első talált hibánál leáll, abbaagyja a fordítást. Más fordítók egy egész listát készítenek a szövegben talált hibákról. Ez utóbbi esetben természetesen előfordulhatnak generált hibák is (amikor egy hiba generálja a többet), ilyenkor a hibalista (különösen annak vége) nem teljesen reális.

Egyes fordítók optimalizálják a kódot, vagyis kiemelik belőle a felesleges lépésekét, és lehetőség szerint tömörítik. Vannak olyan fordítók, amelyek gyorsan fordítanak, de a lefordított kód lassan fog futni, mások viszont ugyanabból a forrásnyelvű szövegből lassan ugyan, de villámgyorsan futó programkódot produkálnak. Jó és gyors fordítóprogramot nehéz dolog írni.

Programszerkesztő program

Az operációs rendszerből indítható program sok esetben több, külön lefordított modulból (tárgykódból) áll, melyeket futás előtt vagy alatt össze kell kapcsolni, szerkeszteni. Az összeszerkesztést a **programszerkesztő program** (linker) végzi. A lefordított tárgykód tehát még összeszerkesztésre vár, az önállóan nem képes futni; a benne levő hivatkozási címek relatív címek, a tárgykód a memóriában áthelyezhető (relocatable code). Egy tárgykód sok esetben futás közben kapcsolódik a hívó programhoz (Windows környezetben például a `dll` állomány).

Csak a programszerkesztő által összeállított program futtatható. Az összeszerkesztendő tárgykódok száma nincs korlátozva, és természetesen egy is lehet. Az `exe` állomány a programszerkesztő által több tárgymodulból összeszerkesztett tárgykód.

Java esetében az összeszerkesztésre várakozó modulok a féllel lefordított bájtkódok (kiterjesztésük `class`). Ezeket még összeszerkesztés előtt egészen le kell fordítani. Mindez a java futtatórendszer (JVM = Java Virtual Machine) feladata.

⚠ A programszerkesztő (linker) nem tévesztendő össze a szövegszerkesztővel (editor)!

Interpreter

Az **interpreter** (értelmező) olyan program, amely a forrásprogramnak egyszerre egyetlen utasítását értelmezi, azt natív kóddá alakítja, rögtön végrehajtja, majd a lefordított kódot elfelejtí. Ha a vezérlés megint erre az utasításra kerül, az értelmezés újból megtörténik. Interpreterrel rendelkezik például a Basic és a Java nyelv. Java esetén az értelmező a JVM (Windows környezetben a `java.exe`), amely a bájtkódot utasításoknál értelmezi és futtatja.

Az értelmezők általában lassúak, ezért sok fejlesztő és felhasználó a program teljes fordítását részesíti előnyben. A Java bájtkódnak létezik egy olyan fordítója/értelmezője, mely a lefordított utasítást a program futása alatt megjegyzi, s csak a legközelebbi betöltéskor fordítja újra. Ez a JIT (Just In Time compiler), mely egyre inkább kezd elterjedni. Speciális környezetekben (pl. Oracle adatbázisban) létezik „valódi” Java fordítóprogram is, amely a Java forráskódból eltárolható natív kódot készít.

A program futtatása

Láttuk, hogy bizonyos fordítók és programszerkesztők együttesen elkészítik a natív kódot. Az ilyen kódot egyszerű elindítani: Windows környezetben például kattintsunk kettőt az ikonjára vagy az `exe` állomány nevére!

Java esetében egy program több, `class` kiterjesztésű modulból (bájtkódból) áll, melyek között van egy megkülönböztetett, belépési ponttal rendelkező modul. A virtuális gép ezt a fő modult indítja el, és ebből történik a többi modul hívása.

Van tehát egy program, amelyet elkészítettünk, és működik – legyen az játékprogram, fordítóprogram, szövegszerkesztő, alkatrész-nyilvántartó vagy könyvelő program. Hogy mit használ fel inputként a program, az a program írójától függ. Természetesen bármely program futtathat egy másik programot is, de a legáltalánosabb az, hogy egy program inputként adatokat használ, és az outputja is adat. A programból persze olyasmik is „kijöhetnek”, amiket a programozó nem akart, hiszen tévedni programozói dolog. Ilyenkor a program kiszámíthatatlan dolgokat produkálhat. A futás közben előforduló hibák legtöbbször a program leállásához vezetnek – ezek a **futás alatti hibák** (runtime error).

Forráskód (forrásprogram): A fordítóprogram által lefordítandó szöveg, illetve szöveges állomány.

Forrásprogram szerkesztése: A forráskód megírása, javítása szövegszerkesztő program (editor) segítségével.

Program fordítása: A forráskód átalakítása a számítógép által érthető tárgykóddá, illetve egy közbenső bájtkóddá. A fordítást a fordítóprogram (compiler) végzi.

Programszerkesztő (linker): Program, amely a lefordított tárgykódokat összeszerkeszti, futtatható állapotba hozza.

Interpreter: Értelmező program, mely a forrásprogramot vagy a bájtkódot utasításonként értelmezi, fordítja le, és hajtja végre.

Program futtatása: Egy adott számítógépen a tárgykód, illetve bájtkód véghajtása. Bájtkód esetén a tényleges futtatás előtt még egy fordítás, illetve értelmezés szükséges.

1.4. A szoftverek osztályozása

A futó programokat (szoftvereket) funkciójuk szerint osztályozni lehet. A legtöbb program besorolható a következő csoportok valamelyikébe:

Operációs rendszerek

(Operating systems)

pl. DOS, OS/2, Unix, Linux, Windows XP, Mac OS

Szoftverfejlesztő rendszerek

(Software development systems)

A szoftverfejlesztő rendszerek a programok fejlesztését, elkészítését szolgálják. Ilyen eszközök segítségével lehet akár operációs rendszereket, akár alkalmazói programokat, akár újabb szoftverfejlesztő rendszereket készíteni.

- ◆ Programnyelvi fordítóprogramok, keretrendszerök – pl. JDK, JBuilder, JCreator, JDeveloper, Borland C++, Visual C++, Delphi, Macro Assembler, Turbo Pascal
- ◆ Adatbáziskezelők – pl. Oracle, Borland InterBase, Microsoft SQL Server, IBM DB2, Sybase, ObjectStore
- ◆ CASE eszközök (Computer-Aided Software Engineering = Számítógéppel támogatott szoftvertechnológia) – pl. Rational Rose, Enterprise Architect, Together

Alkalmazói (felhasználói) programok, programcsomagok

(Application programs, application packages)

Alkalmazói programnak nevezük azt a programot, mely egy speciális feladat elvégzésére készült, és amely közvetlenül hozzájárul a feladat megoldásához.

- ◆ Szövegszerkesztők – pl. Word, NotePad
- ◆ Táblázatkezelők – pl. Excel
- ◆ Kiadványszerkesztő programok – pl. Pagemaker, Ventura
- ◆ Tervező programok – pl. 3D Design, ArchiCAD, ORCAD, Home Design
- ◆ Kommunikációs programok – pl. Internet Explorer, OutLook, WinFax
- ◆ Vállalatirányítási, pénzügyi rendszerek – pl. SAP, MS Money, BookKeeper
- ◆ Tudományos programok – pl. Mathematica, Maple
- ◆ Segédprogramok – pl. WinZip, Calculator, Clock
- ◆ Oktató programok – pl. Java Tutorial
- ◆ Szórakoztató programok – pl. CD lejátszó, játékok

1.5. Szoftverkrízis

Az orvos, a mérnök és a programozó vitatkozik, melyikük mestersége a régebbi:

- ◆ *Orvos: Éva Ádám bordájából való – ehhez Istennek orvosi beavatkozásra volt szüksége. Nyilvánvaló tehát, hogy az orvosi mesterség a legrégebbi.*
- ◆ *Mérnök: Csakhogy Isten előbb teremtette a Földet, mégpedig a káoszból, és ez kétségkívül mérnöki bravúr volt.*
- ◆ *Programozó: Na ja, de ki teremtette a káoszt?*

A szoftverek bonyolulttak

A szoftverek segítségével az ember a valós világot próbálja **modellezni** annak érdekében, hogy munkáját könnyebbé, életét kényelmesebbé tegye. S minthogy az ember folyamatosan újabb és újabb ötletekkel áll elő, és a piacot is meg szeretné hódítani, a szoftverek állandó fejlődésben vannak, egyre bonyolultabbakká válnak. A multimédia betört életünkbe: természetes számunkra, hogy a számítógép beszél, zenél és filmeket játszik – mindenekkel a bonyolult világot szeretnénk minél valósághűbben visszaadni. De menjünk egy kicsit távolabbi a számító-

gépek „személyi” használatától. Egy légiirányító rendszer olyan bonyolult, hogy működését teljes egészében senki sem képes átlátni. Kifejlesztéséhez rengeteg szakembert kell bevonni: a szoftverkészítés tudományán kívül érteni kell a repülőgépekhez, a radarokhoz, a meteorológiahoz és még számos egyébhez. Egy-egy ilyen rendszer sok ember sokéves munkájának eredménye lehet, melyre horribilis összegeket fordítanak a megrendelők.

Minőségi szoftver kell!

Természetes, hogy minden megrendelő **minőségi szoftvert** szeretne. A felhasználó a minőséget **külső jegyekben** méri le: számára az a fontos, hogy a program **helyesen** és **gyorsan működék**, legyen **megbízható**, **felhasználóbarát** és **továbbfejleszthető**. Ha ez nem így van, akkor a megvásárolt szoftver az embert inkább hátrálta, mintsem segíti munkájában. Ilyen esetben a megrendelő lemond a szoftverről, s visszaköveteli a pénzét. A szoftver minőségét a külső jegyek határozzák meg: amikor használók egy programot, akkor az a fontos, hogy az a legteljesebb mértékben kiszolgáljon, és soha ne hagyjon cserben – hogy milyen a program belülről, az engem mint felhasználót nem nagyon érdekel. Tudjuk viszont, hogy a szoftverfejlesztő a kívülről látható és mérhető minőséget csak egy belső rend megteremtésével érheti el. Ezért a **belső minőség** kialakítása a szoftverfejlesztők állandó törekvése.

Szoftverkrízis

A **szoftverkrízis** fogalmát egy 1968-as NATO szoftverfejlesztési konferencián vezették be. A szoftverfejlesztők számára világossá vált, hogy a programok bonyolultsága következtében a régi strukturált módszer felmondta a szolgálatot, ily módon a minőséget nem lehet fenntartani. Egyfajta káosz kezdett eluralkodni: a rendszereket késve szállították, elképesztően sokba kerültek, működésük megbízhatatlan volt, és az átvevőnek ezer kifogása akadt. A strukturált módszerekkel történő szoftverkészítés tehát válságba került, az már nem volt képes kezelni a belső rendet: a szoftverfejlesztő társadalom valamilyen újabb szemléletet, módszert követelt. A válságból kivezető út az objektumorientált paradigmára (szemléletmód), ennek segítségével gyorsabban és biztonságosabban érhetünk célit a szoftverfejlesztés területén.

Nem valószínű természetesen, hogy az objektumorientált módszertan egyszer és mindenkorra megoldotta a szoftverfejlesztési gondokat. Az ember a szoftverrel szemben egyre magasabb igényeket támaszt, és ezt az éppen aktuális technika és módszertanok nem tudják egyenletesen követni. Ilyenkor könnyen keletkezhet újabb krízis, amelyből kivezető utat kell keresni. A szoftverkrízis tehát folyamatosan újra és újra előállhat.

Szoftverkrízis: A szoftverfejlesztés válsága, amikor egy hagyományos módszer már nem képes az igényeknek megfelelő minőségi szoftver előállítására.

1.6. A szoftver minőségének jellemzői

Mielőtt a minőségi szoftver előállításának lehetőségeit boncolgatnánk, nézzük meg a minőségi szoftver kritériumait! Ezek a következők:

- ◆ **Helyesség:** A szoftver helyes, ha egrészt a feladat specifikációja helyes (megfelel a felhasználói igényeknek), másrészt a szoftver a feladat specifikációja szerint működik. Nyilvánvaló dolog, hogy a helyesség az elsődleges minőségi követelmény, hiszen ha a program nem úgy működik, ahogyan azt a megrendelő kérte, akkor nincs értelme tovább minőségről beszálni.
- ◆ **Hibatűrés:** A szoftver hibatűrő, ha az abnormális esetekben is – a lehetőségekhez képest – normálisan működik. A feladat specifikációja például általában nem taglalja a teendőket kevés memória esetére. Ez azonban nem jelentheti azt, hogy ha nincs elég memória, akkor a program minden figyelmeztetés nélkül egyszerűen abortál, félbehagyva az adatfelvitelt és egyéb fontos teendőket.
- ◆ **Karbantarthatóság, bővíthetőség:** Egy szoftver karbantartható, ha az könnyen javítható, illetve módosítható. Egy szoftver bővíthető, ha azon az újabb felhasználói igények könnyen átvezethetők. Rosszul megtervezett program esetén előfordulhat, hogy a program egyik részén elvégzett változtatás követhetetlenül befolyásolja a program egyéb részeit. Csak a könnyen áttekinthető programot tudjuk könnyedén karbantartani, illetve továbbfejleszteni.
- ◆ **Újrafelhasználhatóság:** Egy szoftver újra felhasználható, ha az vagy annak részei újabb szoftverekben hasznosíthatók. A fejlesztőknek sokszor kell olyan feladatot megoldaniuk, melyet már egyszer pontosan úgy vagy hasonlóképpen megoldottak. Érthető, hogy ezeket a már jól bevált, alaposan tesztelt elemeket újra fel szeretnék használni annak érdekében, hogy az új program egyedi részeire koncentrálhassanak.
- ◆ **Kompatibilitás:** Egy szoftver kompatibilis egy másikkal, ha együttműködhet vele. Gondolunk például a rengeteg file- és képformátumra: az egyik szoftver csak akkor tudja használni a másik szoftver által előállított adatokat, ha az adatformátumok megfeleltethetők egymásnak.
- ◆ **Felhasználóbaráság:** A szoftver felhasználóbarát, ha a megjelenése kellemes, használata kényelmes, egyértelmű, logikus, és minden lehetséges módon segíti a felhasználót.
- ◆ **Hordozhatóság:** Egy szoftver hordozható, ha az könnyedén átvihető más hardver-, illetve szoftverkörnyezetbe.
- ◆ **Hatókonyúság:** A szoftver hatékony, ha az a rendelkezésre álló hardver- és szoftvererőforrásokat a legteljesebb mértékben kihasználja, a lehető legkisebbre csökkentve ezzel a futási időt.
- ◆ **Ellenőrizhetőség:** Egy szoftver ellenőrizhető, ha a tesztelési adatok és eljárások könnyedén összeállíthatók.

- ◆ **Integritás (sérthatetlenség):** A szoftver sérthatetlen, ha a különböző rendszerhibák nem okoznak helyrehozhatatlan hibákat (adatokban, programokban, számítógépen stb.).
- ◆ **Szabványosság:** A szabvány valamely egyezmény eredményeképpen létrejött, nyilvánosan elérhető definíció. Egy szoftver szabványos, ha az működésében, külalakjában és dokumentálásában megfelel a szabványnak (például F1-re jön a help, a szövegmezőben a delete gombra karaktert lehet törlni, a felhasználói leírás tartalmazza a biztonsági előírásokat stb.). Az ISO (International Standard Organization) nemzetközi szabványokat megállapító testület, amelynek fennhatósága mindenre kiterjed. Egyelőre azonban a szoftverek területén még kevés dolog van az ISO által szabványosítva, ezért a szabványokat hallgatólagosan a nagyobb szoftverházak diktálják.

1.7. Moduláris programozás

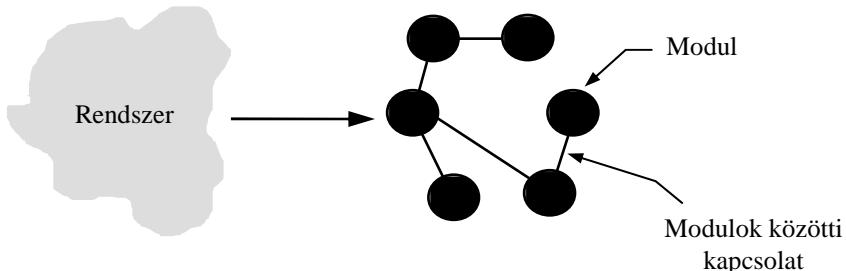
Egy szoftver bonyolultsága csak úgy áttekinthető, ha az az ember számára érthető, kezelhető modulokból áll (1.4. ábra). A szoftver bonyolultságának kezelésére az embernek alapvetően két eszköz áll a rendelkezésére: az **absztrakció** és a **dekompozíció**. Az absztrakció segítségével kiemeljük a feladat szempontjából lényeges dolgokat, a többöt pedig figyelmen kívül hagyjuk. A dekompozíció segítségével a feladatot részekre, **modulokra** bontjuk, mivel az egészet egyszerre nem tudjuk áttekinteni. A részfeladatokat meg kell oldani, majd a modulokat újra össze kell állítani, hogy együtt működhessenek. A kérdés az, hogyan bontsuk részekre a nagy feladatot, a részekre bontás után hogy fogunk hozzá a megoldáshoz, végül hogy történék az összefűzések. Számos programtervezési módszer született már – ezek mindegyikének kivétel nélkül az a célja, hogy az ember munkáját megkönnyítsék, módszert adjanak arra vonatkozólag, hogyan lehet egy feladatot a legrövidebb idő alatt számítógépre vinni úgy, hogy az a legjobb, a legszebb és a legmegbízhatóbb legyen, valamint nyitott legyen a bővítésre, továbbfejlesztésre. Az eddig megalkotott tervezési módszerek egy dologban hasonlítanak egymásra: mindegyik igyekszik betartani a modulokra bontás alapvető szabályait. A moduláris programozás olyan programozási mód, amelyben a teljes program modulokból áll. Az egyes modulok kezelhető méretűek, egyértelmű céljuk van, és jól definiáltan csatlakoznak környezetükhez.

A moduláris programozás irányelvei a következők:

- ◆ **„Oszd meg és uralkodj” elv:** A feladatokat egyértelműen le kell bontani modulokra. A modulok működésébe más modul nem szól bele, de elvárható, hogy minden modul a saját feladatát tökéletesen elvégezze: a modul felelős a feladata elvégzéséért. Nagyon fontos, hogy a modulok – amennyire lehetséges – egymástól függetlenek legyenek. Tartsuk szem előtt a két irányelvet:
 - **Laza külső kötés:** minél kevesebb legyen a modulok közötti kapcsolatok száma;
 - **Erős belső kötés:** az egy modulba csak azokat a dolgokat tegyük, amelyek szoros kapcsolatban állnak egymással (1.4. ábra).

Így a hibák egyszerűbben kiszűrhetők, a feladat átláthatóbb, könnyebben módosítható. Ha a tökéletesen működő modulokat, programrészeket végül pontosan egymáshoz illesztjük, akkor sokkal nagyobb áttekintésünk lehet a program egészéről.

- ◆ **Az adatok (az információ) elrejtésének elve:** A megfelelő programrészek lehetőleg csak a saját adataikon dolgozzanak. A részek csak akkor használjanak közös adatokat, ha az feltétlenül szükséges. „Közös lónak túros a háta”: sokkal könnyebb egy hibát felderíteni, ha egy programrész csak a maga adataiért felelős, és azok változtatásába más programrész nem is nyúlhat bele. Az a jó, ha a programrészek csak a kész adatokat adják át egymásnak.
- ◆ **A döntések elhalasztásának elve:** Csak akkor hozzunk meg egy döntést, ha az ellen- gedhetetlenül szükséges. Azokat a döntéseket, amelyekhez még nincs elegendő ismerek- tünk, halasszuk későbbre – egyébként előfordulhat, hogy a megoldás egy későbbi szá- kaszában felül kell bírálnunk eddigி döntéseinket.
- ◆ **A döntések kimondásának elve:** A feladat megoldása során ne hallgassunk el egyetlen döntést sem. Ha már meghoztunk egy döntést, akkor azt meg kell fogalmazni, le kell rögzíteni – mert később megfeledkezhetünk róla, és így ellentmondásos döntéseket hozhatunk.



1.4. ábra. Moduláris programozás

A modulokra bontás „iránya” kétféle lehet:

- ◆ **A felülről lefelé** (top-down) tervezés esetén a megoldást felülről lefelé, fokozatosan, lépésenként finomítjuk (stepwise refinement), és így a kis feladatokat csak a végső fázi- sokban oldjuk meg.
- ◆ **Az alulról felfelé** (bottom-up) tervezés lényege, hogy már kész modulokból építkezünk. Erre akkor kerül sor, amikor bizonyos részfeladatokat egy előző feladat kapcsán már megoldottunk, vagy amikor egy modulgyűjteményt (osztálykönyvtárat, rutingyűje- ményt) vásárolunk. A meglévő modulok közötti válogatás és azok pontos illesztése csak nagy programozói tapasztalattal lehetséges.

Egy feladat megoldásának tervezésekor rendszerint felülről lefelé indulunk. Amikor a feladatot már sikerült kellően kis modulokra lebontani, akkor a technikai részletek kidolgozásában már az alulról felfelé történő építkezés is fontos szerepet játszik.

Moduláris programozás: Olyan programozási mód, amelyben a teljes program modulokból áll. Az egyes modulok kezelhető méretűek, egyértelmű céljuk van, és jól definiáltan csatlakoznak környezetükhez. A moduláris programozás irányelvei:

- Oszd meg és uralkodj!
- Az adatok elrejtése
- A döntések elhalasztása, illetve kimondása.

A modulokra bontás irányára lehet **felülről lefelé** (top-down), és lehet **alulról felfelé** (bottom-up).

1.8. Módszertanok

Programot azért írunk, mert a számítógép nagy segítség, és egy jól megírt program segítségével az elvégzett munka mennyisége és minősége a sokszorosára növelhető. Az életben meg kell oldani bizonyos feladatokat, lehetőség szerint gyorsan és jól. A lehetőség adott: oldjuk meg a feladatot számítógéppel! Az első programnyelv tanulásakor az ember azt gondolná, hogy egy feladat számítógépre vitelében a legnehezebb dolog az utasítások kiválasztása egy adott programnyelvből. De ez nem így van. A legnehezebb dolog a megoldási **módszer** megtalálása, ez viszont majdnem nyelvfüggetlen. Ha megvan a módszer, akkor már „gyerekjáték” azt megfogalmazni bármely programozási nyelven, legyen az Java, Pascal vagy C++.

Egy jó szoftver sohasem születik véletlenszerűen. Még a legegyszerűbb program írásához is határozott elképzélés szükséges. Egy számítógépen megoldandó feladat felvetődésének pillanatában már is erős a csábítás, hogy az ember leüljön a számítógép elől, és már üsse is be az utasításokat, melyek – reményünk szerint – a feladat megoldásához vezetnek. Miután készen vagyunk a begépeléssel, a fordító által jelzett hibákat több-kevesebb idő alatt kijavítjuk. Az ilyen összedobott program azonban a legritkább esetben működik helyesen. És akkor elkezdődik a javítgatás. A program bizonyos adatokkal működik, másokkal nem. Ha valamit „kijavítunk”, helyette valami más hiba jelentkezik. Előbb-utóbb úgy összekeveredünk, hogy már azt sem tudjuk, mi is volt a feladat tulajdonképpen. A toldozás-foltozás nem a jó programozó mestersége –, **megbízható programot kizárolag módszeres tervezéssel készíthetünk!**

A különböző programtervezési módszertanok leginkább abban különböznek egymástól, hogy az adatokat és az adatokon dolgozó programrészleteket milyen modulokra bontják le, és a dekompozíciót milyen módon végzik el. Egy módszertanhoz szükségképpen valamilyen **grafikus jelölésrendszer** (modellező nyelv) is tartozik. Ezenkívül manapság egy módszertan már **CASE eszköz** nélkül is elképzelhetetlen. Egy CASE (Computer-Aided Software Engineering =

számítógéppel támogatott rendszertervezés) eszköz segítségével számítógépen tervezhetjük meg programunkat, és a terv alapján forráskódot is generálhatunk.

A programtervezési módszereket illetően hosszú ideig a **strukturált programozás, programtervezés** volt a jelszó –, a különböző módszerek ilyen szellemben születtek. Lényege, hogy a programot felülről lefelé, funkcionálisan, vagyis működés szerint egyre kisebb lépésekre (modulokra, eljárásokra) bontják. A **Jackson féle programtervezési módszer** olyan speciális strukturált módszer, amelyben a programszerkezetet a bemenő és kimenő adatszerkezetek összefésüléséből állítják fel.

A strukturált fejlesztéssel járó szoftverkritizist az objektumorientált (OO) szemlélet hivatott feloldani. Napjainkban már szinte kizárolag **objektumorientált szoftverfejlesztési módszereket** és szoftvereket használnak a fejlesztők. Az objektumorientált módszertanokban a dekompozíció alapja az objektum, melyben az adatok és programrészek egyaránt fontos szerepet játszanak. Itt az információ elrejtésének elve minden eddiginél jobban érvényesül: az egyes objektumok (modulok) adataikkal és viselkedésekkel együtt teljes mértékben felelősek a rájuk bízott feladatokért. Objektumorientált szoftverfejlesztési módszer alkalmazásával kisebb energia befektetésével és sokkal biztonságosabban tudunk elkészíteni egy, a mai követelményeknek megfelelő programot.

Könyvünkben a legelterjedtebb objektumorientált modellező nyelvet, az **UML-t** (Unified Modeling Language, Egységesített Modellező Nyelv) és a hozzá tartozó szoftverfejlesztési módszertant, az **Egységesített Eljárást** (Unified Process) fogjuk alkalmazni. Az UML-t és az Egységesített Eljárást CASE eszközök is támogatják, ilyen például a Rational Rose. CASE eszközzel azonban könyvünk nem foglalkozik, az UML-t és az Egységesített Eljárást pedig csak alapjaiban érintjük.

Egy **programfejlesztési módszertan** útmutatást ad a program elkészítésének módjára vonatkozóan. A módszertan segítséget nyújt az adatok (objektumok) és algoritmusok meghatározásában, a program modulokra bontásában, a modulok elkészítésében, majd a modulok összerakásában.

Egy módszertanhoz rendszerint tartozik valamelyen **grafikus jelölésrendszer**. A módszertanokat és a grafikus jelölésrendszereket különböző **CASE eszközök** támogatják.

Tesztkérdések

1.1. Jelölje meg az összes igaz állítást a következők közül!

- a) Az objektumnak vannak tulajdonságai és van viselkedése.
- b) A szoftver a számítógép kézzelfogható része.
- c) A programkészítés célja pusztán adatok szolgáltatása a felhasználó számára.
- d) A program a számítógép számára érthető instrukciók sorozata.

- 1.2. Jelölje meg az összes igaz állítást a következők közül!
 - a) A gépi kódú program a hardver szerves része.
 - b) A natív kód az ember számára könnyen értelmezhető programkód.
 - c) Az assembler fordítóprogram.
 - d) Az alacsony szintű nyelv a számítógéphez közelebb álló nyelv, míg a magas szintű nyelv emberközeli nyelv.
- 1.3. Jelölje meg az összes igaz állítást a következők közül!
 - a) A programozási nyelv olyan program, amely az ember által készített szöveges leírást értelmezi.
 - b) A Java fordító platformfüggetlen közbenső kódot, bájtkódot generál.
 - c) Az interpreter a tárgykódot visszaalakítja forráskódá.
 - d) A forráskód egy szöveg, melyet a fordítóprogram értelmez, illetve fordít le.
- 1.4. Jelölje meg az összes igaz állítást a következők közül!
 - a) A forráskód a számítógép által értelmezhető, futtatható kód.
 - b) A fordítóprogram a forráskódóból a számítógéphez közelebb álló kódot készít.
 - c) A tárgykód a gép által értelmezhető, futtatható kód.
 - d) A programszerkesztő a forrásprogram szövegszerkesztője.
- 1.5. Melyik négy fogalom állítható párba a következő négy fogalommal? Jelölje meg az egyetlen jó választ!

bájtkód – interpreter – compiler – linker

 - a) programszerkesztő – fordító – értelmező – platformfüggetlen közbenső kód
 - b) fordító – értelmező – platformfüggetlen közbenső kód – programszerkesztő
 - c) platformfüggetlen közbenső kód – programszerkesztő – értelmező – fordító
 - d) platformfüggetlen közbenső kód – értelmező – fordító – programszerkesztő
- 1.6. Jelölje meg az összes igaz állítást a következők közül!
 - a) A moduláris programozás lényege, hogy a programot kész modulokból állítsuk össze.
 - b) A szoftverkritizis a hagyományos szoftverfejlesztési módszer válsága.
 - c) A moduláris programozás egyik alapelve, hogy a modulok között minél erősebb kötés legyen.
 - d) A modulokra bontást minden esetben felülről lefelé kell elvégezni.
- 1.7. Mely szavak jellemzik a szoftver minőséget? Jelölje meg az összes jó választ!
 - a) Hatékonyúság
 - b) Helyesség
 - c) Szabványosság
 - d) Gyors fejlesztés
- 1.8. Jelölje meg az összes igaz állítást a következők közül!
 - a) A strukturált programozás az objektumorientált programozás továbbfejlesztése.
 - b) A programfejlesztési módszertan egy grafikus jelölésrendszer.
 - c) Az objektumorientált módszertanokban a dekompozíció alapja az objektum.
 - d) Az UML nem módszertan, hanem egy modellező nyelv.

2. Adat, algoritmus

A fejezet pontjai:

1. Az algoritmus fogalma
 2. Változó, típus
 3. Tevékenységsdiagram
 4. Pszeudokód
 5. Az algoritmus tulajdonságai
-

Egy program alapvetően adatokból és az adatok kezelésére szolgáló instrukciók (utasítások) sorozatából áll. A szoftverfejlesztés célja a végső adatok és algoritmusok megadása. Egy nagy feladatot nem lehet egyszerre megoldani, azt kisebb részekre kell bontani. A dekompozíció azt célozza, hogy a rendszer adatait és instrukciót (műveleteit) úgy csoportosítsuk, hogy a rendszer átlátható és könnyen programozható legyen. Nagyon fontos, hogy egyszerre csak egy áttekinthető adathalmazon dolgozzunk, és az adathalmazon áttekinthető műveletsort alkalmazzunk, másiképp a program bonyolulttá és érthetetlenné válik. Objektumorientált lebontás esetén a rendszer adatait és algoritmusait objektumokra bontjuk szét; minden objektum az adatoknak és algoritmusoknak egy jól meghatározott, nem túl nagy részét képviseli. Egy objektumorientált rendszerben a rendszer moduljai objektumok, melyeken újabb algoritmusok definiálhatók, megadva ezzel az objektumok együttműködését.

Bármilyen lebontási módszert követünk is, **adatainkat (objektumainkat)** és **algoritmusainkat meg kell tervezni!** E fejezetben tisztázzuk az adat és az algoritmus fogalmát, majd megismerünk két algoritmustervező eszközt, az UML tevékenységsdiagramját és egy pszeudokódot.

2.1. Az algoritmus fogalma

Az algoritmus az adatokat mozgató, manipuláló lépések, instrukciók sorozata, mely egy feladat megoldásához vezet. Algoritmusokkal a hétköznapokban is léptén-nyomon találkozunk: egy algoritmus sokféleképpen adható meg, például: szóban, rajzban, írásban vagy egy programozási nyelven. Egy számítógép által érthető algoritmust **programnak** nevezünk. Ezért hívják a

számítógépes program leíró nyelvét **programozási nyelvnek**. Mi a Java nyelvet használjuk majd adataink és algoritmusaink végső megadásához.

Az algoritmus megtervezése előtt meg kell határoznunk a feladatban szereplő adatokat, adatszerkezeteket. (Ennek analógiájára a magasabb szintű objektumorientált programtervezési módszer esetén majd objektumokat határozunk meg, és az azokat mozgató együttműködési algoritmust.)

Az adat és algoritmus fogalmát egy feladat segítségével tisztázzuk.

Feladat – A legfiatalabb lány kiválasztása

Hannibál tanár úr azt kapja feladatul: határozza meg a legfiatalabb lánytanulót az osztályában. Meg kell adnia a kérdéses lány nevét és születési dátumát.

Hannibál tanár úr egy feladatspecifikációt kapott. Először meg kell vizsgálnia (analizálnia kell), hogy egyáltalán képes-e a feladat megoldására. Hogyan gondolkodik Hannibál tanár úr? Ahhoz, hogy megoldja a feladatot, bemenő adatokra van szüksége, mégpedig az osztály összes tanulójának nevére, nemére és születési dátumára. Ha ez adva van, akkor a feladat egyértelmű, és a tanár minden bizonnal produkálni tudja az eredményt.

Honnan szerzi be Hannibál tanár úr az algoritmus bemenő adatait? Két választása van: vagy bemegy az osztályba, és begyűjt az adatokat, vagy az osztálynaplóból kinézi őket. Hannibál tanár úr ez utóbbit választja, mert a feladat sürgős, és az osztály már régen hazament. Némi gondolkodás után veszi tehát az osztálynaplót, és elkezdi lapozni. Az az elképzelése, hogy végignézi a naplót, és folyamatosan megjegyzi, ki az addigi legfiatalabb lány. A napló végiglapozása után az utoljára megjegyzett lány lesz a keresett. Elkezdi tehát lapozni a naplót, és minden egyes tanulónál megáll. Először azt nézi meg, hogy a tanuló fiú-e vagy lány. Ha fiú, akkor továbblapoz. Ha lány, akkor megnézi a születési dátumot. Ha ez későbbi, mint az utoljára megjegyzett lány születési dátuma, akkor az előző helyett most ezt a lányt jegyzi meg.

A tanuló neme nem szerepel a naplóban, azt a tanár a névből találja ki. A számítógép számára azonban csak az egyértelmű utak járhatók: vagy szerepelnie kell ennek az adatnak a naplóban, vagy kap a tanár egy listát a keresztnévekről, amelyben minden egyes keresztnév mellett szerepel a nemre vonatkozó információ.

Megjegyzés: A feladat megoldásához nem kellenek programozói ismeretek, elég a józan paraszti ész. A feladat számítógépes megoldásához természetesen már némi programozási tudás is szükséges.

Az algoritmus akkor egyértelmű, ha minden egyes lépésénél világos, mivel mit kell csinálni. Az algoritmust valamilyen módon formalizálni kell. Meg kell határozni az algoritmusban résztvevő, megjegyzendő, illetve karbantartandó adatokat, és pontosan le kell írni, hogy mitől függően milyen lépést kell tenni.

Egy algoritmusban a következő vezérlőszerkezetek lehetnek:

- ◆ Szekvencia: egymás után végrehajtandó tevékenységek sorozata
- ◆ Szelekció: választás megadott tevékenységek közül
- ◆ Iteráció: egy vagy több tevékenység feltételtől függő, ismételt végrehajtása
- ◆ Feltétel nélküli ugrás: vezérlés átadása az algoritmus egy megadott pontjára

Egy bonyolultabb algoritmust nem lehet „fejben” megtervezni, ahhoz eszközök kellenek. Ha ötleteinket saját, egyéni „módszerrel” vetjük papírra, azt mások nem fogják megérteni. Olyan eszközre van szükség, amely általánosan elfogadott, és a környezetünkben más emberek is ismerik, használják. A következőkben tisztázzuk a változó és a típus fogalmát, majd áttekintjük a tevékenységsdiagram és a pszeudokód **algoritmusrészletező eszközöket**. Mindkét eszközzel megadjuk A legfiatalabb lány kiválasztása algoritmusát.

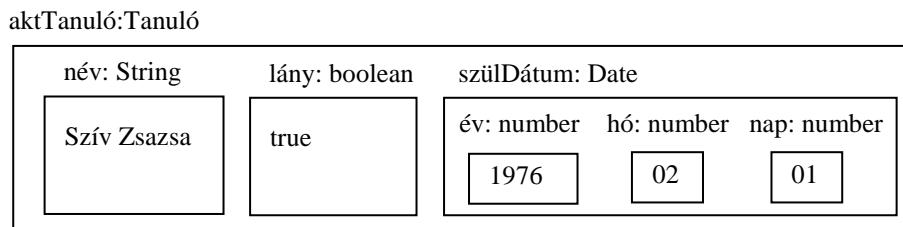
Algoritmus: Utasítások (tevékenységek) sorozata, amely egy feladat megoldásához vezet.

2.2. Változó, típus

Az algoritmus folyamán Hannibál tanár úrnak két tanuló adatlapját kell megjegyeznie: az aktuálisat, és a legfiatalabb lányét. Vesz tehát két „dobozt”, az egyikre ráírja, hogy aktTanuló, ebben tartja annak a tanulónak az adatlapját, amelyiknél éppen tart. A másik dobozra felírja, hogy legTanuló, ebben tartja az addigi legfiatalabb lánytanuló adatlapjának másolatát. A dobozokban egyforma, Tanuló típusú adatlapok vannak. A dobozok tartalmát a tanár úr állandóan cserélgeti, a feladat megoldását majd az utolsó állapot adja.

Változó

Az aktuális tanuló adatlapja így fest (a legtanuló adatlapja ugyanilyen, csak a doboz neve más):



A dobozokban levő értékek változhatnak, ezért a dobozokat változóknak nevezzük. A változókat logikailag csoportosítottuk: vannak elemi és összetett változók. Az aktuális tanuló adatlapja összetett változó, mely három változóból áll: név, lány és születési dátum. A születési dátum három további változóra bontható: évre, hónapra és napra. minden egyes változónak nevet

adunk (azonosítjuk), hogy később egyértelműen hivatkozhassunk rájuk (aktTanuló, név, lány, szüldátum, év, hó, nap).

Az összetett változó egy elemére a . (pont) minősítő segítségével hivatkozhatunk. Példánkban helyes változóhivatkozások a következők:

```
aktTanuló
aktTanuló.név
aktTanuló.lány
aktTanuló.szüldátum
aktTanuló.szüldátum.év
aktTanuló.szüldátum.hó
aktTanuló.szüldátum.nap
```

Egy programban a **változók** olyan memóriaterületek, amelyek különböző **értékeket** vehetnek fel. Egy változónak az algoritmus végrehajtása során változhat az értéke. Az aktTanuló.név változó értéke például "Szív Zsazsa"-ról könnyen kicserélődhet "Egri Kata"-ra, a hozzá tartozó születési dátum pedig (1978,06,20)-ra.

Típus

Minden változónak van egy jól meghatározott típusa. A változó csak a típusának megfelelő értékeket veheti fel, például egy szöveg típusú változóba csak szöveget tehetünk, dátum típusába dátumot stb. Szinte minden programozási nyelvben megtalálhatók a következő típusok:

- ◆ **number** (szám). Például 45, 99.9, -1.2, 1000000000.
- ◆ **boolean** (logikai), értéke true (igaz) vagy false (hamis).
- ◆ **String** (szöveg). Például: "Szív Zsazsa", "Egri Kata".
- ◆ **Date** (dátum), összetett típus, 3 számból áll: év, hó és nap. Például (1978,06,20).

A number és boolean típusú adatok primitív adatok, a String és Date objektumok (összetett típusúak). Az UML és Java szabvány szerint a primitív típusokat kisbetűvel, míg az összetett, objektum típusokat nagybetűvel szokás írni. Ezt a szokást követjük most mi is.

Példánkban a név változó String típusú, a lány változó boolean típusú, a szüldátum Date típusú, az év, hó és nap pedig number típusúak.

Egy változót az algoritmus csak típusának megfelelően kezelhet. Egy szöveget nem lehet megsorozni egy számmal, de le lehet például vágni belőle két karaktert. Két számot össze lehet adni, szorozni stb. Két szöveget, két dátumot és két számot össze lehet hasonlítani egymással (melyikük a nagyobb), de egy dátum és egy valós szám már nem hasonlítható össze.

Az algoritmusban használt változókat deklarálnunk kell: meg kell adnunk azok neveit és típusait. Például:

```
legTanuló: Tanuló
aktTanuló: Tanuló
```

Értékkadás

Egy változónak értékkadási utasítással (művelettel, tevékenységgel) adhatunk értéket. Az értékkadási utasítás (=) baloldalán egy változó szerepel, jobboldalán pedig egy érték, amely lehet akár egyetlen változó aktuális értéke, akár egy változókat is tartalmazó kifejezés, például:

```
legTanuló = aktTanuló
aktTanuló.név = "Egri Kata"
aktTanuló.szülDátum = (1976,02,01)
aktTanuló.szülDátum.év = aktEv-25
```

A változó olyan memóriaterület, melynek változhat az értéke. **Minden változónak van egy jó megfogalmazott típusa.** A változó csak típusának megfelelően kezelhető, és abba csak olyan értékek tehetők, melyek beletartoznak az adott típus értékkészletébe. **Az algoritmus változóit deklarálni kell:** meg kell adni azok neveit és típusait. **Egy változónak értékkadási utasítással adhatunk értéket.**

2.3. Tevékenysésgdiagram

A **tevékenysésgdiagram** (aktivitási diagram, activity diagram) algoritmus leírására szolgáló grafikus jelölésrendszer. Segítségével a program dinamikus viselkedését tudjuk ábrázolni.

A diagramból kiolvasható, hogy adott feltételek mellett mely tevékenységek kerülnek végrehajtásra egymás után. A program „folyásának” irányát nyilak mutatják. A tevékenységek egymásutánjának minden előzőre következője előtt előfordulhat, hogy egy elágazásnál nem lehet előtérrel, merre kell menni.

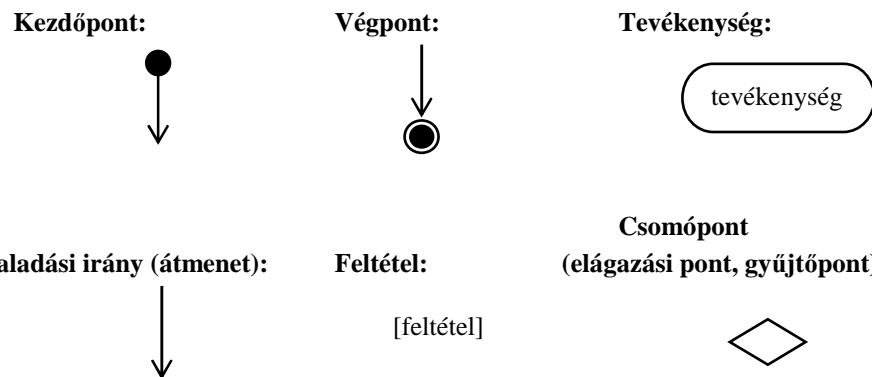
Megjegyzés: A tevékenysésgdiagram az UML diagramja (az állapotdiagram egy speciális formája), és nagyon hasonlít a sokak számára jól ismert folyamatábrához.

A tevékenysésgdiagram alapelemei

A tevékenysésgdiagram alapelemeit a 2.1. ábra mutatja. Az alapelemek magyarázata:

- ◆ **Kezdőpont, végpont:** Az algoritmus bemenete a kezdőpont, itt kezdődik az algoritmus végrehajtása (példaként tekintse a 2.2. ábrát). A végpontnál az algoritmus befejeződik. Egy algoritmusnak egyetlen kezdőpontja (belépési pontja) van, de lehet több végpontja (kilépési pontja) is.
- ◆ **Tevékenység:** Az adatokon dolgozó utasítás vagy utasítássorozat, amely véges idő alatt lezajlik (a tevékenység jelenthet egy beágyazott, összetett algoritmust is).
- ◆ **Haladási irány (átmenet):** A nyíl minden algoritmusban a haladási irányt mutatja időben (átmenet az egyik tevékenységből a másikba). Ha egy tevékenység végrehajtása befejeződött, akkor ezt követően a belőle kiinduló nyíl által mutatott tevékenység kerül végrehajtára.

- ◆ **Feltétel:** Ha a nyíl mellett egy feltétel található, akkor a mutatott tevékenység csak akkor kerül végrehajtásra, ha a feltétel igaz. Ha nincs feltétel, akkor a mutatott tevékenység mindenki által végrehajtódik. Ha több nyíl indul ki a tevékenységből, akkor azok mindenki által végrehajtásra kerülnek. Ha több nyíl indul ki a tevékenységből, akkor azok mindenki által végrehajtásra kerülnek.
- ◆ **Csomópont (elágazási pont, gyűjtőpont):** Szükség esetén a diagramra csomópontokat lehetünk. Egy elágazási pontból a program több irányba mehet, egy gyűjtőpontba a program több irányból érkezhet. Egy csomópont lehet elágazási és gyűjtőpont is egyszerre.



2.1. ábra. A tevékenységgdiagram alapelemei

Megjegyzések:

- Részalgoritmus esetén a kezdőpontot nem minden ábrázolják.
- Az aktivitásdiagram párhuzamos (egy időben futó) algoritmusok ábrázolására is alkalmas, de erről ebben a könyvben nem lesz szó.

Az aktivitásdiagramon bármely tevékenység lehet összetett tevékenység is, vagyis az algoritmusban bármely tevékenység helyettesíthető egy részalgoritmussal.

Egy feltétel értéke `true` vagy `false` lehet, ezek általában értékek összehasonlításából, valamint feltételek logikai összekapcsolásából adódnak. A feltételek megadásakor mi elsősorban a Java összehasonlító és logikai műveleteit alkalmazzuk, de elfogadhatók a jól érthető, természetes megfogalmazások is:

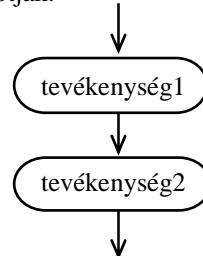
- ◆ Összehasonlító műveletek: `==` (egyenlő-e), `!=` (nem egyenlő-e), `>` (nagyobb-e), `>=` (nagyobb vagy egyenlő-e), `<` (kisebb-e), `<=` (kisebb vagy egyenlő-e)
- ◆ Logikai műveletek: `&` (and/és); `|` (or/vagy); `!` (not/nem)

A következő feltétel azt fogalmazza meg, hogy az a változó értéke 5 és 10 közé esik, beleértve a határokat is: $(a >= 5) \& (a <= 10)$. Ha $a = 7$, akkor a feltétel értéke true, azaz teljesült a feltétel.

Egy algoritmusban vannak jellegzetes vezérlési struktúrák, ilyenek a szekvencia, a szelekció és az iteráció (az ugró utasításokkal most nem foglalkozunk). Nézzük most sorra ezeknek a tevékenységsdiagramját!

Szekvencia

A szekvencia egymás utáni tevékenységek sorozata. A szekvenciát a nyilak irányában felsorolt egymás utáni tevékenységek alkotják:

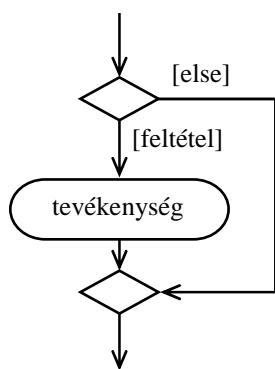


Szelekciók

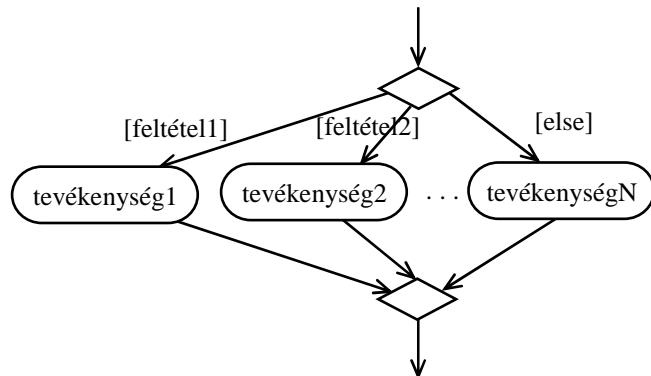
A szelekció programelágazást jelent; egy adott ponton a tevékenységek végrehajtása feltételektől függ. Szelekció esetén a tevékenységből vagy az elágazási pontból kifelé vezető nyilak (programágak) mindenikén szerepel egy feltétel. Ha igaz egy adott feltétel, akkor a feltétel vonalának irányában halad tovább a vezérlés, és a mutatott tevékenység hajtódik végre. Megadható egy **egyébként** (else) ág: ha a többi feltétel nem teljesül, akkor erre az ágra kerül a vezérlés. Az egyébként ágra az [else] feltételt írjuk.

A feltételek között nem lehet átfedés, és le kell fedniük minden lehetséges esetet. minden esetben pontosan egy ágra kerül a vezérlés. A feltélesen végrehajtandó tevékenységek számától függően szokás egyágú, kétágú és többágú szelekcióról beszélni. Az egyágú és többágú szelekció tevékenységsdiagramja a következő:

Egyágú szelekció:



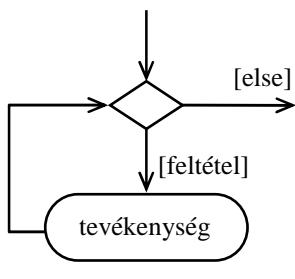
Többágú szelekció:



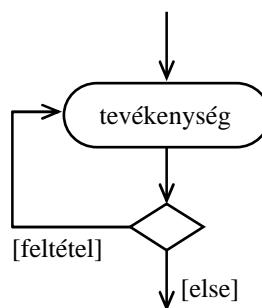
Iterációk

Az iteráció bizonyos tevékenység(ek) ismételt végrehajtása. Az ismétlendő tevékenysége(ke)t **ciklusmagnak** nevezünk. Az iterációt kissé pongyolán ciklusnak is szokás nevezni, bár a ciklus szoros értelemben a ciklusmagot jelenti. Iteráció esetén a vezérlés ismételten viszszatér a ciklusmag elé, de az újból végrehajtás előtt egy elágazási pont található, mely megengedi a ciklus elhagyását. Ha a vezérlés nem tudja elhagyni a ciklust, **végtelen ciklusról** beszélünk. Azt a ciklust, melynek nincsen magja (nincs benne utasítás), **üres ciklusnak** nevezünk. Kétfajta iteráció létezik: az egyik a ciklusmag előtt tesztel, a másik a ciklusmag után, hátul. A hátultesztelő iteráció egyszer mindenkorban végrehajtódik:

Elöltesztelő ciklus:



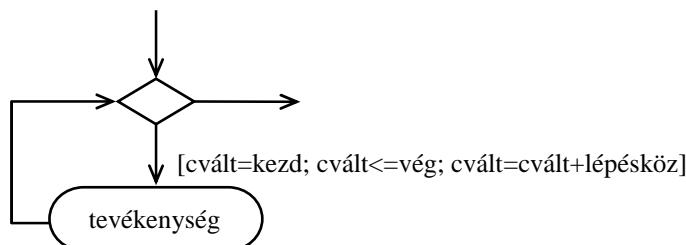
Hátultesztelő ciklus:



Az iterációkat aszerint is szokás osztályozni, hogy **belépési vagy kilépési feltételt** adunk-e meg (a feltétel megfordítható, ekkor az igaz és hamis ágak felcserélődnek). Az ábrán levő minden ciklus belépési feltételt fogalmaz meg.

Léptető (számláló, növekményes) ciklus:

A **léptető ciklus** azt jelenti, hogy a ciklusmag minden egyes végrehajtásakor egy ún. ciklusváltozó automatikusan lép egyet (megváltozik, például növekszik vagy csökken). A ciklusváltozót, annak kezdeti értékét, a ciklusba való belépés feltételét és a léptetési utasítást (lépésköz) a ciklus fejében adhatjuk meg. A ciklusmag használhatja ciklusváltozót. A léptető ciklust a tevékenységszabályzónak elöltesztelő ciklussal tudjuk megvalósítani:



A legfiatalabb lány kiválasztása

Készítsük most el A legfiatalabb lány kiválasztása feladat megoldásának tervét! Először meghatározzuk az adatokat, melyeken az algoritmus dolgozik majd:

Adatok:

Mindössze két Tanuló típusú változót (adatlapot) kell felvennünk:

- ◆ aktTanuló: Tanuló (név:String, lány:boolean, szüldátum:Date)
- ◆ legTanuló: Tanuló (név:String, lány:boolean, szüldátum:Date)

Az aktTanuló változóba tesszük majd minden az aktuális tanuló adatlapját. Az algoritmus egyszerre csak egy aktTanuló adatlapot érhet el. A legTanuló változóba tesszük az addigi legfiatalabb lány adatlapját, ez adja majd az összes adatlap feldolgozása után a feladat megoldását.

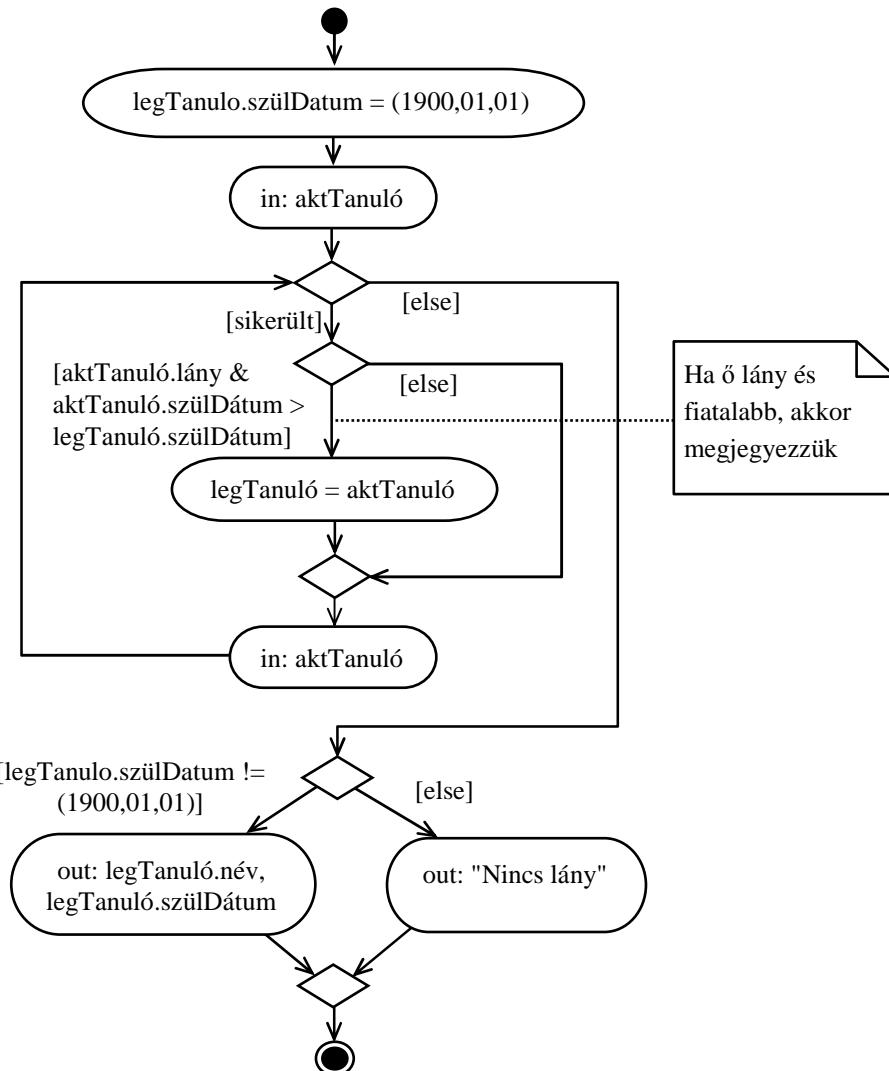
Algoritmus:

Az algoritmus tevékenyséendiagramja a 2.2. ábrán látható. Elemezzük az algoritmust:

Első tevékenységeként a legTanuló adatlap szüldátum változójába egy olyan dátumot teszünk, amelynél bármely lánytanuló születési dátuma biztosan későbbi: (1900,01,01). Feltételezzük, hogy a lányok között van 100 évesnél fiatalabb; ha nem így lenne, algoritmusunk nem működnék helyesen. Második tevékenységeként bekérjük a felhasználótól a következő tanuló adatlapját, melyet az aktTanuló változóban helyezünk el (az in=be adatbevitelt jelez). Ha van még tanuló a naplóban, akkor a bevitel sikeres, és „feldolgozzuk” az adatlapot, egyébként az iteráció befejeződik.

A ciklusmagban megnézzük, hogy az aktTanuló lány-e, és fiatalabb-e, mint az eddigi legfiatalabb. Ha minden teljesül, akkor az eddigi legfiatalabb lány adatlapját (a legTanuló-t) kicseréljük az aktTanuló adatlapjának egy másolatára. A ciklus végén bekérjük a következő tanuló adatlapját. A ciklus addig tart, amíg maradt még tanuló, vagyis sikeres a bekérés. Hogy egy tanuló fiatalabb-e, mint a másik, azt úgy dönthetjük el, hogy összehasonlítjuk születési dátumaiat. Két dátum összehasonlítása nem természetes, azt vagy tudja egy adott programnyelv, vagy nem. Az összehasonlítás lehet egy újabb algoritmus, melyben az évet, a hónapot és a napot egyenként össze kell hasonlítani. Ezt az algoritmust a következő pontban, a pszeudokódnál meg fogjuk adni.

Végül kiírjuk az eredményt. Elfordulhat, hogy a felhasználó egyetlen tanulót sem vitt be, vagy pedig nem volt a tanulók között lány (a programozónak minden lehetőségre fel kell készülnie), ezért ezt az esetet külön kezeljük. Ha volt lány, akkor kiírjuk az eredményt (out=ki), egyébként kiírjuk, hogy „Nincs lány”.



2.2. ábra. Tevékenységsdiagram – A legfiatalabb lány kiválasztása

A tevékenységsdiagramban a nyilak elvileg akárhova húzhatók, ezért nagyon könnyű teljesen áttekinthetően diagramot készíteni. **A tevékenységsdiagramnak legyen minden tiszta a szerkeze:** egyértelműen lehessen látni benne a vezérlőszerkezeteket! A következő pontban tárgyalandó pszeudokódnak sokkal kötöttebb formája van, így sok esetben könnyebb azt áttekinteni. Egy aktivitásdiagramnak akkor tiszta a szerkezete, ha könnyedén meg tudjuk adni annak pszeudokódos változatát.

2.4. Pszeudokód

A **pszeudokód** lényege, hogy a programot az emberi gondolkodáshoz közel álló mondat szerű elemekből építjük fel. Annyiban tér el a folyamatos magyar írástól, hogy itt be kell tartanunk bizonyos szabályokat – a vezérlőszerkezetek képzésére megállapodás szerinti formákat és szavakat veszünk igénybe. A tevékenységek (utasítások) szabad formában is megadhatók. Az adatok deklarálása a pszeudokód része. Adatok bekérésére/megjelenítésére szokás az `in/out` kulcsszó használata:

```
in: változó1, változó2, ...
out: kifejezés1, kifejezés2, ...
```

A könnyebb érthetőség kedvéért előre összefoglaljuk az itt tárgyalt angol szavak jelentését ábécérendben:

| | | | |
|-------------------|----------------------------------|--------------------|-------------------|
| <code>do</code> | <code>csináld</code> | <code>if</code> | <code>ha</code> |
| <code>else</code> | <code>egyébként, különben</code> | <code>in</code> | <code>be</code> |
| <code>end</code> | <code>vége</code> | <code>out</code> | <code>ki</code> |
| <code>for</code> | <code>-ra, -re</code> | <code>while</code> | <code>amíg</code> |

Könyvünk egy olyan „egyéni” pszeudokódot alkalmaz, amelyben a vezérlőszerkezeteket megadó kulcsszavak erősen hasonlítanak a legtöbb programozási nyelvben alkalmazott kulcsszavakhoz. Amikor szükség van rá, ezt a pszeudokódot fogjuk használni.

Nézzük most sorra a vezérlőszerkezetek pszeudokódjait! Előrebocsátjuk, hogy ahol a vezérlőszerkezetben egy tevékenység szerepel, oda több tevékenység is beilleszthető.

Szekvencia

```
tevékenység1
tevékenység2
...

```

`tevékenység1, tevékenység2` stb. egymás után kerülnek végrehajtásra.

Például három szám bekérése egymás után:

```
in: szam1
in: szam2
in: szam3
```

Szelekciók

Egyágú szelekció

```
if feltétel
  tevékenység
end if
```

Ha a `feltétel` teljesül, akkor a `tevékenység` végrehajtásra kerül, egyébként nem. A program az `end if` után folytatódik.

Például: ha az életkor 100-nál több, kiírjuk az "idős" szöveget:

```
...
if eletkor>100
  out: "idős"
end if
```

Többágú szelekció

```
if feltételi
  tevékenység1
else if feltétel2
  tevékenység2
else if feltétel3
  tevékenység3
...
else
  tevékenységN
end if
```

A `feltételi` teljesülése esetén a `tevékenység1` kerül végrehajtásra. Ha egyik feltétel sem teljesül, akkor az `else` ágra kerül a vezérlés. Az `else` ág elhagyható; ilyenkor ha egyik feltétel sem teljesült, nem hajtódi végre semmi. A program minden esetben az `end if` utáni `tevékenységgel` folytatódik.

Például: bekérünk egy életkort, és kiírjuk a korának megfelelő jellemzést:

```
életkor: number
in: életkor
if életkor<14
  out: "gyerek"
else if életkor<18
  out: "fiatal"
else
  out: "felnőtt"
end if
```

Ciklusok

Egy tevékenység a megadott feltétel függvényében ismételten végrehajtódik.

Elöltesztelő ciklus

```
while feltétel
  tevékenység
end while
```

A feltétel itt belépési feltétel. Megengedett az üres ciklus is.

Például: négyzetszámok kiírása 1000-ig:

```
szam: number
szam=1
while szam*szam<=1000
    out: szam*szam
    szam=szam+1
end while
```

Hátultesztelő ciklus

```
do
    tevékenység
end do while feltétel
```

Az újbóli belépésről a ciklus végén születik döntés. A ciklusmag egyszer mindenkorban végrehajtódik. A ciklus végén belépési feltételt adunk meg.

Például: szám bekérése, amíg nem ütnek be egy 10 és 20 közötti értéket (határokat is beleérve):

```
szam: number
do
    in: szam
end do while szam<=10 | szam>=20
```

Léptető (számláló, növekményes) ciklus

```
for cvált=kezd; cvált<=vég; cvált=cvált+lépésköz
    tevékenység
end for
```

A tevékenység a ciklusváltozó kezdő és végértéke között minden olyan értékre végrehajtódik, amelyik a léptetési utasítás hatására előáll.

Például számok kiírása 200-tól 250-ig:

```
i:number
for i=200; i<=250; i=i+1
    out: i
end for
```

Rutin (eljárás, függvény)

A rutin egy külön névvel ellátott összetett tevékenység. A rutin meghívható a nevére történő hivatkozással. A rutinnak lehetnek paraméterei, melyek a rutin végrehajtásakor felhasználhatók, és lehet visszatérési értéke, amely a hívás helyén használható fel. A visszatérési érték nélküli rutint eljárásnak, míg a visszatérési értékkel rendelkező rutint függvénynek nevezzük.

Megjegyzés: A C++ nyelv szerint minden rutin függvény, amelyek közül egyeseknek a visszatérési értéke void (semleges), azaz nincs visszatérési értékük. A Java terminológia hasonlít ehhez, de ott nem függvényekről, hanem metódusokról beszélünk. Ettől függetlenül fogalmi szinten eljárás és függvény egyaránt létezik.

Eljárás

Az eljárás egy visszatérési értékkel nem rendelkező rutin. Egy eljárást a következőképpen írnunk meg (definiálunk):

```
eljárásnév(par1:Típus1, par2:Típus2, ...)
    tevékenység(ek)
end eljárásnév
```

Az eljárás meghívható a nevére hivatkozással:

```
eljárásnév(érték1, érték2, ...)
```

A híváskor megadott paraméterek számának meg kell egyeznie az eljárás definíciójában megadott paraméterek számával, és a paraméterek típusainak is rendre meg kell egyezniük. Végrehajtáskor par1 értéke érték1 lesz, par2 értéke érték2 stb. Az eljárás a paraméterül kapott értékekkel dolgozik. Előfordulhat, hogy az eljárásnak nincsen paramétere; a nyitó-csukó zárójel-párost ilyenkor is le kell írnunk mind a deklarációban, mind a hívás helyén.

Eljárás például:

- ◆ Megadott két szám összeadása, és az eredmény kiírása. Az összead eljárást meghívunk egy kiir(n) eljárást, amely a paraméterben megadott számot 4 karakterhelyen jobbra igazítva írja ki:

```
összead(a:number, b:number)
    kiir(a)
    kiir(b)
    out: "----"
    kiir(a+b)
end összead
```

Az eljárás hívása:

```
összead(986, 45)
```

Az eljárás végrehajtásakor a konzolon ez fog megjelenni:

```
986
45
-----
1031
```

Függvény

A függvény olyan rutin, amely visszatérési értékkel rendelkezik. Egy függvény belsejében a visszatérési értéket egy return utasítással adjuk vissza:

```
függvénynév(par1: Típus1, par2: Típus2, ...): Típus
    tevékenység(ek)
        return érték      // a függvény visszatérési értéke
    end függvénynév
```

Ahol egy adott típusú érték leírható, ott függvényt is hívhatunk. A függvény hívása történhet például az értékdás jobboldalán:

```
változó = függvénynév(érték1, érték2, ...)
```

Függvények például:

- ◆ Négyzetfüggvény (egy szám négyzetét adja vissza):

```
sqr(x:number): number
    return x*x
end sqr
```

A függvény hívása például:

```
out: sqr(15)           // 225-öt ír ki
```

- ◆ Maximumfüggvény (két szám közül a nagyobbikat adja vissza):

```
max(a:number, b:number): number
    if a>b
        return a
    else
        return b
    end if
end max
```

A függvény hívása például:

```
n,x: number
n = 5
x = max(n,17)    // x == 17
```

A kód strukturálása

A pszeudokódban, illetve a program forráskódjában a vezérlőszerkezetek (szekvencia, szelekció és iteráció) jelenlétét a forrásszöveg vízszintes eltolásával fejezzük ki. Ha egy tevékenység beljebb kezdődik, mint egy valamelyik felette álló elem, az minden esetben egy vezérlőszerkezet jelenlétét mutatja. **A vezérlőszerkezetek belső alarendelt egységeit a forráskód-ban jobbra toljuk. A szép, strukturált forráskód olvashatóvá teszi a programot.** A struktúrálás szabályai a programozóra nézve kötelezők!

A pszeudokód sokkal kötöttebb forma, mint a tevékenységsdiagram. minden pszeudokód könnyen átalakítható tevékenységsdiagrammá, de ez fordítva nem igaz. Egy tevékenységsdia-

gramban könnyű olyan átmenetet létrehozni két tevékenység között, hogy az ábra bizonyos részei a három strukturált vezérlőszerkezet felhasználásával nem alakíthatók át pszeudokóddá!

A legfiatalabb lány kiválasztása

Most megadjuk A legfiatalabb lány kiválasztása feladat algoritmusának pszeudokódját:

Adatok (mint a tevékenysédiagramnál)

- ◆ aktTanuló: Tanuló(név:String, lány:boolean, szülDátum:Date)
- ◆ legTanuló: Tanuló(név:String, lány:boolean, szülDátum:Date)

Algoritmus

A nagyobb függvény megadja, hogy az első dátum későbbi-e, mint a második:

```
nagyobb(dátum1:Date, dátum2:Date): boolean
    nagy : boolean = false
    if (dátum1.év>dátum2.év)
        nagy = true
    else if (dátum1.év==dátum2.év) & (dátum1.hó>dátum2.hó)
        nagy = true
    else if (dátum1.év==dátum2.év) & (dátum1.hó==dátum2.hó) &
        (dátum1.nap>dátum2.nap)
        nagy = true
    end if
    return nagy
end nagyobb
```

A kiir eljárás kiírja a megadott tanuló adatait:

```
kiir(tanulo: Tanuló)
    out: tanuló.név, tanuló.szülDátum.év, tanuló.szülDátum.hó,
          tanuló.szülDátum.nap
end kiir
```

A fő algoritmus:

```
legfiatalabbLány()
    legTanuló.szülDátum = (1900,01,01)
    in: aktTanuló
    while sikerült
        if (aktTanuló.lány &
            nagyobb(aktTanuló.szülDátum,legTanuló.szülDátum))
            legTanulo = aktTanulo
        end if
        in: aktTanulo
    end while
    if legTanulo.szulDatum != (1900,01,01)
        out: "Legfiatalabb lány:"
        kiir(legTanuló)
    else
        out: "Nincs lány"
    end if
end legfiatalabbLány
```

Pszeudokód: algoritmus leírására szolgáló, mondatszerű elemekből felépülő szöveges jelölésrendszer. Segítségével a program dinamikus viselkedését tudjuk leírni. A vezérlő-szerkezetek (szekvencia, szelekció és iteráció) belső alarendelt egységeit a forráskódban jobbra toljuk. A szép, strukturált forráskód olvashatóvá teszi a programot.

A rutin egy külön névvel ellátott összetett tevékenység, amelynek lehetnek paraméterei. A rutin lehet **eljárás** vagy **függvény**. Az eljárásnak nincs visszatérési értéke, a függvénynek van.

Mielőtt hozzájárunk az **algoritmus megtervezéséhez**, meg kell határoznunk azokat az adatokat, objektumokat, melyeken az algoritmus dolgozni fog! Ügyeljünk a következőkre:

- Az adatok többsnyire főnevek, a tevékenységek pedig igék vagy igéből képzett főnevek!
- A megtervezett algoritmus legyen áttekinthető, olvasható és egyértelmű! A tervezéshez lehetőleg szabványos jelölésrendszert alkalmazzunk!
- A hosszú és bonyolult algoritmusokat osszuk részekre!

2.5. Az algoritmus tulajdonságai

Az algoritmus leglényegesebb tulajdonságait a következő pontokban foglalhatjuk össze:

- ◆ **Az algoritmus lépésekkel** (elemi tevékenységekből, instrukciókból, utasításokból) **áll**.
- ◆ **Minden lépésnek egyértelműen végrehajthatónak kell lennie.** Az algoritmus leírásában a végrehajtót minden lehetséges esetre előre fel kell készíteni. A végrehajtó egység minden lépés után eldönti, mi lesz a következő lépés.
- ◆ **Egy algoritmusban hivatkozhatunk összetett lépésekre is**, ezek részletezését külön megadhatjuk.
- ◆ **A végrehajtandó instrukciónak valamilyen célja van.** A végrehajtás során valamilyen változás következik be. Általában megváltoznak az adatok értékei.
- ◆ **Az algoritmus véges számú lépésből áll.** Ez azt jelenti, hogy az algoritmizáló ember a feladat megoldásával, vagyis az algoritmus leírásával előbb-utóbb végez. A végrehajtás, vagyis a program is általában véges számú lépés után befejeződik, de vannak esetek, amikor a megoldásnak nem az a lényege, hogy valahonnan valahová eljussunk, hanem a megoldás maga az algoritmus. Tekintsük a MALÉV számítógépes rendszerét! Mikor állhat le a program? Jó lenne, ha nem éppen akkor szűnne meg a számítógépes irányítás, amikor mi ülünk egyik gépük fedélzetén.
- ◆ **Az algoritmusnak általában vannak bemenő (input) adatai, melyeket felhasznál.** A bemenő adatok példánkban a tanulók adatlapjai.
- ◆ **Az algoritmusnak legalább egy kimenő (output) adatot eredményeznie kell.** Az a folyamat, amely briliáns dolgokat művel, de nem kommunikál a külvilággal, definíció szerint nem algoritmus. Példánkban a kimenő adat a legfiatalabb lány adatlapja.

Strukturált algoritmus

Bebizonyították, hogy minden olyan algoritmus, amelynek egy belépési és egy kilépési pontja van, a szekvencia, a szelekció és az iteráció segítségével felépíthető (Böhm és Jacopini tétele). A strukturált programozás általános módszerét E. W. Dijkstra dolgozta ki. A strukturált programozásban ismeretlen a feltétel nélküli ugrás fogalma, így például nem ugorhatunk ki egy ciklusból. Ebből következik, hogy a program minden szekvenciájának – és így az egész programnak is – egyetlen belépési és egyetlen kilépési pontja van.

Megjegyezzük, hogy ebben a könyvben mi az objektumorientált programozás elveit fogjuk követni. Az objektumorientált programozásban az algoritmusok kicsik, és így az azokból való kiugrás nem rontja az áttekinthetőséget. Ok nélkül természetesen nem ugrunk ki sem ciklusból, sem rutinból. **Az objektumorientált programozás alapelvei a strukturált és a moduláris programozás tapasztalataira épülnek.**

Strukturált algoritmus: Olyan algoritmus, amely csak szekvenciákból, szelekciókból és iterációkból építkezik.

Tesztkérdések

- 2.1. Jelölje meg az összes igaz állítást a következők közül!
 - a) minden algoritmusnak kell, hogy legyen bemenő adata.
 - b) Az algoritmus olyan instrukciók sorozata, amely egy feladat megoldásához vezet.
 - c) Az algoritmus akárhány lépésből állhat.
 - d) A program a számítógép által érhető algoritmus.
- 2.2. Jelölje meg az összes igaz állítást a következők közül!
 - a) A változó olyan memóriaterület, mely a program futása során változhat.
 - b) Egy változónak többféle típusa is lehet.
 - c) A tevékenységsdiagram olyan diagram, amelynek segítségével követni lehet a programban található tevékenységek egymásutániságát.
 - d) A szelekció egymás utáni tevékenységek sorozata.
- 2.3. Jelölje meg az összes igaz állítást a következők közül!
 - a) A pszeudokód programozási nyelv.
 - b) A pszeudokód strukturálása a programozóra nézve kötelező.
 - c) Az iteráció a ciklusmagot egy előre megadott számszor hajtja végre.
 - d) A léptető ciklus más elnevezése növekményes ciklus.
- 2.4. Jelölje meg az összes igaz állítást a következők közül!
 - a) Szekvencia esetén feltételtől függően egy vagy több tevékenység közül választhatunk.
 - b) A léptető ciklus hátul tesztel.
 - c) A hátultesztelő ciklus megengedi, hogy a ciklusmag egyszer se hajtódék végre.
 - d) Azt a ciklust, melynek nincsen magja, üres ciklusnak nevezzük.

2.5. Jelölje meg az összes igaz állítást a következők közül!

- a) A rutin egy külön névvel ellátott összetett tevékenység.
- b) Az eljárásnak nincs visszatérési értéke.
- c) Egy strukturált algoritmusban nem lehet a ciklusból kiugrani.
- d) Ajánlatos a tevékenységeket főnevekkel azonosítani.

Feladatok

Tervezze meg minden két algoritmusleíró eszköz (tevékenységsdiagram és pszeudokód) segítséggel a következő algoritmusokat (egyeztesse a tervezetet, hogy megfelelnek-e egymásnak!)

- 2.1. **(A)**¹ Egy adott napon történő vásárlásokról az összes számla a háziasszonny asztalán hever. minden számlán szerepel az elköltött összeg. Segítsünk a háziasszonynak, és készítsünk olyan algoritmust, amely meghatározza az aznap elköltött összeget! Jelenítsük meg számára ezt az információt! A feladatot kétféleképpen is oldjuk meg:
 - a) Tegyük fel, hogy összesen 10 darab számla van az asztalon, s ezt előre tudjuk.
 - b) Nem tudjuk előre a számlák számát.*(Szamlak.html)*²
- 2.2. **(A)** Az olimpiai játékok idején sorban jönnek a hírek az érmekről. Készítsünk egy olyan algoritmust, amely jegyzi a híreket, és az olimpia végén megmondja, hány arany-, ezüst-, illetve bronzérmünk született összesen! (*Olimpia.html*)
- 2.3. A program kérje be sorban a hallgatók nevét és tanulmányi átlagát! Hogy nincs több hallgató, azt a név="*-gal jelezzük. A program írja ki a legjobb és a legrosszabb átlagú hallgató nevét! (*Tanulmány.html*)
 - a) **(B)** Több egyforma hallgató esetén elegendő csak egyiküket kiírni.
 - b) **(C)** Több egyforma hallgató esetén írjuk ki az összes hallgató nevét!

Csoportmunka

- 2.4. **(B)** Alakítsunk ki három, egyenként 3 fős csoportot!
 - a) Az első csoport készítsen egy feladatspecifikációt! Ügyeljenek a feladatspecifikáció egyértelműségére és a megoldhatóságra!
 - b) A második csoport ellenőrizze a feladatspecifikációt, majd készítse el a feladat tervezett tevékenységsdiagramban!
 - c) A harmadik csoport készítse el a terv alapján a pszeudokódot!

A csoportok végül beszéljék meg, mire kell ügyelni a feladat megfogalmazásakor és a tervezésben, hogy a feladat és a megoldás menete más számára is érthető legyen!

¹ A feladat nehézségi szintjét ABC-vel jelöljük: **(A)** rutinfeladat; **(B)** könnyű feladat; **(C)** nehezebb feladat. (Lásd Tanulási és jelölési útmutató)

² A megoldások a könyv mellékletében megtalálhatók.

3. A szoftver fejlesztése

A fejezet pontjai:

1. A szoftverfejlesztés alkotómunka
 2. Az Egységesített Eljárás
 3. Követelményfeltárás
 4. Analízis
 5. Tervezés
 6. Implementálás (kódolás)
 7. Tesztelés
 8. Dokumentálás
-

Egy adott feladat megoldása számítógép segítségével hosszú és bonyolult munka. Egy program, illetve programrendszer fejlesztése heteket, hónapokat, sőt éveket vehet igénybe – gondoljuk csak el, mekkora lehet például egy nemzetközi biztosító társaság számítógépes rendszere. A számítástechnika fejlődésével a munka egyre nagyobb része van ugyan gépesítve, de a lényegi részt mégiscsak embernek kell elvégeznie, hiszen a számítógép nem tud gondolkodni, nincsen akarata, ízlése. A munka ember által készített részei azonban mindenkor lehetnek, mert az ember természeténél fogva meglehetősen pontatlanságos. A számítógépnek persze könnyű pontosnak lennie, az csak a betáplált szabályok szerint dolgozik. De az ember éppen azért más, mint a gép, hogy olyan problémákkal is meg tud birkózni, amilyeneket még nem tápláltak belé. Az az ember, aki ésszerű szabályok szerint dolgozik, nemcsak okos lehet, hanem pontos is. Ez a fejezet a szoftverfejlesztéssel kapcsolatos alapvető ismereteket, szabályokat tárgyalja.

3.1. A szoftverfejlesztés alkotómunka

A szoftverfejlesztők hamar rájöttek, hogy egy rendszer csak akkor lesz könnyen karbantartható, illetve továbbfejleszthető, ha a rendszer fejlesztése megadott szabályok szerint történik. A feladatot pontosan definiálni kell, a megoldást pedig módszeresen modulokra kell bontani, hogy minél rövidebb idő alatt áttekinthető, hibamentes és újrafelhasználható szoftvert fejleszthessünk.

Ha nagyon kicsi a program, és nem szükséges a programot dokumentálni (későbbi felhasználás céljából érhetően leírni), akkor elképzelhető, hogy az ember leül a gép elé, és rögtön elkezdi írni a forrásprogramot; de csak akkor szabad ilyent tenni, ha a program terve a fejünkben már teljes egészében összeállt. Van, aki könnyebben át tud gondolni bizonyos programokat, másnak papírra és ceruzára van szüksége mindenhez. Egy a lényeg: **a programozás nem improvizáció, hanem felelősségteljes alkotás, mivel felelősek vagyunk mind a program működéséért, mind annak olvashatoságáért és javíthatóságáért.** Jó és megbízható programot készíteni csak tervezéssel, alapos rákészüléssel lehet. Nagyon fontos, hogy egy leendő programozó már az első lépéseket is ilyen szellemben tegye meg.

A rendszerfejlesztéssel, illetve a fejlesztés lépéseivel kapcsolatban már rengeteg elképzelés, módszer született. A szoftverekkel szemben támasztott növekvő igények mindenkorban újabb és újabb szoftverfejlesztési módszereket szülnak.

E könyv célja, hogy megismertesse az Olvasót a szoftverfejlesztés és programkészítés alapjai-
val. Szoftvereink elkészítéséhez a ma széleskörűen elfogadott legmodernebb szoftverfejlesztési
eszközöket használjuk:

- ◆ **UML** (Unified Modeling Language, Egységesített Modellező Nyelv): Grafikus jelölés-
rendszer a szoftver különböző nézeteinek modellezésére. Segítségével tervezni és
dokumentálni tudjuk programjainkat.
- ◆ **Egységesített Eljárás** (Unified Process): Módszertan a fejlesztés módjára vonatkozóan.
- ◆ **Java**: Magas szintű programnyelv programjaink implementálásához (kódolásához).

3.2. Az Egységesített Eljárás

Az UML, majd a hozzá tartozó Egységesített Eljárás annak a három eljárásnak (OMT, Booch és OOSE) az egységesítéseként jött létre 1997-ben, amelyek a világon legelterjedtebb objektumorientált szoftverfejlesztési módszerek voltak. A szoftverfejlesztéssel kapcsolatos alapismereteket erre az eljárásra alapozva mutatjuk meg.

Egy szoftver elkészítésének folyamatát a felmerülő problémától a kész, eladható termék átadá-
sáig **szoftverfejlesztésnek** (software development) nevezzük. Kisebb szoftverek esetén prog-
ramfejlesztésről (program development), összetettebb, a valós folyamatokkal is foglalkozó
szoftverek esetén pedig rendszerfejlesztésről (system development) szokás még beszélni. A
szoftvert a **megrendelő** rendeli meg, a **szoftverfejlesztő** készíti el, és a kész szoftvert a
felhasználó használja. A szoftverfejlesztés olyan projekt (feladat, munka), melyet a szoftver-
fejlesztő végez el a megrendelővel, illetve a felhasználóval folytatott megbeszélés, illetve a
vele kötött szerződés alapján. Amikor a szoftver elkészül, azt a fejlesztő átadja a megrendelő-
nek. A felhasználó a szoftvert a fejlesztő útmutatásai alapján használja.

Megjegyzés: Elképzelhető olyan eset is, hogy a szoftverfejlesztő maga a megrendelő vagy a felhasználó (ha valaki a saját részére készít programot).

A szoftver élete, fejlesztési ciklus, szoftver verzió

A szoftverkészítés ötletétől egy kész szoftver átadásáig a szoftverfejlesztés egy **ciklusáról** beszélünk. A fejlesztési ciklus végén a szoftverfejlesztő átadja a megrendelőnek a szoftver egy verzióját (3.1. ábra). A szoftver átadásával azonban legtöbbször nem fejeződik be a szoftver gondozása, hiszen a felhasználó leginkább a szoftver „éles” használata közben veszi észre a szoftverben rejlő hibákat, kényelmetlenségeket, és természetesen újabb ötletekkel is előállhat: ahogy mondani szokták: „evés közben jön meg az étvágy”. A további törökésnek tehát kétféle oka lehet:

- ◆ **A szoftver hibás:** Egy szoftver hibás, ha az szemantikai (tartalmi) hibákat tartalmaz, vagyis
 - nem a megállapodás szerint működik;
 - nem logikusan működik;
 - a futás alatti hiba következtében a program egyszerűen használhatatlanná válik.
 Vannak alattomos, rejttet hibák –, a számítástechnika zsargonjával kifejezve bugok (bogár), ezek általában a programozó figyelmetlenségből erednek. A hibákat természetesen a szoftverfejlesztő – a szerződésben foglaltak szerint – köteles garanciálisan kijavítani.
- ◆ **A szoftvert tovább kell fejleszteni:** Ha a felhasználó az eredeti megállapodáshoz képest új vagy más funkciókat szeretne beépíttetni a szoftverbe, akkor nyilvánvalóan új szerződés szükséges. Ilyenkor a szoftver továbbfejlesztéséről beszélünk, és a szoftverfejlesztés egy újabb ciklusa kezdődik.



3.1. ábra. A szoftver élete

Egy véglegesen átadott (eladott) szoftver javított, illetve továbbfejlesztett változatait verziószámmal szokás elláttni. Kisebb javítások esetén a verziószám csak egy tizeddel vagy századdal növekszik, nagyobb változtatások esetén a verziószám egész része változik. Erre vonatkozólag általános szabály nincsen, a fejlesztők, illetve a gyártók a verziószámot saját belátásuk szerint adják meg. A JDK (Java Development Kit, Java fejlesztőkészlet) verziószámai például az 1.0, 1.1, ... 1.1.8, 1.2, ..., 1.3 stb. Nagy változtatás esetén új név megadása sincs kizárva.

Aktorok, használati esetek

Aktornak nevezzük a felhasználót (a rendszert használó embert), vagy bármilyen más, a rendszerhez csatlakozó külső egységet, tárgyat (például mérőeszközt vagy modemet). A rendszert az aktorok használják. A használatnak egy értelmes, kerek egységét **használati esetnek** (use case) nevezzük. Az egyes használati eseteknek lehetnek előfeltételei. Például:

- ◆ Egy szövegszerkesztő program aktora bárki, aki szöveget szerkeszt. A program jellegzetes használati esetei:
 - az aktor betölt egy általa kiválasztott szöveget. Előfeltétel, hogy a szöveg létezzék, vagyis betölthető legyen.
 - az aktor kinyomtat egy szöveget. Előfeltétel, hogy legyen betöltve egy szöveg.
- ◆ Egy vállalatirányítási rendszernek aktora többek között a könyvelő vagy az igazgató. A könyvelő könyvel. Az igazgató egészen más dolgokra használja a rendszert: ő olyan információkat kér a rendszertől, amelyek az átlagdolgozó számára elérhetetlenek; ilyen lehet például egy fizetési lista.

Az Egységesített Eljárásban a használati esetek végigkísérlik a rendszert: a fejlesztés elején összegyűjtött használati esetek alapján történik a teljes rendszer fejlesztése, és ugyanezek a használati esetek szolgáltatják a kész rendszer kipróbálásának, tesztelésének az alapját is.

A szoftver architektúrája

A szoftver, illetve a rendszer architektúrájának nevezzük a rendszer szerkezeti vázát és főbb jellemzőit, amelyek meghatározzák a rendszer lényegét. Az architektúra részei a rendszer fő moduljai, a modulok közötti kapcsolatok, valamint a szoftver működésének és fizikai megvalósításának alapkoncepciói, mint például a hardver- és szoftverkörnyezet megválasztása, a hálózati kiépítés stb. A szoftverfejlesztés során az egyik legelső törekvés a rendszer architektúrájának kialakítása.

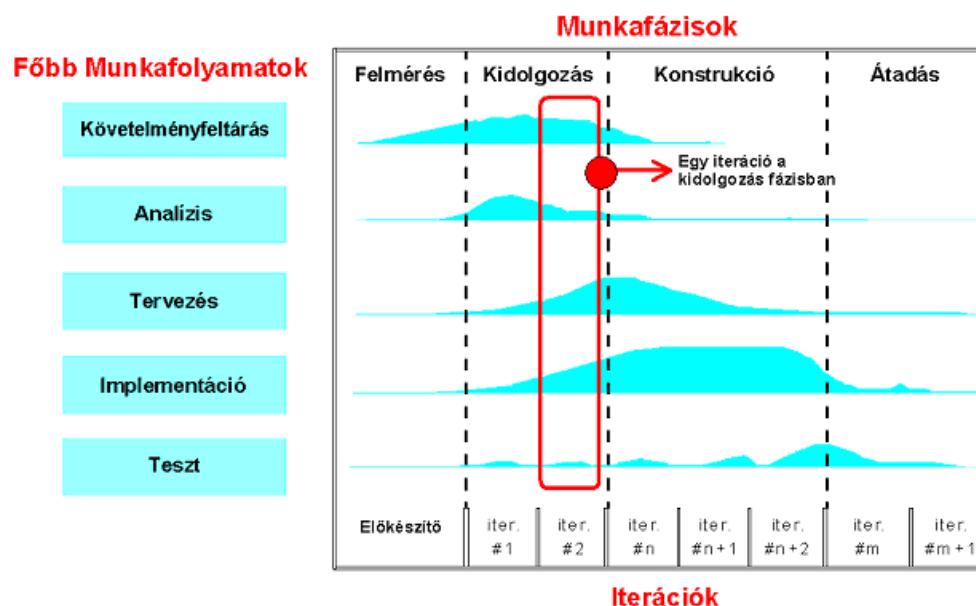
A fejlesztési ciklus munkafázisai

Egy szoftverfejlesztési ciklus alapvetően négy, időben egymást követő munkafázisból áll (3.2. ábra):

- ◆ **Felmérés** (inception): A projekt indítása: a feladat elkezdése, elképzelése. Nagy vonalakban meg kell határozni a feladat nagyságát, fő funkciót, és a szoftver elkészítéséhez

szükséges erőforrásigényeket (pénz, ember, tudás, számítógép, szoftver stb.). Ebben a kezdeti fázisban elsősorban azt kell elhatározni, hogy legyen-e projekt.

- ◆ **Kidolgozás** (elaboration): Ebben a fázisban lényegében kialakítjuk a rendszer architektúráját, és elkészítjük a rendszer tervét. Megtervezzük a rendszerben szereplő modulokat és a modulok közti együttműködéseket. Eldöntjük, hogy a leendő szoftver milyen fizikai egységekre bomlik, és azokat milyen hardver egységekre telepítjük majd.
- ◆ **Konstrukció** (construction): Az előző fázisokban elkészített tervek alapján ekkor következik a rendszer építésének lényegi része. Bár a fázisba jócskán átnyúlik a tervezés is, itt főleg implementációs (kódolási) munka folyik. A fázis végére elkészül egy futó program, a rendszer béta verziója.
- ◆ **Átadás** (transition): Ebben a fázisban a fejlesztők fokozatosan átadják a rendszert a felhasználónak, és szükség esetén betanítják a rendszer használatára. A felhasználó elkezdi használni a rendszert, és közben jelzi a hibákat és a további igényeket. A fejlesztő a hibákat kijavítja, és megállapodás szerint kielégíti azokat a kisebb igényeket, amelyek nem borítják fel a rendszer architektúráját.



3.2. ábra. A fejlesztési ciklus munkafázisai, iterációi és munkafolyamatai

Egy iteráció főbb munkafolyamatai

A fejlesztési ciklus munkafázisai kisebb részfeladatokból állnak (egy ilyen részfeladat lehet például a szoftver általános külalakjának megtervezése). A szoftver fejlesztése iteratív tevékenység, amelynek során a részfeladatok megoldásával egyre közelebb kerültünk a teljes megoldáshoz. Egy részfeladat megoldási folyamatát iterációknak nevezzük. Egy iterációban a következő munkafolyamatokon kell végigmenni (3.2. ábra):

- ◆ **Követelményfeltárás:** A probléma feltárása, a feladattal szemben támasztott alapkötetelmények megfogalmazása. A követelményfeltárás eredménye a feladatspecifikáció. Számos esetben a programozó kész feladatspecifikációt kap – ekkor a feladat megfogalmazója végezte el a követelményfeltárást (például a tanár, aki kiadja a hallgatónak a feladatot). Sok esetben a követelményfeltárás igényli a legnagyobb szakmai gyakorlatot.
- ◆ **Analízis:** A megoldhatóság feltérképezése, elemzése, a feladat pontosítása, irányvonalak, nagyvonalú tervezés megadása. Megoldási eszközök, módszerek kiválasztása.
- ◆ **Tervezés:** A megoldás elvi megadása, körvonalazása. Egy olyan elképzelés dokumentálása, mely már egyértelműen kódolható egy számítógépes nyelv segítségével.
- ◆ **Implementálás:** A forráskód előállítása.
- ◆ **Tesztelés:** A futó szoftver (programkód) kipróbálása tudatosan összeállított tesztadatokkal.

A teljes fejlesztés során képtelenség a különböző munkafolyamatokat időben elhatárolni. Nem tehetjük meg, hogy először minden követelményt feltárunk, majd a pontos feladatspecifikáció alapján analizálunk, majd tervezünk, kódolunk, és végül a teljes rendszert teszteljük. Előfordulhat például, hogy a felmérés során programfejlesztő eszközöket kell kipróbaálnunk, kódolással meg kell győződnünk arról, hogy bizonyos feladatok megoldhatók-e egy adott szoftver segítségével. Sok esetben működő prototípusokat kell készítenünk, hogy a felhasználó el tudja képzelni a leendő szoftvert. A prototípusok a további megbeszélések alapját is képezhetik.

A munkafolyamatok részfeladatonként értendők, és a részfeladat jellegétől függően más-más munkafolyamatok dominálhatnak. Ahogy az a 3.2. ábrán látható dombocskából kiderül, a felmérés során elsősorban követelményeket tárunk fel, de már egy pici kódolás és tesztelés is jelen van. A kidolgozási fázisban az analízis játszik döntő szerepet, de lényegében minden munkafolyamat megtalálható. Elvileg tehát minden fázisban minden csinálhatunk, de

- ◆ a felmérési szakaszban főleg követelményeket tárunk fel;
- ◆ a kidolgozási fázisban főleg analizálunk és tervezünk;
- ◆ a konstrukciós fázisban főleg implementálunk;
- ◆ az átadási fázisban pedig főleg tesztelünk.

Az Egységesített Eljárás főbb jellemzői

Az Egységesített Eljárás főbb jellemzői a következők:

- ◆ **Használatieset-vezérelt:** A rendszer fejlesztésének elején meghatározott használati esetek végigkísérlik a teljes fejlesztést.
- ◆ **Architektúraközpontú:** A teljes fejlesztést meghatározza a rendszer architektúrája.
- ◆ **Iteratív és inkrementális:** A rendszert iteratív módon fejlesztjük. minden egyes részmunka (iteráció) esetén követelményeket tárunk fel, analizálunk, tervezünk, implementálunk és tesztelünk. minden egyes iteráció során a rendszer növekszik, illetve logikailag tiszta.

Megjegyzés: Az Egységesített Eljárásnak további munkafolyamatai is lehetnek, például: üzleti modellezés, menedzsment jellegű munkafolyamatok stb. Kisebb munka esetén a követelményfeltárás és az analízis egybemosódhat.

Kis szoftver (program) fejlesztése

Egy kisebb feladat megoldásához sokszor elegendő egyetlen iteráció:

- ◆ Feltárjuk a követelményeket, ennek során elkészítjük a feladatspecifikációt;
- ◆ Ha a megoldás nem evidens, akkor analizáljuk a feladatot;
- ◆ Megtervezzük a feladatot (egész pici program esetén fejben is lehet);
- ◆ Implementálunk, vagyis lekódoljuk a tervet (a forráskódba megjegyzésekkel teszünk);
- ◆ Teszteljük a futó programot.

Egy iteráció hiba nélkül történő végigvezetéséhez nagy tapasztalatra van szükség. Természetes, hogy a programozó a kódolás közben néha visszanyúl tervéhez, és kijavítja az esetleges hibákat. Az is előfordulhat, hogy tesztelés közben derül ki, hogy nem tárunk fel minden követelményt. Nagyon fontos, hogy minden egyes munkafázis dokumentálva legyen, hogy adott esetben könnyen javíthatssuk, illetve továbbfejleszthessük a programot.

Könyünkben mi kizárolag kis szoftvereket fogunk fejleszteni. A főbb munkafolyamatokat a következő pontokban egyenként, részletesebben tárgyaljuk.

Szoftverfejlesztés: A szoftver elkészítésének folyamata a felmerülő ötlettől a kész, eladható termék átadásáig.

Szoftverfejlesztő: Aki elkészíti a szoftvert.

Felhasználó: A szoftvert használó személy.

Aktor: Aki vagy ami a szoftvert használja. Felhasználó, vagy egy rendszerhez csatlakozó hardveregység.

Használati eset: A használatnak egy értelmes, kerek egysége.

A szoftver élete, fejlesztési ciklus, szoftververzió: A szoftver élete fejlesztési ciklusokból áll. Egy fejlesztési ciklus végén átadnak egy verziót. Az átadás után újabb fejlesztési ciklus következhet.

Munkafázis: Egy fejlesztési ciklus időben jól elhatárolható szakasza. A ciklus négy, egymást követő munkafázisból áll: felmérés, kidolgozás, konstrukció és átadás.

Munkafolyamat: Egy részfeladat megoldását célzó munka. Alapvető munkafolyamatok: követelményfeltárás, analízis, tervezés, kódolás és tesztelés.

Iteráció: A szoftverfejlesztés részfeladatok megoldásainak iterációja. Egy részfeladat fejlesztése munkafolyamatokon megy keresztül.

3.3. Követelményfeltárás

Amikor felmerül egy probléma, sokszor nem is tudjuk pontosan, mi a feladat, azt pedig különösen nem tudjuk, hogy elvégzése mennyi munkával fog járni, és egyáltalán meg tudjuk-e oldani. A munkát azzal kezdjük, hogy összegyűjtjük, feltárnak a feladattal kapcsolatos követelményeket, vagyis megfogalmazzuk azokat a fontos dolgokat, amelyeket mindenki meg akarunk valósítani. Ebben a munkafolyamatban együtt kell működni a szoftver megrendelőjével, azzal a szakemberrel, aki a szoftvert használni fogja majd. A követelmények gyűjteményét követelményspecifikációnak vagy feladatspecifikációnak nevezzük. Egy jó feladatspecifikáció teljes, érthető, egyértelmű, pontos, tömör, szemléletes, előrelátó, jól felépített és tagolt.

Követelményfeltárás: A feladattal szemben támasztott követelmények összegyűjtése. A követelményfeltárás dokumentációja a **követelményspecifikáció (feladatspecifikáció)**, mely egyértelműen leírja a feladattal szemben támasztott követelményeket.

3.4. Analízis

Az analízis során meg kell vizsgálnunk, hogy a feladat megoldható-e, meg kell becsülnünk a feladat nagyságát, a szükséges erőforrásokat (pénz, ember, idő, hardver, szoftver stb.). A megoldáshoz irányvonalaikat kell kijelölni, eszközöket, módszereket kell választani. A szoftver megrendelőjével, használójával ebben a munkafázisban is együtt kell működni. Az analízis során készítünk egy nagyvonalú megoldási tervet, egy ún. **szakterületi objektummodellt** (adatfeldolgozási feladat esetén ez lényegében egy adatmodell), melyet a számítógéphez nem értő szakember is ért – lehetőség szerint kerüljük a számítógéppel kapcsolatos szakkifejezéseket. Ebben a munkafolyamatban össze kell gyűjteni a fontosabb **használati eseteket**, vagyis el-

kell dönten, mire akarjuk használni ezt a programot vagy programrészt (modult). minden egyes használati esetnél át kell gondolni, milyen adatokat vár a program, és milyen adatokat kell produkálnia. Csak így válhat világossá, érthetővé a feladat.

Az analízis dokumentációja a szakterületi modell és a használati esetek.

Analízis: A megoldhatóság és az erőforrásigény feltérképezése; a feladat elemzése, pontosítása; irányvonalak, nagyvonalú tervezés megadása; megoldási eszközök, módszerek kiválasztása. Az analízis dokumentációja a **szakterületi objektummodell** és a **használati esetek**.

3.5. Tervezés

A programtervezés feladata, hogy az analízis során kialakított elképzéléseket továbbfejlesztve, az ott fölvázolt tervet részletesen kidolgozza. A szakterületi modellből programterv lesz, amely már tartalmaz számítógéppel kapcsolatos elemeket is, és amelyet már csak a tapasztalattal rendelkező szoftverfejlesztő képes átlátni. A tervezési szakasz dokumentációja a **programterv**. A programtervet általában valamilyen grafikus jelölésrendszer alkalmazásával készítjük el. Mi az UML diagramjait (osztálydiagramot, együttműködési diagramot) fogjuk alkalmazni egyszerű programjaink megtervezéséhez. Egy jó programterv alapján egyértelműen el lehet készíteni a forráskódot.

A program megtervezésével a probléma lényegileg meg van oldva. Nagyon fontos, hogy a terv minél tökéletesebb legyen, mert a hibák javítása a későbbi munkák során egyre drágább.

Tervezés: A feladat lényegi megoldása, egy olyan dokumentáció előállítása, amelynek alapján a forráskód már elkészíthető. A tervezési szakasz dokumentációja a **programterv**.

3.6. Implementálás (kódolás)

Ha a terv elkészült, akkor jöhet annak **implementálása** (kivitelezése). Az implementáció a programterv kódolása egy adott programnyelven, vagyis a **forrásprogram** elkészítése. Ez a szakasz egy rutinos programozó számára elég mechanikus, hiszen az előző szakaszokban a feladat logikai megoldása elkészült. A programnyelv, amelyen kódolunk, sokféle lehet – mi ebben a könyvben a Javát fogjuk használni. A számítógépes program megírásához jól kell ismerni az adott programnyelvet, hiszen most csak olyan instrukciókat adhatunk, amelyeket a fordító ismer. A kódolási szakasz dokumentációja a megjegyzésekkel ellátott **forrásprogram**.

Egy forrásprogram akkor jó, ha az

- ◆ pontosan a feladatspecifikáció, illetve a programterv alapján készül;
- ◆ áttekinthető, olvasható;

- ◆ tömör és egyértelmű megjegyzésekkel van megtüzdelve;
- ◆ nem tartalmaz sem formai (szintaktikai), sem logikai (szemantikai) hibát.

Kódolás: Forráskód (forrásprogram) elkészítése a programterv alapján.

3.7. Tesztelés

A forrásprogram begépelésével természetesen még nem vagyunk készen, hiszen bármilyen gondosan tervezünk, és bármilyen jól gépeltünk is, a kód még tele lehet hibákkal. A tesztelés a program tudatos ellenőrzése, figyelése hibakeresés céljából.

A forráskódban kétféle hiba lehetséges:

- ◆ **Szintaktikai (formai) hiba:** A forráskód része, melyet a fordítóprogram nem tud értelmezni. Ez lehet elgépeléstől, vagy fakadhat abból, hogy nem ismerjük eléggyé a programnyelvet.
- ◆ **Szemantikai (logikai, tartalmi) hiba:** A program nem logikusan vagy nem a leírás szerint működik, vagy egyszerűen használhatatlanná válik.

A szintaktikai hibákat a fordítóprogram segítségével szűrjük ki. A szemantikai hiba sokkal kellemetlenebb, mert ekkor a probléma mélyen gyökerezhet – az is elképzelhető, hogy a hibát a probléma analizálásának fázisától kezdve hurcoljuk, ez azonban csak felületes munka esetén következik be. Természetesen minél alaposabbak voltak az előző fázisok, annál nagyobb a valószínűsége, hogy a programrész előbb vagy utóbb úgy fog működni, ahogyan szeretnénk.

A program **tesztelésének** külön tudománya van – nem is egyszerű dolog előre feltárni a majd bekövetkezhető szélsőséges eseteket, és kezelní őket. A tesztelés folyamán különböző próbatudatokkal futtatjuk a programot. A **tesztadatokat** úgy kell összeállítani, hogy azok minden lehetséges esetet lefedjenek

Egy programot szárazon is lehet tesztelni: ilyenkor a forráskódot böngészve az elköpzelt környezetben és adatokkal „fejben futtatjuk” a programot. Az ilyen tesztet **száraztesztnek** nevezzük. Ugyanígy ajánlatos a programtervet és más dokumentációkat is tesztelni, mielőtt azokat késznek nyilvánítjuk.

A program tesztelésekor a következőkre kell figyelni:

- ◆ Pontosan úgy működik a program, ahogy az a feladat leírásában szerepel?
- ◆ Nem lehet elrontani?
- ◆ Elég hatékony?
- ◆ Biztonságos a használata?
- ◆ Felhasználóbarát? (Teljes mértékben szolgálja a program a felhasználót? Szép a program? Sehol sem idegesítő?)

A tesztelési fázis dokumentációja a **kész program** és a **tesztadatok**.

Tesztelés: A program működésének ellenőrzése tudatosan összeállított tesztadatokkal.

Szintaktikai hiba: Forráskódrészlet, melyet a fordító nem tud értelmezni.

Szemantikai hiba: A program nem logikusan vagy nem a leírás szerint működik.

Szárazteszt: A program működésének ellenőrzése „fejben”, a program futása nélkül.

3.8. Dokumentálás

Ahogy az egyes munkafolyamatok leírásában láthattuk, minden munkafolyamatnak megvan a maga „terméke”, dokumentációja. Nagyobb szoftverfejlesztési munkák esetén az egyes munkafázisok végén készítenek egy összefoglaló, több modellből álló dokumentációt. A dokumentációt meg kell őrizni, mert csak annak birtokában lehet később a programon változtatni.

Lényegében kétféle dokumentációról szokás beszélni:

- ◆ **Fejlesztői dokumentáció:** A program fejlesztését végigkísérő és a megoldást definiáló dokumentációk összességét fejlesztői dokumentációknak nevezzük.

A fejlesztői dokumentáció részei:

- Analízismodell (feladatspecifikáció, szakterületi modell, használati esetek)
- Tervezési modell (programterv, képernyőtervezek, listatervezek)
- Implementációs modell (forrásprogram)
- Telepítési modell (megmondja, hogy milyen programot milyen gépre telepítünk)
- Tesztelési modell (tesztadatok, teszteredmények)

Telepítési és tesztelési modellt általában csak nagyobb szoftverek esetén adnak meg.

- ◆ **Felhasználói dokumentáció:** A kész rendszerhez tartozik még egy dokumentáció, melyet a felhasználó kap meg, és amely a program használatával kapcsolatos tudnivalókat tartalmazza. A felhasználói dokumentáció részei:

- A feladat leírása (kiknek a részére, milyen célból készült a program, a használati esetek megadása)
- Szükséges hardver- és szoftverkörnyezet (a számítógép típusa, a minimálisan szükséges konfiguráció; az operációs rendszer, a futtatáshoz szükséges kiegészítő szoftverek)
- A program betöltése, indítása
- A program általános használatának leírása: billentyűk használata, működési leírás minden részletre kitérően, a segítségkérés lehetőségei
- A program menüfunkcióinak részletes leírása
- Képernyőképek, listáképek

- Hibaüzenetek felsorolása, útmutatás, teendők
- Biztonsági előírások (pl. adatok időszakos mentése)

Dokumentálás: A szoftver fejlesztésének, illetve működésének leírása megőrzési és megértési célból.

- **Fejlesztői dokumentáció:** A fejlesztőknek szóló dokumentáció, mely az egyes fejlesztési szakaszok eredményeit rögzíti.
- **Felhasználói dokumentáció:** A felhasználóknak szóló dokumentáció, mely útmutatást ad a szoftver használatára.

Tesztkérdések

- 3.1. Jelölje meg az összes igaz állítást a következők közül!
 - a) Az UML szoftverfejlesztési módszertan.
 - b) Az Egységesített Eljárás szoftverfejlesztési módszertan.
 - c) Egy fejlesztési ciklus alatt a szoftvernek több verzióját is átadják.
 - d) A fejlesztési ciklus munkafázisai: felmérés, kidolgozás, konstrukció és átadás
- 3.2. Jelölje meg az összes igaz állítást a következők közül!
 - a) Egy iteráció munkafolyamatai: követelményfeltárás, analízis, tervezés, kódolás és tesztelés.
 - b) A szoftver architektúrája a programterv.
 - c) A szoftverfejlesztés kezdeti fázisa a felmérés.
 - d) Az aktor az, aki teszteli a programot.
- 3.3. Jelölje meg az összes igaz állítást a következők közül!
 - a) A használati eset a program használatának egy értelmes egysége.
 - b) Egy iteráció több munkafázisból áll.
 - c) Egy munkafázis több iterációból áll.
 - d) A felmérési fázisban több analízis van, mint tervezés.
- 3.4. Jelölje meg az összes igaz állítást a következők közül!
 - a) A „használatieset-vezérelt” kifejezés azt jelenti, hogy a használati esetek végigkísérik a szoftver fejlesztését.
 - b) A szemantikai hiba a fordítóprogram által kiszűrhető hiba.
 - c) A szakterületi modell a követelményfeltárás munkafolyamat dokumentációja.
 - d) A tesztelés a program működésének ellenőrzése hibakeresés céljából.
- 3.5. Jelölje meg az összes igaz állítást a következők közül!
 - a) Az implementálás dokumentációja a programterv.
 - b) A forráskódot a programterv alapján kell előállítani.
 - c) A szoftver elkészítése után a fejlesztői dokumentációt meg kell őrizni.
 - d) A fejlesztői dokumentációk az egyes fejlesztési szakaszok eredményeinek leírásai.

I. BEVEZETÉS A PROGRAMOZÁSBA

1. A számítógép és a szoftver
2. Adat, algoritmus
3. A szoftver fejlesztése



II. OBJEKTUMORIENTÁLT PARADIGMA

4. Mitől objektumorientált egy program?
5. Objektum, osztály
6. Társítási kapcsolatok
7. Öröklődés
8. Egyszerű OO terv – Esettanulmány

II.

III. JAVA KÖRNYEZET

9. Fejlesztési környezet – Első programunk
10. A Java nyelvről

IV. JAVA PROGRAMOZÁSI ALAPOK

11. Alapfogalmak
12. Kifejezések, értékadás
13. Szelekciók
14. Iterációk
15. Metódusok írása

V. OSZTÁLYOK HASZNÁLATA, KÉSZÍTÉSE

16. Objektumok, karakterláncok, csomagolók
17. Osztály készítése

VI. KONTÉNEREK

18. Tömbök
19. Rendezés, keresés, karbantartás
20. A Vector és a Collections osztály

FÜGGELÉK

- A tesztkérdések megoldásai
Irodalomjegyzék
Tárgymutató

4. Mitől objektumorientált egy program?

A fejezet pontjai:

1. A valós világ modellezése
 2. Az objektumorientált program főbb jellemzői
-

Az objektumorientált módszertanok – amennyire lehet – megpróbálják becsempészni a természetes emberi gondolkodás szabályait a szoftverkészítés menetébe. A következő fejezet először áttekintést ad arról, milyen gondolatok vezérlik az embert a valós világ modellezésekor, majd egy, az életből merített példa segítségével megpróbálja érzékeltetni az objektumorientált rendszer főbb jellemzőit.

4.1. A valós világ modellezése

Az objektumorientált gondolkodásmód nagyon hasonlít a természetes emberi gondolkodáshoz. Ahhoz, hogy ezt beláthassuk, vizsgáljuk meg az emberi gondolkodás alapvető jellemzőit. Az ember a körülötte lévő tárgyakat (objektumokat) észreveszi, leegyszerűsíti, megkülönbözteti és rendszerezi – így tudja ezt a végtelenül bonyolult világot a maga számára többé-kevésbé érthetővé tenni. Az embert a valós világ modellezésekor a következő gondolatok vezérlik:

◆ **Absztrakció**

A valós világ végtelen és bonyolult. Minél mélyebbre hatolunk az elemzésben, annál bonyolultabb. A lebontásban tehát valahol meg kell állnunk, illetve szelektálnunk kell.

Az absztrakció a valós világ leegyszerűsítése a lényegre koncentrálás érdekében.

Az egyes objektumoknak csak azon tulajdonságait és viselkedésmódját vesszük figyelembe, melyek célunk elérése érdekében feltétlenül szükségesek. Egy embernek például mérhetetlen sok tulajdonsága van, és rengeteg dologra képes – mégis, amikor az utcán meglátunk egy embert, adott esetben csak arra figyelünk, hol tartózkodik, és hogy éppen mozog-e. Talán eszünkbe sem jut, hogy milyen a szeme színe és mi van a zsebében.

◆ **Megkülönböztetés**

Rengeteg objektum van körülöttünk, s ezek mind-mind külön létező, konkrét, egyedi objektumok. **Az objektumokat a számunkra lényeges tulajdonságok alapján meg-**

különböztetjük (magas ember – alacsony ember; piros labda – kék labda; az én piros labdám – a te piros labdád). Előfordulhat persze, hogy két objektum között képtelenek vagyunk különbséget tenni – például két hangya között, de ez csak a mi gyengeségünk, attól még a hangyák nem ugyanazok.

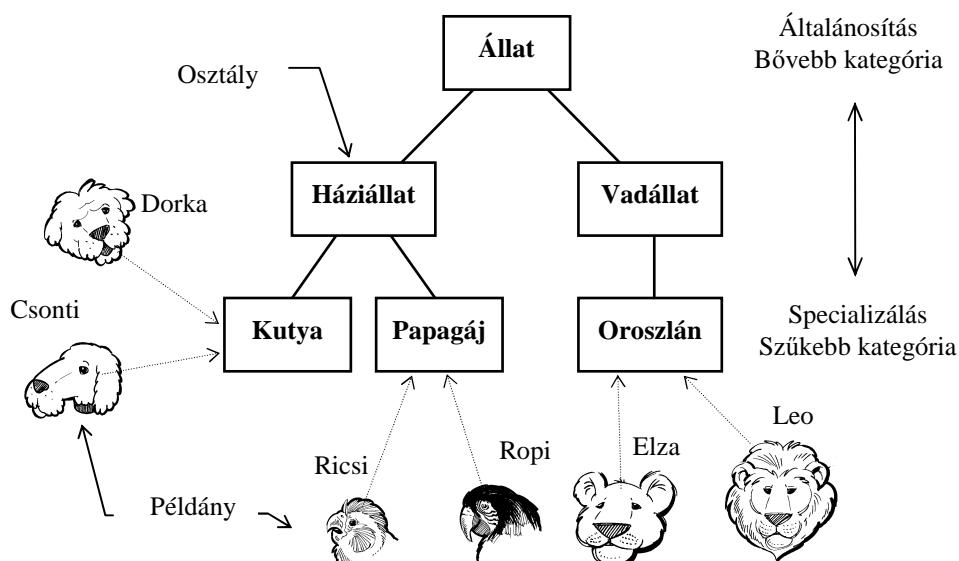
♦ Osztályozás

Az ember az objektumokat automatikusan rendszerezi, s azokat a számára fontos tulajdonságok alapján kategóriákba, osztályokba sorolja: „Az az objektum ugyanolyan, mint az, tehát ugyanahhoz az osztályhoz tartozik.” (Lásd 4.1. ábra.) Az egy osztályba sorolható objektumokat az osztály példányainak szokás nevezni:
 „Dorka és Csonti kutya” – Dorka és Csonti a Kutya osztály példányai;
 „Ricsi és Ropi papagáj” – Ricsi és Ropi a Papagáj osztály példányai.

♦ Általánosítás, specializálás

Az ember az objektumok között állandóan keresi a hasonlatosságokat és különbözőségeket, ezek alapján **bővebb, illetve szűkebb kategóriákat állít fel** (lásd 4.1. ábra):

- „Az az objektum olyan, mint az, de rá ez is jellemző ... ”,
- „Az az objektum egy ..., de ... ”,
- „A kutya háziállat, de olyan, amelynek négy lába van.”,
- „Az oroszlán vadállat, de van sörénye, és nagyon aranyos.”
- „A kutya, a papagáj és az oroszlán mind állat.”
- „Elza állat.”



4.1. ábra. Objektumok osztályba sorolása

◆ **Kapcsolatok felépítése, részekre bontás**

Az ember az objektumok között állandóan keresi a kapcsolatokat:

- „A sütemény alkotóelemei a következők: 50 dkg liszt, 10 tojás ...”
- „Jancsi és Juliska ismeri egymást.”
- „A kutyának van feje, teste, négy lába és egy farka.”
- „A kutya őrzi a házat.”

Alapvetően két fajta kapcsolat létezik:

- ismeretségi, más néven használati kapcsolat;
- tartalmazási, más néven egész–rész kapcsolat.

Ismeretségi kapcsolat akkor áll fenn két objektum között, ha azok egymástól függetlenül is létezhetnek, vagyis egyik léte sem függ a másiktól. Ilyen a kutya és a ház, hiszen ha a kutya elpusztul, attól még a ház megmarad, legfeljebb nem őrzi azt senki. Ha pedig a ház pusztul el, akkor a kutya egy új házat fog őrizni.

Egész–rész kapcsolatról akkor beszélünk, ha az egyik objektum határozottan része a másiknak, mégpedig úgy, hogy ha az egész objektum elpusztul, akkor pusztul vele a rész is. Ilyen a kutya és annak négy lába.

A kapcsolat jellegének megállapítása természetesen nem minden esetben egyszerű. Milyen kapcsolatban van például az autó és a kereke? A roncstelepre szállított autó kerekét, ha kell, nyugodtan átszerelhetjük egy új autóra.

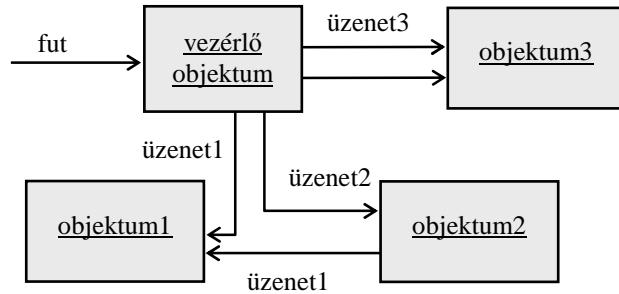
Az objektumorientált paradigma az emberi gondolkodást igyekezik utánozni; erről az Olvasó maga is meggyőződhet, hiszen e paradigma tárgyalásakor ugyanezekkel a fogalmakkal fog majd találkozni.

4.2. Az objektumorientált program főbb jellemzői

Ahhoz, hogy egy program objektumorientált legyen, nem elég az OO nyelv, OO szellem is szükséges!

Egy objektumorientált (OO) program egymással kommunikáló objektumok összessége, melyben minden objektumnak megvan a jól meghatározott feladatköre (4.2. ábra). Egy objektum szolgáltatásokat ad, mégpedig olyan szolgáltatásokat, amelyekre őt beprogramozták. A képernyön levő ablak objektumot például beprogramozták arra, hogy menjen odébb valamekkora távolsággal – ezért, ha ezt a szolgáltatást kérjük tőle, megteszí. Ahhoz, hogy egy objektum végrehajthassa feladatát, bizonyos adatokat meg kell jegyeznie. Ezek az adatok legtöbbször az objektum privát adatai, azokat a szolgáltatást kérő objektum nem ismeri. Egy embernek is mondhatjuk, hogy menjen arrébb egy méterrel anélkül, hogy tudnánk, hol van. A szolgáltatást kérőnek csak tudnia kell, mire lehet egy objektumot megkérni, valamint meg kell tudni szólítania az objektumot. A feladat végrehajtásának „hogyan”-ja az objektum belügye. minden objektumnak van egy interfész része, amelyen keresztül kommunikálni lehet vele. Ha az ember szóból ért, akkor az ember interfésznek része a fül, és ha az ember be van programozva a

„megy” szolgáltatásra, akkor az ember úgy mozgatja a lábait, ahogy azt belétáplálták, ebbe a szolgáltatást kérő már nem avatkozhat bele.



4.2. ábra. Objektumorientált rendszer

Egy OO rendszer tervezésekor a feladatspecifikáció alapján meg kell határozni a rendszerben szereplő objektumokat, és az objektumok között szét kell osztani a feladatokat. A program általában egy kiválasztott (vezérlő) objektum „megszólításával” indul, ez az objektum felelős az egész program végrehajtásáért (a 4.2. ábrán a `fut` üzenettel indul a program). Egy objektum aztán megkérhet más objektumokat bizonyos részfeladatok elvégzésére – ilyenkor az objektum delegálja feladatának, illetve felelősségenek egy részét. A rendszerben minden objektumnak megvan a jól meghatározott feladatköre. Egy objektum felelős feladatának elvégzéséért, akkor is, ha bizonyos feladatokat „mással csináltat meg”. A 4.2. ábrán a szürke téglalapok az objektumokat, a nyilak a kérelmet üzeneteket jelölik. A kérelmet üzeneteket is szokás nevezni (sőt, ez az elterjedtebb neve): az egyik objektum üzen a másiknak, hogy végezze el a megadott feladatot.

Objektum: Az objektum információt tárol, és kérésre feladatokat hajt végre. Az objektum felelős feladatainak korrekt elvégzéséért. Az objektum logikailag összetartozó adatok és rajtuk dolgozó algoritmusok (rutin, metódus, programkód) összessége.

Objektumorientált program: Egy objektumorientált program egymással kommunikáló objektumok összessége, melyben minden objektumnak megvan a jól meghatározott feladatköre (felelősségi köre).

Készül a húsleves

Néhány objektumorientált fogalom előzetes megvilágítására most vegyük az életből egy példát (4.3. ábra). Adva van egy család: Laci, Erzsébet, és két főzni tudó leánygyermek: Zsófi és Kati. Lacinak ma ebédre húslevest kell készítenie, megkérte erre a nagymama:

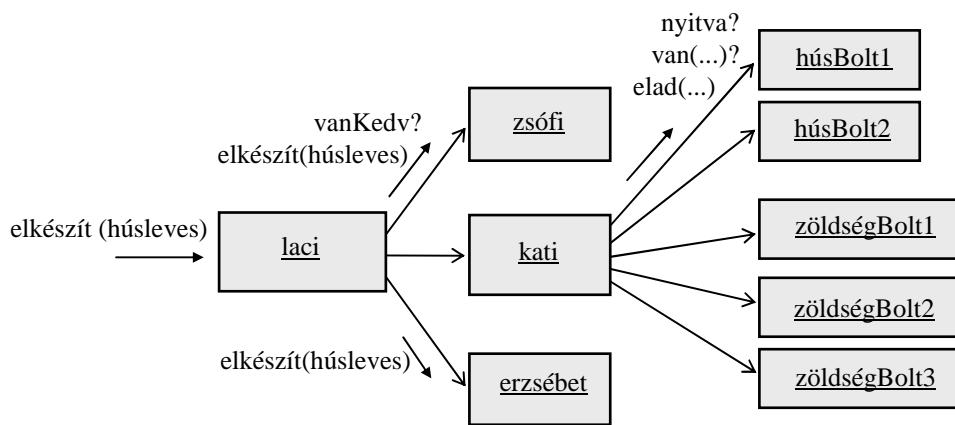
```
laci.elkészít(húsleves)
```

Üzenetküldéskor a megszólított objektum és az üzenet neve közé pontot teszünk. Az üzenetek lehetnek paraméterei. Az üzenet nem más, mint egy az objektumba beprogramozott rutin (eljárás vagy függvény) hívása. Az OO paradigmában a rutint metódusnak nevezük.

Egy objektum csak akkor küldhet üzenetet egy másik objektumnak, ha azzal kapcsolatban áll, vagyis ismeri vagy tartalmazza az üzenetet fogadó objektumot. Példánkban Laci ismeri a három hölgyet, a hölgyek pedig ismerik a boltokat (a hölgyeknek nem kell ismerniük Lacit, és lehet, hogy Laci nem is tudja, mi az a bolt).

Megjegyzések:

- Az objektum nevét megállapodás szerint kis kezdőbetűvel írjuk, a név tagolásához már használhatók nagybetűk is.
- A példa egy régimódi családképet sugall. Kérem az Olvasót, tekintsen el most ettől. A szerzőt semmilyen ironikus gondolat nem vezette, a példa egyszerűen így sikerült.



4.3. ábra. Készül a húsleves

Példánkban a megszólított objektum a `laci`, az üzenet az `elkészít`, melynek van egy paramétere, a `húsleves`. Az `elkészít` üzenetre Laci csak reagál, ha az `elkészít` algoritmus belé van programozva. Laciba szerencsére ez a rutin be van programozva, de úgy, hogy a feladatot továbbadja (delegálja): sorban megkérdezi családjá három hölgytagját, van-e kedvük húslevest készíteni. Aki először válaszol igennel, az fogja elkészíteni a húslevest. Ha Zsófi és Kati egyaránt nemmel válaszolnak, akkor Erzsébet készíti el, húsleves tehát mindenkihez lesz. Laci `elkészít` rutinja a következőképpen fest:

```

elkészít(húsleves)
  if zsófi.vanKedv()
    zsófi.elkészít(húsleves)
  else if kati.vanKedv()
    kati.elkészít(húsleves)
  else
    erzsébet.elkészít(húsleves)
  end if
end elkészít

```

Az `elkészít` üzenetre Zsófi másképpen reagál, mint Laci. Hogy egyáltalán reagál-e, azt Laci-nak tudnia kell, mert egyébként a program kudarcra van ítélezve. Ez az életben csak annyit jelentene, hogy nincs húsleves, de egy programban a „süket fülek” fordítási vagy futási hibát eredményeznek (a Javában ez fordítási hiba). A `zsófi.vanKedv` szintén kérelem, és nem más, mint a megszólított objektum pillanatnyi állapotának lekérdezése.

Üzenet (kérelem): Az objektumokat üzeneteken keresztül kérjük meg különböző feladatok elvégzésére. Az üzenet nem más, mint egy az objektumba beprogramozott rutin hívása. Egy objektum csak akkor küldhet üzenetet egy másik objektumnak, ha azzal kapcsolatban áll, vagyis ismeri vagy tartalmazza ezt a másik objektumot.

Az ebéd elkészítése a hölgyek esetén három részfeladatból áll:

```

elkészít(húsleves)
  vásárol(...)
  főz()
  mosogat()
end elkészít

```

Tegyük fel, hogy a részfeladatok közül a főzéshez és a mosogatáshoz nem kell más objektum-tól segítséget kérni, a vásárláshoz viszont különböző boltokhoz kell fordulni.

A hölgyek mindenhol egyformán csinálnak, kivéve a főzést. Zsófi átlagosan főz, Erzsébetnek nagy gyakorlata van, Ő egy kicsit jobban főz, mint az átlag, Kati elszózza az ételt. A három hölgy közül valamelyik mindenkorban megkapja a feladatot: tegyük fel, hogy Zsófinak most nincs kedve, Katinak pedig van (így aztán sós lesz a húsleves).

Katinak először is be kell szereznie a húst és a zöldségeket. A környéken két húsbolt és három zöldséges található, reméljük, ezekben kaphatók a szükséges kellékek. Bemegy tehát Kati a `húsBolt1` azonosítójú boltba. Tegyük fel, hogy rögtön kap szép marhahúst. Ezután megnézi zöldségBolt1-et, de az zárva van, ezért elmegy zöldségBolt2-be. Ott megkapja a hozzávalók felét, zöldségBolt3-ban pedig megkapja a még hiányzó dolgokat. A boltokban így vásárol (részletek):

```

if húsBolt1.nyitva() és húsBolt1.van(marhahús)
  húsBolt1.elad(marhahús,1.5 kg)
...
if zöldségBolt3.nyitva() és zöldségBolt3.van(sárgarépa)
  zöldségBolt3.elad(sárgarépa,1 kg)
...

```

Végül hazamegy a hozzávalókkal, főz és mosogat.

A rendszer objektumait és az objektumok közötti üzeneteket a 4.3. ábrán látható **együttműködési diagram** tünteti föl. Az UML diagram neve azt fejezi ki, hogy a feladat elvégzése érdekében az objektumok együttműködnek. Figyeljük meg, hogy a rendszerben minden objektumnak megvan a jól meghatározott feladata. minden objektum felelős saját feladatainak elvégzéséért:

A férfi feladata:

- ◆ Elkészíti a húslevest (a módszer nem számít, a lényeg az, hogy legyen húsleves)

A hölgyek feladatai a következők:

- ◆ Megmondja, van-e kedve
- ◆ Elkészíti a húslevest
- ◆ Vásárol
- ◆ Főz
- ◆ Mosogat

Ezek közül a feladatok közül a hölgyeket csak kettőre kérjük meg (vanKedv és elkészít). A többi feladat – vásárol, főz, mosogat – az elkészít feladat automatikus velejárója, az objektum belügye.

Egy húsbolt feladatai:

- ◆ Megmondja, nyitva van-e
- ◆ Megmondja, van-e hús
- ◆ Eladja a húst

Egy zöldségbolt feladatai:

- ◆ Megmondja, nyitva van-e
- ◆ Megmondja, van-e zöldség
- ◆ Eladja a zöldséget

Felelősség: minden objektumnak megvan a jól meghatározott felelősségi köre. Az objektum felelős feladatai elvégzéséért.

Az objektumok zártak; ez azt jelenti, hogy a más objektumok számára lényegtelen információkat elrejtik. minden objektum a saját dolgával töröklik, s amiért felelős, azt az erejéhez képest a legjobban próbálja végrehajtani. Egyik objektum sem szólhat bele a másik dolgába. Laci nem szól Katira, hogy ne melegvízben mossa a húst, és Kati sem szól a zöldségesnek, hogy tartsa tisztán a rekeszeket. Kati csak kér, kap és fizet.

Bezárás, információ elrejtése: A feladatok elvégzésének „hogyan”-ja az objektum belügye. Az objektum belseje sérthetetlen. Az objektummal csak az interfészen keresztül lehet kommunikálni.

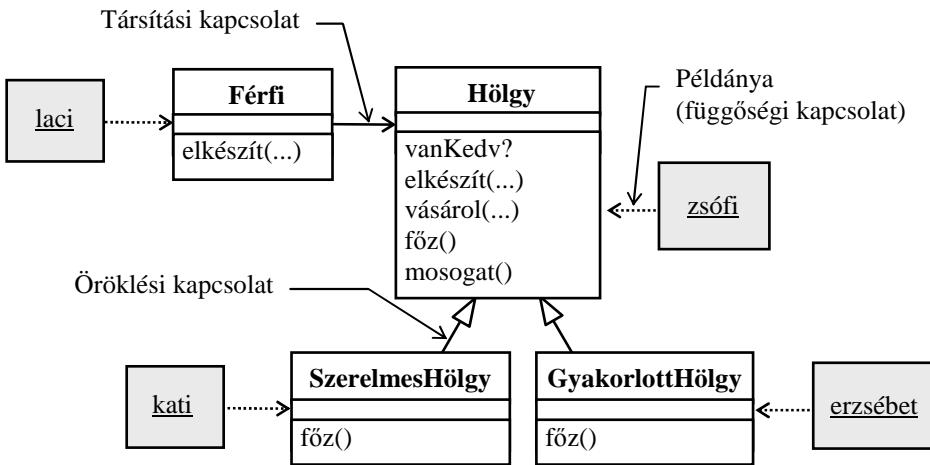
Az elkészít operáció négy objektumba is be van programozva. Mind a négyen reagálnak tehát erre az üzenetre, csakhogy nem egyformán. Laci egészen másképp reagál, mint a három hölgy, de a hölgyek sem teljesen egyformán, hiszen a főz operáció mindegyikükbe másképp van beprogramozva. Az üzenetet küldőnek nem kell tudnia, hogy mi lesz a megszólított objektum reakciója, neki csak azt kell tudnia, megkérheti-e őt egyáltalán az üzenetben foglaltak végrehajtására.

Polimorfizmus (többalakúság): Ugyanarra a kérelemre a különböző objektumok különbözőképpen reagálhatnak.

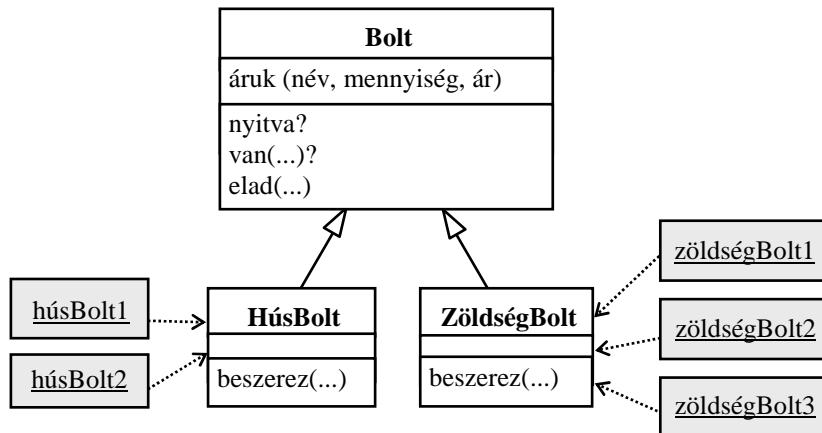
Próbáljuk osztályokba sorolni a program objektumait (hogy az „egyforma” objektumokat csak egyszer kelljen beprogramozni). laci, erzsébet, zsófi és kati mind ember ugyan, de laci nagyon kilóg a sorból. A feladatok szempontjából semmi közös nincs benne és a három hölgyben. A három hölgy azonban nagyon hasonlít egymásra, mindegyik tud húslevest készíteni, vásárolni, főzni stb. De nem teljesen egyformák, mert másképp főznek, ezért végül is mindegyikük más osztályhoz tartozik. A 4.4. ábrán látható módon a húsleveskészítőket négy osztályba soroltuk – van egy Férfi osztály és háromféle Hölgy osztály: (átlagos) Hölgy, Szerelmes Hölgy és GyakorlottHölgy (ellentétben az objektumokkal, az osztályokat nagy kezdőbetűvel írjuk). Mindhárom osztályhoz elvileg akárhány példány tartozhat, példánkban mindegyikból pontosan egy van. A két húsbolt ugyanabba az osztályba sorolható, hiszen azok hasonlóképpen viselkednek (mindegyik ad húst, ha nyitva van és van hús). A három zöldségbolt a ZöldségBolt osztályhoz tartozik. A kétféle bolt között az a különbség, hogy az egyik húst tart, illetve szerez be, a másik pedig zöldséget (4.5. ábra).

Osztályozás: Az objektumpéldányokat viselkedésük szerint osztályokba soroljuk.

Az osztályok egymás alá, illetve fölé rendelhetők. A szerelmes hölgy és a gyakorlott hölgy minden ugyanúgy csinál, mint az átlagos hölgy, csak a főz operáció van másképp beléjük programozva. „A szerelmes hölgy is hölgy, csak elsőzza az ételt.” Az öröklés tehát logikai és kódolási könnyebbseg. Amikor a szerelmes hölgyet „beprogramozzuk”, nem kell minden előlről megadnunk, hiszen ő majdnem minden ugyanúgy csinál, mint egy átlagos hölgy. A SzerelmesHölgy öröklí a Hölgy összes tulajdonságát és viselkedési normáját, és ilyenkor nem kell másat csinálni, mint megadni a viselkedésbeli különbségeket – megadhatunk újabb viselkedéseket, régieket pedig átdefiniálhatunk, illetve letakarhatunk. A szerelmes hölgy a hölgy egy specializációja: „A SzerelmesHölgy is Hölgy.”. Azt mondjuk, hogy a SzerelmesHölgy osztálya a Hölgy osztályából származik. A Hölgy az ősosztály, a SzerelmesHölgy pedig az utód, vagy más szavakkal: kiterjesztett, leszármazott vagy származtatott osztály (4.4. ábra). Ahogy az ábrán is látható, az ősosztály öröklési kapcsolatban áll az utódosztállyal (ezt üres fejű nyíllal jelöljük), az egyes objektumok pedig függési viszonyban állnak a maguk osztályával (ezt a szaggatott nyíl jelzi).



4.4. ábra. A „húsleveskészítők” osztályai



4.5. ábra. A boltok osztályai

Most nézzük meg a boltok osztályozását (4.5. ábra). A `Bolt` osztálynak két utódosztálya van: a `HúsBolt` és a `ZöldségBolt`. A két bolt abban különbözik egymástól, hogy azok más képp szerzik be az árat. Példánkban az ősosztályból nincsen példány, az utódokból viszont osztályonként több is van (két húsbolt és három zöldségbolt).

Öröklődés: Egy osztály örökölihet tulajdonságokat és viselkedésformákat egy másik osztálytól. Az utódosztályban csak az ősosztálytól való eltéréseket kell megadni.

Van az objektumorientált programozásnak még egy fontos jellemzője: a **késői** vagy más néven **futás alatti kötés**. Amikor egy objektum beprogramozásakor „megszólítunk” egy másik objektumot, nem minden tudjuk, hogy pontosan milyen operáció(k) fog(nak) végrehajtódni. Példánkban az elkészít operációt a Hölgy osztály beprogramozásakor írjuk meg, amely operáció három másik operáció hívásából áll (vásárol, főz, mosogat). De a főz operáció melyik hölgyhöz kapcsolódik végül is? Van sózás, vagy nincs? Amikor a Hölgyet programozzuk, akkor még nem tudjuk, hogy lesz szerelmes hölgy is. Hogy ki főzi majd az ebédet, az csak futáskor derül ki. Ha Kati készíti az ebédet, lesz sózás, ha más, akkor nem. A főz operáció csak később, futáskor „kötődik hozzá” a programhoz – minden annak az objektumnak az operációja, amelyik éppen „dolgozik”. Ha Kati dolgozik, akkor sóz. A Hölgy osztály kódját (az elkészít rutinnal együtt) már korábban lefordítottuk, esetleg kész szoftverként megvettük. De ha abba a főz címét fordításkor „beleégették” volna, akkor a SzerelmesHölgy programozása nem definiálhatná át a főz operációt. Ez a kérdés tehát fordításkor nyitva marad, a kötetet későbbre, a futási időre halasztjuk.

Futás alatti (késői) kötés: Hogy egy üzenet pontosan melyik metódus hívását jelenti, az sokszor csak a program futásakor derülhet ki. Ezt a jelenséget futás alatti kötésnek nevezzük.

Tesztkérdések

- 4.1. Jelölje meg az összes igaz állítást a következők közül!
 - a) Az absztrakció az objektumok közötti hasonlatosságok figyelése, összegyűjtése.
 - b) Az osztályozás a világ objektumainak rendszerezése.
 - c) Az általánosítás a világ objektumainak leegyszerűsítése.
 - d) A specializálás egy szűkebb kategória meghatározása az objektumok különbözősége alapján.
- 4.2. Mely állítás jellemző egy objektumorientált programra? Jelölje meg az összes jó választ!
 - a) Az OO program egymással kommunikáló objektumok összessége, ahol minden objektumnak megvan a jól meghatározott feladatköre.
 - b) Az objektumoknak nagyjából egyenlő számú felelősségi van.
 - c) Az objektum felelős feladatainak elvégzéséért, és a feladatokat nem is adhatja át más objektumoknak.
 - d) Az objektum példányokat tulajdonságaik és viselkedésük alapján osztályokba soroljuk.
- 4.3. Jelölje meg az összes igaz állítást a következők közül!
 - a) Csak akkor küldhető üzenet egy objektumnak, ha a küldő és a fogadó objektum kapcsolatban áll egymással.
 - b) A futás alatti kötés azt jelenti, hogy csak futáskor derül ki, melyik metódus hajtódik végre.
 - c) A polimorfizmus azt jelenti, hogy ugyanarra az üzenetre a különböző objektumok egyformán reagálnak.
 - d) Az információ elrejtése azt jelenti, hogy az objektum feladatai kódolva vannak.

5. Objektum, osztály

A fejezet pontjai:

1. Az objektum
 2. Az objektum állapota
 3. Az objektum azonossága
 4. Osztály, példány
 5. Kliens üzen a szervernek
 6. Objektum létrehozása, inicializálása
 7. Példányváltozó, példánymetódus
 8. Osztályváltozó, osztálymetódus
 9. Bezáras, az információ elrejtése
 10. A kód újrafelhasználása
 11. Objektumok, osztályok sztereotípusai
-

Ebben a fejezetben tisztázzuk az objektum és az osztály fogalmát. Szó lesz az osztály szerkezetéről, az UML-ben való megadásának módjáról, és arról, hogyan lehet egy osztályból objektumokat létrehozni, és azokat használni.

5.1. Az objektum

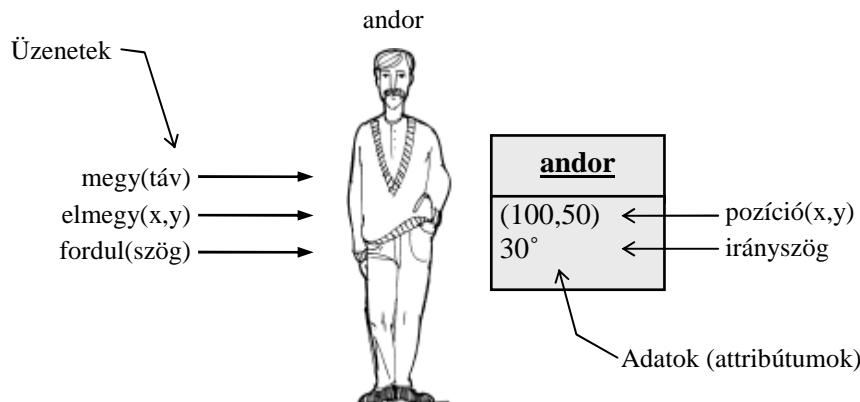
Az **objektum** (object) információt tárol, és kérésre feladatokat hajt végre. Az objektumot üzenetek által lehet megkérni a feladatok elvégzésére. Az objektum felelős feladatainak korrekt elvégzéséért.

Minden objektum valamilyen viselkedésminta szerint működik. Egy objektumnak vannak adatai és metódusai (operációi):

- ◆ **Adatok:** Az objektum az információt adatok, attribútumok formájában tárolja.
- ◆ **Metódusok:** A metódus olyan rutin (eljárás vagy függvény), amely az objektum adatain dolgozik. Az objektumot a feladatokra üzenetek által lehet megkérni. Egy üzenet hatására végrehajtásra kerül az objektumnak egy, az üzenettel azonos nevű metódusa, s ezáltal az objektum adatai megváltozhatnak.

Példaként legyen az objektum egy *andor* nevű ember, aki menni és fordulni tud (5.1. ábra). Egy absztrakció eredményeként ennek az embernek csak az *(x,y)* pozícióját és az aktuális irányszögét tároljuk. Összesen három üzenet küldhető neki, a *megy(táv)*, az *elmegy(x,y)* és a *fordul(szög)*. Emberünk tehát, ha megkérlik, a következőre képes:

- ◆ *megy(táv)*: Megy egy adott távolságnyit. Hogyan milyen irányban? Az állapotának megfelelő pillanatnyi irányba. És mi lesz az új pozíciója? Ennek kiszámolása az objektum feladata.
- ◆ *elmegy(x,y)*: Elmegy egyenesen a megadott *(x,y)* pozícióra. Eredeti irányát most már természetesen nem tudja tartani, ettől az operációtól megváltozik az irány.
- ◆ *fordul(szög)*: Irányszögéhez képest elfordul a megadott szöggel.

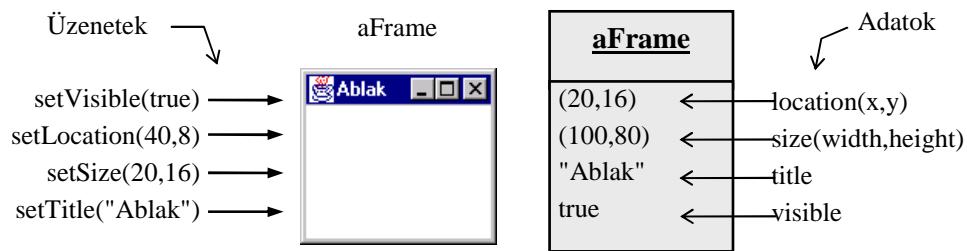


5.1. ábra. Ember objektum

Mivel ugyanarra az üzenetre több objektum is be lehet tanítva, meg kell mondani, melyik objektumnak szánjuk az üzenetet. Az objektumnak úgy küldjük az üzenetet, hogy az üzenet előtérül álljon az objektum neve: *andor.megy(8)*, *andor.elmegy(3,7)*, *andor.fordul(180)*. Katinak így küldhetünk üzenetet: *kati.megy(20)*.

Másik példaként (5.2. ábra) vegyük a képernyőn egy ablakot, melynek neve: *aFrame* (egy ablakkeret). Az ablakra jellemző annak helyzete (*location*), mérete (*size*), címe (*title*), valamint az a tény, hogy látható-e vagy sem (*visible=true* vagy *false*). Az objektumnak ezeket az adatokat kell megjegyeznie ahhoz, hogy el tudja végezni a rá szabott feladatokat:

- ◆ *setVisible(visible)*: Az ablak láthatóvá/láthatatlanná tétele (*visible=true/false*).
- ◆ *setLocation(x,y)*: Az ablak elmozgatása a megadott pozícióba.
- ◆ *setSize(width,height)*: Az ablak átméretezése (szélesség, magasság)-ra.
- ◆ *setTitle(title)*: Az ablak címének megváltoztatása.



5.2. ábra. Ablak objektum

5.2. Az objektum állapota

Az objektumnak minden van valamilyen **állapota** (state) – ez megfelel az adatok pillanatnyi értékeinek. Egy feladat elvégzése után az objektum állapota megváltozhat.

Az objektum minden „emlékszik” az állapotára: Lehet, hogy az 5.1. ábrán látható ember sokáig nem csinál semmit, de ha kap egy üzenetet, hogy megy (5), akkor onnan indul, ahol előzőleg megállt. Feladata elvégzése után más lesz a pozíciója, és legközelebb ugyanilyen utasításra erről a pozíóról indul, mégpedig a megjegyzett irányban. Ez persze csak akkor lehetséges, ha az ember minden tisztában van a maga attribútumaival.

Az 5.2. ábra ablakának állapotát együtt határozza meg az ablak négy adata. Ha az ablakot elmozdítjuk, átméretezzük, vagy megváltoztatjuk a címét, akkor az ablaknak márás megváltozik az állapota. Az ablak is csak akkor tud korrekt módon elmozdulni, ha tudja saját helyzetét, méretét stb.

Két azonos osztályhoz tartozó objektumnak akkor és csak akkor ugyanaz az állapota, ha az adatok értékei rendre megegyeznek.



5.3. Az objektum azonossága

Az objektumok egyértelműen azonosíthatók. Az **objektum azonossága** (identity) független a tárolt értékektől.

Az életben is minden objektum azonosítható, bár előfordulhat, hogy az azonosítás kisebb-nagyobb fejfájást okoz. Ha egy ikerpár például nagyon egyforma, akkor ember legyen a talpán, aki eldönti, melyikük írja éppen vizsgafeladatát Programozásból. Az üres A/4-es papírlapokat sem tudjuk megkülönböztetni, de erre általában nincs is szükség. A valóságban két objektum állapota sohasem egyezhet meg, hiszen ha más nem is, a pontos tartózkodási helyük biztosan különbözik. Absztrakció révén azonban két objektum állapota mégis megegyezhet (ha például egy embernek csak a nevét és születési évét tároljuk). Az objektumokhoz ki kell találni egy-egy minden más azonosítótól különböző azonosítót. Programjainkban ezt a problémát könnyen meg fogjuk oldani, hiszen az objektumot egy változónévvvel (andor, zsófi), illetve memóriacímmel azonosítjuk majd.

Két objektum akkor sem azonos, ha állapotaik megegyeznek!

◆ Az objektumot lehetőleg ne azonosítsuk egyik adatával se, mert az adat megváltozhat!



5.4. Osztály, példány

Az **osztályozás** a természetes emberi gondolkodás szerves része. Az ugyanolyan adatokat tartalmazó és ugyanolyan viselkedésleírással jellemezhető objektumokat egy osztályba soroljuk.

Ha katiról ugyanazokat az adatokat tartjuk nyilván, mint andorról, valamint ha azonos állapot mellett a két ember ugyanazokat a feladatokat ugyanúgy tudja végrehajtani, akkor kati és andor ugyanahoz az osztályhoz tartozik: kati „**olyan, mint**” andor. Kati és Andor ember.

Az **osztály** (class) olyan objektumminta vagy típus, amelynek alapján **példányokat** (**objektumokat**) hozhatunk létre. minden objektum egy jól meghatározott osztályhoz tartozik.

Az osztályban definiáljuk

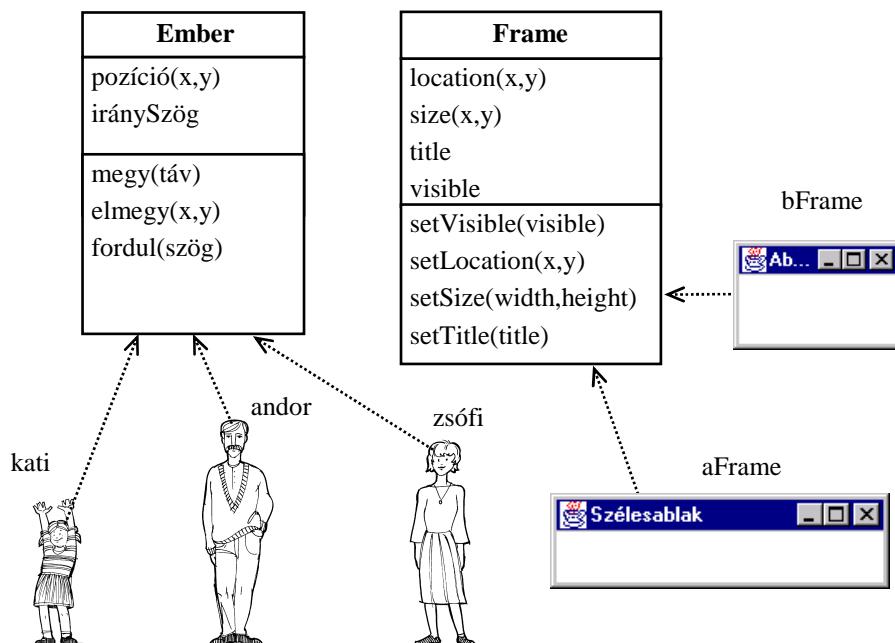
- ◆ az objektum adatait (hogy az egyes objektumok milyen adatokat jegyeznek meg),
- ◆ az objektum által elvégzendő műveleteket (metódusokat). A metódus tulajdonképpen rutin (eljárás, függvény), mely az adott objektum adatain dolgozik. Az üzenet nem más, mint egy rutin hívása.

Egy feladat elvégzésének eredménye függ egyrészt az objektum osztályától (típusától), másrészt attól, hogy az objektum milyen állapotból "indul". A fenti két ember például ugyanazon megy(táv) operáció szerint „működik” – de mivel pozíciójuk és irányszögük aktuális értéke különböző lehet, a két ember valószínűleg más pozícióba érkezik.

Az Ember: osztály; ennek az osztálynak az alapján "készült" Andor, Kati és Zsófi (5.3. ábra). Mindhárman az Ember osztály példányai, ők létező, „hús-vér” objektumok. A példányt szokás előfordulásnak vagy egyszerűen objektumnak is nevezni. Az ábrán az Ember osztályból három, míg a Frame osztályból két példányunk van. Az objektumot a programban azonosítani kell. Példánkban az embereknek kati, andor és zsófi az azonosítójuk, az ablakoké pedig aFrame és bFrame:

```
andor, zsófi, kati: Ember
aFrame, bFrame: Frame
```

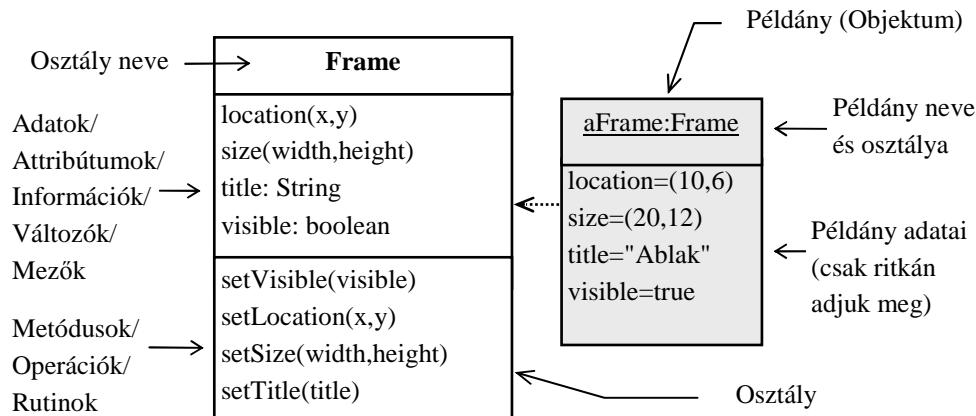
Az osztály–objektum viszony a hagyományos típus–változó viszony objektumorientált megfelelője. Ahogyan egy (hagyományos) változóhoz hozzátarozik a típusa, amely meghatározza a változó értékkészletét és a rajta végezhető műveleteket, éppen úgy az objektumhoz is hozzátarozik annak osztálya (azaz a típusa), amely leírja a benne tárolt adatokat és a vele végezhető műveleteket.



5.3. ábra. Osztályok, példányok

Könyvünk az objektumorientált paradigmák jelölésében az **UML** (Unified Modeling Language, Egységesített Modellező Nyelv) eszköztárát alkalmazza. Eszerint az osztályokat és példányokat a következőképpen adjuk meg (5.4. ábra):

- ◆ Az osztályt egy három részre osztott téglalappal jelöljük (a második és harmadik rész elhagyható):
 - a felső részbe, középre írjuk az osztály nevét, vastag betűvel;
 - a középsőbe kerülnek az osztály adatai;
 - egy adat nevét, típusát és kezdőértékét a következő formában adjuk meg:
változónév:Típus=érték (a típus és érték elhagyhatók);
 - a legsötébbbe tesszük az osztályban szereplő metódusokat;
 - egy metódust így adunk meg (csak a metódusnév kötelező, a többi elhagyható):
metódusnév(paraméter:Típus, ...):Típus
- Az adat-, illetve a metódusrész közül bármelyik elhagyható. A szóba jöhető típusok és értékek programozási nyelvenként mások és mások lehetnek.
- ◆ A példányt szintén téglalappal jelöljük:
 - A példányba beírjuk annak azonosítóját és osztályát kettősponttal elválasztva, aláhúzva (nem kell vastag betűvel): példány:Osztály. A példány- vagy osztályazonosítók egyike elhagyható; ekkor a jelölés így fest:
 - Csak példány megadása esetén: példány
 - Csak osztály megadása esetén: :Osztály (Az osztály előtt egy kettőspont van.)
 - Szükség esetén – egy vonallal elválasztott részben – megadhatjuk az objektum állapotát vagy bármit, ami a feladat szempontjából kívánatos.
 - A példányt egy szaggatott nyíllal az osztályához „köthetjük”, ha az segíti a megérteést. A nyíl mindenkor az osztály felé irányul: a példány függ az osztálytól.



5.4. ábra. Osztály és példány jelölése UML-ben

Megjegyzések:

- A példányt csak e könyv sötétíti a jobb kiemelés érdekében.
- Az osztályokba, illetve objektumokba nem kell minden információt beírni – minden csak annyit, amennyit a feladat éppen megkíván.
- Az UML ajánlása, hogy az osztály nevét nagybetűvel, az adatok és a metódusok nevét pedig kisbetűvel kezdjük.
- Az osztály megfelelő részeit ajánlatos akkor is megrajzolni, ha azok történetesen üresek. Csak az egyik részből ugyanis nem minden lehet megállapítani, hogy az adatokat vagy metódusokat tartalmaz.
- Az osztályokba, ha kell, kapcsos zárójelek közé megjegyzést írhatunk.

Egy objektum születésekor annak osztálya egyértelműen meg van határozva. Ettől kezdve az objektum a szabályoknak megfelelően él, egész életében „tudja”, hova tartozik.

Az osztály és a példány összetartozását a rendszer felügyeli, arra a programozónak nem kell figyelnie.

Ha két – azonos osztályhoz tartozó – objektumnak ugyanaz az állapota, akkor ugyanarra a kérésre pontosan ugyanúgy reagálnak.

Egy objektum minden a saját osztályában megadott metódusok szerint működik. Tegyük fel, hogy van két osztályunk: Ember és Autó, és minden osztályban van olyan metódus, hogy megy. Az egyEmber az Ember osztály, az egyAutó pedig az Autó osztály egy példánya. Az egyEmber.megy üzenet az embernek szól, ő nyilván az Ember osztályban megadott módon fog menni. Az egyAutó.megy az autónak szól. Az ember sosem fog „véletlenül” négy keréken, az autó pedig sosem fog „véletlenül” két lábon menni. Az ember, amíg csak él, „tudja”, hogy ő ember, az autó pedig „tudja”, hogy ő autó.

5.5. Kliens üzen a szervernek

Egy objektum „felkérhet” más objektumokat különböző feladatok elvégzésére. A felkérést **üzenetküldésnek** vagy **kérelemnek** nevezünk. Az egyes objektumok tehát üzenetek révén „szólítják” meg egymást, és ebben a szereposztás a következő (5.5. ábra):

- ◆ **Kliens:** a feladatot elvégeztető objektum (ügyfél, kérő, üzenetet küldő)
- ◆ **Szerver:** a feladatot elvégző objektum (kiszolgáló, végrehajtó, választ adó, üzenetet fogadó)

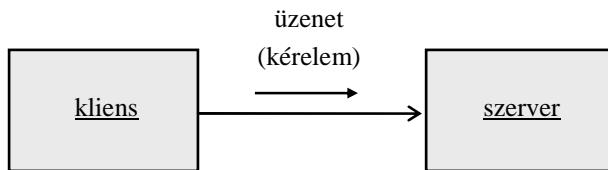
Amikor egy szerverobjektumot felkérünk egy feladat elvégzésére, akkor az objektumnak üzenetet küldünk a következő formában: objektum.metódus. Itt a metódus szükségszerűen az objektum osztályában deklarált metódus. Az objektum.metódus üzenet hatására minden az objektum osztályának megfelelő metódus hajtódik végre.

Az üzenet (message) nem más, mint egy kívülről elérhető metódus hívása. Az üzenetet a kliens (client) küldi, és a szerver (server) fogadja.

Az objektumnak vannak belső, kívülről elérhetetlen metódusai is. Az aFrame objektum kívülről elérhető metódusai például a setVisible, a setTitle stb. Az ablak keretrajzoló metódusa belső metódus, hiszen a keret rajzolására az ablakot nem szokás és nem is ajánlatos külön megkérni.

Ugyanaz az objektum lehet egyszer kliens, máskor szerver. Általános dolog, hogy egy objektum a számára kiszabott feladat bizonyos részeit más objektumokkal (az arra legalkalmasabbakkal) végezeti el. Vannak tipikus kliensobjektumok (például egy vezérlő objektum), illetve tipikus szerverobjektumok (például printer vagy adatbázis objektumok). Szokásos elnevezések még: az **aktor** (actor) csak kér, az **ügynök** (agent) kér és végrehajt.

Az üzenetet a megszólítandó objektum azonosítójával minősítjük, és az üzenetnek lehetnek paraméterei: **objektum.üzenet(paraméterek)**. Ha a megszólított objektumtól választ (információt) is kérünk, akkor azt paraméterek révén, illetve a függvény visszatérési értékének kaphatjuk meg. Egy objektum önmagának is küldhet üzenetet.



5.5. ábra. Szereposztás az üzenetküldésben

5.6. Objektum létrehozása, inicializálása

Az objektumnak életciklusa van. minden objektum egyszer „megszületik”, aztán „él”, végül „meghal”. Születéskor az objektumba be kell táplálnunk a kezdeti adatait – amíg egy objektumnak nincsen állapota, addig azt semmilyen feladatra nem lehet „rábírni”.

Az objektumot létre kell hozni, majd azonnal inicializálni kell:

- ◆ be kell állítani kezdeti adatait;
- ◆ végre kell hajtani azokat a tevékenységeket, amelyek az objektum működéséhez feltétlenül szükségesek.

Az inicializálást végző metódust **konstruktornak** nevezzük. A konstruktornak lehetnek paraméterei, amelyek segítségével az objektumba kívülről „betölthetünk” bizonyos kezdeti értéke-

ket. Vannak olyan adatok, amelyeket az objektumot életre lehelfő „egyen” nem határozhat meg: ezek értékét az osztály tervezője előre eldönti, és a konstrukturban belülről beállítja. Ha a konstruktur egyetlen adatot sem kér kívülről, akkor egyáltalán nem kellenek neki paraméterek. A Javában a konstruktur neve megegyezik az osztály nevével.

Példaként nézzük meg, hogyan írták meg az Ember osztály konstrukturát:

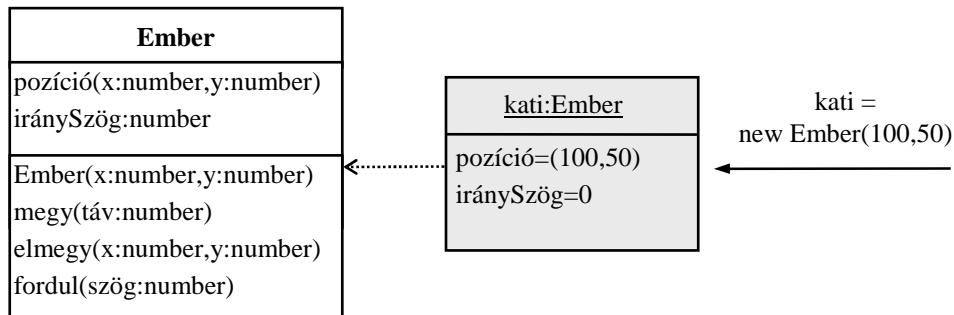
```
Ember(x:number,y:number)
  // Egy metódus hivatkozhat osztályának bármelyik adatára:
  pozíció.x = x
  pozíció.y = y
  iránySzög = 0    // kezdetben minden ember iránysszöge 0°.
end Ember
```

Az objektumot létre kell hozni, és inicializálni kell. Az inicializálást elvégző metódust **konstruktornak** (constructor) nevezzük.

A Javában az objektumot a new (új) operátorral hozzuk létre. Az operátor után meg kell adni az osztály nevét (milyen osztályból kívánjuk létrehozni az objektumot). Mivel az osztály neve egyben a konstruktur neve is, az objektum automatikusan inicializálódik.

Példaként hozzunk létre egy Ember osztályhoz tartozó objektumot, kati-t (5.6. ábra).

```
kati = new Ember(100,50)
```



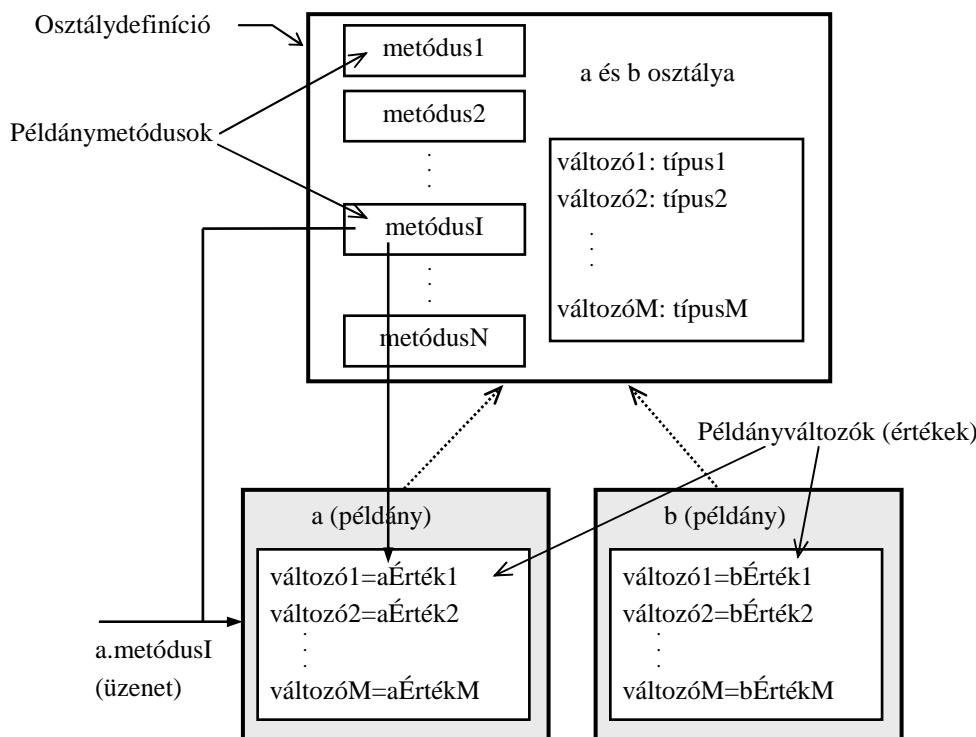
5.6. ábra. Objektum inicializálása

Egy Ember születésekor tehát megadjuk annak pozícióját. Az iránysszöget nem adjuk meg kívülről. Az osztály megalkotója úgy döntött, hogy születéskor minden ember iránysszöge 0°.

Ha az objektumra már nincs szükségünk, akkor meg kell szüntetnünk, el kell pusztítanunk. A Javában nem kell foglalkozni az objektumok megszüntetésével, memóriahelyük felszabadításával; azok automatikusan megsünnek, ha nincs rájuk semmiféle hivatkozás.

5.7. Példányváltozó, példánymetódus

Az objektum minden a saját adatain dolgozik, az operációkat (metódusokat) pedig az osztály leírásából „nézi ki”. Az 5.7. ábrán az a és b példány ugyanahhoz az osztályhoz tartozik – a példányok „össze vannak kötve” osztályukkal. Mindkét példányban megtalálhatjuk az osztály leírásában szereplő összes adatot, azok példányonként más-más értékeket vehetnek fel. A metódusokat azonban elegendő csak egyszer, az osztályban tárolni, s azok majd tekintetbe veszik a megfelelő objektum adatait (állapotát).



5.7. ábra. A példány az osztályából „nézi ki” viselkedésleírását

Kövessük most végig egy üzenet végrehajtását: kérjük fel az a példányt a metódusI feladat elvégzésére (a.metódusI)! Mivel az a példányt kértük fel, azért

- ◆ a program az a osztályában található metódusI-t fogja végrehajtani;
- ◆ metódusI az a példány adatain fog dolgozni.

A példányonként helyet foglaló változók a **példányváltozók** (instance variable), más néven példányadatok. Az osztálynak azokat a metódusait, amelyek példányonkon (példányváltozókon) dolgoznak, **példánymetódusoknak** (instance method) nevezzük.

Nézzük meg most az Ember osztály teljes definícióját! Pszeudokódban az osztály definícióját a class és az end class kulcsszavak közé tesszük. Az osztályban meg kell adnunk az összes adatot és metódust.

Az Ember osztály pszeudokódja

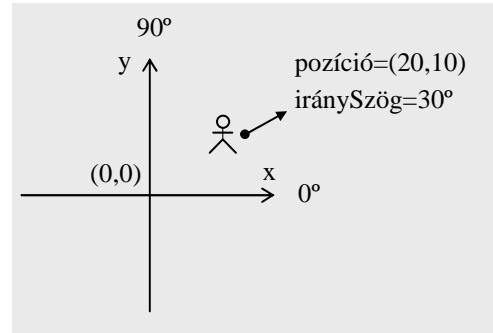
```
class Ember
    // Adatok:
    pozíció(x:number,y:number)
    iránySzög: number

    // Konstruktur:
    Ember(x:number,y:number)
        pozíció.x = x
        pozíció.y = y
        iránySzög = 0
    end Ember

    // Megerjeszti a pozíciót. Megváltozik a pozíciója:
    megy(táv:number)
        pozíció.x = pozíció.x + táv*cos(iránySzög)
        pozíció.y = pozíció.y + táv*sin(iránySzög)
    end megy

    // Elmegy az (x,y) pozícióra. Megváltozik az irányszöge is:
    elmegy(x:number,y:number)
        pozíció(x,y) kiszámítása a paraméterben megadott x és y
        értékekből.
        irányszög kiszámítása.
    end elmegy

    // Elfordul egy adott szöggel:
    fordul(szög:number)
        iránySzög = iránySzög + szög
    end fordul
end class Ember
```



Megjegyzés: Az elmegy metódusban az új irányszög kiszámítása implementációs részlet. Mivel a pszeudokódot csak nagyvonalú tervezésre szokták használni, itt elegendő az irányszög kiszámítása tevékenység megadása. A részletes megoldást majd kódoláskor dolgozzuk ki.

Az osztály teszteléseként hozunk létre egy jános azonosítójú Embert, és adjunk át neki üzeneteket! Kísérjük figyelemmel jános életének egy részét: vizsgáljuk meg, mi az állapota megszületésekor és minden további üzenet küldése után! Az eredményt a következő táblázat mutatja:

| Konstruktur és további üzenetek | János állapota pozíció(x,y) | irányszög |
|---------------------------------|--------------------------------|-----------|
| jános = new Ember(20,10) | (20,10) | 0 |
| jános.fordul(30) | (20,10) | 30 |
| jános.fordul(60) | (20,10) | 90 |
| jános.megy(6) | (20,16) | 90 |
| jános.elmegy(5,16) | (5,16) | 180 |
| jános.megy(5) | (0,16) | 180 |
| jános.fordul(-150) | (0,16) | 30 |

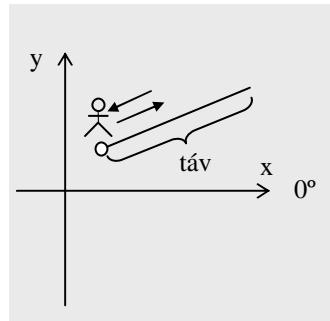
Feladat

Írunk egy olyan eljárást, amelyben egy ember tízszer fel-alá sétál egy megadott távolságon!

Az eljárás egyik paramétere az ember, a másik a távolság. Először elküldjük a szóban forgó embert az adott távolságra – amilyen irányban áll, arra fog menni: ember.elmegy(táv), majd egy 180 fokos fordulatot tesz: ember.fordul(180). Ezután ismét megtesszi a megadott távolságot, ismét megfordul stb. Ahhoz, hogy az ember a távolságot n-szer oda-vissza megtegye, a megy-fordul algoritmust $2*n$ -szer kell végrehajtania. Eljárásunkban $n=10$.

A megoldás pszeudokódja

```
sétál(ember: Ember, táv: number)
  for i=1; i<2*10; i=i+1
    ember.megy(táv)
    ember.fordul(180)
  end for
end sétál
```



Feladat

Hozzuk létre Sárát és Jánost, és az előbbi eljárást felhasználva sétáltassuk meg őket! Sára távolsága 10, Jánosé pedig 15 legyen! Felváltva sétáljanak – amíg az egyik sétál, a másik pihenjen!

Az előbbi, sétál eljárást bármilyen emberre alkalmazhatjuk, és az embert bármilyen távolságú sétálásra megkérhetjük. Először is létrehozzuk Sárát és Jánost: hogy hova, milyen pozícióba születnek, az most teljesen mindegy – bárhol voltak is előtte, most mindenkorban sétálni fognak. Sétálás után állapotuk mindenkorban visszaáll az eredetibe.

```
sára, jános: Ember  
sára = new Ember(50,20)  
jános = new Ember(100,60)  
sétál(sára,10)  
sétál(jános,15)  
sétál(sára,10)  
sétál(jános,15)
```

5.8. Osztályváltozó, osztálymetódus

Vannak adatok, amelyek nem egy konkrét példányra, hanem az egész osztályra jellemzőek. Ilyen például az adott osztályhoz tartozó, „élő” példányok száma, vagy egy olyan tulajdonság, érték, amely minden objektumra egyformán jellemző (például az egy súlycsoportba tartozó birkózó „objektumok” esetén a súlycsoport). Az ilyen közös, osztályonként egyszer előforduló változót **osztályváltozónak** (osztályadatnak) nevezzük. Az osztályváltozó értéke az osztály összes példányára ugyanaz, teljesen felesleges lenne tehát ezt az értéket példányonként eltárolni. Az osztály változóját az osztály összes objektuma ismeri és eléri.

Osztálymetódusnak nevezzük az olyan metódust, amely objektumok nélkül is tud dolgozni. Az osztálymetódus a példányadatokat nem éri el, az csak az osztályváltozókat manipulálhatja. Egy osztálymetódus meghívható az osztálynak küldött üzenettel:

```
Osztály.osztályMetódus
```

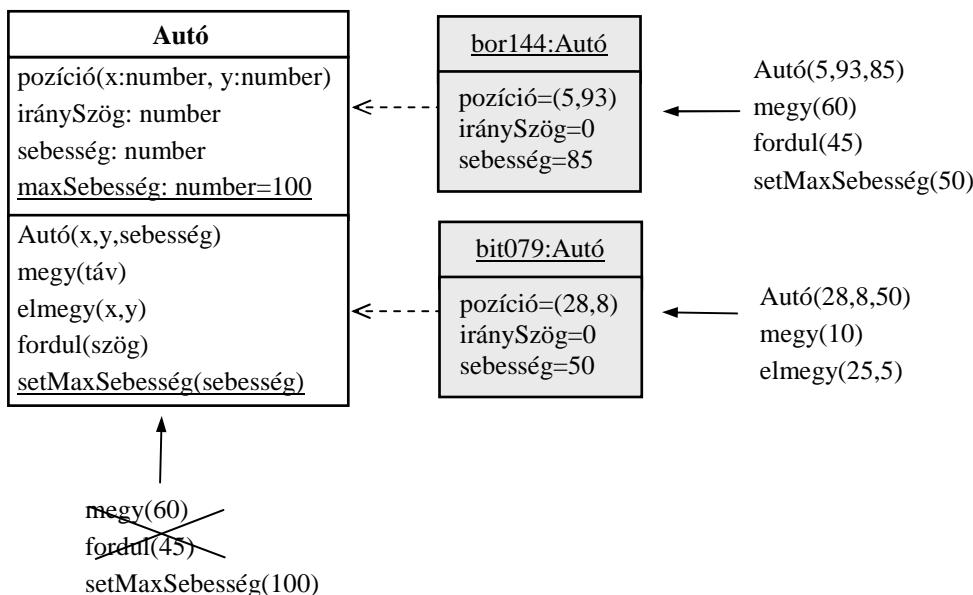
Egy osztályban az osztályváltozó és példányváltozó leírásai egyaránt szerepelnek. Az osztályváltozó az osztály születésekor keletkezik, és az osztályban kap helyet; a példányváltozó az objektum születésekor keletkezik az osztályban található leírás alapján, és az a megszületett példányban kap helyet.

A példányváltozókat csak a példánymetódusok érik el, míg az osztályváltozót egyaránt eléríti a példánymetódusok és az osztálymetódusok. Ennek az az egyszerű oka, hogy egy objektumhoz mindenkorban tartozik osztály, de ez fordítva nem mindenkorban igaz: előfordulhat, hogy egy osztálynak az adott pillanatban nincsen egyetlen előfordulása sem. Az osztályváltozó, illetve osztálymetódus nevét az UML-ben aláhúzzuk (5.8. ábra).

Példaként nézzünk egy Autó osztályt, ahol minden autónak van egy (x,y) pozíciója, egy iránya és egy sebessége (5.8. ábra). Az autó képes egy adott távolságra és egy adott pozícióra „menni”, és tud fordulni. minden autónak más a pozíciója, és más sebességgel megy; ezek az adatok tehát példányadatok. Létezik ezenkívül az autókra vonatkozóan egy sebességkorlátozás (maxSebesség), és a maximális sebességet az osztály egyetlen példánya sem lépheti túl. A maxSebesség adat minden autóra ugyanaz; ez tehát osztályadat. A maximális sebességet a setMaxSebesseg osztálymetódussal lehet beállítani, például:

```
Autó.setMaxSebesseg(100)           // beállítás 100-ra
Autó.setMaxSebesseg(Autó.maxSebesseg+10) // 10-zel többre
```

Az autó konstruktőrben (vagyis egy egyedi példány létrehozásakor) a programozónak természetesen figyelnie kell a kívülről megadott sebességet, és ha az az osztályban tárolt értéket meghaladja, vissza kell utasítania.



5.8. ábra. Osztályváltozó, osztálymetódus

Az autó példányt meg lehet szólítani példánymetódussal és osztálymetódussal egyaránt (ez utóbbi általában felesleges, és kerülendő). Egy osztálymetódus sosem dolgozhat példányadaton. A bor144 rendszámú autónak szóló setMaxSebesseg(50) üzenet az osztály maxSebesség változóját állítja be 50-re. Arra az objektumoknak maguknak kell figyelniük,

hogy ha valaki átállítja az osztály `maxSebesség` változőját, akkor ne menjenek ennél többel. Az osztály sosem figyeli a hozzá tartozó objektumokat, hiszen nem is ismeri őket.

Egy osztályt csak annak osztálymetódusával szólíthatjuk meg. Lehet, hogy az osztálynak nincs egyetlen példánya sem.

Az **osztályváltozó** (class variable) az osztály saját változója, az az egyes példányokban nem szerepel.

Az **osztálymetódus** (class method) az osztály saját metódusa, amely csak osztályváltozókon dolgozik. Egy osztályt csak osztálymetódussal lehet megszólítani; egy objektumot meg lehet szólítani példánymetódussal és osztálymetódussal egyaránt.

5.9. Bezárás, az információ elrejtése

Az objektum egyik legnagyobb ereje abban áll, hogy zárt és „sérthetetlen”. Ezt úgy tudja megvalósítani, hogy az adatokat és a hozzá tartozó metódusokat összezárja, és a kliens számára felesleges információkat egyszerűen elrejti. Vannak olyan adatok, metódusok, amelyeknek állításával, meghívásával könnyen elromolhat az objektum állapotának egysége, logikája, konzisztenciája. Ezért az objektum csak azokat az adatokat, metódusokat engedi elérni, amelyeknek az elérésével a kliens biztosan nem okozhat bajt – erről a szerver osztályt elkészítő programozónak kell gondoskodnia. Egy kliens csak azt tudhatja, hogy a szerver MIT csinál – a viselkedés HOGYAN-ja az objektum „műhelytitka”. Egy jó objektumorientált nyelv megfelelő védelem beépítésével elősegíti e szabályok betartását. Miért nagyon jó ez?

- ◆ Mert más programrész nem tudja elrontani az objektum belsejét.
- ◆ Mert az objektumban esetlegesen keletkezett hiba nem tud átterjedni más programrésekre.

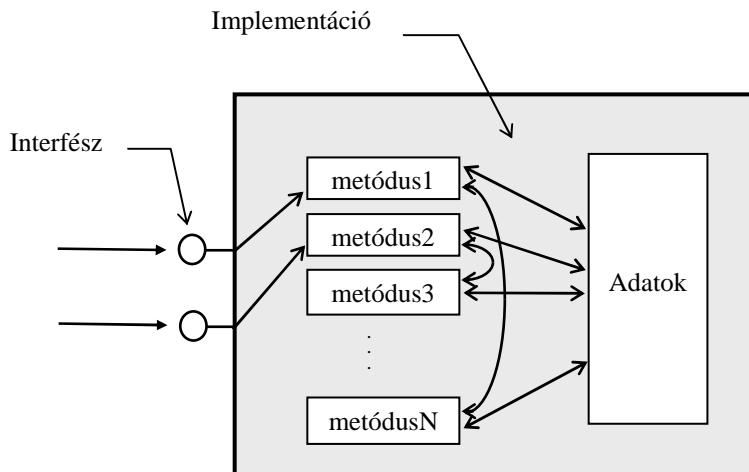
Az információ elrejtésével a hibák mennyisége lényegesen csökkenthető. De hogyan érjük el, hogy egy objektum zárt legyen? A következő szabályok (5.9. ábra) azt szolgálják, hogy más ne piszkálhasson bele az objektum belsejébe:

- ◆ Az objektumnak van interfésze: az a programozó által kijelölt metódusokból áll (ritkábban adatokból is).
- ◆ Az objektum csak az interfészén keresztül "közelíthető meg".
- ◆ Az adatokat, ha lehet, csak metódusokon át tegyük elérhetővé!
- ◆ Az objektum interfészét tegyük a lehető legkisebbé!

A **bezárás** (encapsulation) az adatok és metódusok összezárását, betokozását jelenti. Más elnevezések: egybezárás, egységbezárás.

Az információ elrejtése (information hiding) azt jelenti, hogy az objektum elrejti „belügyeit”, azokat csak az interfészén át "hagyja" megközelíteni.

A bezárás és az információelrejtés fogalma összemosódott, sokan egyként is kezelik.



5.9. ábra. Az információ elrejtése

5.10. A kód újrafelhasználása

A programozónak mindenkor törekvése volt, hogy az egyszer már jól megírt, kitesztelt programkódját akár ugyanabban, akár egy másik programban újra felhasználhassa. Az objektumorientált programozás egyik nagy előnye, hogy a régi kódokat könnyűszerrel – és hibátlanul vagy majdnem hibátlanul – használhatjuk fel más programokban.

A kód újrafelhasználásának egyik módja, hogy a már megalkotott osztályból példányokat hozunk létre. Ahány példányt hozunk létre, annyiszor használjuk újra a megfelelő osztály kódját. Ha ezt az osztályt egyszer sikerült jól beprogramozni, akkor azt nyugodtan használhatjuk akárhányszor, sosem fogunk benne csalódni. A kód újrafelhasználásának van egy másik módja is: egy már meglevő osztály öröklés révén való továbbfejlesztése, átírása.

A kód újrafelhasználása (code reuse) egy már megírt kód felhasználása akár példány létrehozásával, akár osztály továbbfejlesztésével.

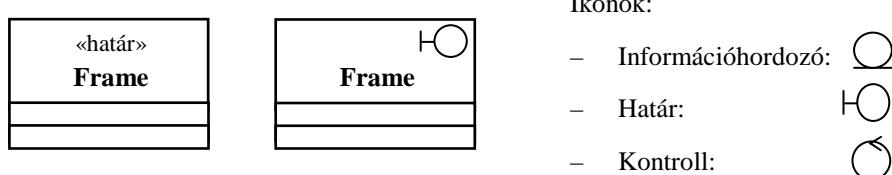
5.11. Objektumok, osztályok sztereotípusai

Az objektumokat, illetve azok osztályait feladataik jellege szerint csoportosítani lehet. Egy objektum, illetve osztály **sztereotípusának** (stereotype) nevezük annak fajtáját, jellegét. A sztereotípusok között nem mindig húzható éles határ, és előfordulhat, hogy egy objektumra két sztereotípus is jellemző. Az objektumok, osztályok alapvető sztereotípusai: információhordozó, határ, kontroll, konténer.

Az UML szerint az objektumok, osztályok lényegesebb sztereotípusai a következők:

- ◆ **Információhordozó (egyed, üzleti) objektum:** Az információhordozó objektum információt tárol. Rendszerint hosszabb életű (maradandó, perzisztens), vagyis „túléli” a program futását. Ezeket az objektumokat gyakran adatbázisban tárolják. Az információhordozó objektum általában valós világbeli személy, dolog, hely, fogalom vagy esemény. Például egy banki rendszerben: bank, ügyfél, számla stb.
- ◆ **Hatóobjektum (interfészobjektum):** A határobjectum a külvilággal kapcsolatot teremtő objektum. Az aktor határobjectumokon keresztül kommunikál a rendszerrel. Határobjectumok például egy grafikus felhasználói felület elemei: ablakok (szövegszerkesztő, dialógus, ...), beviteli mezők, nyomógombok stb. Később látni fogjuk: **mindig a határobjectum ismeri az információhordozó objektumot, sosem fordítva!**
- ◆ **Kontrollobjectum:** Vezérlést, számolást végrehajtó objektum. Ilyen például egy folyamatvezérlő, egy statisztikai adatgyűjtő, vagy egy más objektumokat koordináló objektum.
- ◆ **Konténerobjektum:** A konténerobjektum tipikusan implementációs (a program kódolását elősegítő) objektum. A konténer az objektumok közötti kapcsolatok megvalósítását szolgálja. Konténerekről a következő, Társítási kapcsolatok fejezetben lesz szó.

A sztereotípusok ismerete olyankor hasznos például, amikor egy rendszer fejlesztésekor objektumokat próbálunk azonosítani, csoportosítani, vagy a feladat megoldására használható osztályt keresgélünk egy osztálykönyvtárban. Az osztály sztereotípusát címkével (az osztály neve felett, angyalzárójelben), vagy ikonnal szokás feltüntetni (5.10 ábra).



5.10 ábra. A sztereotípus jelölése az UML-ben

Tesztkérdések

- 5.1. Jelölje meg az összes igaz állítást a következők közül!
- Feladatának elvégzése után az objektum definiálatlan állapotú lesz.
 - Ha két objektum állapota megegyezik, akkor a két objektum azonos.
 - Az osztály meghatározza objektumainak viselkedését.
 - Ha két objektum ugyanahoz az osztályhoz tartozik és ugyanabban az állapotban van, akkor egyformán reagál két egyforma üzenetre.
- 5.2. Jelölje meg az összes igaz állítást a következők közül!
- Az UML-ben az osztályt egy három részre osztott téglalappal jelöljük. A felső részbe írjuk az osztály nevét, aláhúzva.
 - Az UML-ben a példányon fel lehet tüntetni a példánymetódusokat.
 - Az objektum születésekor meg kell adni, hogy ez az objektum melyik osztályhoz tartozik, és azt az osztályt a továbbiakban már nem lehet megváltoztatni.
 - Az üzenet küldője a szerver.
- 5.3. Jelölje meg az összes igaz állítást a következők közül!
- Az objektumot születésekor inicializálni kell!
 - Az objektum kezdeti állapotát a konstruktur állítja be.
 - Az osztálymetódus elvileg elérheti a példányváltozót.
 - A példánymetódus elvileg elérheti az osztályváltozót.
- 5.4. Mi igaz az információ elrejtésére? Jelölje meg az összes igaz állítást!
- Az információhoz senki sem férhet hozzá.
 - A feladat elvégzésének hogyanja a feladatot végző objektum belügye.
 - Az objektum bizonyos információihoz nem lehet kívülről hozzáférni.
 - Csak jelszóval lehet elérni bizonyos adatokat.

Feladatok

- 5.1. Használja fel a fejezetben található Ember osztályt:
- (A)** Hozzon létre egy Gergő nevű embert! Kezdeti pozíciója legyen (30,5), irány-szöge pedig 120! Kövesse végig Gergő állapotát, ha Gergő a következő üzeneteket kapja: megy(10), fordul(120), megy(13), elmegy(0,0)!
 - (B)** Hozzon létre egy Matyi nevű embert is, és a két ember folyamatosan menjen egy adott pont felé, amíg el nem éri azt a pontot!
- 5.2. **(B)** Képzeljen el egy számla objektumot! Szimulálja a pénzbetétet és -kivétet! Készítse el a Számla osztály UML tervét és pszeudokódját! Ezután hozzon létre három számlát, amelyekre különböző összegeket betesz, illetve amelyekről különféle összegeket kivesz.
- 5.3. **(C)** Találjon ki, és specifikáljon egy tetszőleges osztályt: határozza meg és írja le pontosan a feladatát! Ezután készítse el UML tervét és pszeudokódját! Végül hozzon létre az osztályból példányokat, és használja őket!

6. Társítási kapcsolatok

A fejezet pontjai:

1. Objektumok közötti társítási kapcsolatok
 2. Osztályok közötti társítási kapcsolatok
 3. A társítási kapcsolat megvalósítása
-

Egy objektum csak akkor tud egy másik objektumnak üzenetet küldeni, azzal feladatot végrehajtatni, ha társítási kapcsolatban áll vele. A társítási kapcsolat objektumokra és osztályokra egyaránt értelmezhető; ez utóbbi az előbbinek az általánosítása. Ebben a fejezetben az objektumok és osztályok közötti társítási (ismeretségi és tartalmazási) kapcsolatokról lesz szó.

6.1. Objektumok közötti társítási kapcsolatok

Az objektumok csak úgy tudnak együttműködni, ha azok társítási kapcsolatban állnak egymás-sal. Ha a kliensobjektum üzenetet akar küldeni egy másik objektumnak, akkor tartalmaznia kell a megszólítani kívánt szerverobjektum azonosítóját (referenciáját, mutatóját). Ha az objektumok kölcsönösen akarnak üzenni egymásnak, vagyis a kliens–szerver szereposztás váltakozik, akkor minden objektumban fel kell vennünk a másikra vonatkozó referenciát.

Kétféle társítási kapcsolat létezik:

- ◆ **ismeretségi**, más szóval használati kapcsolat;
- ◆ **tartalmazási**, más szóval egész–rész kapcsolat.

Az UML-ben úgy jelöljük a kapcsolatot két objektum között, hogy az objektumokat összekötjük egy vonallal (6.1. ábra):

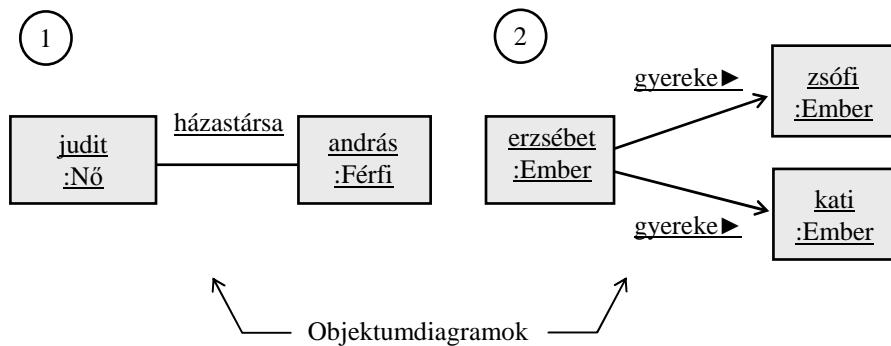
- ◆ A vonal végén levő nyíl a **navigálás irányát** mutatja: a nyíl irányában minden képpen lehet az objektumnak üzenetet küldeni. A nyílat nem kötelező kitenni, ha a kapcsolat kétirányú vagy még nem tudjuk, hogy milyen.
- ◆ Információs jelleggel a vonal fölé, illetve alá írhatjuk a **kapcsolat nevét** és **irányát**. A nevet itt aláhúzzuk, mert objektumok közötti kapcsolatról van szó (a két osztály közötti kapcsolat egy példánya). Az irányt egy tömör nyílhegy jelöli: a kapcsolatot a nyíl irá-

nyában kell olvasni, mint például a 2. esetben: „erzsébet gyereke zsófi”. Ha nincs nyíl, akkor a kapcsolat neve minden irányban olvasható. „judit házastársa andrás” és „andrás házastársa judit”.

Az objektumokat és azok kapcsolatait ábrázoló diagramot **objektumdiagramnak** vagy **példánydiagramnak** nevezzük. Egy objektum természetesen több objektummal is kapcsolatban állhat, és az is előfordulhat, hogy nincs kapcsolatban egyetlen másik objektummal sem.

Ismeretségi kapcsolat

Két objektum **ismeretségi** (használati) kapcsolatban áll egymással, ha egyik léte sem függ a másikétől, és legalább az egyik ismeri, illetve használja a másikat. Az ismeretségi kapcsolatban álló objektumok közül bármelyik megszüntethető, csak arra kell vigyázni, hogy a „túlélő” objektum a továbbiakban ne hivatkozzék a „meghalt”-ra, és ne maradjon referencia nélkül az az objektum, melynek még üzenetet akarunk küldeni.



6.1. ábra. Objektumok közötti ismeretségi kapcsolatok

A 6.1. ábra csupa ismeretségi kapcsolatot ábrázol. Elemezzük a két esetet:

1. judit Nő, andrás pedig Férfi. A kapcsolat kétirányú: „judit házastársa andrás”, „andrás házastársa judit”. judit és andrás ismeri egymást, mindkettő küldhet üzenetet a másiknak.
2. Ezen az objektumdiagramon három Ember szerepel, s közülük ketten-ketten vannak egymással kapcsolatban. A kapcsolatok egyirányúak: „erzsébet gyereke zsófi” és „erzsébet gyereke kati”. Erzsébet ismeri a gyerekeit, de a gyerekek nem ismerik őt (a példában legalábbis nem, és így nem üzenhetnek vissza).

Tartalmazási kapcsolat

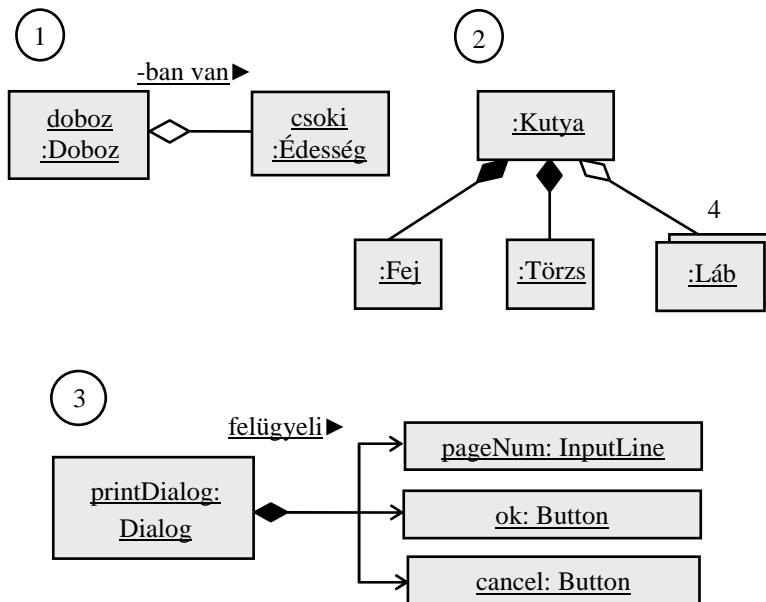
Két objektum **tartalmazási** (egész–rész) kapcsolatban áll egymással, ha az egyik objektum fizikailag tartalmazza vagy birtokolja a másik objektumot. A tartalmazó objektum az **összetett** vagy **egész objektum**, a benne levő pedig a **részobjektum**. A részobjektum léte az egész objektumtól függ, vagyis ha az egész objektumot megszüntetjük, vele együtt „pusztul” a rész is. A tartalmazási kapcsolatra sokszor jellemző, hogy az egész objektum delegálja feladatát a részeire is (például egy dialógusablak elmozgatásakor annak nyomógombjai is elmozdulnak).

Kétféle tartalmazási kapcsolat létezik:

- ◆ **gyenge tartalmazás**: ha a rész kivehető az egészből;
- ◆ **erős tartalmazás**: ha a rész nem vehető ki az egészből.

Tartalmazási kapcsolat esetén a kapcsolat vonalára egy kis rombuszt teszünk a tartalmazó objektum oldalára: a gyenge tartalmazást üres, míg az erős tartalmazást tömör rombusz jelöli (6.2. ábra). **Kompozíció**nak nevezzük azt a tartalmazást, amelyben az egész objektum erős tartalmazási kapcsolatban áll valamennyi részével, vagyis az egész létrehozásakor összeáll a végleges kompozíció, és később nem vehető ki belőle egyetlen rész sem.

A tartalmazási kapcsolat erősebb, mint az ismeretségi: az egész objektum minden ismeri a részét.



6.2. ábra. Objektumok közötti tartalmazási kapcsolat

Nézzük végig a 6.2. ábra eseteit:

1. „A doboz-ban van csoki.” A doboz osztálya Dobozi, a csokié Édesség. Ez gyenge tartalmazás: ha a dobozt ellopják, ellopják a csokit is; de amíg megvan a doboz, kivehetjük belőle a csokit (ekkor megszűnik a kapcsolat).
2. „A Kutyának van Feje, Törzse és négy Lába.” A kutya a fejével és a törzsével erős tartalmazási kapcsolatban áll, hiszen azok nélkül egyáltalán nem tud létezni. A lábakat gyenge tartalmazásként jelöltük; ennek helyessége természetesen vitatható. A lábakat **multiobjektumként** ábrázoltuk: ha egy objektum több egyforma (egy osztályhoz tartozó) objektummal áll kapcsolatban, akkor azokat objektumcsoportként, egymást majdnem takaró téglalapokkal ábrázolhatjuk. A csoportban szereplő objektumok számát a kapcsolat vonalára írhatjuk a csoport felőli oldalra. Ilyenkor a csoport elemei egyenként nem nevezhetők meg.
3. A `printDialog` tipikus kompozíció: a printelést vezérlő dialógusdoboz tartalmazza az oldalak számát megadó `pageNum` inputsort, valamint egy `ok` és egy `cancel` nyomógombot. „`printDialog` felügyeli `pageNum`-ot, `ok-t` és a `cancel-t`”. Ha a dialógusdobjozt megszüntetjük, akkor meg kell szünniük a benne levő objektumoknak is: hogy festene az, ha az ablak eltűnése után a nyomógombok ott maradnának a képernyön? De annak sincs értelme, hogy egy nyomógombot kivegyünk a dialógusdobozból: ezek az objektumok csak így, együtt dolgoznak tökéletesen.

Megjegyzések:

- A magyarban a ragozás miatt sokkal nehezebb a kapcsolatnak olyan nevet találni, hogy az összeolvasás következetes legyen, és jól hangozzék. Angolul a Kutya és a Fej közötti kapcsolatot így mondanánk: „Dog has a Head”, „Head belongs to Dog”. Próbálja meg ezt ragozás nélkül lefordítani magyarra: „Kutya birtokolja Fej”?, „Fej tartozik Kutya”? Az egyik objektumot a magyarban ragoznunk kell.
- Annak elődöntése, hogy egy kapcsolat tartalmazási vagy ismeretségi kapcsolat-e, későbbre halasztható. A tartalmazási kapcsolatot nem szükséges mindenáron feltüntetni.

Az objektumok közötti társítási kapcsolat (association) lehet ismeretségi vagy tartalmazási:

- két objektum **ismeretségi** vagy használati **kapcsolatban** (acquaintance association) áll egymással, ha létezésük független egymástól, és legalább az egyik ismeri, illetve használja a másikat.
- két objektum **tartalmazási** vagy egész–rész **kapcsolatban** (aggregation, whole-part association) áll egymással, ha az egyik objektum fizikailag tartalmazza vagy birtokolja a másik objektumot. A tartalmazás **erős**, ha a tartalmazott objektum nem vehető ki a tartalmazóból, egyébként **gyenge**. **Kompozíció**nak nevezzük azt a tartalmazást, amelyben az egész objektumot erős tartalmazási kapcsolat köti össze valamennyi részével.

Az objektumokat és kapcsolataikat ábrázoló diagramot **objektumdiagramnak** vagy **példánydiagramnak** nevezünk.

6.2. Osztályok közötti társítási kapcsolatok

Egy feladat kapcsán általában nem konkrét objektumok közötti kapcsolatokról van szó, hanem osztályok közötti kapcsolatokról, hiszen a szabályokat legtöbbször általánosan kell meghatározni. Ilyenkor definiálni kell, hogy az egyik osztály objektumai milyen kapcsolatban állhatnak a másik osztály objektumaival. A példányok között ezek után csak olyan kapcsolatok állhatnak fenn, amelyeket az osztályok közötti kapcsolatok megengednek. Az így megadott szabályoknak egy rendszerben minden körülmények között érvényesülniük kell, akkor is, ha az egyes osztályokból újabb példányokat hozunk létre, illetve szüntetünk meg.

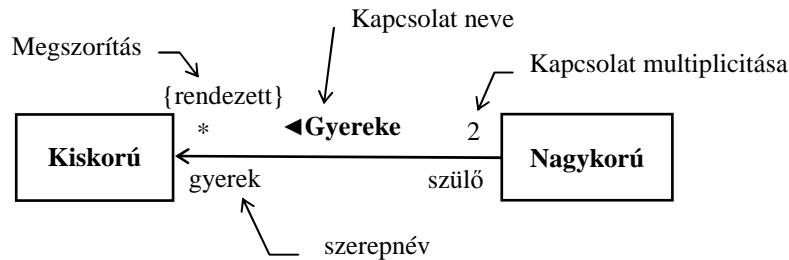
Osztályok között ugyanúgy értelmezünk ismeretségi, illetve tartalmazási kapcsolatot, mint objektumok között. Az osztályok közötti társítási kapcsolat az objektumok közötti társítási kapcsolat általánosítása: kifejezi, hogy az egyik osztály egy-egy objektuma hány objektummal állhat a kapcsolatban a másik osztályból, és milyen lehet ez a kapcsolat. Két osztály között a kapcsolat lehet ismeretségi vagy tartalmazási (az egyes objektumpárosok kapcsolatai egyformák). Az osztályokat és azok kapcsolatait ábrázoló diagramot **osztálydiagramnak** nevezük.

Két osztály kapcsolatát a következők jellemzik (a jelölésüket is felsoroljuk; 6.3. ábra):

- ◆ **Kapcsolat:** Vonalat húzunk a két osztály közé. Az ismeretségi, illetve tartalmazási (erős vagy gyenge) kapcsolatokat ugyanúgy jelöljük, mint objektumok esetében. Opcionális, vagyis nem kötelező megadni.
- ◆ **Navigálási irány:** A vonal végein levő nyilak mutatják. Opcionális.
- ◆ **Kapcsolat neve és iránya:** A vonalra tesszük nagy kezdőbetűvel, vastag betűsen, aláhúzás nélkül. Opcionális.
- ◆ **Multiplicitás (kardinalitás):** A vonal két végére egy-egy számot vagy számhalmazt írunk: az osztályhoz közel eső szám azt jelenti, hogy a szemközti osztály egy objektumához hány objektum tartozhat ebben az osztályban. Számhalmazt számok és számintravallumok felsorolásával adhatunk meg. A * jelentése: akárhány. Ha nem írunk a vonalra semmit, az 1-et jelent. Például:
 - 3 : pontosan három
 - 0..1 : nulla vagy egy
 - 10..20 : 10 és 20 közé eső szám
 - * : 0..∞, vagyis akárhány
 - 1..* : 1..∞, vagyis legalább egy
 - 1,3,10..* : 1 vagy 3, vagy 10-től kezdve akárhány.
- ◆ **Szerepnév:** Két osztály kapcsolata nagyon jól jellemzhető azzal, hogy mi az osztályok szerepe ebben a kapcsolatban. A szerepnevet kisbetűvel írjuk. Opcionális.

- ◆ **Megszorítások (kitételek):** Ilyen például a kapcsolat rendezett volta (ha egy objektummal kapcsolatban álló objektumok valamelyen szempont szerint rendezve vannak). A megszorításokat kapcsos zárójelbe tesszük, például: {rendezett}. Opcionális.

A 6.3. ábrán egy osztálydiagram látható. minden egyes kiskorú „objektumnak” pontosan két nagykorú szülője van, valamint minden egyes nagykorú „objektumnak” akárhány kiskorú gyereke van. Az ugyanahhoz a szülőhöz tartozó gyerekek sorba vannak rendezve (például kor szerint). A navigálás egyirányú, vagyis csak a szülők „parancsolhatnak” a gyerekeknek, fordítva ez nem lehetséges.



6.3. ábra: Osztályok közötti társítási kapcsolat (osztálydiagram)

A kapcsolatok multiplicitásuk szerint osztályozzuk. Eszerint alapvetően háromféle kapcsolatot különböztetünk meg:

- ◆ egy–egy kapcsolat
- ◆ egy–sok kapcsolat
- ◆ sok–sok kapcsolat

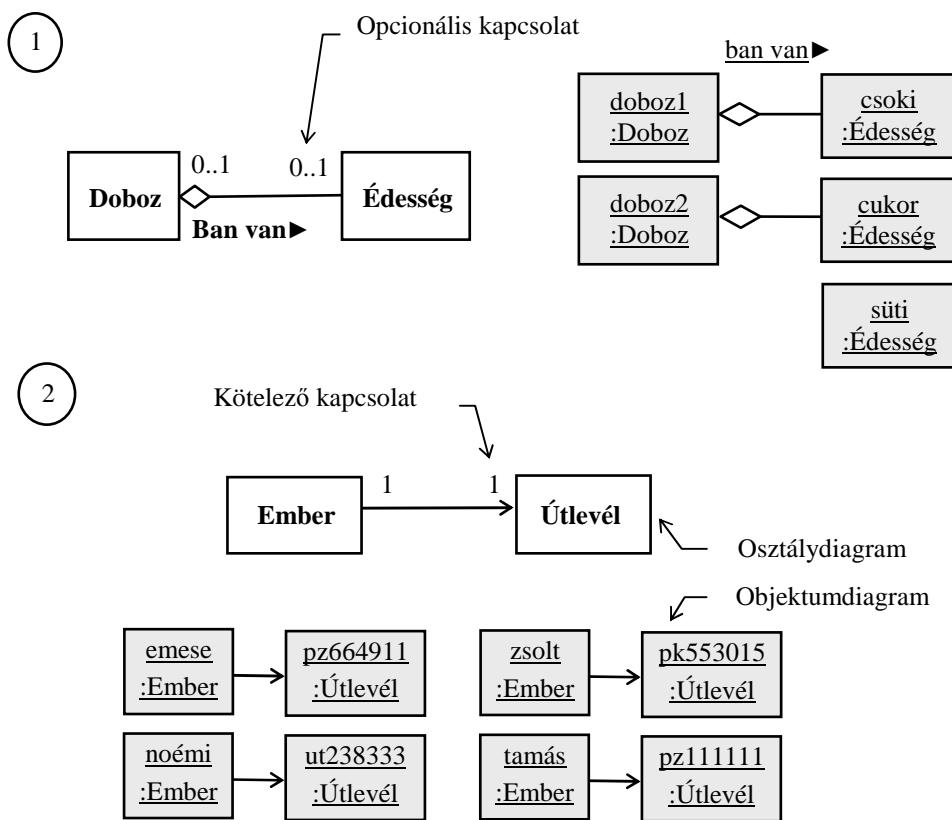
Ezen belül egy kapcsolat lehet **kötelező** vagy **opcionális** jellegű aszerint, hogy a túloldalon kell-e egyáltalán társ, vagy nem (a nulla megengedhető vagy nem). Nézzük végig ezeket az eseteket példák segítségével. Az egyes osztálydiagramokhoz elvileg akárhány példánydiagram megadható; mi a példákban csak egyet adunk meg. A példánydiagramra nem kell felenni a megfelelő osztályok összes objektumát. Egy objektumdiagram az osztálydiagram egy előfordulása, példánya.

Egy–egy kapcsolat

Ha két osztály egy–egy kapcsolatban van egymással, akkor az egyik osztály egy példánya a másik osztály legfeljebb egy (0..1) példányával állhat kapcsolatban. A másik osztályra ugyanez vonatkozik.

Elemezzük a 6.4. ábra eseteit:

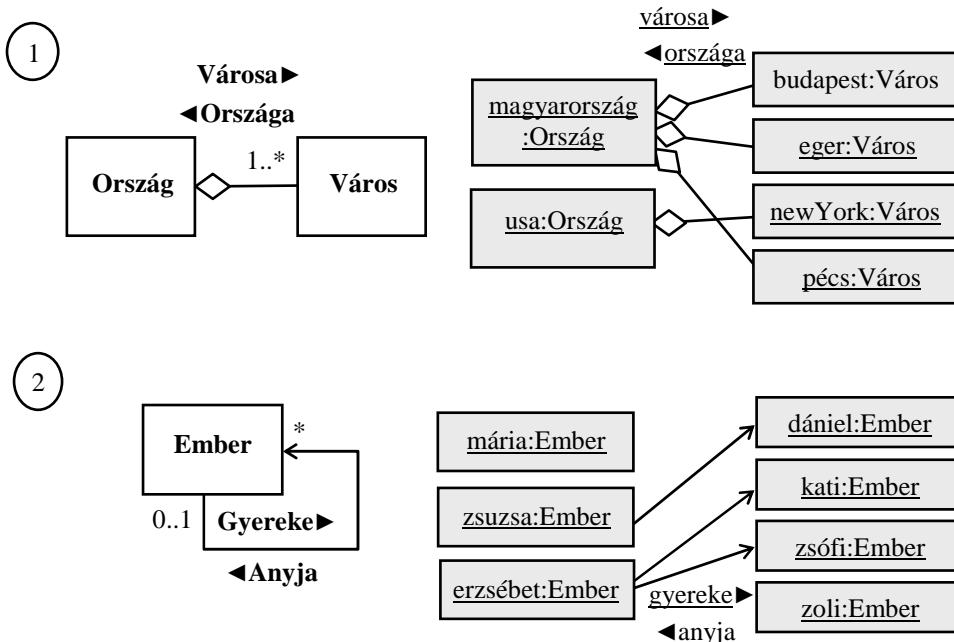
1. Az első eset azt fejezi ki, hogy vannak Doboz típusú (osztályú) objektumok, és vannak Édesség típusú objektumok. Egy dobozban vagy van édesség, vagy nincs, de egynél több édesség nem lehet benne. Fordítva: egy édességhöz vagy nulla vagy egy doboz tartozik (lehet, hogy az édesség nem dobozban van, és egy édességet két dobozba egyszerre nem lehet beletenni). Az objektumdiagram az osztálydiagram egy lehetséges előfordulását adja meg: van két doboz és három édesség. A süti nincsen dobozban. Ez az objektumdiagram kielégíti az osztálydiagram által leírt szabályokat.
2. A második esetben a kapcsolat minden végén egy 1-es szerepel (ez az alapértelmezés, ezt nem kötelező kiírni), és ez azt jelenti, hogy a kapcsolat minden irányban kötelező jellegű: minden egyes emberhez pontosan egy útlevél tartozik, és minden egyes útlevélhez pontosan egy ember tartozik. Az életben természetesen ez nem így van; a diagramok idealizáltak, absztrakcióval jönnek létre.



6.4. ábra. Egy–egy kapcsolat

Egy–sok kapcsolat

Két osztály között akkor áll fenn **egy–sok kapcsolat**, ha az első osztály egy példánya a második osztálynak sok példányával állhat kapcsolatban, a második osztály példányai viszont legfeljebb csak eggyel az első osztály példányai közül.



6.5. ábra. Egy–sok kapcsolat

Elemezzük a 6.5. ábra eseteit:

1. Az **Ország** és **Város** kapcsolata egy–sok. minden országhoz legalább egy város tartozik (**1..***; nincsen ország város nélkül), és minden városhoz kötelezően egy ország kapcsolódik (nincs olyan város, amelyiknek ne lenne országa, illetve több országhoz tartoznék). A kapcsolat tehát minden két irányban kötelező. Az objektumdiagram szerint három magyar város van, és egy amerikai. A kapcsolat gyenge tartalmazási kapcsolat: egy város, ha kell, átkapcsolható egy másik országhoz. A kapcsolat kétirányú, az ország és város meg tudja szólítani egymást.
2. Itt az **Ember** saját magával áll kapcsolatban. A szereplők tehát emberek, de a környezet az anya–gyerek kapcsolatot hangsúlyozza. Elvileg mindenki lehet anya, és egy anyának

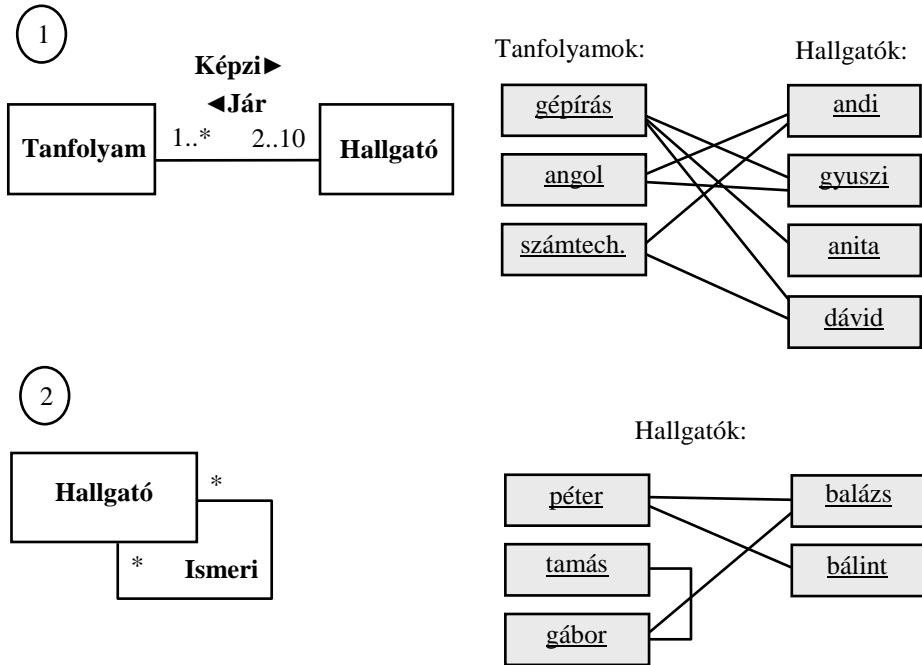
akárhány gyereke lehet (*). Egy embernek nulla vagy egy anyja lehet (0..1), legalábbis ebben a rendszerben. Az objektumdiagramon Máriának nincs gyereke; Zsuzsának egy gyereke van: Dániel; Erzsébetnek két gyereke van: Zsófi és Kati; Zoli anyját nem regisztrálták.

Sok–sok kapcsolat

Két osztály közötti sok–sok kapcsolatban a két osztály bármely példánya a másik osztálynak sok példányával állhat kapcsolatban.

A 6.6. ábra esetei:

1. Első esetben a kapcsolat a Tanfolyam és a Hallgató osztály között áll fenn. Egy tanfolyam létszáma 2 és 10 között lehet, egyébként a tanfolyamot meg kell szüntetni vagy ketté kell osztani. Egy hallgató egy vagy több tanfolyamra jár. Ha egyre sem jár, törölni kell a nyilvántartásból.
2. Itt a Hallgató osztály önmagával áll kapcsolatban: a hallgatók ismerik, illetve ismerhetik egymást. Bárminelyik hallgató ismerhet bármely más hallgatót, de lehet olyan hallgató is, aki senkit sem ismer, vagy akit senki sem ismer.



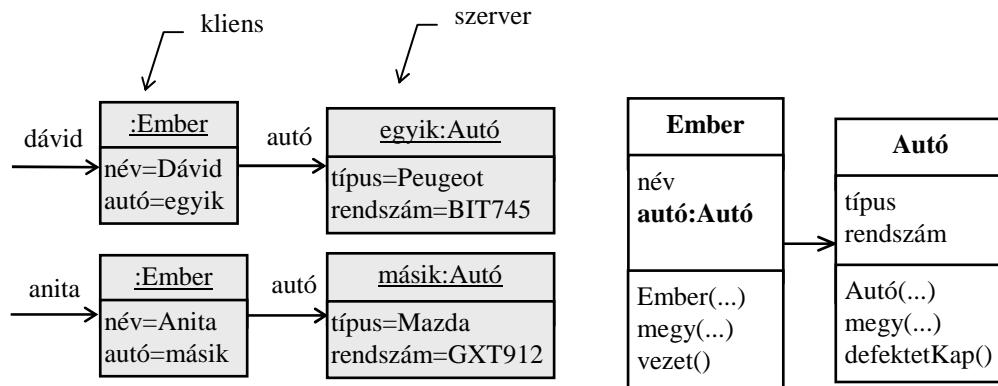
6.6. ábra. Sok–sok kapcsolat

Az osztálydiagram (class diagram) az osztályokat és azok kapcsolatait ábrázoló diagram. Az objektumdiagram az osztálydiagram előfordulása, példánya. Az osztálydiagram rögzíti az objektumok közötti kapcsolatok szabályait.

Két osztály közötti társítási kapcsolat főbb jellemzői: ismeretségi vagy tartalmazási kapcsolat (ha tartalmazási kapcsolat, akkor erős vagy gyenge); név; multiplicitás (egy–egy, egy–sok vagy sok–sok jellegű; kötelező vagy opcionális); szerepnév; megszorítás.

6.3. A társítási kapcsolat megvalósítása

A kliensobjektumnak (a kérő objektumnak) ismernie kell a szerverobjektumot (kiszolgáló objektumot), másnéven nem tudja azt megszólítani. A program készítésekor a társítási kapcsolatot nyelvi szinten meg kell valósítani. Ezt úgy tehetjük meg, hogy a kliensobjektumban felveszünk egy szerverre vonatkozó referenciát. Az, hogy egy objektum kapcsolatban áll egy másik objektummal, az objektum tulajdonsága. Ezt a tulajdonságot azonban az UML ábrán nem szükséges explicit módon megadni, mert a társítási tulajdonságokat a nyilak egyértelműen kifejezik. A kapcsolatokat megadó tulajdonságokat csak a program megvalósításakor, a kódban (pszeudokódban) kell felvennünk.



6.7. ábra. Az egy–egy kapcsolat megvalósítása

Az egy–egy kapcsolat megvalósítása

A 6.7. ábrán látható Ember-Autó – egy–egy multiplicitású, kötelező – kapcsolat a következő jelenti: ha létrehozunk egy Ember objektumot, akkor annal lesz egy autója is (ezt az Ember konstruktora fogja biztosítani). Az objektumdiagramon két ember szerepel: referenciajuk dávid és anita – más objektumokból így lehet rájuk hivatkozni. Mivel az ember ismeri

az autót, az emberben szerepelnie kell egy autóra vonatkozó referenciajának (`autó:Autó`), hogy meg tudja azt szólítani, például így: `autó.defektetKap()`. A két emberben lévő referencia természetesen más-más objektumot azonosít: a Dávid objektum `autó` nevű referenciaja az `egyik`, az Anita objektum `autó` nevű referenciajá pedig a másik autóra mutat. Két különböző autóról van tehát szó, például így lehet őket megszólítani:

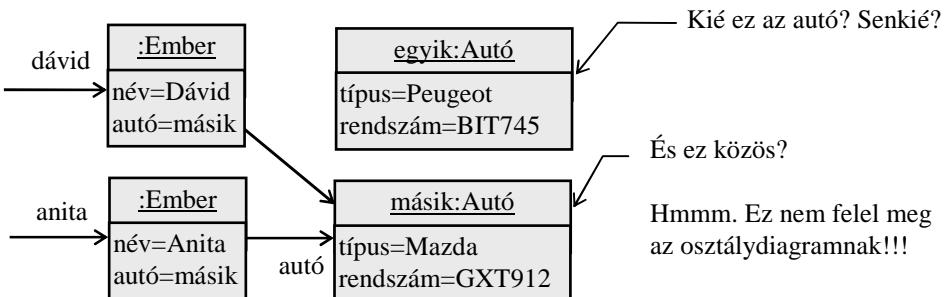
```
dávid.autó.defektetKap();
anita.autó.megy();
```

Egy-egy kapcsolat esetén a referencia kétféleképpen kaphat értéket:

- ◆ A kliens maga létrehozza a szervert, ekkor a referencia automatikusan azonosítja az új objektumot:
`autó = new Autó()`
- ◆ Egy már létező objektumra hivatkozunk. Ennek érdekében a kliensobjektumban felveszünk egy referenciátulajdonságot, és ezt ráirányítjuk a már létező szerverobjektumra, például:
`dávid.autó = anita.autó`

Ez utóbbi értékkadás itt azt jelenti, hogy Anita autója Dávidé (is) lett: az értékkadás után Dávid ugyanarra az autóra mutat, amelyikre Anita. A referencia átírányítását a 6.8. ábra szemlélteti. Dávid autója most senkié, azt mindenki elviszi a szemetes. No és kié most Anita autója? Dávidé vagy Anitáé? Anitától az autót nem vettük el, így hát ez az autó most közös. Csak az a probléma, hogy ez az objektumdiagram egyáltalán nem tartja be az osztálydiagram által felállított szabályt, miszerint mindenki pontosan egy autója van.

A kapcsolat minden jellemzőjét (multiplicitás, tartalmazási/ismeretségi stb.) a programozónak kell megvalósítania: a programot úgy kell kódolnia, hogy érvényesüljenek a megadott szabályok. A 6.8. ábra objektumdiagramja nem felel meg a 6.7. ábra osztálydiagramjának, hiszen itt nem valósul meg az egy-egy kapcsolat.



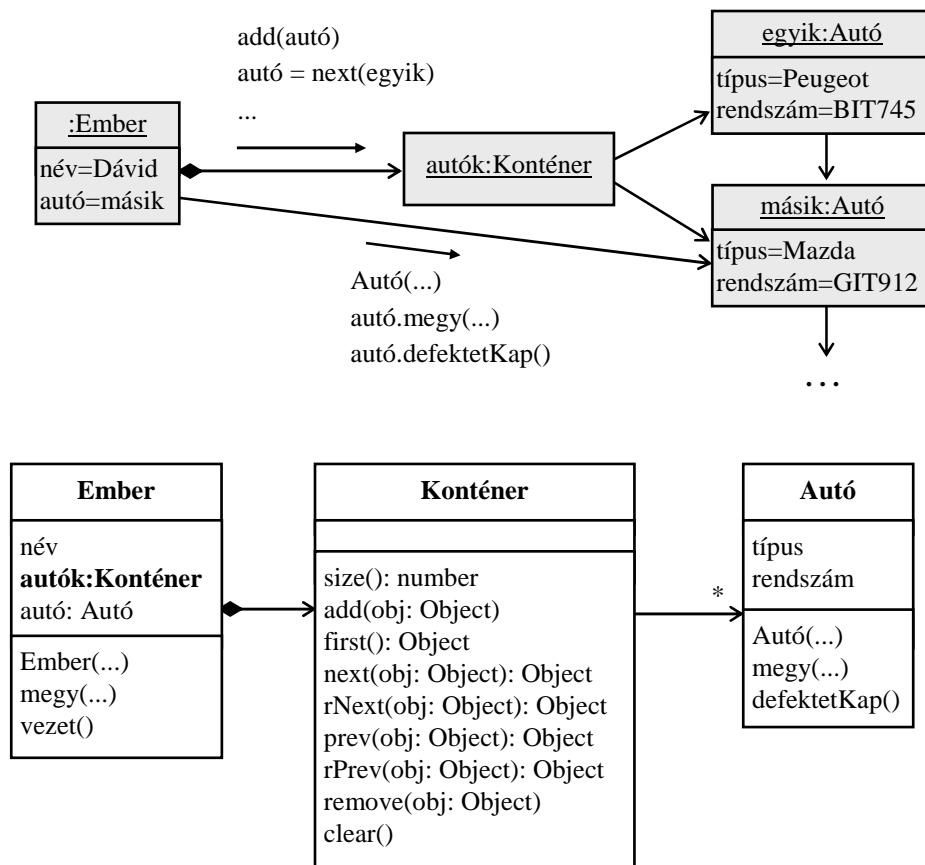
6.8. ábra. A referencia átírányítása

Az egy–sok kapcsolat megvalósítása – a konténer

Az egy–sok kapcsolat megvalósítható úgy, hogy a kliensben egyszerre több referenciát veszünk fel:

```
auto1, auto2, auto3, ... : Auto
```

Az egy–sok kapcsolatot azonban tipikusan konténerobjektumok segítségével szokás kódolni. A konténer dolga, hogy a „beledobott” objektumokat tárolja, megjegyezze azok sorrendiségét, abban egy adott objektumot megkeressen, és szükség szerint „kiadja” a kérő objektum referenciaját (az objektum a konténerben marad, csak a kliens kapcsolatba kerül a konténerben "lakó" szerverrel). A kiadott objektumnak aztán üzeneteket lehet küldeni. A konténer-objektumot az osztálydiagramon az „egy” és a „sok” objektum közé tesszük.



6.9. ábra. Egy–sok kapcsolat megvalósítása: a konténer

A 6.9. ábra együttműködési diagramja szerint egy embernek sok autója lehet. Az ember és az autók között egy autók nevű konténerobjektum „dolgozik”, az ő felelősége az autókat nyilvántartani. Az embernek nem kell bajlódnia annak megjegyzésével, hogy hány és milyen típusú autója van; csak sorban „kikéri” autót a konténertől, megkérdezi a tulajdonságaikat és üzeneteket küld nekik. A konténer ismeri összes autóját, tölök egy kis segítséget is kér: minden gyiket megkéri, hogy jegyezze meg az utána következő objektumot (ezt az ábrán úgy fejeztük ki, hogy minden autó mutatja a következő autót).

Példaként tegye be Dávid az ábrán látható két autót az autók konténerébe. Ezután üzenje azt a konténerben levő első autónak, hogy menjen, a másodiknak pedig azt, hogy kapjon defektet:

```

autó: Autó
autók: Konténer = new Konténer()
autók.add(new Autó("Peugeot", "BIT745"))
autók.add(new Autó("Mazda", "GIT912"))

...
autó = autók.first()
autó.megy()
autó = autók.next(autó)
autó.defektetKap()
```

Az osztálydiagramon – az együttműködési diagramnak megfelelően – az Ember és az Autó osztály közé tesszük a Konténer osztályt, és beleírjuk a feladatait. Az Ember osztályban felvettünk egy autó referenciát a konténer egyes autóinak azonosításához.

A Konténer osztály metódusai

- ▶ **size(): number**
Visszaadja a konténer méretét, vagyis azt, hogy hány objektum van benne pillanatnyilag.
- ▶ **add(obj: Object)**
Objektum hozzáadása a konténerhez.
- ▶ **first(): Object**
A tárolás szerinti első objektum referenciájának kiadása.
- ▶ **next(obj: Object): Object**
rNext(obj: Object): Object
obj-hoz képest a következő objektum referenciájának kiadása. Az utolsó objektum után next a null referenciát adja vissz (vége a listának), rNext pedig az elsőt (round Next, vagyis körbe megy).
- ▶ **prev(obj: Object): Object**
rPrev(obj: Object): Object
obj-hoz képest az előző objektum referenciájának kiadása. Az első objektum után prev a null referenciát adja vissza (vége a listának), rPrev pedig az utolsót (round Prev, vagyis körbe megy).
- ▶ **remove(obj: Object)**
obj objektumot törli a konténerből.
- ▶ **clear()**
Az összes objektumot törli a konténerből.

A kapcsolat minden jellemzőjét (multiplicitás, tartalmazási/ismeretségi stb.) a programozónak kell megvalósítania: **a programot úgy kell kódolni, hogy érvényesüljenek a megadott szabályok.**

Az egy–egy kapcsolatot úgy valósítjuk meg, hogy az egyik osztályban **felveszünk egy referenciatalajdonságot.**

Az egy–sok kapcsolatot tipikusan **konténerobjektumokkal** valósítjuk meg, de megvalósítható több referenciatalajdonság definiálásával is.

Tesztkérdések

- 6.1. Jelölje meg az összes igaz állítást a következők közül!
 - a) Az ismeretségi kapcsolat egyben tartalmazási kapcsolat is.
 - b) A tartalmazó objektum megszűnése maga után vonja a tartalmazott objektum megszűnését.
 - c) Kompozíció esetén a tartalmazó objektum erős tartalmazási viszonyban áll objektumival.
 - d) Két osztály kapcsolatának multiplicitása megadja, hogy a kapcsolat két oldalán összesen legalább hány objektumnak kell szerepelnie.

- 6.2. Jelölje meg az összes igaz állítást a következők közül!
 - a) Az objektumdiagram objektumokat és azok társítási kapcsolatait ábrázoló diagram.
 - b) Egy objektumdiagramból egyértelműen megállapítható az osztálydiagram.
 - c) A gyenge tartalmazás azt jelenti, hogy a tartalmazott objektumot ki lehet venni a tartalmazóból.
 - d) A kötelező kapcsolat azt jelenti, hogy az osztály minden objektumának kapcsolatban kell lennie legalább egy objektummal a másik osztályból.

- 6.3. Jelölje meg az összes igaz állítást a következők közül!
 - a) Az osztálydiagram az osztályokat és azok kapcsolatait ábrázoló diagram.
 - b) Az osztálydiagram rögzíti a lehetséges objektumok közötti kapcsolatok szabályait.
 - c) Ha két osztály között nem adjuk meg a kapcsolat multiplicitását, akkor az nulla.
 - d) Két objektum között a navigálás irányában küldhetők üzenetek.

- 6.4. Jelölje meg az összes igaz állítást a következők közül!
 - a) Két osztály között kötelező egy–egy kapcsolat esetén az egyik osztály mindegyik objektumához a másik osztályból nulla vagy egy objektum tartozik.
 - b) Két osztály között egy–sok kapcsolat esetén nem lehetséges olyan eset, hogy az egy oldalon levő objektumhoz ne tartozzék objektum a sok oldalon.
 - c) Két objektum között egy kapcsolatot úgy valósítunk meg, hogy az egyik objektumban felvesszük a kapcsolódó objektum referenciáját.
 - d) Csak objektumok között lehet tartalmazási kapcsolat, osztályok között nem.

Feladatok

6.1 (A) Határozza meg a következő objektumcsoportok osztálydiagramját, és készítsen mindegyikhez legalább két objektumdiagramot:

- Autó, sofőr
- Ember, testrészek
- Lakás, telefon
- Ház, lakás
- Emberek házastársi viszonya

Találjon ki további példákat!

7. Öröklődés

A fejezet pontjai:

1. Az öröklődés fogalma, szabályai
 2. Az utód osztály példányának adatai és a küldhető üzenetek
 3. Egyszeres, többszörös öröklés
 4. Az interfész fogalma
 5. Láthatóság (hozzáférési mód, védelem)
-

Az öröklődés két osztály között értelmezett kapcsolat: az öröklődéssel speciális osztályt készítünk egy általános osztály tulajdonságaira támaszkodva. E fejezetben az öröklődéssel kapcsolatos alapismereteket tárgyaljuk.

7.1. Az öröklődés fogalma, szabályai

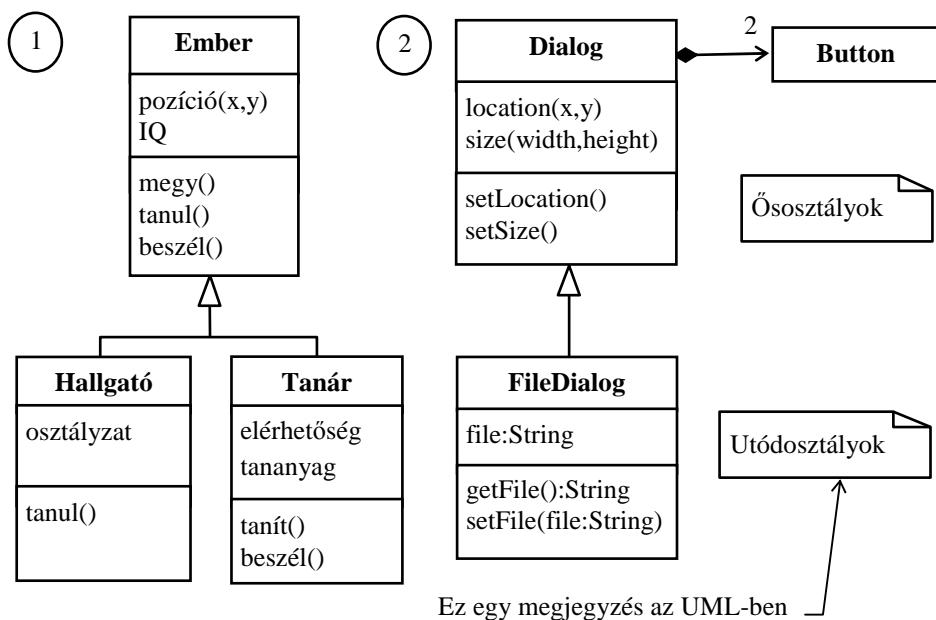
Az **általánosítás** folyamatában közös jellemzőket emelünk ki több doleg leírásából. A specializálással egyedi jellemzőket adunk egy már meglevő doleg leírásához. **Specializálással** egy már meglevő osztály tulajdonságait és képességeit felhasználva újabb osztályt készíthetünk. A meglevő osztály az **ősosztály**, a specializált pedig az **utódosztály**, más szóval **kiterjesztett osztály**. A specializált osztály örökli az ősosztály adatait és metódusait, azokhoz újabbakat adhatunk, vagy módosíthatjuk a már meglévőket. A 7.1. ábra az öröklődés jelölését mutatja UML-ben. Az utódosztályból egy nyíl mutat az ősosztályra; a nyíl hegye egy üres, zárt háromszög. A nyíl iranya a függőséget szándékozik kifejezni: az utódosztály függ az ősosztálytól. Több utód esetén rajzolhatunk külön nyílakat is, de össze is húzhatjuk őket az ábrán látható módon. A specializálást az OO paradigmában az **osztályok közötti „az egy ...” kapcsolatnak**, vagy más néven „olyan, mint ...” kapcsolatnak is szokás nevezni.

Elemezzük a 7.1. ábrán található két esetet:

1. Az **Ember**-nek van pozíciója és IQ-ja (intelligence quotient = intelligenciahányados), valamint megy, tanul és beszél. A **Hallgató** osztály az **Ember** egy specializációja („A hallgató az egy ember.”). A hallgató örökli az ember tulajdonságait és képességeit. Összességében van tehát pozíciója, IQ-ja és osztályzata, valamint megy, tanul és beszél. De a **tanul**

képessége át van írva, így a hallgató másképp tanul, mint az átlagember – valószínűleg intenzívebben. A `Tanár` osztály is az `Ember` osztály specializációja („A tanár is ember.”) – a tanárnak van pozíciója, IQ-ja, elérhetősége és tananyaga (amelyet kifejlesztett), valamint megy, tanul, beszél és tanít. Látható, hogy itt egy új képesség jelent meg, hiszen nem minden ember tud tanítani, csak a tanár. A tanár beszél képessége az emberhez képest át van írva, ő nyilván sokkal gördülékenyebben beszél.

2. A `Dialog` ablaknak van helyzete (location), mérete (size) és van két nyomógombja (`Button`). A `FileDialog` osztály a `Dialog` osztály kiterjesztése, segítségével bekérhetjük egy fájl nevét a felhasználótól. A `FileDialog` osztály természetesen öröklí a `Dialog` osztály adatait és kapcsolatait is, ezért egy `FileDialog` ablaknak is van helyzete és mérete, és lesznek nyomógombjai is. Új képességgéként megjelennek a fájl lekérdezését/beállítását elvégző metódusok (`getFile/setFile`).

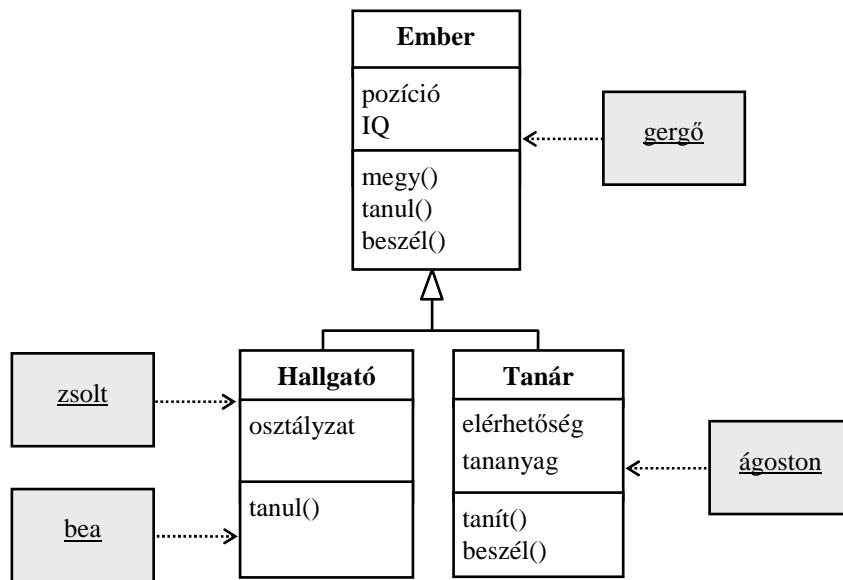


7.1. ábra. Öröklődés jelölése az UML-ben

A származtatás több szinten át folytatódhat – a `Hallgató`ból származtathatnánk például újabb osztályokat: `SzorgalmasHallgató`, `LustaHallgató`. Bármely metódus elvileg ismeri és

használhatja a saját osztályában és a felette lévő osztályokban deklarált összes adatot és metódust. A belőle származó dolgokat természetesen még nem ismerheti.

Hozzunk létre különböző példányokat az Ember, a Hallgató és a Tanár osztályokból, és vizsgáljuk meg, ki mire képes (7.2. ábra): legyen gergő egy Ember, zsolt és bea hallgató, ágoston pedig tanár. zsolt és bea ugyanúgy beszél, mint gergő; ágoston már sokkal meggondoltabban. Sem gergő, sem zsolt, sem bea nem tud tanítani, ágoston viszont igen. ágoston és gergő egyformán tud tanulni, zsolt és bea viszont sokkal intenzívebben. A példányok egész életük során pontosan tudni fogják osztályukat, és automatikusan úgy viselkednek, ahogy az az osztályukban meg van határozva.



7.2. ábra. Ki mire képes?

Egy osztály örökítésekor három lehetőségünk van:

- ◆ új változókat adunk hozzá az ősosztályhoz;
- ◆ új metódusokat adunk hozzá az ősosztályhoz;
- ◆ az ősosztály metódusait átírjuk (felülírjuk).

Az első két út az osztály bővítése, a harmadik annak megváltoztatása. Az ősosztály adatait sosem lehet átírni.

Megjegyzés: Új adatok megadása új metódusok nélkül általában értelmetlen, hiszen egyszerűt az új adattal új viselkedés is megjelenik, másrészt az adatokat metódusokkal szokás elérni (információ elrejtése). Az ös metódusok nyilvánvalóan nem ismerhetik az újonnan deklarált adatokat.

Elnevezések

- ◆ **Ősosztály** (ancestor class) = **szuperosztály** (superclass): Amiből örökítünk.
- ◆ **Utód** (descendant) = **leszármazott** = **származtatott** (derived) = **kiterjesztett** (extended) = **specializált** (specialized) **osztály** = **alosztály** (subclass): Az öröklő osztály.
- ◆ **Közvetlen ős** (immediate ancestor): A legközelebbi örökítő valamelyike.
- ◆ **Közvetlen utód** (immediate descendant): A legközelebbi öröklő valamelyike.
- ◆ **Alaposztály** (base class): A hierarchia legfelső osztálya(i).

Megjegyzés: Az ős–utód kapcsolatra szokás még a szülő–gyerek kapcsolat (parent–child relationship) elnevezést is használni. Ez azonban néha zavart okozhat, mert a szakirodalom sok helyen az egész–rész kapcsolatra használja ezt az elnevezést.

Szabályok

- ◆ **Egy osztályból több osztály is származtható.**
- ◆ **A hierarchia mélysége tetszőleges**, de: 5 még ajánlott, 10 felett áttekinthetetlen.
- ◆ **Az öröklés tranzitív**: Ha A örökli B-t és B örökli C-t, akkor A örökli C-t.
- ◆ Példányadatok öröklése: Az utódosztály példányainak adatai = ős adatok + saját (utód) adatok. Az utódosztály metódusaiban használható bármely saját és ős adat.
- ◆ Példánymetódusok öröklése: Bármely metódust felül lehet írni.
 - Az utódosztály példányának küldött üzenet minden felülírt (az öröklési ágon az utódhoz legközelebbi) metódus végrehajtását jelenti.
 - Az utódosztály metódusaiban használhatók a legközelebbi ős metódusok.
- ◆ **Az utódosztály az ősosztály kapcsolatait is örökli.**

Az általanosítás (generalization) az a folyamat, amellyel több dolgozó leírásából kiemeljük a közös jellemzőket. A **specializálás** (specialization) olyan folyamat, amelyben egy már meglévő dolgozó leírásához egyedi jellemzőket adunk hozzá.

Az öröklődés (inheritance) két osztály közötti kapcsolat; az abban szereplő **utódosztály**, más szóval **kiterjesztett osztály** (descendant class, extended class) az **ősosztály** (ancestor class, superclass) specializálása (kiterjesztése, származtatása, továbbfejlesztése) során keletkezett.

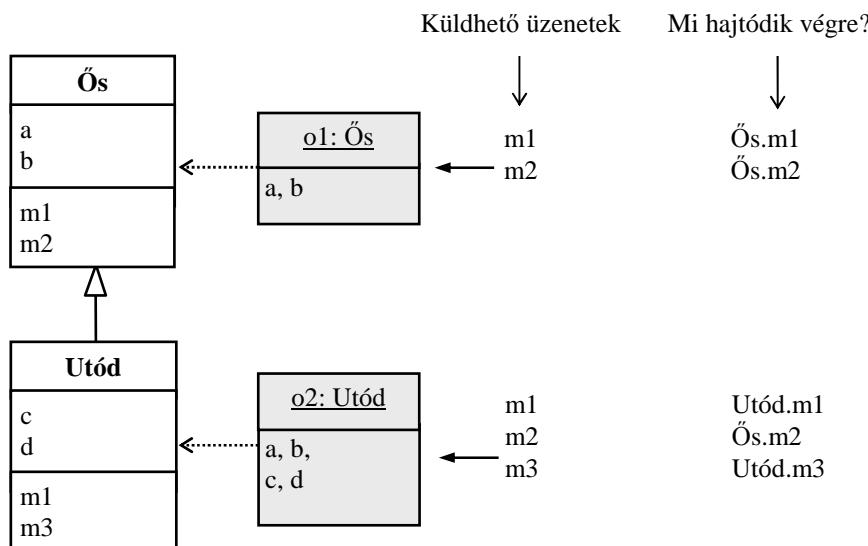
Osztálydiagram (class diagram): Olyan diagram, amely az osztályokat és a közöttük lévő társítási és öröklési kapcsolatokat ábrázolja. **Osztályhierarchia-diagram** (class hierarchy diagram): csak öröklési kapcsolatokat ábrázoló osztálydiagram.

7.2. Az utód osztály példányának adatai és a küldhető üzenetek

Kövessük most végig az adatok és metódusok öröklési szabályait egy példa segítségével! A 7.3. ábrán az Ős osztály adatai: az a és b, metódusai pedig: az m1 és m2. Az Utód osztályban a c és d adatot, valamint az m1 és m3 metódust deklaráltuk. Az utóban tehát m1 az Ős m1 felülírása, m3 pedig egy újonnan bevezetett metódus. Mindkét osztályból létrehozunk egy-egy példányt: az o1 és o2 objektumot.

Milyen adatokat tartalmaznak az egyes példányok? o1-nek két adata lesz: a és b; o2-nek már négy: a, b, c és d, hiszen c-t és d-t az utóban deklaráltuk, a és b pedig örökolt adat.

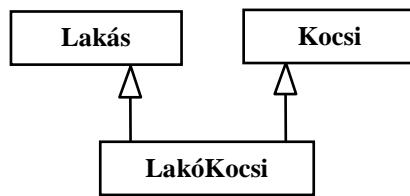
Milyen üzenetek küldhetők az egyes példányoknak? Az Ős osztály példányának küldhető üzenetek egyértelműek, hiszen az osztályában deklarált összes metódus meghívható ($\text{Ős}.m1$ és $\text{Ős}.m2$). Nézzük most meg, milyen üzenetek küldhetők az Utód osztály példányának, o2-nek! minden olyan metódus hívható, amely az öröklési ágon megtalálható – a kérdés csak az: valójában melyik hajtódiik végre? Természetesen az, amelyik az öröklési ágon közelebb áll az Utód osztályhoz. Ez azt jelenti, hogy ha van az objektum osztályában ilyen nevű metódus, akkor az hajtódiik végre, ha nincs, akkor a keresés felfelé halad az öröklési ágon. Az o2-nek három üzenet küldhető: m1, m2 és m3; o2.m1 és o2.m3 hatására az Utód osztály metódusai hajtódnak végre, hiszen m1-et felülírtuk, m3 pedig új metódus; o2.m2 hatására pedig az Ős metódusa kerül végrehajtásra, mert az Utód osztályban nincs ilyen nevű metódus.



7.3. ábra. Utód példány adatai, küldhető üzenetek

7.3. Egyszeres, többszörös öröklés

Egyszeres öröklésről akkor beszélünk, ha egy osztálynak csak egy közvetlen őse lehet. A többszörös öröklés azt jelenti, hogy egy osztálynak több közvetlen őse is lehet. A 7.4. ábrán a **LakóKocsi** ugyanúgy örökli a **Lakás** jellemzőit (lehet benne lakni, van benne hűtőszekrény, ülögarnitúra stb.), mint a **Kocsi**-ét (rész vesz a közlekedésben, kereke van, autószerelőhöz kell vinni stb.). Egyes programozási nyelvek megengedik a többszörös öröklést, mások nem. Ennek az az oka, hogy a többszörös öröklés megvalósítása lényegesen bonyolultabb, mint az egyszeresé, hiszen az, hogy az űsökben előfordulhatnak ugyanazok az azonosítók (pl. a lakásban és a kocsiban is lehet ablak vagy küszöb), elvi és programtechnikai nehézségekkel jár.



7.4. ábra. Többszörös öröklés

Ha az osztálynak több közvetlen őse is lehet, akkor **többszörös öröklésről** (multiple inheritance) beszélünk, egyébként **egyszeres öröklésről** (single inheritance).

Megjegyzések:

- A Javában, a Pascalban és a Smalltalkban csak egyszeres öröklés van, a C++ és a CLOS nyelv megengedi a többszörös öröklés alkalmazását.
- A többszörös öröklést helyettesíteni lehet ismeretségi/tartalmazási kapcsolatok felépítésével, illetve interfészkekkel.

7.4. Az interfész fogalma

Egy **interfész** (interface) metódusfejeket definiál abból a célból, hogy valamely osztály azt a későbbiekben implementálja, megvalósítsa. Az implementáló osztály speciális utódja az interfésznek. Egy osztály több interfészt is implementálhat, függetlenül attól, hogy milyen osztályokat örökölt.

Az interfész: képességek gyűjteménye. Az interfész a metódusoknak csak a formáját adja meg, azokat nem valósítja meg. Egy osztály úgy implementál egy interfészt, hogy megírja az ott

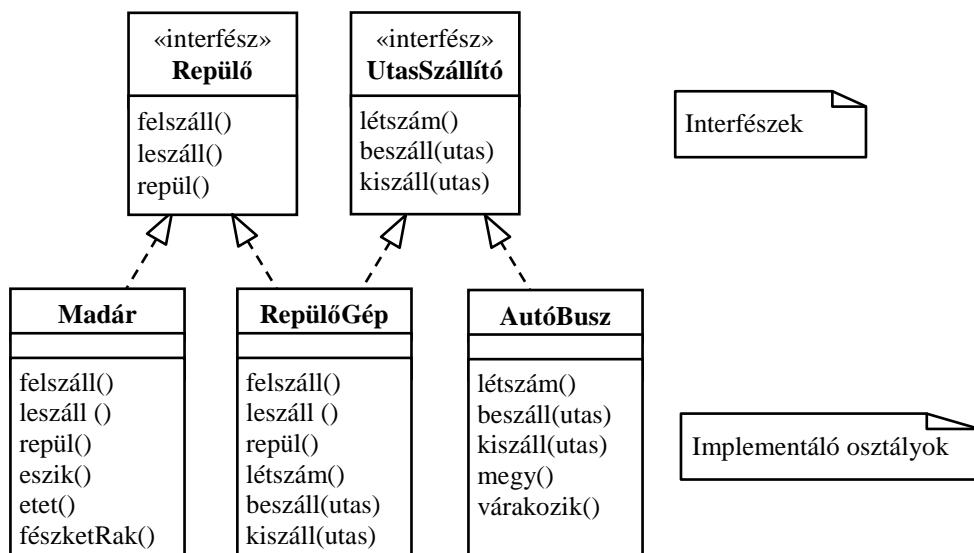
megadott metódusokat. Az implementációt az UML-ben az öröklésnél bemutatotthoz hasonló, de szaggatott nyíl jelzi (7.5. ábra). A 7.5. ábrán két interfész szerepel:

- ◆ a Repülő interfész: definiálja a felszáll(), leszáll() és repül() metódusokat;
- ◆ az UtasSzállító interfész: definiálja a létszám(), beszáll(utas) és kiszáll(utas) metódusokat.

A repülő és az utasszállító szó itt nem főnév, hanem melléknév – mindenktől képességeket fejez ki (repülő madár, utasszállító jármű stb.). Ha egy osztály implementál egy ilyen interfészt, akkor biztosak lehetünk benne, hogy az osztályban implementálva vannak ezek a metódusok; objektumaiban tehát meglesznek a megadott képességek, és meg lehet őket kérni az interfészben felsorolt feladatok elvégzésére. A 7.5. ábrán három osztály szerepel:

- ◆ a Madár Repülő, mert implementálja a Repülő interfészt.
- ◆ a RepülőGép szintén Repülő, mert ő is implementálja a Repülő interfészt – nyilvánvalóan egészen másképp, mint a madár. A Repülő UtasSzállító is egyben.
- ◆ az AutóBusz UtasSzállító, de nem tud repülni.

Az interfész implementáló osztály természetesen más metódusokat is tartalmazhat. A megfelelő interfész megvalósítására „szerződést” kötött, így az osztály használója számíthat rá, hogy az interfészben megadott metódusok mindenképpen meg vannak valósítva.



7.5. ábra. Interfészek

7.5. Láthatóság (hozzáférési mód, védelem)

A mosógépnek legyenek egyértelműek a gombjai – én majd kapcsolgom, és ha nem végzi el azt, ami le van írva, akkor elviszem a szervizbe. Amihez nem szabad hozzájárlnom, ahoz ne is lehessen hozzájárni – ezt a szerviz is így akarja, meg én is!

Már szó volt róla, hogy az adatokat semmiképpen sem ajánlatos kívülről manipulálni, és vannak olyan metódusok is, amelyeket a külső felhasználók elől el kell zární, mert a külső felhasználók bajt okozhatnak ezeknek a meghívásával. Nézzük most meg részletesebben, ki elől és hogyan lehet elrejteni egy osztály deklarációját!

Az osztály deklarációinak lényegében háromféle **láthatósága** vagy **hozzáférési módja** (visibility, access mode) létezik:

- **Nyilvános** (public): minden vele kapcsolatban álló kliens eléri és használhatja; UML jelölése: +
- **Védeott** (protected): hozzáférés csak az osztályból és az osztály leszármazottaiból lehetséges; UML jelölése: #
- **Privát** (private): az osztály privát deklarációja, csak az osztály saját metódusai férhetnek hozzá; UML jelölése: -

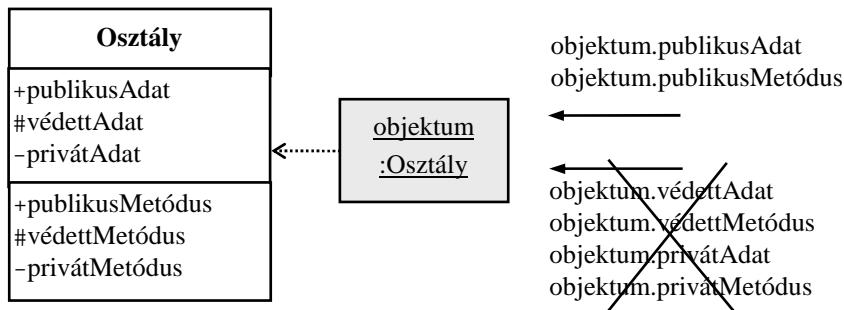
Az osztály deklarációját az elől kell vagy lehet védeni, aki az osztályt használni akarja. Egy kész osztályt lényegében kétféleképpen lehet használni:

- ◆ Az osztályból példányt hozunk létre (7.6. ábra):

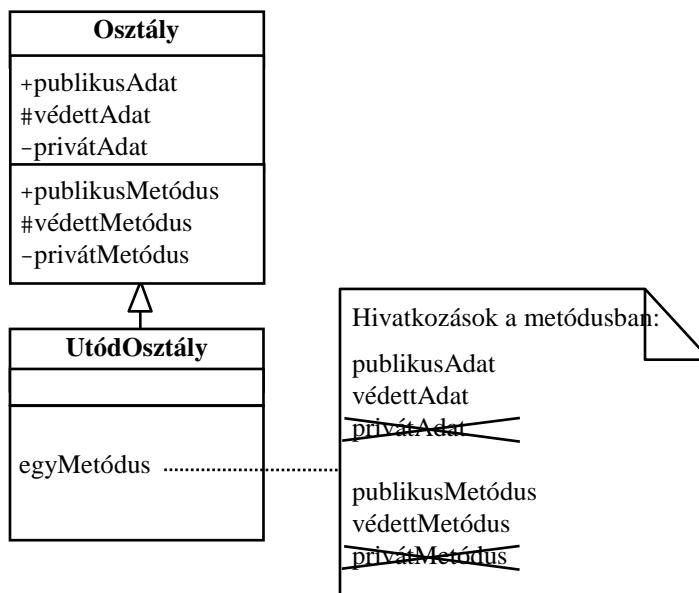
Egy objektumnak kizárolag csak a publikus deklarációt lehet elérni. A privát (private) és védeott (protected) deklarációkat az objektum megszólításával nem lehet használni.

- ◆ Az osztályból örökítéssel új osztályt hozunk létre (7.7. ábra):

Az örökítéskor az új, utódosztályban felhasználjuk a már meglévő osztály adatait és metódusait. Igaz, hogy az új osztályt valószínűleg gyakorlott programozó készíti el, mégis felmerül a kérdés: megengedjük-e az összes adat, illetve metódus használatát? Egy jól beprogramozott osztály adatai, metódusai általában csak együtt képesek jól dolgozni – ha azok közül némelyeket átírunk (akár véletlenül is), akkor tönkretehetjük az osztály működését. Ezért egy osztály írása előiránytalan, hogy mely adatokat és metódusokat enged felülírni, s melyeket nem. A gyakorlott programozónak is kényelmesebb, ha az osztályt bátran használhatja, nem kell félnie attól, hogy elront valamit. A privát (private) deklarációt csak az osztály programozója érheti el, azokhoz még öröklés révén sem lehet hozzáférni. A nyilvános (public) és védeott (protected) deklarációt az utódosztály használhatja, hivatkozhat rájuk.



7.6. ábra. Objektum védelme



7.7. ábra. Osztály védelme

Az osztályokat csoportosítani lehet: a logikailag összetartozó osztályokat csomagokba (package) foglalják. A teljességehez hozzátaroznak még a következő szabályok:

- ◆ A láthatóságot nem kötelező megadni. A láthatóság alapértelmezése az ún. **csomag szintű láthatóság**, ez azt jelenti, hogy a deklaráció az aktuális csomagban nyilvános.

- ◆ Az osztályoknak is van láthatóságuk: a publikus osztály más csomagokból is látható; alapértelmezésben egy osztály csak a saját csomagjában látható.

Az összes láthatósági jel elhagyása egy UML ábrán azt is jelentheti, hogy nem adjuk meg a láthatóságot.

Tesztkérdések

- 7.1. Jelölje meg az összes igaz állítást a következők közül!
 - a) A specializálás az a folyamat, melyben több doleg leírásából kiemeljük a közös jellemzőket.
 - b) Az általánosítás az a folyamat, melyben több doleg leírásából kiemeljük a közös jellemzőket.
 - c) Az öröklődés objektumok között értelmezett kapcsolat.
 - d) Ha a B osztály utódja az A osztálynak, akkor a B osztály hivatkozhat az A osztály összes privát metódusára.
- 7.2. Jelölje meg az összes igaz állítást a következők közül!
 - a) Az osztályok közötti öröklési kapcsolatot „az egy ...” kapcsolatnak is szokás nevezni.
 - b) Az öröklő osztály az ős kapcsolatait is öröklí.
 - c) Az utódosztály példánya csak azokat az adatokat tartalmazza, melyeket az utódosztályban deklaráltak.
 - d) Az alaposztály az osztályhierarchy teteje.
- 7.3. Jelölje meg az összes igaz állítást a következők közül!
 - a) Egy osztályhierarchy-diagram az osztályok közötti öröklési és társítási kapcsolatokat egyaránt ábrázolja.
 - b) Egy osztályhierarchy-diagram csak az osztályok közötti öröklési kapcsolatokat ábrázolja.
 - c) A többszörös öröklés jelentése: az ősosztálynak több közvetlen utódja is lehet.
 - d) A többszörös öröklés jelentése: az utódosztálynak több közvetlen őse is lehet.
- 7.4. Jelölje meg az összes igaz állítást a következők közül!
 - a) Az osztálydiagram csak az osztályokat és azok társítási kapcsolatait ábrázolja.
 - b) Az interfész metódusfejeket definiál abból a célból, hogy azokat majd egy osztály implementálja.
 - c) Egy osztály több interfészét is implementálhat.
 - d) Ha azt akarjuk, hogy egy metódust az osztályon kívül más senki se használhasson, akkor azt védettnek (protected) kell definiálnunk.
- 7.5. Jelölje meg az összes igaz állítást a következők közül!
 - a) A nyilvános deklarációt mindenki látja, tehát az osztály egy példányának használója és az osztály utódja is.
 - b) A védett deklarációt mindenki láthatja, aki megszólítja az osztály valamely objektumát.
 - c) A privát deklarációt kizárolag az osztály és annak leszármazottai látják.
 - d) A védett deklaráció erősebb védelmet ad, mint a privát deklaráció.

8. Egyszerű OO terv – Esettanulmány

A fejezet pontjai:

1. A fogalmak tisztázása
 2. Gyuszi játéka – fejlesztési dokumentáció
-

A szoftver fejlesztésének fázisairól és az egyes munkafolyamatokról a könyv 3. fejezetében esett szó. E fejezet célja, hogy alátámaszsa eddigi szoftverfejlesztési és OO tervezési ismereteinket, és megmutassa, hogyan lehet egy kis szoftvert lépésről lépésre elkészíteni. A fejezet első részében tisztázunk néhány, a megoldáshoz nélkülözhetetlen fogalmat, és megadunk néhány fejlesztési fogást. A második rész a mintafeladat fejlesztési dokumentációja: a követelményspecifikáció, a képernyőterv, a szakterületi objektummodell, a használati esetek felsorolása, a programterv és a pszeudokód.

Fontos tudni, hogy az itt tárgyalt mintaprogram Java nyelven történő megírása még nagyon sok ismeret elsajátítását igényli – ettől függetlenül a forráskód megtekinthető a programozással való barátkozás céljából. Nem gondolom azonban, hogy a fejezet elolvasása után a fejlesztésben még nem jártas Olvasó meg tudna oldani ehhez hasonló feladatokat, vagy legalább a tervezéssel megbirkózna. De most nem is ez a cél, hanem az, hogy az Olvasó kellő áttekintést kapjon az OO rendszerekről, majd annak birtokában könnyedén elsajátíthassa a Java nyelv elemeit, és ne vesszen el a kódolás meg a hibakeresés útvesztőiben.

Futtassa a GyusziJateka programot! Ehhez lapozzon a 9. fejezethez, és járjon el az ott leírtak szerint!

Megjegyzés: A Gyuszi játékához hasonló programok elkészítéséhez csak a 2. kötet ad majd elegendő ismeretet.

8.1. A fogalmak tisztázása

Először megfogalmazzuk, milyen szoftvert szeretnénk készíteni, ezután tisztázzuk a számítógépes szimuláció fogalmát, átismétljük az egyszerű OO fejlesztés munkafolyamatait, dokumentációit, s végül beszélünk majd az algoritmusvezérelt és az eseményvezérelt programról.

A készítendő szoftver

Gyuszinak hamarosan itt a születésnapja, erre szeretnénk egy személyes játékot készíteni. Leülünk vele beszélgetni, megpróbáljuk kifaggatni, hogy mire vágyna. Kiderül, hogy leginkább egy olyan számítógépes játéknak örülne, amely a már létező terepasztalát és a rajta robogó autóit szimulálja. Gyuszi úgy szokott játszani az autóival, hogy kénye-kedve szerint megfordítja vagy arrébb teszi őket.

A feladatot meg szeretnénk oldani, vagyis el szeretnénk készíteni a **Gyuszi játéka** szoftvert. Mielőtt hozzálátnánk a megoldáshoz, tisztázunk néhány, a fejlesztéshez szükséges alapfogalmat.

Számítógépes szimuláció

A szimuláció valamely létező vagy elképzelt rendszer viselkedésének utánzása. **Számítógépes szimulációval** szoftveresen utánozzuk a rendszer működését úgy, hogy a rendszer minden egyes eleméhez hozzárendelünk egy a szoftverben értelmezhető dolgot.

Gyuszi játékát számítógépen kell létrehoznunk. Ha ezt Gyuszi nem mondta volna, akkor lehet, hogy a játékboltban kötnénk ki, és méretre vágatnánk egy farostlemez a terepasztalhoz. Így azonban a feladatot szimulálni fogjuk: a terepasztal nem „igazi” lesz, hanem egy ablakkeret a képernyőn. A kisautók a számítógép képernyőjén fognak robogni úgy, hogy a megadott helyen kirajzolódnak. Gyuszi aktor (a program használója), ő a kisautókat most nem a kezével, hanem az egérrel fogja manipulálni: ha kettőt kattint az autón, akkor az 180 fokos fordulatot tesz, vonszolással pedig oda teheti, ahová akarja.

Egy egyszerű OO fejlesztés munkafolyamatai, dokumentációi

A Gyuszi játéka feladat aránylag kicsi, így a fejlesztésre elegendő egyetlen iteráció; ennek a követelményfeltárás, az analízis, a tervezés, az implementálás és a tesztelés lesznek a munkafolyamatai. Ismételjük át ezeket a munkafolyamatokat nagy vonalakban!

Követelményfeltárás

Ebben a munkafolyamatban tisztázni kell a feladatot. A követelményfeltárás dokumentációja a **követelményspecifikáció** (feladatspecifikáció), amely egyértelműen leírja a feladat iránti követelményeket.

Analízis

Az analízisnek az a feladata, hogy megvizsgálja a feladat megoldhatóságát, és felvázolja a megoldás fő vonalát. **Egyszerű feladat esetén az analízis dokumentációi a következők:**

- szakterületi objektummodell;
- a használati esetek felsorolása;
- képernyötervezek;
- a számítógépes környezet és a fejlesztőeszközök megadása.

A szakterületi modell felállításához lényegében nincs szükség speciális számítógépes tudásra, csupán szakterületi ismeretekre. Esetünkben ez azt jelenti, hogy a játék modelljének felállításához érteni kell az autóhoz, de nem kell tudnunk például programozni. A szakterületi modellt a „csak” a maga területén jártas szakember is érti. Az objektummodell elkészítéséhez be fogunk mutatni egy egyszerű fogást, amely megkönyítheti az objektumok megtalálását és a felelősségek szétosztását.

Tervezés

A programtervezésnek az a feladata, hogy az analízzel kialakult elképzéléseket továbbfejlesztve részletesen kidolgozza az ott megszületett tervet. A szakterületi modellből programterv lesz; az már magába foglal számítógéppel kapcsolatos részeket is, és már csak tapasztalt szoftverfejlesztőnek látható át.

Egy egyszerű objektumorientált program terve a következő részekból áll:

- **Osztálydiagram** (class diagram): Az osztálydiagram a feladatban szereplő objektumok osztályát és azok (társítási és öröklési) kapcsolatait ábrázolja. A rendszert egyetlen osztálydiagram írja le. Az osztálydiagramon szerepel a rendszer összes objektumának osztálya, mégpedig minden osztály pontosan egyszer. Az osztálydiagramhoz **osztályleírások** tartoznak. Ezekben az osztálydiagramon szereplő osztályokat dokumentáljuk, részletezzük. minden egyes osztálynak röviden leírjuk a feladatát, megadjuk legfontosabb jellemzőit, és szükség esetén leírjuk az egyes metódusok működését. Csak a feladatspecifikus, a fejlesztő által tervezett osztályokat kell dokumentálni, a fejlesztő-eszköz által kínált osztályokat nem. Az osztálydiagram képezi az implementálás (kódolás) alapját.
- **Együttműködési diagramok** (collaboration diagram): Az együttműködési diagram olyan objektumdiagram, amelyen feltüntetjük az egyes objektumoknak küldött üzeneteket. Az együttműködési diagramon osztályok is szerepelhetnek az osztályváltozók, illetve osztálymetódusok használata miatt. Az együttműködési diagram segítségével szemléletessé tehetjük az egyes használati eseteket, illetve operációkat működés közben. Egy osztálydiagramhoz több együttműködési diagram is tartozhat. Az együttműködési diagram az osztálydiagram egy példánya, a működő program pillanatfelvétele.

Implementálás

A programterv alapján a forráskód elkészítése. Mi most az implementációt pszeudokódban fogjuk megadni. A Java kód a könyv mellékletében megtalálható.

Tesztelés

Ebben a munkafolyamatban a számítógépen valóban futó szoftver hibáit keressük. Kérem az Olvasót, hogy tesztelje a futó játékot, és ha hibát észlel, írja meg a szerzőnek! ☺

Egy módszer az analízis objektummodelljének felállításához

Az analízis objektummodelljének felállításához össze kell gyűjteni a feladatban található összes megoldandó részfeladatot, használati esetet, és ki kell jelölni azokat az objektumokat, amelyek majd végrehajtják ezeket a feladatokat. Ezért megpróbáljuk megkeresni a feladat főbb objektumait, és meghatározni, melyik miért legyen felelős. A használati eset olyan feladat, amelynek az aktor (használó) indítja el a végrehajtását.

A főnevek és az igék szerepe a feladat szövegében

Vizsgáljuk meg a feladat szövegezését! Nagyon fontos észrevétel, hogy az objektum neve általában fönév, az objektum által végrehajtandó tevékenységek (metódusok) pedig igék. Következésképpen, ha a feladat megfogalmazása jó, akkor szinte minden „tálalva van”. Olvassuk tehát végig a feladat szövegét figyelmesen, és húzzuk alá a főneveket folytonos, az igéket pedig szaggatott vonallal!

Miért nagy segítség ez?

- ◆ A főnevek kiválasztása azért ad segítséget, mert a keresett objektumok szinte bizonyosan „nevükön vannak nevezve” a feladatban. Az OOP (objektumorientált programozás) „emberközel” programozás. Adjunk olyan nevet objektumainknak, amilyeneket a hétköznapokban is használunk! Persze nem mindegyik fönév lesz objektum, hiszen az objektumok tulajdonságai is főnevek. A talált tulajdonságokat csoportosítsuk a megfelelő objektumok köré.
- ◆ Az igék kiválasztása azért nyújt segítséget, mert az igék fogalmazzák meg azokat a feladatokat, amelyeket a program objektumai végre fognak hajtani. Az autó megfordulása például az autó dolga. Vegyük tehát sorra a feladat megfogalmazásában szereplő igéket (feladatokat), és gondolkodjunk el: van-e olyan objektum, amelyik ezt a feladatot el fogja végezni? Végül minden feladatot ki kell osztani.

Egy főnevet, illetve igét elegendő egyszer aláhúzni. A ragozott, képzett és gyanús eseteket is vizsgáljuk meg! Elképzelhető természetesen, hogy a feladat megfogalmazása kissé hiányos, vagy „túl sok a szöveg”. Persze nem nyelvtanórán vagyunk, csak egyszerűen segítséget szeretnénk kapni a feladat tervének elkészítéséhez. Egy biztos: az ilyen átgondolás mindenképpen segít az OO terv elkészítésében.

Példaként próbáljuk megkeresni Gyuszi játékának objektumait, és határozzuk meg nagy vonalakban a felelősségeket! Csak a funkcionális követelményeket boncolgatjuk. Íme, a feladat megfogalmazása (feladatspecifikáció), az aláhúzott főnevekkel és igékkel:

Gyuszinak van egy 7x5 m-es terepasztala, amelyen **kisautók robognak különböző irányokban**. minden autójára jellemző annak **színe** és **sebessége**. Ha egy autó a terepasztal **falához ér**, ott rugalmasan **ütközik**, és a megváltozott irányban **folytatja útját**. Gyuszi úgy játszik, hogy kedve szerint egy-egy autót **megfordít** vagy **arrébb tol**. Előbb vagy utóbb Gyuszi **befejezi** a játékot – ha **ráüt** egyet az asztalra, az autók engedelmesen **megállnak**.

A feladatot egy egyprocesszoros számítógépes környezetben kell szimulálni! Könyítés: tegyük fel, hogy az autók nem ütköznek soha, vagyis „**átmennek** egymáson”.

Csak a funkcionális követelményeket boncolgatjuk. A fejlesztés körülményeire utaló kitételek, mint „A feladatot egy egyprocesszoros számítógépes környezetben kell szimulálni!”, majd az implementációnál játszanak szerepet.

A szövegben talált **főnevek**:

- ◆ **Gyuszi** = aktor (a program használója)
- ◆ **terepasztal** = asztal = grafikus felület, keret, frame (7*5 m)
- ◆ **kisautó** = **autó**
- ◆ **irány** (az autó pillanatnyi tulajdonsága)
- ◆ **szín** (az autó állandó tulajdonsága)
- ◆ **sebesség** (az autó állandó tulajdonsága)
- ◆ **fal** = a terepasztal széle
- ◆ **út** = az autó **pozíciójának** változása
- ◆ **játék** = **Gyuszi játéka** = feladat = program

A szövegben talált **igék**:

- ◆ robog (autó a terepasztalon) = tovább folytatja (útját) = **megy**
- ◆ ér = **ütközik** (az autó fordul, irányt változtat)
- ◆ játszik (fut a program)
- ◆ **megfordít** (az autót) – használati eset
- ◆ **arrébb tol** (az autót) – használati eset
- ◆ **befejezi** = **ráüt** = **vége a programnak**, lenyomja az Esc billentyűt – használati eset
- ◆ **megállnak** (az autók a játék végén)
- ◆ **átmennek** (egymáson) = nem ütköznek – ezt a problémát nem kell kezelni

A szinonimák közül kiválasztjuk azt a szót, amelyik a leghétköznapibb, leglogikusabb és legszérdesebb. Időnként előállhat több jó megoldás is, és az is elképzelhető, hogy egyik szó sem az „igazi”, és egy újabbat választunk. A kiválasztott szavakat valamelyen módon megjelöljük, például az első helyre tesszük (mi most vastag betűvel szedtük).

Megjegyzések:

- Az egyszer már „megtalált” főneveket, illetve igéket többször nem húzzuk alá. A rag természetesen nem számít.
- Az objektumok megtalálásához és a felelősségek helyes szétosztásához nagyobb feladatokban jelentős tapasztalatra van szükség.

Minden feladathoz megkeressük az alkalmas objektumot: „Ki végzi el ezt a feladatot?”. Tartunk szem előtt, hogy minden feladatot ki kell osztanunk, és hogy az egyes objektumokat valakinek létre is kell hoznia, életre kell keltenie.

Mielőtt véglegesen szétosztanánk a felelősségeket, elmélkedjünk most egy kicsit a teljes program végrehajtásának felelősségéről!

A fő felelős

Tegyük fel, hogy elromlik a Mazdám, és megbízom a „Kétjobbkéz” Kft.-t, hogy javítsa meg. Én az autót egy bizonylat ellenében átadom. Az autó X kezébe jut, aki kellő szaktudás hiányában átadja azt Y-nak. Y pedig bezáratlanul az utcán felejti, s az autónak ettől "kereke kél". Kérdés: ki fizeti meg az autót? Ki a felelős? Szerencsére az én kezemben van egy bizonylat a „Kétjobbkéz” Kft. pecsétjével, és ennek alapján az autót vissza kell kapnom. Engem nem érdekel sem X, sem Y, az egész ügyért a „Kétjobbkéz” Kft. a felelős, amely időközben valószínűleg „Kétbalkéz”-re változtatta nevét.

Szoftver esetén is van „valaki”, aki a program egészéért felelős. Amikor szövegszerkesztőm egy „Application error” üzenet kíséretében „elszáll”, nem a hibáért felelős rutint vagy annak íróját fogom szidni, hanem a szövegszerkesztőt magát, és azt a céget, amely ehhez az alkalmazáshoz a nevét adta. Persze a fő felelős a legtöbb esetben áthárítja valakire a felelősséget, de ez az ó magánügye, és csak akkor sikerül neki, ha korábban egyértelműen osztódtak szét a rendszer feladatai.

A kérdés most az, hogyan kezdődjék a program, kit kérjünk meg először, és mire?

Egy objektumorientált programban mindenig van egy vezér, aki a teljes feladatot elvégzi és mindenért felel. Ezt a fő felelőst **fő objektumnak**, **vezérlőnek** (**kontrollnak**), **programnak** vagy **alkalmazásnak** (**applikációnak**) szokás nevezni.

Egy Java programban a vezér a `main` metódust tartalmazó objektum vagy osztály. A mi programunk vezére a **Gyuszi játéka** objektum. Statikus `main` metódusa azzal kezdi, hogy létrehoz az osztályból egy példányt.

A feladat objektumai, felelősségeik

Objektumjelöltjeink, az azokhoz rendelt tulajdonságok és felelősségek a következők:

- ◆ **Gyuszi:** ō aktor. Kívülről fogja manipulálni a programot a billentyűzettel és az egérrel.
- ◆ **Terepasztal**
 - tulajdonság: méret
 - felelősség: összetartja és működteti az autókat.
- ◆ **Autó**
 - tulajdonságok: pozíció, szín, sebesség, irány
 - felelősség: megy valamekkora sebességgel, megfordul, arrébb megy
- ◆ **Gyuszi játéka**
 - tulajdonság: -
 - felelősség: ez az objektum vezéri az egész programot.

A szakterületi objektummodell a fejlesztési dokumentációban megtalálható (8.4. ábra).

Algoritmusvezérelt program – eseményvezérelt program

Van egy lényeges kérdés: **Ki figyeli pontosan, hogy Gyuszi (az aktor) mikor és milyen billentyű-, illetve egéreseményeket kelt?** A továbbiakban tisztázzuk az algoritmusvezérelt és az eseményvezérelt program fogalmát. Feladatunkat eseményvezérelt környezetben fogjuk megoldani.

A programvezérlési módszereket osztályozhatjuk azszerint, hogy az aktor hogyan avatkozik be a program menetébe. Egy program kétféle lehet azszerint, hogy lehet-e vele beszélgetni:

- **Kötegelt:** Kötegelt (batch) programnak az aktor megadhat indítási paramétereket, de a futó program működésébe már nem szólhat bele.
- **Interaktív:** Interaktív (párbeszédes) programmal az aktor programfutás közben is kommunikálhat.

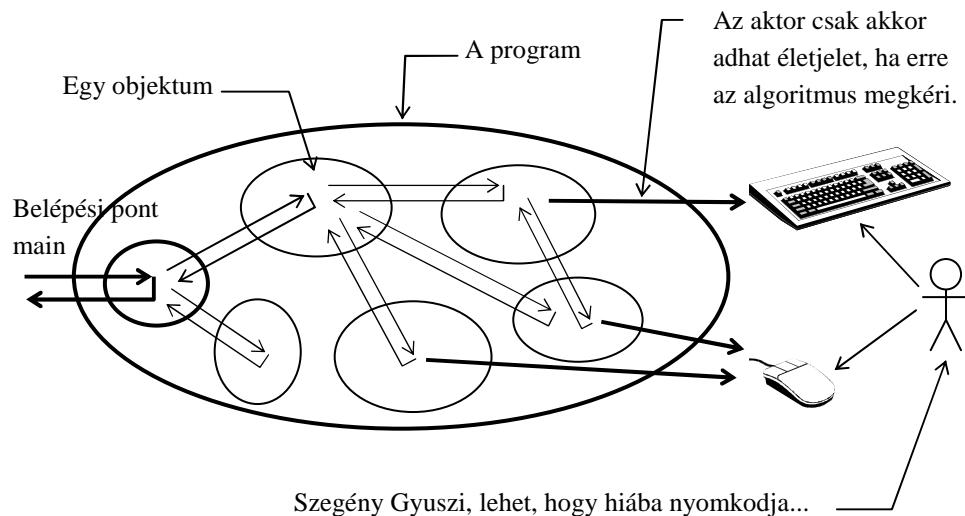
Egy interaktív program a párbeszéd módjától függően kétféle lehet:

- **Algoritmusvezérelt:** Egy algoritmusvezérelt programban az interakciót a program irányítja. Az aktor csak válaszol a program által feltett kérdésekre. **Az aktor pontosan akkor és azt mondhatja, amikor és amit a program kérdez tőle.**
- **Eseményvezérelt:** Egy tökéletesen eseményvezérelt program reagál az aktor által indukált kérdésekre, kérdésekre. **Az aktor azt mondhatja, amit ō akar, és akkor, amikor akarja!**

Az algoritmusvezéreltség a „vaskalapos” szülő (program) és a „jólnevelt” gyerek (aktor) párbeszéde – a gyerek megvárja, amíg a szülő befejezi mondandóját, majd arra válaszol, amit kérdeztek tőle. Ezzel szemben az eseményvezérelt „beszélgetés” látszólag kész zúrzavar, a két fél mintha mindenkor egymás szavába vágna. E látszólagos zúrzavar mögött azonban a program részéről egy szilárd belső rend áll – igaz, hogy az aktor elvileg azt csinál, amit akar, de a program jól nevelő szülő módjára okosan a helyére teszi a "hiperaktív" gyereket: csak azokat a kéréseit elégíti ki, amelyeknek értelmük van, a többöt – ha kell, határozott, egyértelmű útmutatás kíséretében – elutasítja. A program, ha lehet, mindenben kiszolgálja a felhasználó értelmes kérdéseit, kéréseit: párhuzamosan elvégzi a kijelölt feladatokat és figyel arra, hogy ne romoljék el a belső konzisztenciája.

Mintapéldánkban lehet, hogy Gyuszika össze-vissza nyomogatja az egeret és a billentyűzetet. Semmi baj: ha jó a program, akkor nem engedi, hogy Gyuszika bajt csináljon; nem fagy le az Enter lenyomásától, és nem engedi „leszállni” az autót a terepasztalról, ha Gyuszika netán túlhúzná az egérrel.

Az eseményvezéreltség együtt szokott járni a grafikus felhasználói felülettel, de ez nem szükségszerű követelmény. Egy grafikus felületen is működhet kötegelt vagy algoritmusvezérelt program; és fordítva: egy karakteres felületen is elképzelhető eseményvezérelt program.



8.1. ábra. Algoritmusvezérelt program

Az algoritmusvezérelt program szerkezete

Minden programnak van egy belépési pontja: a Javában egy program a vezérlőobjektum `main` metódusának meghívásával kezdődik (8.1. ábra). Ez a metódus aztán meghív egy másik metódust, amely ugyanazon az objektumon vagy egy másikon dolgozik. A metódushívások objektumról objektumra vándorolnak: a hívott metódus vagy rögtön lefut, vagy egy újabb metódust hív meg. Az egyes objektumok pillanatnyi állapotától függően a vezérlés természeten más és más ágra kerülhet. Közben bármely objektum lekérdezheti a billentyűzet vagy az egér pillanatnyi állapotát, hogy azt megnyomták-e, és pontosan milyen eredménnyel. Az algoritmusvezérelt programra az jellemző, hogy az aktor csak akkor avatkozhat bele a programba, amikor a program megkérdezi. A 8.1. ábrán látható, hogy a program objektumai és az aktor közötti kapcsolat egyirányú: **a program kérdezi az aktort** (pontosabban az aktor eszközeit: a billentyűzetet és az egeret).

Az eseményvezérelt program szerkezete

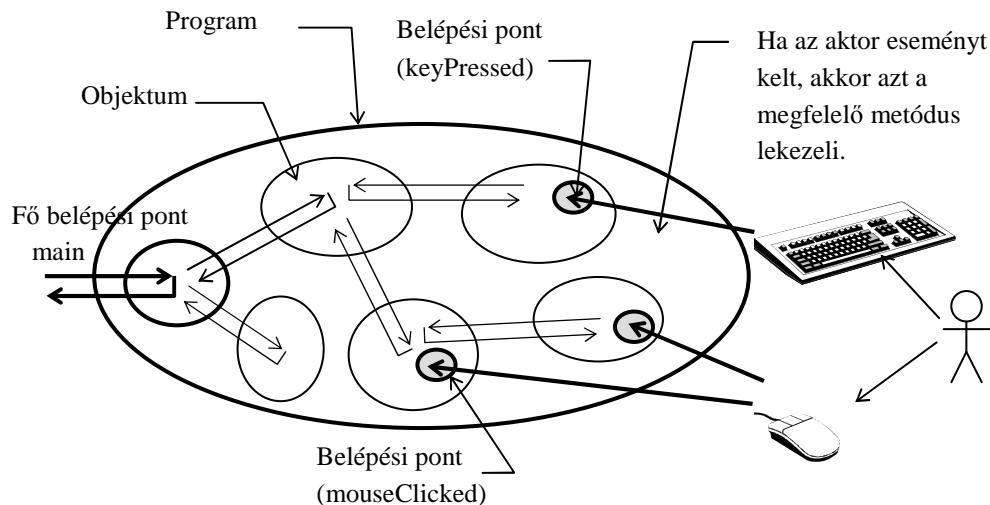
Az eseményvezérelt program (event driven program) futása (8.2. ábra) szintén a `main` meghívásával kezdődik szintén a `main` meghívásával kezdődik, és a metódusok ugyanolyan szabályok szerint kerülnek meghívásra, mint egy algoritmusvezérelt programban. Van azonban egy lényeges különbség: egy eseményvezérelt programban a program fő szála mellett működik egy eseményelosztó szál, benne egy óriási **eseményelosztó ciklus**, amely folyamatosan figyeli, hogy bekövetkezett-e valamilyen esemény (event). Ha igen, akkor végignézi, hogy melyik komponenset (objektumot) illeti az esemény lekezelése (billentyűzetesemény esetén azt, amelyik éppen fókuszban van, egéresemény esetén azt, amelyiken rajta van az egér). **A ciklus oda-adja az eseményt a jogos tulajdonosnak, hogy az lekezelhesse.** A lekezeléseket az eseménylekezelő metódusok végzik. A programozónak mindenkorra annyi a dolga, hogy megírja az objektum eseménylekezelő metódusát, és közli a rendszerrel, hogy ezen az objektumon milyen eseményeket figyeljen. Ilyen értelemben **a programnak a programozó szempontjából több belépési pontja van:** lefut egy eseménylekezelő metódus, aztán lefut egy másik eseménylekezelő metódus stb.

Az eseményelosztás roppant terhétől az eseményvezéreltséget támogató komponenskészlet használatával szabadulhatunk meg. Ez a komponenskészlet esetünkben a Java osztálykönyvtár csomagjai: a `java.awt` és a `javax.swing`.

Egy grafikus felhasználói interfésszel (Graphical User Interface, GUI) rendelkező programban a képernyőelemek téglalap alakú komponensek (a `JComponent` osztály utódainak példányai), amelyeknek van pozíciójuk, méretük, színük stb. A rendszer egy adott szabályrendszer szerint meg tudja állapítani, hogy egy egér-, illetve billentyűzetesemény melyik komponensen keletkezett. Ha – mint programozók – azt szeretnénk, hogy egy objektum reagáljon bizonyos eseményekre, akkor a következőket kell tennünk:

- ◆ meg kell írnunk a megfelelő lekezelő metódus(okat) az objektum osztályában. (A Javában az egéresemény lekezelő metódusai a mouseClicked és a mouseDragged, a billentyűzetesemény lekezelő metódusa a keyPressed.)
- ◆ az objektumot rá kell tennünk a megfelelő figyelőláncra (a Javában ezt az addMouseListener, illetve az addKeyListener parancccsal tehetjük meg). Ezzel elérjük, hogy az objektum figyelje a megfelelő eseményt. Az esemény bekövetkeztekor a lekezelő metódus automatikusan meghívódik.

Példánkban az autó egy GUI komponens, ez figyeli az egéreseményeket: ha kettőt kattintanak rajta, akkor majd megfordul, ha pedig vonszolás közben éppen rajta van az egér, akkor folyamatosan áthelyeződik abba a pozícióba, ahová az egér irányítja.



8.2. ábra. Eseményvezérelt program

8.2. Gyuszi játéka – fejlesztési dokumentáció

A következő dokumentációkat adjuk meg:

- ◆ Követelményspecifikáció A követelményfeltárás dokumentációja
- ◆ Képernyőterv (prototípus)
- ◆ Szakterületi objektummodell
- ◆ Használati esetek
- ◆ Számítógépes környezet, fejlesztőeszközök
- ◆ Programterv A tervezés dokumentációja
- ◆ Pszeudokód (Java kód helyett) Az implementálás dokumentációja

} Az analízis dokumentációi

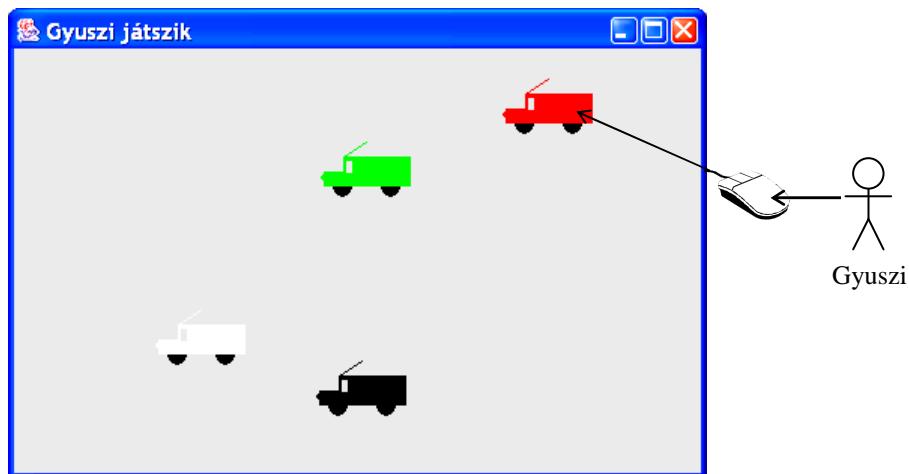
Követelményspecifikáció

Gyuszinak van egy 7*5 m-es terepasztala, melyen kisautók robognak különböző irányokban. minden autóra jellemző a színe és a sebessége. Ha egy autó a terepasztal falához ér, ott rugalmasan ütközik, és a megváltozott irányban folytatja útját.

Gyuszi úgy játszik, hogy kedve szerint egy-egy autót megfordít vagy arrébb tol. Előbb vagy utóbb Gyuszi be szeretné fejezni a játéket – ha ráüt egyet az asztalra, akkor az autók engedelmesen megállnak.

A feladatot egy egyprocesszoros számítógépes környezetben kell szimulálni! Tegyük fel, hogy az autók soha nem ütköznek össze, hanem „átmennek egymáson”.

A követelményspecifikációnak maradtak „szabad vegyértékei”: itt nincsen szó a terepasztal színéről, az autók számáról, azok formájáról, az egyes autók színéről, sebességéről stb. Ezeket a továbbiakban kell majd tisztázni!



8.3. ábra. A Gyuszi játéka program számítógépes szimulációja

Képernyőterv (prototípus)

A feladatot egy egyprocesszoros számítógépes környezetben szimuláljuk. A feladat egyes objektumainak számítógépes megfelelői a következők:

- ◆ **Terepasztal:** egy ablakkeret a képernyőn. A terepasztal fala az ablakkeret széle.
- ◆ **Autók:** rajzok a képernyőn. Úgy robognak, hogy a megadott helyen kirajzolódnak. Az autók nem ütköznek, hanem átmennek egymáson.

- ◆ **Gyuszi** aktor, az egér segítségével manipulálja a kisautókat: ha kettőt kattint valamelyiken, akkor az 180 fokos fordulatot tesz, vonszolással pedig arrébb viszi őket. A program akkor fejeződik be, amikor Gyuszi leüti az Esc billentyűt (ráüt az asztalra).

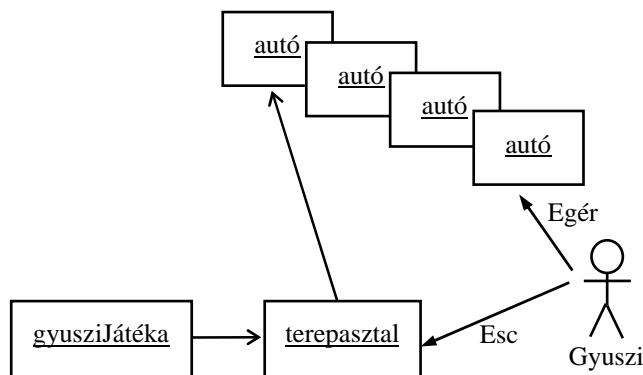
A Gyuszi játéka program képernyőterve (itt most prototípusa) a 8.3. ábrán látható.

Szakterületi objektummodell

A játék objektummodelljét a 8.4. ábra mutatja.

Aktorok és objektumok leírása, felelőssége

- ◆ **Gyuszi**: Ő aktor. A programot kívülről manipulálja billentyűzettel és egérrrel.
- ◆ **Gyuszi játéka**: Vezér. Először **létrehozza** a terepasztalt, aztán létrehoz egy csomó autót, és **rátesszi** őket az asztalra. Végül megkéri az asztalt, hogy legyen **robogás**, vagyis az autók menjenek folyamatosan a megadott módon.
- ◆ **Terepasztal**: A terepasztalnak van mérete, és az a fő feladata, hogy összetartsa és működtesse, **robogtassa** az autókat. A terepasztal olyan konténer, amelybe betesszük az összes autót. Mivel csak egy processzorunk van (és programszálakat most nem szeretnénk írni), a párhuzamos robogást a terepasztal a következőképpen érheti el: sorban, egymás után minden autót megkér, hogy menjen egy kicsit. Ha az autó megkapta a lehetőséget a működésre, akkor neki már tudnia kell, hogyan menjen. A terepasztal másik feladata, hogy **figyelje** Gyuszit, nem nyomta-e meg az Esc billentyűt. Mert ha igen, akkor azonnal be kell fejezni a robogatást.
- ◆ **Autó**: Az a fő feladata, hogy egyenletes sebességgel menjen: egyhuzamban megtegyen valamekkora, a sebességtől függő utat. Az autó figyeli Gyuszit, nem nyomta-e meg az egeret. Ha igen, akkor **megfordul** vagy **áthelyeződik**.



8.4. ábra. Gyuszi játéka – Szakterületi objektummodell

Használati esetek

A program aktora Gyuszi. Gyuszi a következő dolgokra használja a programot:

- ◆ Megfordítja az autót (kettőt kattint az autón az egérrel)
- ◆ Arrébb tolja az autót (az egérrel vonszolja az autót)
- ◆ Véget vet a programnak (leüti az Esc billentyűt)

Ezek a kezdeti használati esetek, ezeket kell a programban megvalósítani. Mivel kis feladatról van szó, ez lesz a végeleges használatieset-összeállítás is.

Számítógépes környezet, fejlesztőeszközök

A feladatot grafikus felületen, eseményvezérelten oldjuk meg. A megoldásban a következő eszközöket használjuk:

- ◆ Operációs rendszer: Windows
- ◆ Programfejlesztési módszer: Egységesített eljárás, UML
- ◆ Programnyelv: Java
- ◆ Fejlesztőeszköz: JBuilder 8.0, JDK 1.4

Megjegyzés: Az implementációt itt csak pszeudokódban készítjük el. A végső kódolás természetesen Java nyelven történik, ez megtalálható a könyv mellékletében. (*GyusziJateka.java*)

Programterv

A program együttműködési és osztálydiagramja a 8.5. ábrán látható; a tervhez tartoznak a megfelelő osztályok leírásai is. Az osztályleírások után megadjuk a program pszeudokódját (a konkrét Java implementáció helyett). Pszeudokódot egyébként az osztályleírásokban is lehet alkalmazni a jobb érthetőség kedvéért. A programterv előállításához már szükséges a programozásban való jártasság – de mert ennek a megszerzése ezutáni feladat, azért most megadjuk a JPanel és a JFrame API-osztály leírását is. Az osztályleírások után elemezni fogjuk a tervet és a pszeudokódot.

A 8.5. ábrán látható osztálydiagram osztályainak leírásai a következők:

Auto osztály – osztályleírás

Őse: JPanel

Az Auto olyan komponens, amely megjelenik képernyőn, autó formája van, tud menni, forrulni. Rugalmasan ütközik a terepasztal falával. Reagál a kettős egérkattintásra és a vonszolásra.

Metódusok:

- ◆ `Auto(szin:Color, x,y,sebesseg:number, asztal:Terepasztal)`
Konstruktor. Létrehoz egy autót, melynek színe, pozíciója és sebessége a paraméterben megadott érték lesz. Szintén paraméterben adjuk meg a terepasztalt, amelyen mozog.
- ◆ `paintComponent(g:Graphics)`
Kirajzolja az autót.
- ◆ `megy()`
Megy egy sebességének megfelelő egységet.
- ◆ `fordul(fok:number)`
Elfordul a megadott fokkal.
- ◆ `setIrany(fok:number)`
Beáll a megadott irányba.
- ◆ `mouseClicked(e:MouseEvent)`
Lekezeli a dupla kattintás egéreseményt. Megfordul, azaz fordul 180 fokot.
- ◆ `mouseDragged(e:MouseEvent)`
Lekezeli az egérvonzsolás egéreseményt. Áthelyezi az autót az egér által kijelölt pozícióba.

Terepasztal osztály – osztályleírás

Őse: `JFrame`. Mérete 700*500 pont, színe szürke. Egy–sok kapcsolatban áll az autóival. Ismeri autóit, és képes robogtatni őket. Reagál az Esc billentyű leütésére.

Metódusok:

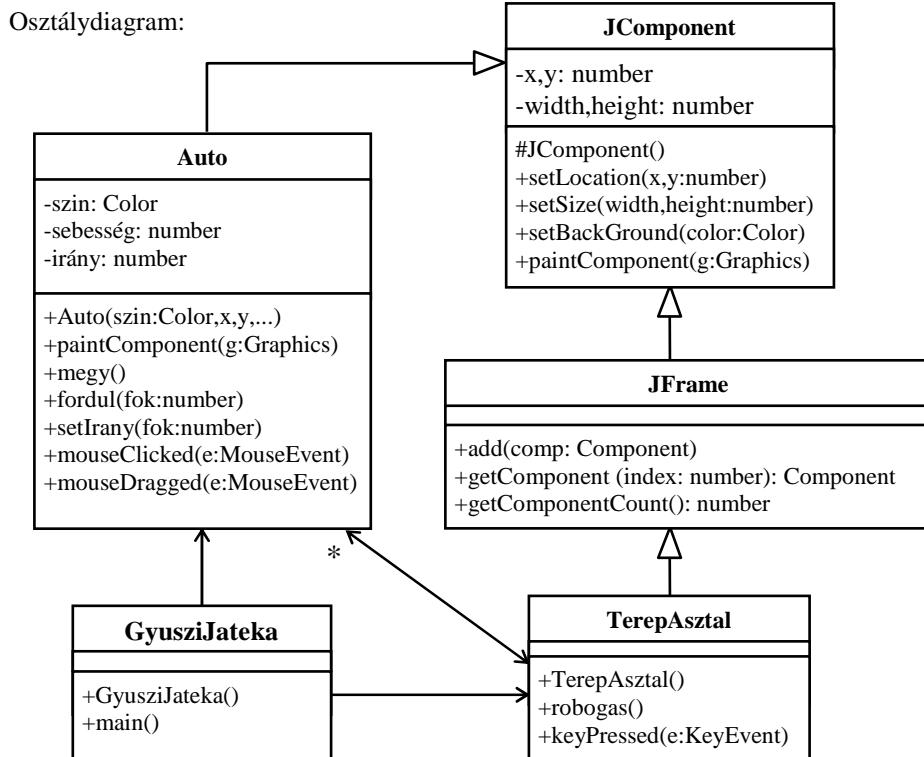
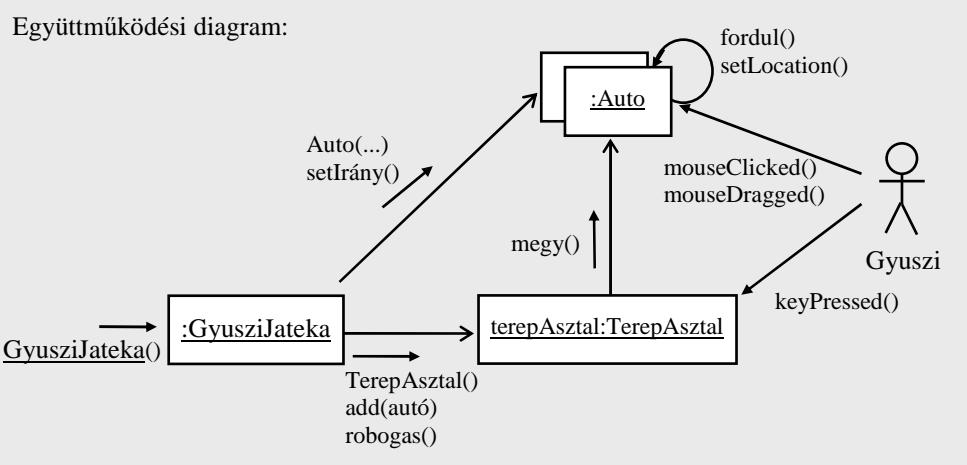
- ◆ `Terepasztal()`
Konstruktor. Létrejön a terepasztal, mérete és színe a megadott.
- ◆ `robogas()`
Robogtatja az autókat. Sorban kikéri a konténerétől az autókat, és mindenkit utasítja, hogy menjen egy sebességének megfelelő távolságot.
- ◆ `keyPressed(e:KeyEvent)`
Lekezeli a billentyűzeteseményt. Ha lenyomták az Esc-et, akkor befejezi a programot.

GyusziJateka osztály – osztályleírás

Őse: nincs. Vezérli a játékot.

Metódusok:

- ◆ `GyusziJateka()`
Konstruktor. Létrehozza a terepasztalt és az autókat. Az autókat ráteszi a terepasztalra, és felkéri a terepasztalt, hogy robogtassa őket.
- ◆ `main()`
A program belépési pontja. Létrehozza a `gyusziJateka` objektumot.



8.5. ábra. Gyuszi játéka programterve – Együttműködési és osztálydiagram

Pszeudokód

```
class Auto extends JComponent // JComponent kiterjesztése
    szin: Color
    sebesseg: number          // m/sec
    irany: number             // fokban
    terepasztal: Terepasztal // objektum, ezen van rajta
    // Konstruktor:
    Auto(aSzín:Color, x,y,aSebesseg:number, azAsztal:Terepasztal)
        szin = aSzín, sebesseg = aSebesseg, irany = 0
        terepasztal = azAsztal
        setLocation(x,y)      // az autó pozíciója a terepasztalon
        setSize(60,60)         // az autó mérete
    end Auto

    paintComponent(g:Graphics)
        autó képének kirajzolása
    end paintComponent

    // megy egy sebességének megfelelő egységet.
    megy()
        x,y: number
        x és y kiszámítása a sebesség és irány alapján.
        Nem lehet le a terepasztalról. Szükség esetén irányváltás
        setLocation(x,y)
    end megy

    fordul(fok: number)      // elfordul a megadott fokkal
        irany = irany + fok
    end fordul

    setIrany(fok: number)    // beáll a megadott irányba
        irany = fok
    end setIrany

    mouseClicked(e:MouseEvent) // az egérkattintás lekezelője
        fordul(180)
    end mouseClicked

    mouseDragged(e:MouseEvent) // az egérvonszolás lekezelője
        autó áthelyezése az egérkurzor helyére (setLocation)
    end mouseDragged
end class Auto



---


class Terepasztal extends JFrame // JFrame kiterjesztése
    Terepasztal()           // konstruktor
    setSize(700,500)         // a terepasztal méretének beállítása
    setBackground(szürke) // a terepasztal színe szürke legyen
end Terepasztal
```

```

robogas()
    index: number           // ez az index megy körbe az autókon
    auto: Auto              // aktuális kapcsolat egy autóval
    index = -1
    do
        index = index+1      // a következő autó indexe
        if index == getComponentCount() // utolsó autó után az első
            index = 0          // számozás nullától
        end if
        auto = getComponent(index) // kapcsolatfelvétel egy autóval
        auto.megy()             // az aktuális autó megy egy kicsit
        kis várakozás
    end do while true       // az autók a végtelenségig robognak
end robogas

keyPressed(e:KeyPressed) // a billentyűleütés lekezelője
    if e==Esc               // ha az Esc-et ütötték le
        vége a programnak
    end if
end keyPressed
end class TerepAsztal

```

```

class GyusziJateka // Kontroll objektum osztálya
    terepAsztal: TerepAsztal

    GyusziJateka()      // konstruktor
        terepAsztal = new TerepAsztal()
        auto: Auto
        auto = new Auto(piros,100,10,5,tereptAsztal)
        terepAsztal.add(auto)
        auto = new Auto(fekete,250,150,10,tereptAsztal)
        auto.setIrany(30)
        terepAsztal.add(auto)
        auto = new Auto(zöld,350,50,5,tereptAsztal)
        auto.setIrany(120)
        terepAsztal.add(auto)
        auto = new Auto(fehér,500,300,15,tereptAsztal)
        auto.setIrany(60)
        terepAsztal.add(auto)

    terepAsztal.robogas()
end GyusziJateka

main()
    new GyusziJateka()
end main

end class GyusziJateka

```

API osztályok leírásai

Component osztály (API) – osztályleírás (nem része a tervnek)

A JComponent a javax.swing csomag absztrakt osztálya. Származtatásra való, közvetlenül nem lehet példányt létrehozni belőle. Az osztály tartalmaz minden olyan alapmetódust, amely egy alakzatnak a képernyőn való megjelenítéséhez szükséges. A JComponent osztály kirajzoló (paintComponent) metódusa csupán egy üres téglalapot jelenít meg. Ahhoz, hogy egy adott alakzatot megjelenítsünk, a JComponent osztályból származtatnunk kell egy osztályt (itt Auto), melyben felül kell írnunk a paintComponent metódust.

Metódusok:

- ◆ `JComponent()`
Védett konstruktur, csak a leszármazott osztály használhatja. Létrehozza a komponenst.
- ◆ `setLocation(x:number, y:number)`
Áthelyezi a komponenst az (x,y) helyre. (x,y) a komponens bal felső sarkának relatív koordinátája az öt tartalmazó komponens (ablak) bal felső sarkához viszonyítva.
- ◆ `setSize(width:number, height:number)`
Beállítja a komponens méretét, vagyis szélességét (width) és magasságát (height).
- ◆ `setBackground(color:Color)`
Beállítja a komponens háttérszínét.
- ◆ `paintComponent(g:Graphics)`
Kirajzolja a komponenst (alakzatot) a képernyőre. A metódus alapértelmezésben csak egy üres téglalapot jelenít meg.
A leszármazott kirajzolásához természetesen további ismeretekhez lenne szükség, de ezzel most nem foglalkozunk. Tegyük fel, hogy megvannak a megfelelő eszközeink, hogy egyeneseket, köröket és egyéb mértani idomokat rajzolunk különböző színekkel.

JFrame osztály (API) – osztályleírás (nem része a tervnek)

Őse (nem közvetlen): JComponent

Egy ablakkeret, amely megjelenik a képernyőn. Jóval többet tud ősénél, mert ő egyúttal konténer is, amely ismeri és összetartja a benne levő látható komponenseket (például a terepasztal az autókat).

Metódusok:

- ◆ `add(comp: JComponent)`
Újabb komponens hozzáadása a kerethez. Ez lesz a legnagyobb sorszámról komponens.
- ◆ `getComponent(index:number): JComponent`
Visszaadja a megadott indexű komponens objektumot.
- ◆ `getComponentCount():number`
Visszaadja a konténerben (az ablakkeretben) levő komponensek számát.

A terv és a pszeudokód elemzése

A terv alapján elkészítettük a program pszeudokódját. Az elemzés során a tervre, az osztályleírásokra és a pszeudokódra egyaránt fogunk hivatkozni.

Az osztálydiagramon a GyusziJateka az induló objektum osztálya. Ennek az objektumnak a létrehozásáról is gondoskodnia kell valakinek, mégpedig a rendszeren kívülről. Az induló objektum létrehozásának technikája a nyelvi környezettől függ. Tegyük fel tehát, hogy végrehajtódik a GyusziJateka osztály konstruktora, és létrehoz egy :GyusziJateka objektumot (ezt a main metódus fogja elvégezni). Ez a konstruktur felelős aztán a többi objektum létrehozásáért és „beindításáért” – létrehozza a terepasztalt: `terepAsztal=new TerepAsztal()` és az autókat: `auto=new Auto(...);` az autókat ráteszi az asztalra: `terepAsztal.add(auto);` majd felkéri az asztalt a robogásra: `terepAsztal.robogas();`. Az együttműködési diagramon látható, hogy a :GyusziJateka objektum a TerepAsztal(), add(autó) és robogas() üzeneteket küldi a terepasztal:TerepAsztal objektumnak. Ezek az üzenetek tehát szerepelnek a TerepAsztal osztályában vagy annak valamelyik ősében.

Ha az együttműködési diagramon egy objektumnak üzenetet küldünk, akkor az üzenethez illeszkedő metódusnak szerepelnie kell az osztálydiagramon a megfelelő osztályban vagy annak egy ősében.

A Terepasztal osztály közvetve a JComponent osztályból származik, az tehát egy komponens: van pozíciója, mérete stb. Közvetlenül azonban a JFrame osztályból származik, amely már konténer is (bele lehet tenni komponenseket), ezért a terepasztal szintén konténer. Az autók a terepasztalon vannak. A 8.5. ábrán az egy–sok kapcsolatot a JFrame és Auto osztályok között is jelöltetük volna, mert a tárolást a JFrame végzi el. Emlékezzünk vissza, hogy a kapcsolatok öröklődnek, ennél fogva a TerepAsztal és az Auto osztályok is egy–sok kapcsolatban állnak egymással.

Az autó is egy komponens: van pozíciója, mérete, színe stb., az ō osztálya is a Component osztályból származik. Az autó tehát minden tud, amit egy komponens; ezen kívül azonban van színe, sebessége, iránya, és tud menni, fordulni stb. Felülírtuk benne a paintComponent() metódust, ezért autó formája van. A mouseClicked() és mouseDragged() egy–egy eseménylekezelő metódus. Ide kell beírnunk, mit kell tennie az autónak az események bekövetkezésekor. A megfelelő eseménylekezelő metódust a rendszer automatikusan meghívja, ha az esemény bekövetkezik.

Tesztkérdések

- 8.1. Jelölje meg az összes igaz állítást a következők közül!
 - a) A számítógépes szimuláció egy létező vagy elképzelt rendszer szoftverrel való utánzása.
 - b) A követelményfeltárásnak nincsen dokumentációja.
 - c) Az analízis dokumentációja a feladatspecifikáció.
 - d) Az osztálydiagram a programterv része.
- 8.2. Miben segítenek a feladat szövegében levő főnevek? Jelölje meg az összes jó választ!
 - a) A feladat objektumainak megkeresésében.
 - b) Az objektumok tulajdonságainak megkeresésében.
 - c) Az objektumok felelősségeinek megkeresésében.
 - d) Az egyes osztályok metódusainak meghatározásában.
- 8.3. Melyik objektum felelős a teljes program működéséért? Jelölje meg az összes jó választ!
 - a) Az alkalmazás
 - b) Valamelyik felhasználói interfész objektum
 - c) A programot vezérlő objektum
 - d) A program objektumai egyformán felelősek a működésért
- 8.4. Jelölje meg az összes igaz állítást a következők közül!
 - a) Kötegelt programmal kommunikálhat a felhasználó.
 - b) Az interaktív program és az eseményvezérelt program fogalma azonos.
 - c) Az eseményvezérelt program az aktor által indukált eseményekre reagál.
 - d) Eseményvezérelt programban az eseményeket egy központi eseményelosztó ciklus osztja el.
- 8.5. A felsoroltak közül melyek az analízis dokumentációi? Jelölje meg az összes jó választ!
 - a) Szakterületi objektummodell
 - b) Használati esetek
 - c) Képernyőterv
 - d) Programterv
- 8.6. Mi igaz az együttműködési diagramra? Jelölje meg az összes jó választ!
 - a) Az együttműködési diagramból kiolvasható, hogy valamely osztályban pontosan milyen metódusok szerepelnek.
 - b) minden egyes osztálydiagramhoz pontosan egy együttműködési diagram tartozik.
 - c) Az együttműködési diagram egy pillanatfelvétel a működő programról.
 - d) Ha az együttműködési diagramon egy objektumnak üzenetet küldünk, akkor az osztálydiagramon a megfelelő osztályban vagy annak valamelyik ősében szerepelnie kell az üzenethez tartozó metódusnak.

I. BEVEZETÉS A PROGRAMOZÁSBA

1. A számítógép és a szoftver
2. Adat, algoritmus
3. A szoftver fejlesztése

II. OBJEKTUMORIENTÁLT PARADIGMA

4. Mitől objektumorientált egy program?
5. Objektum, osztály
6. Társítási kapcsolatok
7. Öröklődés
8. Egyszerű OO terv – Esettanulmány



III. JAVA KÖRNYEZET

9. Fejlesztési környezet – Első programunk
10. A Java nyelvről

III.

IV. JAVA PROGRAMOZÁSI ALAPOK

11. Alapfogalmak
12. Kifejezések, értékadás
13. Szelekciók
14. Iterációk
15. Metódusok írása

V. OSZTÁLYOK HASZNÁLATA, KÉSZÍTÉSE

16. Objektumok, karakterláncok, csomagolók
17. Osztály készítése

VI. KONTÉNEREK

18. Tömbök
19. Rendezés, keresés, karbantartás
20. A Vector és a Collections osztály

FÜGGELÉK

- A tesztkérdések megoldásai
Irodalomjegyzék
Tárgymutató

9. Fejlesztési környezet – Első programunk

A fejezet pontjai:

1. A JBuilder letöltése, indítása
 2. A könyv melléklete
 3. A JBuilder alkalmazásböngészője
 4. JBuilder-projekt fordítása, futtatása
 5. Önálló program fordítása, futtatása
 6. A javalib könyvtár konfigurálása
 7. A javaprog projekt létrehozása
 8. Mintaprogram – Hurrá
 9. A JBuilder szövegszerkesztője
 10. JDK – Java Fejlesztői Készlet
 11. Az API csomagstruktúrája
 12. Fordítás és futtatás több osztály esetén
 13. Integrált fejlesztői környezetek
-

Ebben a fejezetben telepíteni fogjuk a JBuilder integrált fejlesztői környezetet és a könyv elektronikus mellékletét. Megismerkedünk a JBuilder alkalmazásböngészővel, s megtanuljuk futtatni a könyv két kötetének programjait. Némelyik ilyen program egész JBuilder-projekt, mások meg csak egyetlen Java forráskódóból állnak. Először azt fogjuk látni, hogyan lehet egy már megírt JBuilder projektet lefordítani és futtatni. Ezután önálló Java forráskódokkal fogunk foglalkozni – a könyv mellékletében közreadott javaprog projekttel egyszerűen lefordíthatunk és futtathatunk egy tetszőleges önálló, bármilyen környezetben megírt Java forráskódot.

Térképezze fel a két kötet tanulnivalóját: futtassa, próbálghassa minden kötet programjait; nem baj, ha még nem érzi a kódot! A könyv első kötetében nem állítunk össze projektet, csak kész projektet futtatunk; projektkészítéssel majd a könyv 2. kötete foglalkozik.

Végül szó lesz a JDK-ról, a Java programozás alapszoftveréről. A JDK ott van minden Java környezet mögött, s minden Java program fordítható és futtatható vele, csak épp meglehetősen „fapados” módszerrel.

9.1. A JBuilder letöltése, indítása

A JBuilder **integrált fejlesztői környezet**; fejlesztőeszközöket (szövegszerkesztő, fordító, futtató, nyomkövető stb.) építettek össze (integráltak) benne, és rengeteg kényelmi funkciót is belefoglaltak.

A JBuilder Personal (aktuális verziója a 8.0) a www.borland.com lapról tölthető le, és ingyenesen használható. Több platformra is elkészült: Windowsra, Linuxra és Solarisra. A letöltendő tömörített állomány mérete nagyjából 60 MB. Ajánlatos a dokumentációt is letölteni – további 60 MB-ot. Letöltéskor a felhasználónak regisztrálnia kell magát; válaszul a Borland e-levélben aktivációs állományt küld, s abban a regisztrációs kulcsot.

A JBuilder erőforrásigénye:

- ◆ Memória: 256 MB
- ◆ Lemezterület: JBuilder: ~120 MB, dokumentáció: ~70 MB
- ◆ Processzor: Intel Pentium II 233 MHz vagy ezzel kompatibilis.

A telepítés előtt vizsgálja meg, van-e elegendő hely a lemezén!

Telepítse először a JBuilder 8.0 fejlesztőeszközt! Futtassa az install programot; a szoftver helyének adja meg valamelyik lemezének JBuilder8 mappáját (pl. C:/JBuilder8)!

Ezután telepítse a JBuilder 8.0 dokumentációját, vagyis a segítséget (helpet)! A JBuilder mappájának adja meg azt a mappát, ahol a JBuildert telepítette (pl. C:/JBuilder8), a JDK (Java Development Kit, Java Fejlesztői Készlet, lásd később) mappájának pedig az az alatti jdk1.4 mappát (pl. C:/JBuilder8/jdk1.4).

A JBuilder első indításakor meg kell adni az e-levélben kapott aktivációs állományt.

Megjegyzés: A könyvtárneveket elválasztó jel hol / lesz, hol meg \, mert lehetetlen következetesnek lenni. Az elválasztójel operációsrendszer-függő, és a JBuilder is mindenkorán kicseréli a \ jelet /-re. Egy programban ajánlatos a / jelet használni.

9.2. A könyv melléklete

Töltsé le a könyv elektronikus mellékletét a könyv hátán megadott címről! A melléklet a két kötet közös melléklete, neve `javaproj.zip`, mérete kb. 4 MB. Bontsa ki a fájlt, és tegye a C lemezegység főkönyvtárába! Ha más helyre teszi, akkor a könyv és a `javaproj` projekt hivatkozásai értelemszerűen átírandók.

Kibontás után a 9.1. ábrán látható könyvtárstruktúrát kapja. A mappákat vastag betűvel szedtük. Az első 3 mappa neve aláhúzással kezdődik, hogy az ábécé szerint előre kerüljenek. A

_MyPrograms mappa a készülő, saját programokat tartalmazza; hozunk létre almappákat a programok csoportosítására (például tanuló neve, dátum vagy téma szerint)!

A mappák mellett megjegyzésben tüntettük fel a tartalmukat. Bizonyos mappákat csak a könyv 2. kötete használ (pl. icons, images, sounds).

```
c:\javaproj
    _MyPrograms          // saját programjaink helye
    _OTPJava1            // az 1. kötet melléklete
        Esettanulmanyok
        Feladatmegoldasok
        Mintaprogramok
    _OTPJava2            // a 2. kötet melléklete
        Esettanulmanyok
        Feladatmegoldasok
        Mintaprogramok
    doc                  // dokumentációk
    original             // fontosabb fájlok eredeti változata
    icons                // ikonok
    images               // képek
    javaproj_bak         // a javaproj projekt forrásfájljainak másolata
    javaproj_classes     // a javaproj projekt class-fájljai
    javaproj_src          // a javaproj projekt alapértelmezett forrásfájljai (üres)
    lib                  // segédkönyvtárak
        javalib.jar        // a javalib könyvtár tömörítve
        javalib_src.jar    // a javalib könyvtár forráskódjai tömörítve
    sounds              // hangfájlok
    work                 // munkakönyvtár (pl. fájlkezeléshez)
    javaproj.jpx         // önálló forráskódok projektfájlja
```

9.1. ábra. A könyv mellékletének könyvtárstruktúrája

A könyv mellékletében a programoknak kétféle lehet a „kiszerelésük”:

- ◆ **Önálló Java forráskód** (kiterjesztése java): Egyetlen java kiterjesztésű, önállóan fordítható és futtatható forrásállomány. Az első kötetben szinte csak ilyen programokkal foglalkozunk. minden forrásállományt külön mappába tettük. A program fordításához és futtatásához be kell töltenünk a Java forrásállományt.
- ◆ **Projekt** (a projektfájl kiterjesztése jpx): Nagyobb, több állományból álló program. Az állományok lehetnek forráskódok, adat-, kép-, hangfájlok stb. Mindezek az állományok egy projektkönyvtárnak nevezett könyvtár alatt sorakoznak, könyvtárstruktúrába szerelve. A projekt állományait egy projektfájl fogja össze (JBuilderben egy jpx kiterjesztésű fájl). A program fordításához és futtatásához elegendő a projektfájlt betölteni.

Projektkészítéssel majd a 2. kötet foglakozik – ebben a könyvben csak futtatjuk, s ha kell, fordítjuk őket.

Először nagy vonalakban áttekintjük a JBuilder alkalmazásböngésző elemeit, majd lefordítunk és futtatunk egy JBuilder-projektet és egy önálló Java forráskódot.

9.3. A JBuilder alkalmazásböngészője

A **projekt** (**project**) egy szoftver fejlesztésében használt, logikailag összetartozó állományok és környezeti beállítások gyűjteménye. minden projektnek van egy könyvtára; ebben a könyvtárban van a projektet leíró fájl, itt vannak továbbá a projekt alkönyvtárai és állományai.

Például:

```
C:/javaprog/_OOTPJava1/Esettanulmanyok
GyusziJateka
    GyusziJateka.jpx    // projektfájl
    src                  // a projekt forrásfájljai
    classes              // a projekt bájtkódjai
    ...
    ...
```

A GyusziJateka projekt könyvtára: C:/javaprog/_OOTPJava1/Esettanulmanyok/GyusziJateka; projektállománya: GyusziJateka.jpx.

Nyissa meg a GyusziJateka projektet:

- ◆ *File/Open Project...* Válassza ki a következő állományt:
C:/javaprog/_OOTPJava1/Esettanulmanyok/GyusziJateka/GyusziJateka.jpx.

A projektet nem kell lefordítani, mert a lefordított kód része a mellékletnek. Ha kettőt kattintunk a projektfa GyusziJateka.java elemén (9.2. ábra), akkor a tartalompanelen megjelenik a forráskód. Futtassa a GyusziJateka projektet:

- ◆ *Run/Run Project (F9)*

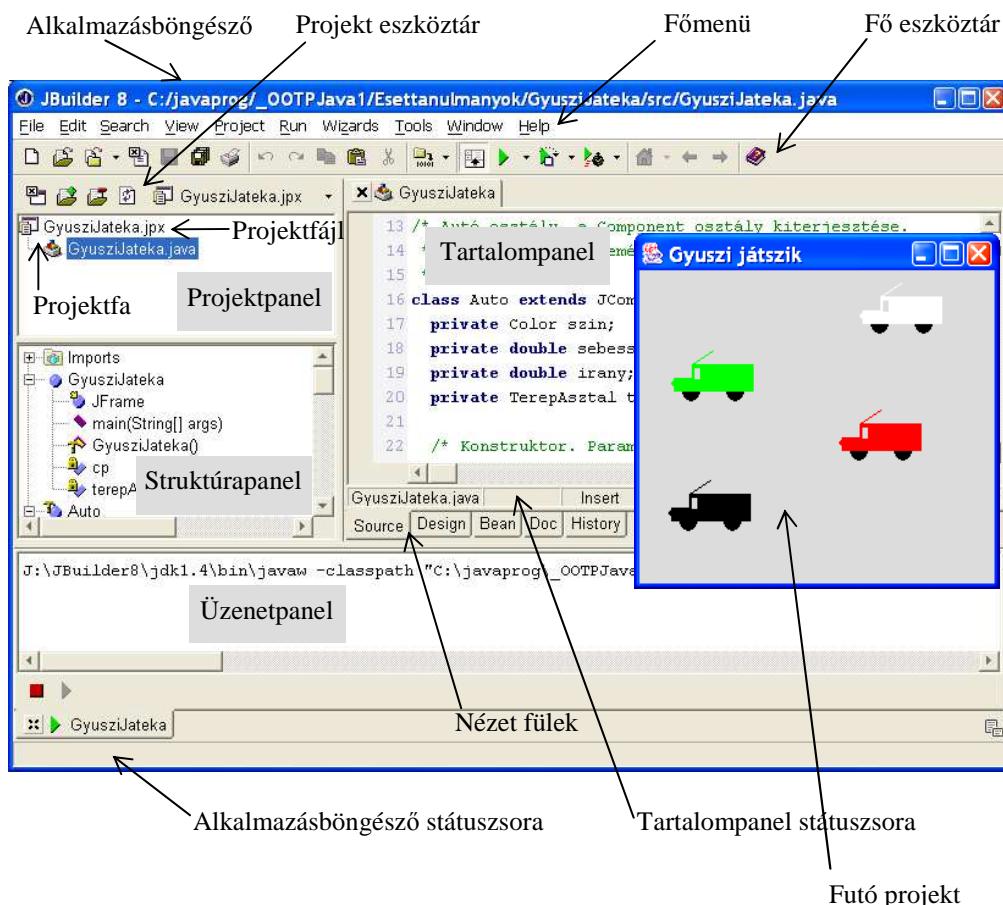
A JBuilder fő ablakát alkalmazásböngészőnek nevezzük (application browser). Ez látható a 9.2. ábrán: most éppen a GyusziJateka projekt fut benne. A program egy ablakot jelenít meg, az üzenetpanelre nem ír semmit.

Az alkalmazásböngésző részei:

- ◆ **Főmenü** (Main menu). Menüpontok: File, Edit, Search, View, Project, Run, Wizards, Tools, Window, Help.
- ◆ **Fő eszköztár** (Main toolbar). Innen érhetők el az alkalmazásböngészőre vonatkozó főbb menüpontok. A gombok funkció szerint vannak csoportosítva: File, Edit, Search, Build,

Run, Debug, Navigation és Help. A különféle csoportok elrejthetők, illetve újra láthatóvá tehetők a View/Toolbars menüpontban.

- ◆ **Projektpanel** (Project pane). Ez mutatja a projekt elemeit, fastruktúrába szervezve. A panelen van a **projekt eszköztára** (project toolbar). Mindig van egy aktuális projekt: az, amelyen dolgozunk – ezt a projektet a **projektválasztó** listából lehet kiválasztani. A **projektfa** minden projekt elemeit mutatja. A + és – ikonnal elemeket adhatunk a projekthez, illetve elemeket vehetünk le róla.



9.2. ábra. A GyusziJateka projekt a JBuilderben

- ◆ **Tartalompanel** (Content pane). A tartalompanelen ott van az összes nyitott állomány (a megnyitott állományoknak egy-egy fül felel meg). Egy állományt úgy nyithatunk meg, hogy duplán kattintunk a projektfa megfelelő elemén. Ha a JBuilder felismeri a megjelenítendő állományt, akkor azt a szokásos módon jeleníti meg – ilyenek például a java, class, txt, html, xml, jpg, gif, wav, zip, és jar kiterjesztésű állományok. A megnyitott

állományok között minden van egy aktuális, ehhez kapcsolódik a struktúrapanel és a tartalompanel státuszsora.

A projekt elemei és a nyitott állományok elvileg függetlenek egymástól. A projekt elemei megnyithatók, de megnyithatók más, a projekthez nem tartozó állományok is (*File/Open File...*).

A tartalompanel részei:

- **Fájlfülek** (felül). A fülre való kattintással kiválasztható az aktuális állomány.
- **Nézetfülek** (alul: Source, Design ...). Az aktuális állománynak különböző nézetei lehetnek. A megfelelő fülre kattintva nézetet válthatunk. A Source fülön található a forráskód; a Design fület vizuális tervezésnél használjuk stb.
- **A tartalompanel státuszsora.** Információk az aktuálisan szerkeszthető állományról: az állomány neve, a kurzor pozíciója (sor:oszlop), Modified, Insert/Overwrite.
- ◆ **Struktúrapanel** (Structure pane). A tartalompanel aktuális elemének struktúráját mutatja – Java forrásfájlban a csomag- és típusdeklarációkat, valamint a különféle típusok tagdeklarációit.
- ◆ **Üzenetpanel** (Message pane). Az üzenetpanel lapjain jelennek meg a hibaüzenetek, valamint a keresések, futtatások és nyomkövetések információi, eredményei. Ez a program „konzolablaka” is: itt kell tehát megadni a program beviteli adatait (például a Console osztály metódusaival bekért adatokat). **Adatbekérés előtt az üzenetpanelt fókuszba kell hozni!**

Az egyes tevékenységekhez külön lap nyílik:

- Fordítás (Compiler). A lap megjeleníti legutóbbi fordítás hibalistáját, ha a fordítás nem sikerült volna. A hibaüzenetre kattintva a hibás sorra állhatunk.
- Futtatás (Az aktív program neve). Külön lap nyílik minden futó programszálnak.
- Nyomkövetés (Debugger).
- Keresés (Search Results). A lapon a legutóbbi keresés eredménye látható.

Minden lapon két gombot találunk: az egyikkel leállítható a szál, a másikkal újraindítható. Az üzenetpanel lapjai törlhetők: *Jobb egérgomb a lap alján/Remove...* Az üzenetpanel elrejthető: *View/Messages*.

- ◆ **Az alkalmazásböngésző státuszsora** (Status line): Egysoros információ az alkalmazás állapotáról.

Kedvencek

Adja hozzá a kedvencekhez a javaprojekt mappát, hogy könnyebben odatajáljon! Ezt az *File/Open Project...* vagy a *Project/Add Files...* ablakban található *Favorites* (kedvencek, jele: szívecske) eszközgombbal teheti meg. A kedvencek használata megkönnyíti a munkát.

9.4. JBuilder-projekt fordítása és futtatása

Nézzük meg egy kicsit részletesebben, hogyan fordítható és futtatható le egy JBuilder-projekt! Közben keressen a könyv mellékletében és a JBuilder mintaprogramjai között további projekteket, és futtassa őket!

Futtassa például a C:/javaprog/_OOTPJava2/Esettanulmanyok/KissDraw/KissDraw.jpx projektet!

Megjegyzés: Ne ijedjen meg! Egyelőre csak használnia kell ezt a programot. Ilyen programot csak a könyv 2. kötetében fogunk írni.

Projekt megnyitása

- ◆ *File/Open Project...* Megjelenik az *Open Project* dialógusablak. Az ablakban csak a projektfájlok jelennek meg (jpr vagy jpx kiterjesztés). Keressük meg a lemezen, és válasszuk ki a kívánt projektfájlt!
- ◆ *File/Open File... (Ctrl + O)* Megjelenik az *Open File* dialógusablak. Az ablakban feltűnik az összes állomány. Keressük meg a lemezen, és válasszuk ki a kívánt projektfájlt!

Ha projektfájlt nyitunk meg, akkor ez a két funkció csak a megjelenített fájlokban tér el egymástól.

Projekt(ek) bezárása

- ◆ *Projekt eszköztár/Kék X* Bezárja az aktuális projektet.
- ◆ *File/Close Projects...* Bejelöljük a bezárandó projekte(ke)t.

Projekt fordítása

- ◆ *Project/Make Project "[Projektfájl]" (Ctrl + F9, Fő eszköztár/Sárga téglalap/Make)*. Lefordítja a projektben az időközben módosított forrásállományokat (azokat, amelyeknek a dátuma későbbi, mint a már lefordított class állományé). Ha szükséges, akkor az összes forráskódot lefordítja. Fordítás előtt elmenti a forráskódot, ha azt megváltoztattuk volna a szövegszerkesztőben.
- ◆ *Project/Rebuild Project "[Projektfájl]" (Fő eszköztár/Sárga téglalap/Rebuild)*. Lefordítja a projekt összes forrásállományát, feltétel nélkül. A megváltozott forráskódokat előbb lemezre menti.

Mindkét esetben megjelenik egy, a fordítás eredményességéről szóló üzenet az alkalmazásböngésző státuszsorában. Például: *Build succeeded with 1 file(s) built. Build took 1 seconds.* Vagyis: 1 állomány fordítása sikeresen befejeződött. A fordítás 1 másodpercig tartott.

Projekt futtatása

- ◆ *Run/Run Project (F9, Fő eszköztár/Zöld háromszög/[Futási konfiguráció]).* Futtatja a projektet. Ha a projekt forráskódjai még nincsenek lefordítva, akkor lefordítja őket.

Egy projekt tulajdonságai közt ott van a lefordítandó projekt típusa (alkalmazás/applet...), s ha alkalmazásról van szó, akkor a `main` metódust tartalmazó osztály neve is. Egy kész projektben (így a könyv projektjeiben is) minden tulajdonság úgy van beállítva, hogy a projekt futtatható legyen, a projektek beállításaival ezért remélhetőleg nem kell törödnünk.

A könyv bizonyos projektjei megkövetelik a `javalib` könyvtár jelenlétét; ez is benne van a könyv mellékletében (`javaproj/lib/javalib.jar`). Ilyenkor a megfelelő könyvtárat hozzá kell adni a környezethez (lásd a fejezet „A `javalib` könyvtár konfigurálása” pontját).

9.5. Önálló program fordítása, futtatása

A JBuilderben csak projekt keretein belül lehet programot írni. A programfejlesztést a programozó általában egy projekt összeállításával kezdi: különböző könyvtárakat, fájlneveket és alapértelmezéseket kell megadnia, beállítania. Nagyobb program fejlesztésekor a projekt nagy segítséget ad. Más esetekben viszont kényelmetlen és értelmetlen lehet külön projektet és projekttartozékokat létrehozni – például akkor, ha

- ◆ mindössze egyetlen, rövid Java forráskódot készítünk, vagy ha
- ◆ egy meglévő Java forráskódot fordítunk és futtatunk. Gyakori eset, hogy a Java forrás-kódhoz nincs projektfájl vagy ha van, akkor az más fejlesztőrendszerből való. A különböző Java fejlesztőkörnyezetek projektállományai általában nem kompatibilisek egymással. Egy szabványos Java forrásállomány független a fejlesztési környezettől.

Könyvünkben az 1. kötet programjaihoz és a 2. kötet programjainak nagyjából 80%-ához nem tartozik külön projektfájl.

Egy önálló Java forrásállományt többféleképpen fordíthatunk és futtathatunk. Legegyszerűbb, ha a kívánt forrásállományt hozzáadjuk a könyv mellékleteként megadott `javaproj` projekt-hez.

Példaként futtassa a 14. fejezet `Csillag.java` programját! A feladat a következő:

Írunk ki a konzolra 10 darab csillagot egy sorba!

Nyissa meg a `javaproj` projektet:

- ◆ *File/Open Project... Válassza ki:*

`C:/javaproj/javaproj.jpx`

Nyissa meg és adja a projekthez a Csillag.java forráskódot:

- ◆ Projekt eszköztár/+ Válassza ki:

C:/javaproj/_OOTPJava1/Mintaprogramok/14Iteraciok/Csillag/Csillag.java

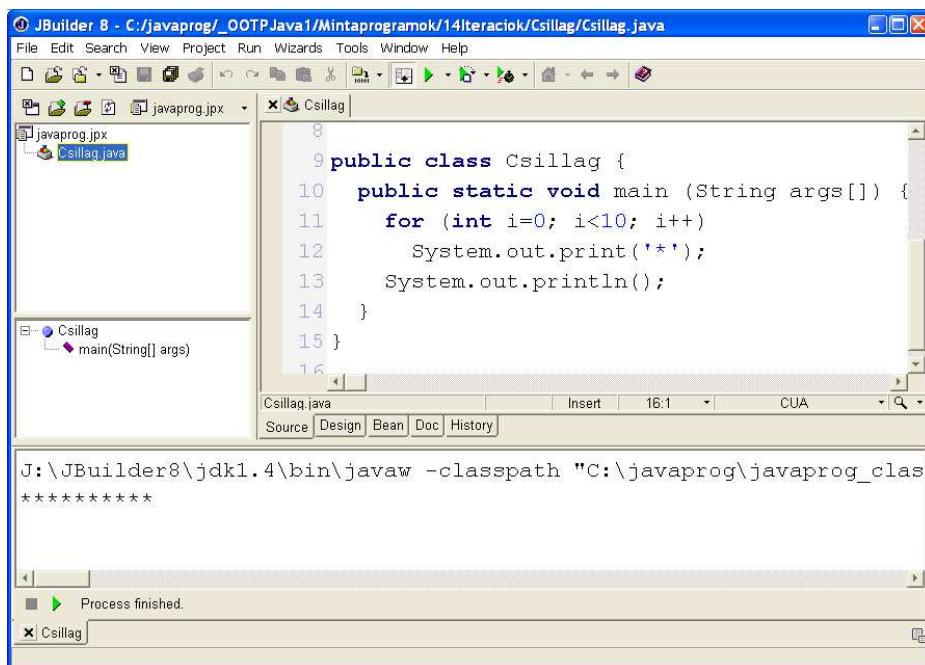
Ha kettőt kattintunk a projektfán a Csillag.java névre, akkor a forráskód nyomban megjelenik a tartalompanelen.

Fordítsa le és futtassa a Csillag.java programot (a lefordított kód nem része a könyv mellékletének, futtatáskor azonban automatikusan lezajlik a fordítás):

- ◆ Jelölje ki a forrásprogramot a projektfán! Jobb egérkomb/Run using "javaproj"

A 9.3. ábra mutatja a Csillag.java forrásprogram futását. Az üzenetpanelen megjelenik 10 darab csillag.

Megjegyzés: A programot egyelőre csak futtatnia kell!



9.3. ábra. A Csillag.java önálló Java program a JBuilderben

A javaproj projekt különálló forráskódok fejlesztéséhez (fordításához és futtatásához) **használható**. A javaproj.jpx projektfájl a könyv mellékletéhez tartozik, egyszerűen megnyitható és használható. Nem kell bezárni; jó, ha minden „kéznél van”.

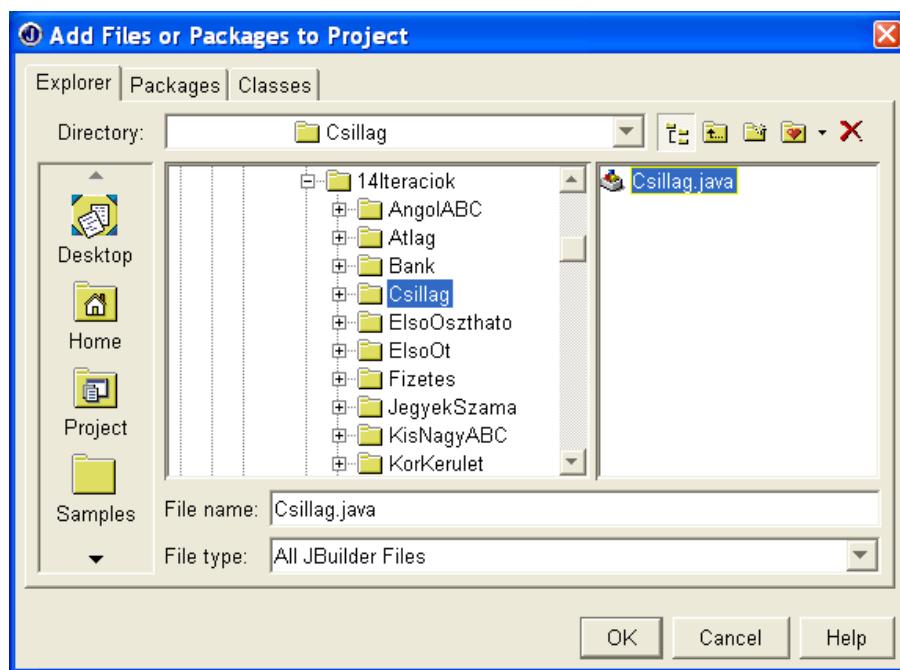
A `javaproj` projekthez hozzáadott önálló forrásprogramok nem fordíthatók és futtathatók projektként, vagyis rájuk nem alkalmazhatók a *Project/Make Project*, a *Project/Rebuild Project*, valamint a *Run/Run Project* menüpontok.

Most egy kicsit részletesebben megmutatjuk, hogyan kell egy Java forrásállományt kiválasztani vagy létrehozni, majd a `javaproj` projekthez hozzáadni, fordítani és futtatni.

Elem hozzáadása a projekthez

- ◆ *Project/Add Files/Packages...* vagy *Projekt eszköztár/+* Megjelenik a 9.4. ábra dialógusablaka. Az Explorer (böngésző) fül segítségével kerességhetünk a lemezen. Ha már meglévő Java forráskódot akarunk hozzáadni a projekthez, akkor egyszerűen csak kiválasztjuk. Ha egy új forráskódot akarunk létrehozni, akkor írjuk be a *File name* beviteli mezőbe a kívánt nevet a java kiterjesztéssel, és nyomjuk le az Ok gombot. Az önállóan futó programok forrásállományait ajánlatos külön könyvtárba tenni – új könyvtárat létrehozhatunk itt a böngészőben is. Ha nem létező fájlnevet adunk meg, akkor a rendszer megkérdezi, hogy létre akarjuk-e hozni. A kiválasztott vagy létrehozott elem ezután megjelenik a projektfán.

- ◆ Új Java forráskód létrehozásakor ne felejtse el megadni a **java kiterjesztést!** Ha nem adunk meg kiterjesztést, akkor az `.txt` lesz, s az nem kezelhető Java forráskódként.



9.4. ábra. Elem hozzáadása a projekthez

Megjegyzések:

- Egy projekthez elvileg bármilyen állományt (szöveget, bájtkódot, képet, hangot, HTML lapot...) hozzáadhatunk, de mi most Java forrásállományt szeretnénk fordítani és futtatni.
- Egy Java forráskódot a *File/Open* menüponttal is megnyithatunk, a projekthez való hozzáadás nélkül. A projekt aktuális beállításai ilyenkor is érvényesek, de körülményesebb a fordítás és futtatás (futtatáskor például nincs automatikus fordítás).

Elem lekapcsolása a projektről

- ◆ *Project/Remove from Project...* vagy *Projekt eszköztár/–* Előtte ki kell jelölni a projekt elemet a projektfán. A fájl nem törlődik a lemezről, csupán kikerül a projektből.

Gyakorlásnéképpen kapcsolja le a programot a projektről, majd újra adja hozzá!

Önálló forrásfájl fordítása

A `javaproj` projekt nem fordítható, mert a forrásprogramok nem a projekt forráskönyvtárában (`C:/javaproj/javaproj_src`) helyezkednek el, hanem bárhol a lemezen. A forráskódokat egyedileg kell fordítani. Lehetőségek:

- ◆ A projektfán jelöljük ki a forrásfájlt! *Jobb egérgomb/Make*
- ◆ Jelenítsük meg a tartalompanelen a forrásfájlt! Ha előzőleg már megjelenítettük, akkor válasszuk ki a tartalompanelen (kattintsunk a névének megfelelő fülre)! *Project/Make "[Forrásfájl]" (Ctrl + Shift + F9)*

Bájtkódok törlése

A lefordított class állomány az *Output Path* helyen keletkezik – a `javaproj` projektben a `C:/javaproj/javaproj_classes` könyvtárban. Mivel a `javaproj` projekt minden bájtkódot ide tesz, azért ajánlatos ezt a könyvtárat időnként kiüríteni:

- ◆ *Jobb egérgomb/Clean*. Ha előtte a projektfájl volt kijelölve, akkor a projekt összes bájtkódja törlődik; ha egy forráskód volt kijelölve, akkor törlődik a neki megfelelő class állomány.

Önálló forrásfájl futtatása

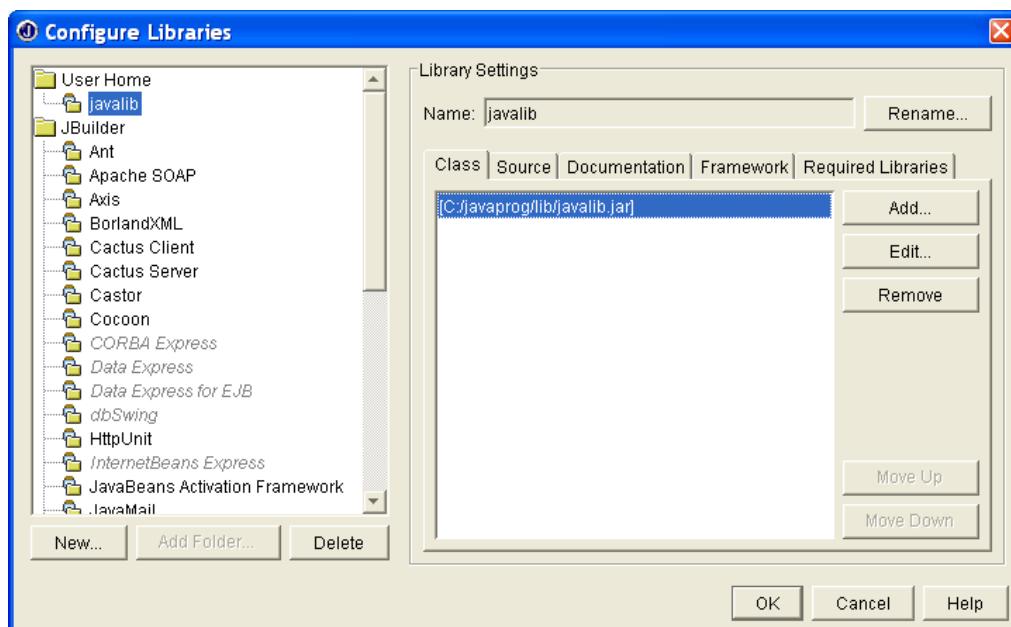
A `javaproj` projekt nem futtatható; a hozzáadott forrásprogramot csak egyedileg lehet futtatni. Lehetőségek:

- ◆ *Run/Run "[Forrásfájl]"*
- ◆ *Jobb egérgomb/Run using "javaproj"*. Előzetesen ki kell jelölni a forrásprogramot a projektpanelen.

Megjegyzés: Ha valamiért használhatatlanná válna a javaprojekt, akkor a javaproj_bak könyvtárban megtaláljuk a eredeti, sérzetlen jpx állományt.

9.6. A javalib könyvtár konfigurálása

Az eddig futtatott programok minden rendszerrel együtt szállított osztályokra támaszkodtak. Más programok azonban „nem érik be” az alapértelmezésben jelen levő API (Application Programming Interface) könyvtárakkal – a benne levő csomagok osztályaival –, hanem további könyvtárakra is hivatkoznak. A könyv programjainak többsége használja például a javalib könyvtárat – az is benne van a mellékletben. A javalib könyvtárban van az extra.Console osztály; a benne szereplő statikus metódusok segítségével adatokat kérhetünk be a konzolról (a Java írói nem gondoltak a kezdő programozókra).



9.5. ábra. A javalib könyvtár konfigurálása

A javaprojekt úgy van beállítva, hogy támaszkodjék a javalib könyvtárra. A tényleges használhatósághoz azonban be kell építenünk ezt a könyvtárat a JBuilder környezetébe, s azután a JBuilder már emlékezni fog rá. Válasszuk ki a *Tools/Configure Libraries...* menüpontot! Válasszuk aktuális könyvtárnak a *User Home* könyvtárat, majd nyomjuk le a *New...* gombot (9.5. ábra)! Megjelenik a *New Library Wizard* ablaka. Adjuk meg a következő adatokat:

- ◆ *Name* (a könyvtár neve): javalib
- ◆ *Library Path* (a könyvtár útvonala):
 - *Class*: javaproj/lib/javalib.jar // class fájlok könyvtára tömörített formában
 - *Source*: javaproj/lib/javalib_src.jar // java fájlok könyvtára tömörített formában

Most már a két kötet összes programja futtatható – vagy önálló projektként, vagy a javaproj projekt segítségével.

Próbaként futtassa a 20. fejezet ValosSzamok.java programját! A feladat a következő:

Kérjünk be tetszőleges sok valós számot 0 végig! Ezután írjuk ki

- a számokat a bevitel sorrendjében!
- a számokat növekvő sorrendben!
- a legkisebb és a legnagyobb számot!

Végül keressünk meg egy számot a bevittek között!

The screenshot shows the JBuilder 8 IDE interface. The title bar says "JBUILDER 8 - C:/javaproj/_OOTPJava1/Mintaprogramok/20Vector/ValosSzamok/ValosSzamok.java". The left pane shows a project named "javaproj.jpx" with a file "ValosSzamok.java" selected. The right pane shows the Java code:

```

17 double szam;
18 // Számok bekérése:
19 while (true) {
20     szam = Console.readDouble("Szám: ");
21     if (szam==VEGJEL)
22         break;
23     szamok.add(new Double(szam));
24 }

```

Below the code editor is a status bar with tabs for "Source", "Design", "Bean", "Doc", and "History". The status bar also displays the current time as 18:1 and the current user as CUA. The bottom pane is a terminal window showing the program's output:

```

J:\JBUILDER8\jdk1.4\bin\javaw -classpath "C:\javaproj\javaproj_classes;C:\
Szám: 56.7
Szám: 3
Szám: 44.1
Szám: 0
56.7 3.0 44.1
A számok: [56.7, 3.0, 44.1]
A számok rendezve: [3.0, 44.1, 56.7]
Legkisebb : 3.0
Legnagyobb: 56.7
Keresendő szám: |

```

9.6. ábra ValosSzamok.java – Beolvasás konzolról

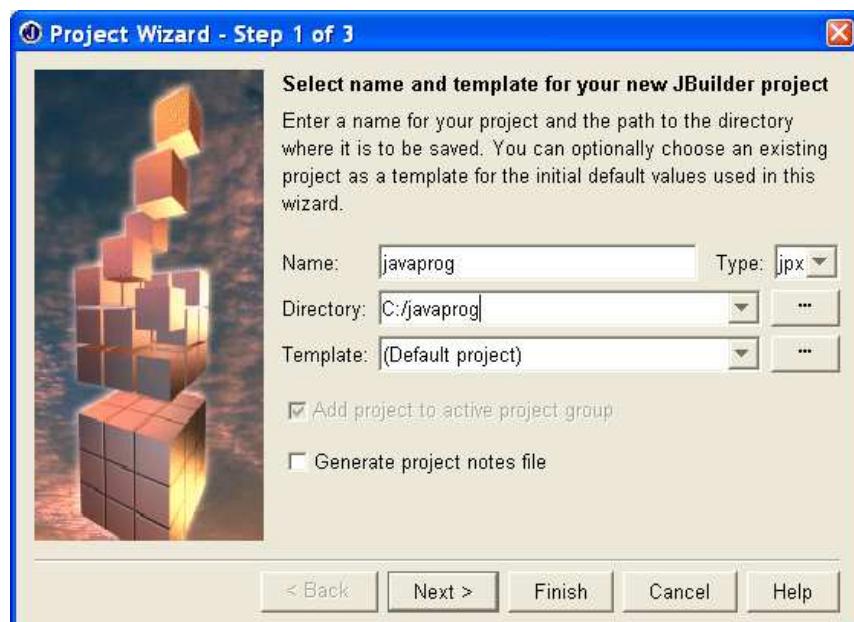
A program futását a 9.6. ábra mutatja. Az üzenetpanelen megjelenik a "Szám: " szöveg, jelezve, hogy a program bevitelre vár. Át kell váltanunk az üzenetpanelre, és be kell ütnünk egy valós számot. A program ezután kéri a következő számot, egészen addig, amíg a 0 végjelet be nem ütjük. Erre a program befejezi a számok gyűjtését, és kiírja a kért adatokat (az eredeti sorrendben kétféle kiírás történik). Végül kér egy számot, hogy megkereshesse azt a beütött számok között. Az ábrán a kurzor éppen bevitelre vár. A program első sora: `import extra.Console;` (nem látszik), jelezve, hogy használni akarja a `Console` osztályt.

Próbálja meg futtatni a 2. kötet esettanulmányait is! A `javalib`_projekt kivételével itt minden mappában egy futtatható projekt van (a `javalib`_projekt önállóan nem futtatható osztálykönyvtár).

9.7. A javaprojekt létrehozása

A javaprojekt a könyv melléklete, azt nem kell létrehozni. S mivel a JBuilder indításkor automatikusan megnyitja az utoljára használt projekteket, azért a `javaproj` projekttel gyakorlatilag nincs semmi dolgunk... Ha mégis létre szeretné hozni a `javaproj` projektet, kövesse a következő útmutatást!

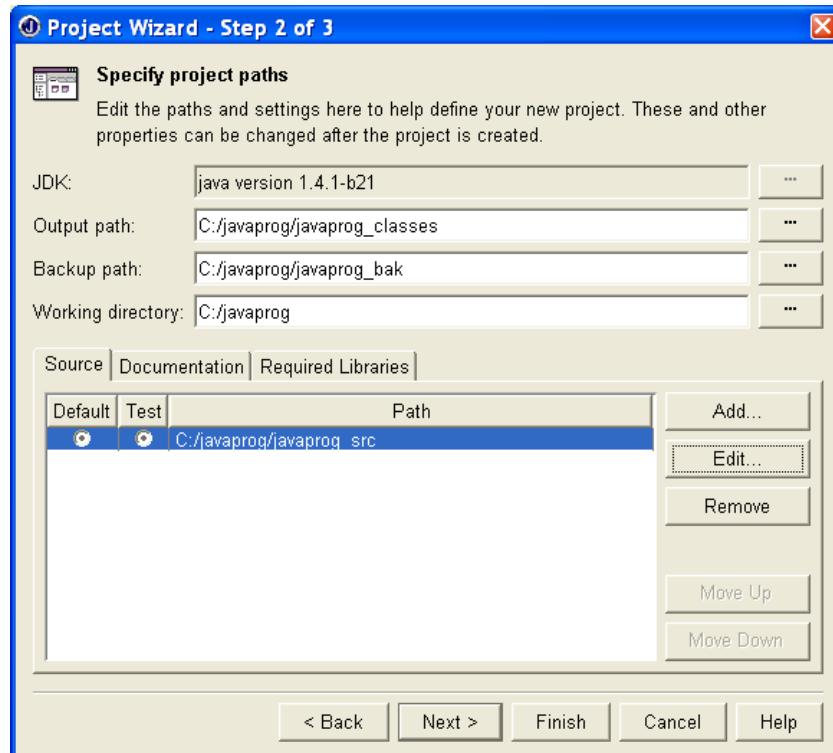
- ◆ *File/New Project...* Megjelenik a projekt varázsló. Állítsa be a mezőket az ábrákon megadott módon!



9.7. ábra. Projekt varázsló – 1. lépés

1. lépés (9.7. ábra):

- ◆ *Name*: javaprog. A projekt neve. Ez lesz a projektfájl neve is, jpx kiterjesztéssel.
- ◆ *Directory*: C:/javaprog A projekt könyvtára. Válasszuk ki a C:\ főkönyvtárat, s az majd automatikusan kiegészül a projekt nevével. Ajánlatos a mappa és a projektfájl nevét egyformának választani.
- ◆ *Template*: (Default project) Projektminták: ennek alapján jön létre az új projekt.



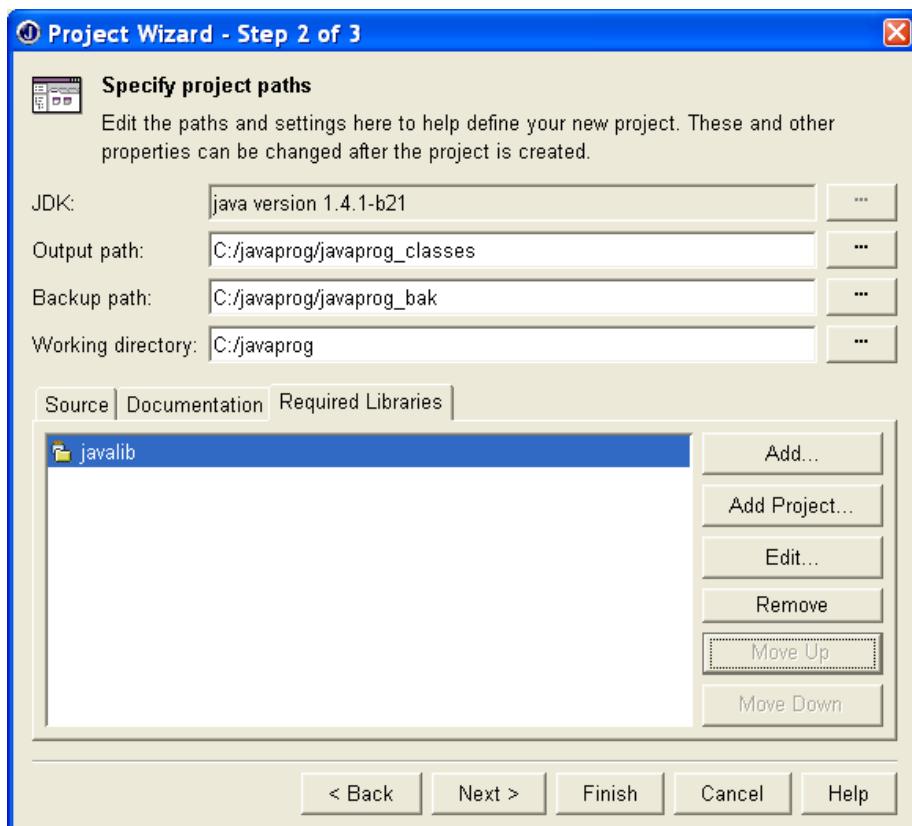
9.8. ábra. Projekt varázsló – 2. lépés, útvonalak beállítása

2. lépés (9.8. ábra):

- ◆ *JDK*: A használt JDK aktuális verziója. A JBuilder Personalban ez nem módosítható.
- ◆ *Output path*: C:/javaprog/javaprog_classes. Itt keletkeznek a projektben lefordított bájtkódok. Időnként ajánlatos törölni őket.
- ◆ *Backup path*: C:/javaprog/javaprog_bak. Itt keletkeznek a projektben szerkesztett forráskódok biztonsági másolatai.
- ◆ *Working directory*: C:/javaprog. Munkakönyvtár – a projekt forráskódjaiban megadott relatív útvonalak kiinduló könyvtára.

- ◆ *Source:* C:/javaproj/javaproj_src. Csak az osztályvarázslóval létrehozott forráskódok keletkeznek itt. Megadása kötelező, de ebben a projektben nem használjuk – ez a mappa tehát általában üres.
- ◆ *Required Libraries:* Itt kell beállítani a programban használt könyvtárakat (a CLASSPATH útvonalait, lásd a JDK leírását). A 9.9. ábra a javalib könyvtár beállítását mutatja. A könyvtára(ka)t előzőleg a New segítségével vagy a Tool/Configure Libraries... menüpontban be kell állítani. A különféle projektekben a már megadott könyvtárakból válogatunk. Ha a környezetben megváltoztatjuk egy könyvtár útvonalát, akkor a beállítás a JBuilder összes projektjét érinti!

Egy projekt ajánlott mappanevei: classes, bak és src. A javaproj projektben azért adtunk tőlük különböző neveket, hogy a mappák ábécé rendben kövessék egymást – így áttekinthetőbb a javaproj könyvtár.



9.9. ábra. Projekt varázsló – 2. lépés, szükséges könyvtár megadása

3. lépés:

Itt nincs mit tennünk. Nyomjuk meg nyugodtan már a 2. lépés végén a *Finish* gombot! A projekt a c:/javaproj könyvtárban keletkezik, javaproj.jpx névvel.

Ezt a javaproj projektet csak egyszer kell létrehozni. Ezután a megfelelő Java forrásállományt csak hozzá kell adni a projekthez, illetve le kell venni róla.

A javaproj projektet nem kell bezárni. Egyszerre több projekt is nyitva lehet.

9.8. Mintaprogram – Hurrá

Ebben a pontban beírunk és elemzünk egy nagyon egyszerű mintaprogramot, majd lefordítjuk és futtatjuk is.

Feladat – Hurrá

Írunk olyan Java programot, amely konzolra írja a következő szöveget:
Hurrá, fut az első Java programom!

A mintaprogram megtalálható a C:/javaproj/_OOTPJava1/Mintaprogramok/09Környezet/Hurra mappában. Készítse el a programot önállóan, a _MyPrograms/Hurra mappában!

- ◆ *File/Open Project...* C:/javaproj/javaproj.jpx. Csak akkor kell betölteni a javaproj projektet, ha még nem volna betöltve!
- ◆ *Projekt eszköztár/+ C:/javaproj/_MyPrograms/Hurra/Hurra.java*

A forráskódot a Hurra.java szöveges állomány fogja tartalmazni.

Forráskód

```
// Első Java programom //1
public class Hurra { //2
    public static void main(String[] args) { //3
        System.out.println("Hurrá, fut az első Java programom!"); //4
    } //5
}
```

A program beírása

Gépelje be a forráskódot! Pontosan azt gépelje be, amit megadtunk – a kisbetű például kisbetűnek, a nagybetű nagybetűnek!

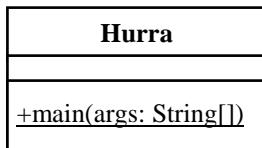
A szövegszerkesztő funkcióit és lehetőségeit a következő pont tárgyalja.

A mintaprogram elemzése

A `Hurra.java` forrásállomány minden összetevője egyetlen osztálynak, a `Hurra` osztálynak a deklarációját tartalmazza. Az osztály UML ábrája a 9.10. ábrán látható. Ennek az osztálynak egyetlen publikus osztálymetódusa van, a `main` (konstruktora nincs, nem készül az osztályból egyetlen objektum sem). A Java értelmező minden programban a `main` metódusát keresi, majd automatikusan el is indítja. A `main` metódus feje kötött formájú:

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

Javában minden sorban minden paraméterformára elfogadott, és az `args` változónév helyett választhatunk más nevet is.



9.10. ábra. A `Hurra` osztály UML ábrája

Elemzés soronként:

- ◆ // Első Java programom
Az 1. sor egy megjegyzéssor. Egy programsor // jelpáros utáni részét a fordító figyelmen kívül hagyja. minden sor végén megjegyzésbe tettük a sor számát.
- ◆ public class Hurra
A 2. sor a `Hurra` nevű osztály deklarációjának a feje (class = osztály). Az osztály publikus (public), azaz más csomagokból is látható. A fejet a deklaráció blokkja követi, kapcsos zárójelek között (a 2. sor vége és a 6. sor). A blokk a fej után kezdődik, és az osztálydeklaráció végéig tart.
- ◆ public static void main(String[] args)
A 3. sor az osztály `main` nevű metódusának feje. Ez a program belépési pontja. A `public` kulcsszó jelzi azt, hogy ez a metódus publikus (a terven ezt a + jel mutatja), a `static` kulcsszó pedig azt, hogy osztálymetódus (a terven aláhúzás). A `void` azt jelenti, hogy a `main` metódusnak nincsen visszatérési értéke, vagyis a `main` eljárás. A zárójelben levő `String[] args` a `main` metódus esetleges futtatási paramétereit adja meg: ezeket minden programban le kell írnunk, bár egyelőre figyelmen kívül hagyjuk őket. A `main` metódus törzse szintén egy nyitó és egy csukó kapcsos zárójel közé van beírva (3. sor vége és az 5. sor).

- ◆ `System.out.println("Hurrá, fut az első Java programom!");`

A 4. sor a main metódus egyetlen utasítása (metódushívása). A System osztály out objektumának println metódusa a paraméterben megadott szöveget ("Hurrá, fut az első Java programom!") kiírja a konzolra, JBuilderben az üzenetpanelre. A `System.out.println` metódusról a 11. fejezetben részletesebben is szó lesz.

9.9. A JBuilder szövegszerkesztője

A tartalompanelen megjelenő forráskód szerkeszthető. Felsoroljuk a szövegszerkesztő fontosabb lehetőségeit, csoportokba szedve.

Szövegírás

- ◆ **Insert:** Váltás a Beszúró mód és Felülíró mód között. Beszúró üzemmódban (a státuszsorban: Insert) a cursor egy vékony vonal, felülíró üzemmódban (a státuszsorban: Overwrite) pedig egy tömör, az aktuális karakteren villogó téglalap.
- ◆ **Karakter:** Beszúró üzemmódban a leütött karakter a cursor helyére kerül, az addigi karakterek pedig eggyel jobbra tolódnak a sorban. Felülíró üzemmódban az újonnan beírt karakterek felülírják a már ott levőket.
- ◆ **Enter:** Új sor kezdése. Felülíró üzemmódban a cursor a következő sor első pozíciójára áll. Beszúró üzemmódban ezzel egy sort szűrünk be, a cursor ebben az új sorban a felette álló sor első karaktere alá áll. Szerkesztés közben a program struktúrája automatikusan alakul a beírt vezérlő utasítások alapján (Indent mód).

Pozicionálás

- | | |
|--------------------|--|
| ◆ → | A cursor egy karakterrel jobbra lép. |
| ◆ ← | A cursor egy karakterrel balra lép. |
| ◆ Ctrl + → | A cursor egy szóval jobbra lép. |
| ◆ Ctrl + ← | A cursor egy szóval balra lép. |
| ◆ ↑ | A cursor egy sorral feljebb lép. |
| ◆ ↓ | A cursor egy sorral lejjebb lép. |
| ◆ Home | A cursor a sor elejére ugrik. |
| ◆ End | A cursor a sor végére (az utolsó karakter mögé) ugrik. |
| ◆ Ctrl + Home | A cursor a szöveg elejére ugrik. |
| ◆ Ctrl + End | A cursor a szöveg végére ugrik. |
| ◆ Ctrl + Page Up | A cursor az aktuális oldal tetejére ugrik. |
| ◆ Ctrl + Page Down | A cursor az aktuális oldal aljára ugrik. |

Törlés

- | | |
|-----------------|---|
| ◆ Delete | A cursor alatt levő karakter törlése. A cursor helyben marad. |
| ◆ BackSpace (←) | A cursor előtti karakter törlése; a cursor eggyel balra lép. |
| ◆ Ctrl + Y | A cursor sorának törlése. Az alatta levő sorok eggyel feljebb húzódnak, s a cursor a következő sor elejére áll. |

Kódbeillesztés – Ctrl + J

A kódbeillesztés megkönnyíti a kódolást: előre elkészített kódrészletek illeszthetők be vele a programba. A Ctrl + J hatására egy lista tűnik fel, különböző kódrészletek azonosítóival. A kiválasztott kódrészlet bekerül a forráskódba.

A main blokkot például minden programba be kell írnunk – ez egyszerz unalmas, másrésztt hiba forrása lehet. A main azonosítójú kódrészlet kiválasztására a szövegbe kerül a main blokk.

Fontosabb elemek:

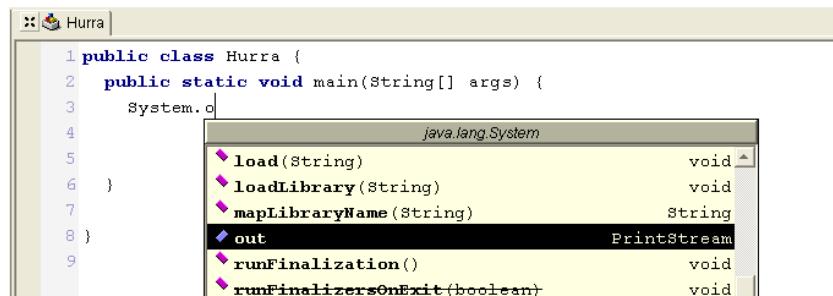
| Azonosító | Kódrészlet |
|-----------|---|
| classp | public class {} |
| main | public static void main(String[] args) {} |
| out | System.out.println(" "); |
| addwin | addWindowListener(new WindowAdapter() { . . . }); |

Automatikus metódusfelkínálás

Ha egy osztály vagy objektum neve után pontot teszünk, akkor a rendszer automatikusan felkínálja azokat az adatokat, metódusokat, amelyeket a fordító ezen a helyen elfogad (9.11. ábra). Mindez csak akkor megy így, ha a program eddig szintaktikailag hibátlan, és tartalmazza a megfelelő importdeklarációkat.

Hibafeltárás

Ha a kódban szintaktikai hiba van, akkor szerkesztés közben a struktúrapanelen megjelenik egy Errors gyökérelem, és felsorolja a hibákat. Ha a hibát kijavítjuk, akkor ez az elem eltűnik. Ajánlatos a hibákat folyamatosan javítani, különben nem működik az automatikus metódus-felkínálás.



9.11. ábra. Automatikus metódusfelkínálás

Automatikus befejezés – Ctrl + Space

Ha elkezdünk írni egy szót, akkor a Ctrl + szóköz lenyomására a rendszer a szót „legjobb tudása szerint” befejezi. Ha a folytatás még nem egyértelmű, akkor a rendszer felkínálja a lehetőségeket.

A szövegszerkesztőben van ezen kívül **szintaktikai kiemelés** (különböző típusú szavak más színnel jelennek meg), valamint **helyzetérzékeny help** (F1 lenyomására megjelenik annak az osztálynak a leírása, amelyen a cursor áll).

9.10. JDK – Java Fejlesztői Készlet

A JDK szabványos fejlesztői környezet; a Sun fejlesztette ki, és belefoglalta a Java programok írásához szükséges

- fejlesztőeszközök; például fordítót, futtatót, nyomkövetőt;
- osztálykönyvtárakat (ez az API, Application Programming Interface).

A JDK önállóan letölthető, és alkalmas Java programok fejlesztésére, de nem integrált fejlesztői környezet, mint a JBuilder: nem tartalmaz szövegszerkesztőt, és a fordítást meg a futtatást külön programmal kell meghívni. A JDK része a JBuildernek.

A Java fejlesztőeszközök valamilyen módon minden használják a JDK-t. Bizonyos környezetek megkövetlik a JDK jelenlétét – ilyen például a JCreator –, mások, mint a JBuilder, maguk telepítik fel. Ha tehát telepítjük a JBuildert, akkor felesleges a JDK-t még külön telepíteni, hiszen az a JBuilder mappájában már úgyis benne lesz!

A JDK-t tudnunk kell alapszinten használni, különben idegen környezetben, szokásos integrált fejlesztői környezetünk nélkül nem boldogulhatnánk. Kétségtelen persze, hogy a JDK csak „fapados” módszerekkel szolgál.

A JDK telepítése, könyvtárstruktúrája

A Java nyelvet a Sun MicroSystems egy csoportja, a JavaSoft fejlesztette ki. A Javához a nyelvi szabályok leírásán kívül tartozik egy osztálykönyvtár is, az API (Application Programming Interface): az alkalmazások által használható osztályok gyűjteménye. A Sun a Java különböző verziót JDK (Java Development Kit = Java Fejlesztői Készlet) névvel bocsátja ki (az aktuális verzió az 1.4-es). A JDK része az API, valamint egy primitív fejlesztői környezet: egyebek között egy fordító, egy értelmező/futtató, valamint egy appletnéző program különböző platformokra (Windows, Linux, Solaris ...). A JDK szabadon letölthető az Internetről, és szabadon használható, címe: <http://java.sun.com/j2se>

Megjegyzések:

- A JDK-t a Sun már minduntalan SDK-nak (Standard Development Kit) emlegeti.
- A kiterjesztett JDK a J2SE – ez a Java2 Standard Edition rövidítése.
- A JDK egyetlen, futtatható állományként tölthető le. (Az 1.4-es verzió mérete kb. 35 MB.) A telepítéshez kattintsunk rá a verziónak és platformnak megfelelő tömörített, futtatható állományra; az állomány erre kicsomagolja magát, és telepíti a JDK-t a kérő könyvtárba. A JDK installált változata mintegy 60 MB. A dokumentációt (az API leírást) külön kell letölteni.

A JDK könyvtárstruktúráját a 9.12 ábra mutatja. A // után minden sorban az aktuális elem rövid magyarázata áll. Vizsgáljuk meg most nagy vonalakban a különféle elemeket!

```
c:\JBUILDER8
jdk1.4
  bin          // fejlesztői programok
    javac.exe   // Java compiler
    java.exe    // Java interpreter (JVM, futtató)
    appletviewer.exe // Java appletnéző
  ...
  demo         // mintaprogramok
    applets     // applet mintaprogramok
    sound       // hang mintaprogramok
  ...
  jre          // futtatókörnyezet (Java Runtime Environment)
    lib
      rt.jar    // API (osztálygyűjtemény)
    bin
      java.exe  // JVM, a jdk1.4\bin\java.exe másolata
  ...
  docs         // API help dokumentáció, külön kell letölteni és
    index.html  telepíteni
  src.zip      // API osztályok forráskódja (source)
  ...
```

9.12. ábra. A JDK könyvtárstruktúrája

A JDK elemei:

- ◆ **bin** könyvtár: Itt találhatók a fejlesztői programok (fordító, futtató, hibakereső stb.). A `javac.exe` lefordítja a Java forrásállományt, és elkészíti a bájtkódot. A `java.exe` és a `javaw.exe` virtuális gép (JVM), vagyis értelmezi és futtatja a bájtkódot. Az `appletviewer.exe` olyasfajta funkciót lát el, mint a böngészőbe beépített JVM: tesztelhetjük appletjeinket, hogy milyenek fogjuk látni őket böngészés közben.
- ◆ **demo** könyvtár: Minta-appleteket és -applikációkat tartalmaz demonstrációs célra.
- ◆ **jre** könyvtár: Java futtatókörnyezet. Ha a Java programokat csak futtatni akarjuk, akkor csupán erre a mappára van szükségünk. Két almappája van:

- **lib** könyvtár: ebben a mappában található a Java osztálykönyvtár, más néven az API (Application Programming Interface) lefordított, bájtkód formában. Több ezer osztályból válogathatunk programjaink elkészítésekor. Az API osztálykönyvtárat tulajdonképpen az óriási, `rt.jar` tömörített állomány tartalmazza; a rendszer ebben találja meg a `class` kiterjesztésű, könyvtárstruktúrába szervezett osztályokat.
- **bin** könyvtár: itt is megtalálható a JVM, a `jdk1.4\bin\java.exe` másolata.
- ◆ **docs** könyvtár: Help, API-specifikáció hiperszöveg (vagy pdf) formában. Az `index.html`-ből elérhető az API összes osztályának és interfészének leírása.
- ◆ **src.zip**: a Sun mellékelt az osztálykönyvtár (API) teljes forráskódját. Aki nem igazodik el kellőképpen a help dokumentációkon, az további lehetőségekért böngészheti a Java forráskódot is. A forrás közszemlére bocsátásának más célja is van: sok szem többet lát, s jóval könnyebben kiszűri az esetleges hibákat. A Java fejlesztőszerek kicsomagolás nélkül olvassák ezt az állományt.

Tegyük fel most, hogy nem is telepítettük a JBuilder-t! Ha telepítettük volna a JDK-t, akkor a `jdk1.4` mappa most a főkönyvtár alatt lenne. Ebben a pontban csak a JDK szolgáltatásaira támaszkodva írunk Java programot.

Java forrásprogramunkat bármilyen tiszta karakteres szövegszerkesztővel elkészíthetjük: az a lényeg, hogy az elmentett állomány csak a begépelezett szöveg karaktereit tartalmazza, ne kerüljön bele semmilyen szövegformázási parancs.

Megjegyzések:

- Windows környezetben szövegszerkesztésre alkalmas például a NotePad vagy az Edit. A Word nem használható erre a célra, mert az elmentett állomány tele lesz a Java fordító számára érthetetlen formázó karakterekkel.
- Érdemes megfigyelnie, hogy a JDK a lemezen mintegy 60 MB helyet foglal el, s ebből a `javac.exe` és a `java.exe` csak nagyjából 20–20 KB!
- A böngészőkbe betett JVM általában le van maradva az aktuális verzióhoz képest.

Környezeti beállítások

Ha JDK-ban dolgozunk, be kell állítanunk a következő környezeti változókat:

- ◆ **PATH környezeti változó:** Ahhoz, hogy a JDK programjait (`javac.exe`, `java.exe` stb.) a maguk könyvtárán kívüli könyvtárból is használhassuk, a `PATH`-t ki kell bővítenünk a könyvtárunkra való hivatkozással – a mi esetünkben ezzel:
`SET PATH=%PATH%;C:\JBuilder8\jdk1.4\bin`
- ◆ **JAVA_HOME környezeti változó:** A könyv programjai fordításkor és futtatáskor az operációs rendszer `JAVA_HOME` környezeti változójára hivatkoznak. Állítsa be azt a JDK könyvtárra! A mi esetünkben:
`SET JAVA_HOME=C:\JBuilder8\jdk1.4`

- ◆ **CLASSPATH környezeti változó:** A JDK fordító- és futtatórendszere megtalálja a maga osztálykönyvtárát, az API-t. Ha programunkban más osztályokat is szeretnénk használni, akkor meg kell adnunk azok elérési útvonalát. A rendszer a CLASSPATH környezeti változóban megadott könyvtárakban keresi azokat az osztályokat, csomagokat, amelyekre programunkban hivatkozunk (az importált osztályokat, csomagokat). A CLASSPATH-ban az aktuális könyvtárat mindenkiéppen meg kell adnunk (erre a könyvtárra a . jellet hivatkozunk), mert itt vannak az aktuális csomag osztályai. Könyvünk használ egy saját, extra nevű csomagot (osztálykönyvtárat) is, a javalib könyváról; erre a csomagra a legegyszerűbb programok írásához is szükségünk lesz. Esetünkben tehát a beállítás:

```
SET CLASSPATH=. ;C:\javaprogram\lib\javalib.jar
```

❖ Vigyázat! Ha több könyvtárat ad meg, ne tegyen közéjük szóközt!

Ajánlatos a futtatáshoz egy kis parancsállományba foglalni ezeket a beállításokat. A könyvmellékletében ez a javaprogram\lib\setjava.bat állomány, és ez van beleírva:

```
SET PATH=%PATH%;c:\JBuilder8\jdk1.4\bin
set JAVA_HOME=c:\JBuilder8\jdk1.4
set CLASSPATH=. ;c:\javaprogram\lib\javalib.jar
doskey
c:
cd \javaprogram\_MyPrograms
```

Az utolsó parancssal átváltunk abba a munkakönyvtárba, ahol dolgozni fogunk. A doskey parancsra azért van szükség, hogy az előzőleg beírt DOS parancsokat a fel-le billentyűvel újra előhozhassuk. A környezeti változókat a rendszerbeállításokkal (Windowsban a vezérlőpulton) végegesen is megadhatjuk – akkor nem kell a set java.

A program begépelése, futtatása

A Hurra.java program fordítását és futását a 9.13. ábra mutatja. A program elkészítéséhez és futtatásához kövesse az alábbiakat:

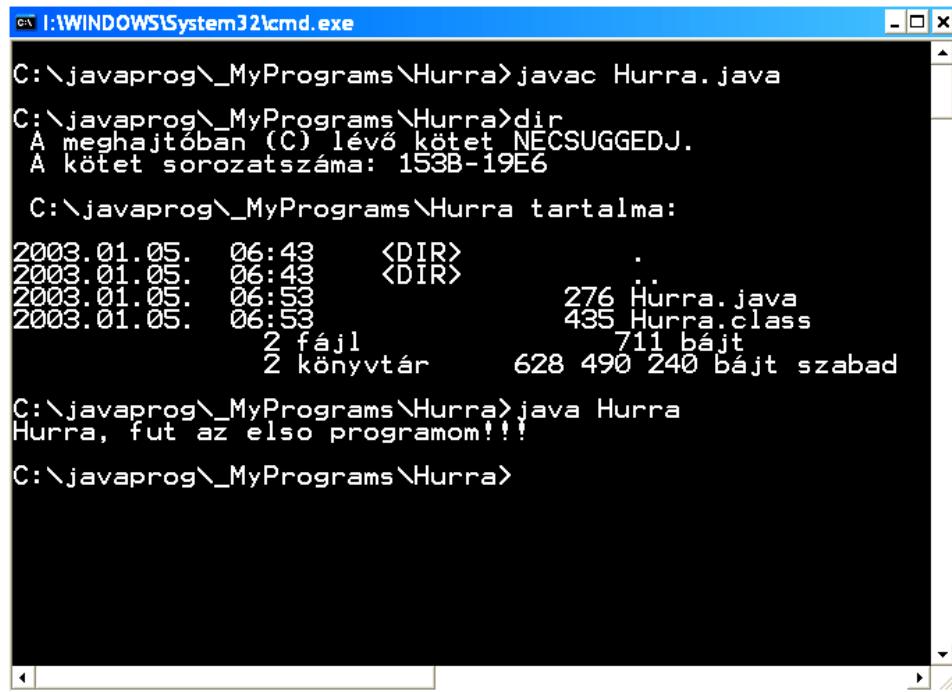
- ◆ Nyisson egy konzolablakot!
Start/Run/cmd.exe, vagy *Start/Programs/Accessories/Command Prompt*
- ◆ Futtassa a javaprogram\lib\setjava.bat programot:
A következőkben gondoskodjon arról, hogy a munka megkezdésekor (a konzolablak megnyitásakor) minden lefusson ez a parancsállomány.

Még egyszerűbben: Kattintson kettőt a javaprogram\work\jdkwork.bat állományon, és már ott találja magát a JDK-s munkára felkészített konzolablakban, a _MyPrograms mappában.

- ◆ Készítsen most a Hurra programnak egy mappát és lépjén át ebbe a könyvtárba!

```
md Hurra
cd Hurra
```

- ◆ Indítsa el a Notepad szövegszerkesztőt! Írja be a parancssorba: Notepad, vagy:
Start/Programs/Accessories/Notepad
 - ◆ Gépelje be a program forráskódját!
 - ◆ Mentse el a begépelt szöveget a c:\javaprog_MyPrograms\Hurra mappába, egy Hurra.java nevű állományba!
 - ◆ A NotePad-ból ne lépjön ki, csak váltsan át a konzolablakra! Ott a C:/javaprog/_MyPrograms/Hurra az aktuális könyvtár!
 - ◆ Fordítás le a Java forrásprogramot a következő parancssal:
javac Hurra.java
- A javac.exe program paramétere az előbb megírt Java forrásállomány neve, kiterjesztéssel együtt (Hurra.java). Az aktuális könyvtárban keletkezik egy Hurra.class állomány (erről meggyőződhet a könyvtár kilistázásával: dir), ez a Hurra.java osztály lefordított, bájtkód formája.
- ◆ Futtassa a Hurra.class állományt a következő parancssal:
java Hurra
- A java.exe program paramétere az előbb lefordított bájtkód (Hurra.class) neve, ezúttal kiterjesztés nélkül!



The screenshot shows a Windows Command Prompt window titled 'cmd I:\WINDOWS\System32\cmd.exe'. The command line shows the following sequence of commands and their outputs:

```
C:\javaprog\_MyPrograms\Hurra>javac Hurra.java
C:\javaprog\_MyPrograms\Hurra>dir
A meghajtóban (C) lévő kötet NECSUGGEDJ.
A kötet sorozatszáma: 153B-19E6
C:\javaprog\_MyPrograms\Hurra tartalma:
2003.01.05. 06:43 <DIR> .
2003.01.05. 06:43 <DIR> ..
2003.01.05. 06:53 276 Hurra.java
2003.01.05. 06:53 435 Hurra.class
      2 fájl    711 bájt
      2 könyvtár   628 490 240 bájt szabad
C:\javaprog\_MyPrograms\Hurra>java Hurra
Hurra, fut az else programom!!!
C:\javaprog\_MyPrograms\Hurra>
```

9.13. ábra. A Hurra program fordítása és futtatása JDK-ban

Képernyőjén remélhetőleg megjelent az ujjongó szöveg! Elképzelhető, hogy az ékezes betűk hibásan jelennek meg – a JBuilder már majd jól fogja megjeleníteni.

Előfordulhat, hogy valamilyen hiba folytán mégsem jelent meg a várva várt szöveg. Ebben az esetben újra ellenőrizni kell a szöveget és a megfelelő műveleteket, nem rontottunk-e el valamit. Legjobb, ha nyitva hagyjuk a konzolablakot és a szövegszerkesztő ablakot is (a Windowsban ezt megtehetjük), és a következő algoritmus szerint járunk el:

```
do
    átváltás a Notepad szövegszerkesztő ablakába
    forrásprogram szerkesztése
    forrásprogram elmentése
    átváltás a konzolablakba
    fordítás
    futtatás
end do while valami nincs rendben
```

Esetleges hibák

Ha valami nincs rendben, ellenőrizze a következőket:

- ◆ Nem mindegy, hogy a Java forrásállomány nevében és a mappák nevében kisbetű írunk-e vagy nagybetűt! A Hurra és a hurra tehát különbözik; az extra csomag neve csupa kisbetűs!
- ◆ A Java forrásállomány nevének meg kell egyeznie az állománybeli publikus osztály nevével: `public class Hurra ↔ Hurra.java`
- ◆ Fordításkor meg kell adnunk a `.java` kiterjesztést, futtatáskor viszont nem szabad megadnunk a `.class` kiterjesztést (`javac Hurra.java` és `java Hurra`)!
- ◆ Ellenőrizze a PATH környezeti változót a DOS `set` parancsával! Benne van a JDK `bin` könyvtára?
- ◆ Ellenőrizze a CLASSPATH környezeti változót a DOS `set` parancsával! Benne van az aktuális könyvtár (`.`) és a `javapath`?
- ◆ A környezeti beállítás módja függhet az operációs rendszertől. A konzolablakból való kilépéskor megszűnhetnek a beállítások!
- ◆ Az operációs rendszer hosszú állományneveket használ (a `.java` kiterjesztés is több mint 3 karakter).
- ◆ Ha olyan sok hibaüzenetet ír ki a fordító, hogy a konzolablakban nem lehet visszalapozni őket, akkor változtassa meg az ablak tulajdonságait: növelje meg a képernyőpuffer sorainak számát!
- ◆ Nézze meg, hogy biztosan `java`-e a kiterjesztést! Az operációs rendszer elrejtheti a kiterjesztést – a konzolablak DIR parancsa azonban mindenkor megmutatja!

- ◆ Előfordulhat, hogy az operációs rendszer átírja a kisbetűt nagybetűre, vagy a szövegszerkesztő hozzáteszi az állománynévhez az általa használt kiterjesztést (például `Hurra.java.txt`). Ilyenkor próbálja meg a nevet elmentéskor idézőjelbe tenni: "Hurra.java".

A JDK-ban nehézkes a programozás. A JBuilderben fontosabb dolgokra is figyelhetünk.

Futtassa JDK-ban a könyv néhány programját!

9.11. Az API csomagstruktúrája

A Java nyelv igen egyszerű felépítésű; nincs benne sok típus és utasításfajta. Tervezői arra törekedtek, hogy könnyen áttekinthető legyen, és egykönnyen ne lehessen hibázni. A Java nyelv „ereje” a fejlesztői környezettel együtt kapott osztálykönyvtárban van: a programozó rengeteg osztályra támaszkodhat a legkülönfélebb programok összeállításához. Megjegyzendő, hogy ennyi osztályt szinte lehetetlen áttekinteni. A könyv célja nem is az, hogy a teljes osztálykönyvtárat ismertesse az Olvasóval. A cél az, hogy az Olvasó megértse a nyelvben és az osztálykönyvtárban rejlı szemléletet, és kialakuljon benne az újrafelhasználható és robusztus alkalmazások írásának képessége.

A **Java osztálykönyvtár (API)** osztályai a tömörített `rt.jar` állományban vannak, könyvtárstruktúrába szervezve. Az osztályokat logikailag csoportosították: az összetartozó osztályok ugyanabba a csomagba (package) tartoznak. Bizonyos csomagokba az osztályokon kívül további csomagok (alcsomagok) is kerültek. A csomagstruktúrának a lemezen egy ugyanolyan szerkezetű könyvtárstruktúra felel meg.

A JDK csomagstruktúrája a 9.14. ábrán látható. A legfelső szinten szerepel például a `java` és a `javax` csomag. A `java` csomag alcsomagai: az `applet`, `awt`, `io`, `lang`, `util` stb. A különféle csomagokba osztályokat foglaltak: az `applet`be az `Applet`et, az `awt`-be a `Containert`, a `Fontot`, a `javax.swing` csomagba a `JButtont`, `JComponentet` stb. Az ábra nem teljes, csak néhány, nekünk fontosabb csomagot és osztályt ábrázol. A csomagok neve minden kisbetűvel kezdődik – az ábrán vastag betűvel szedtük őket –, az osztályoké viszont minden nagybetűvel. Az osztályok `class` kiterjesztéssel szerepelnek a megadott könyvtárban (például `Applet.class`), hiszen futtatáskor a rendszer az osztály lefordított bájtkódját használja. Az osztályokat csomagneveikkkel és az osztály nevével együtt azonosítjuk, s e nevek közé pontot teszünk (pl.: `java.applet.Applet`).

```

java
  applet
    Applet
    ...
  awt
    Container
    Font
    Graphics
    Color
    ...
    event
      KeyListener
      MouseListener
      ...
  io
    File
    PrintStream
    ...
  lang
    Integer
    Math
    Object
    String
    System
    ...
  util
    GregorianCalendar
    Locale
    Vector
    ...
javax
  sound
  swing
    JButton
    JComponent
    JFrame
    ...
...

```

9.14. ábra. A Java API csomagstruktúrája

Az API főbb csomagjainak rövid ismertetése:

- **java:** Legfelső szintű csomag, közvetlenül nem tartalmaz osztályokat.
- **java.applet:** Az appletek írásához szükséges osztályokból áll.
- **java.awt** (Abstract Window Toolkit): Ez a csomag tartalmazza a grafikus felhasználói felület (GUI, Graphical User Interface) alaposztályait. Az itt szereplő vezérlők (keret, nyomógomb, beviteli mező stb.) az operációs rendszer szolgáltatásaira támaszkodnak (ún. nehézsűlyű komponensek, natív vezérlőket futtatnak). A JDK 1.2-es verziójában

- megírták a „pehelysúlyú”, az operációs rendszertől már független swing komponenseket; lásd javax.swing csomagot.
- **java.io** (input/output): Osztályai segítségével adatokat lehet kiírni a különböző perifériákra (például lemezre), valamint adatokat lehet beolvasni róluk.
 - **java.lang** (language): A Java alaposztályait tartalmazza (Integer, Math, Object stb.). Ezt a csomagot minden Java program automatikusan látja, nem kell külön importálni.
 - **java.util** (utility): Ez a csomag kisegítő osztályokat tartalmaz. Itt vannak a konténerek (pl. Vector), a naptárral és nemzetköziséggel kapcsolatos osztályok (pl. GregorianCalendar, Locale) és más hasznos osztályok, interfészek.
 - **javax**: Java kiterjesztés.
 - **javax.swing**: A Swing az AWT kiterjesztése (modernebb és bővebb változata). A Swing platformtól független, könnyed (pehelysúlyú) külalakkal szolgál. A GUI osztályok neve itt J betűvel kezdődik.

Programunkban egy API osztályra csak úgy hivatkozhatunk, ha az osztályt (vagy az osztály teljes csomagját) programunk számára elérhetővé, láthatóvá tesszük, vagyis **importáljuk**.

A következő sorban a java.util csomag Vector osztályát importáljuk:

```
import java.util.Vector;
```

A teljes java.util csomagot – annak összes osztályát – így importálhatjuk:

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

A lang csomagot minden program automatikusan látja, ezért a következő importálás például felesleges:

```
import java.lang.System;
```

9.12. Fordítás és futtatás több osztály esetén

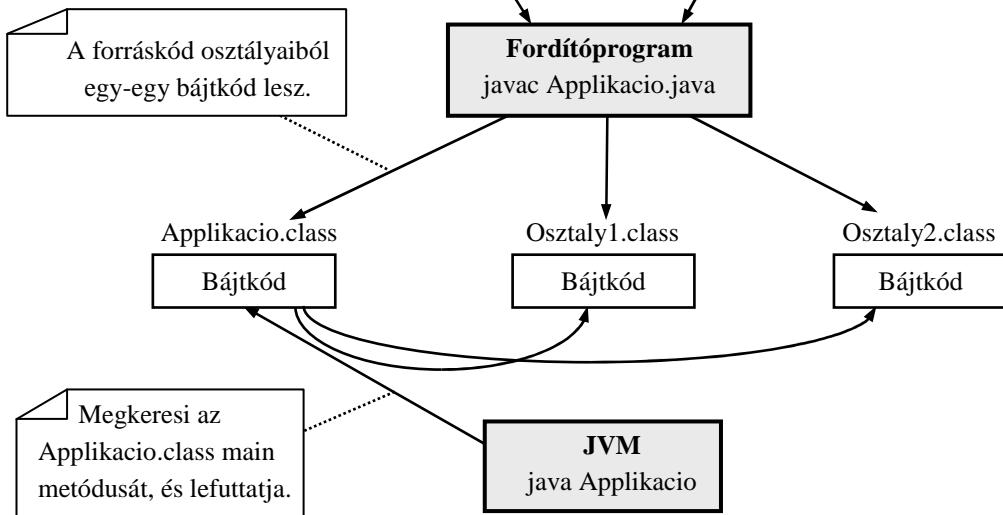
Egy program általában több osztályt is tartalmaz (mi egyelőre csak egyosztályos programokat fogunk írni). Az osztályok forráskódját forrásállományokba tesszük; egy forrásállomány egy vagy több osztályt tartalmazhat, de általában ajánlatos az egy osztály–egy állomány konstrukcióhoz tartanunk magunkat (bizonyos fejlesztői környezetek ezt meg is követelik).

Az egyszerre lefordítható forráskódmenyiséget **fordítási egységnek** szokták nevezni. Javában a fordítási egység a teljes osztály(oka)t tartalmazó forrásállomány. Nézzük át, mi is történik nagy vonalakban egy Java fordítási egység (forrásállomány) fordításakor és futtatásakor, ha abban több osztály is szerepel (9.15. ábra).

Appikacio.java (fordítási egység, forrásállomány)

```
import java.util.*  
public class Applikacio {  
    public static void main(String[] args) {  
        ...  
    }  
    class Osztaly1 {  
        ...  
    }  
    class Osztaly2 {  
        ...  
    }  
}
```

```
package java.util;  
class GregorianCalendar ...  
class Date ...  
...
```



9.15. ábra. A Java program fordítása, futtatása

Példánkban az `Applikacio.java` forrásállományban 3 osztály szerepel: `Applikacio`, `Osztaly1` és `Osztaly2`; hármójuk közül csak egy publikus, az `Applikacio`. Ez az osztály tartalmazza a `main` nevű, kötött formájú publikus osztálymetódust. Az `Applikacio.java` importálja a `java.util` csomagot, mert nyilván hivatkozni szeretne az abban szereplő osztályokra, például a `GregorianCalendar`-re vagy a `Vector`-ra.

A fordítás menete, szabályok:

- Egy Java forrásállomány (fordítási egység) egyszerre több osztályt is tartalmazhat, s azok közül legfeljebb egy lehet publikus. A publikus osztálynak a neve meg kell, hogy egyezzék a forrásfájl nevét: (public class Applikacio ↔ Applikacio.java).
- Egy Java program több fordítási egységből is állhat (mi egyelőre csak egyet írunk), de a programnak pontosan egy belépési pontja van, ez a main metódus. A main metódus feje kötött formájú: public static void main (String[] args). Az Applikacio osztály csak akkor futtatható, ha tartalmaz main-t.
- A fordítóprogram (javac.exe) a fordítási egység minden osztályából készít egy class kiterjesztésű bájtkódot. A class-állományok neve ugyanaz lesz, mint a bennük levő osztályé.
- Futtatni csak olyan bájtkódú állományt lehet, amelyben van main metódus. Akárhány osztályból áll is a programunk, csak egyetlen main metódusa, vagyis egyetlen belépési pontja lehet.
- Lefordított programunk (a class fájlok összessége) futtatáskor is használja az API osztálykönyvtárat. A futási környezetnek (jre = java runtime environment) tehát számítogépünkön minden futáskor jelen kell lennie.

9.13. Integrált fejlesztői környezetek

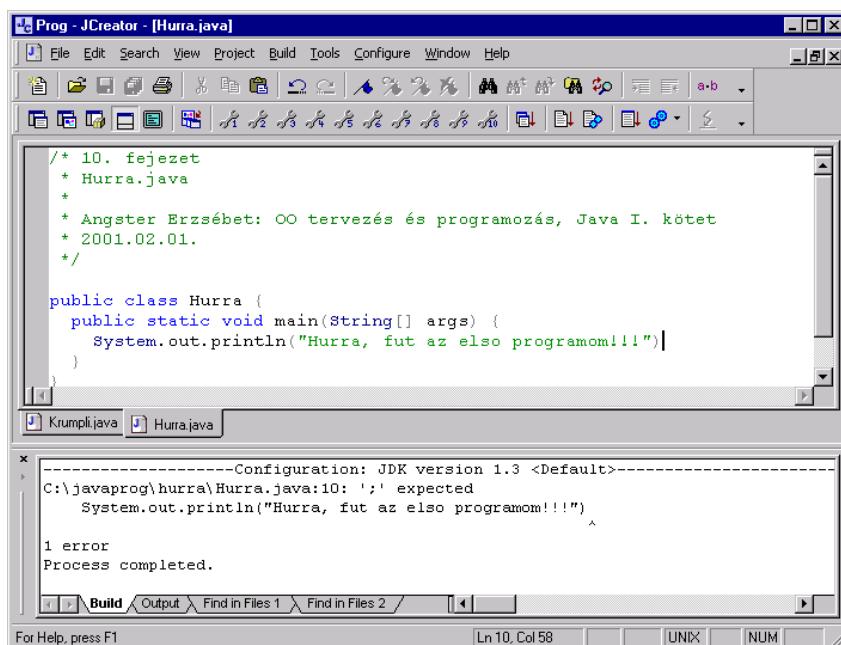
Mint láttuk, a JDK tartalmazza azokat az alaprogramokat, amelyekkel egy forrásprogramot bájtkóddá fordíthatunk (javac.exe), s azt futtathatjuk (java.exe). A program fejlesztése azonban meglehetősen nehézkes, ha pusztán a JDK által kínált minimális lehetőségeket használjuk, mivel folyamatosan váltanunk kell a szövegszerkesztő és konzolablak között, mindig figyelnünk kell a megfelelő környezeti beállításokra, és nem támaszkodhatunk másutt már megszokott kényelmi funkcióra.

Az integrált fejlesztői környezet (Integrated Development Environment) rengeteg lehetőséggel teszi könnyebbé a programozó életét és gyorsítja meg a munkáját. Egy modern fejlesztői keretrendszer összekapcsolja a szövegszerkesztést, a fordítást, a futtatást és a többi funkciót, épedig úgy, hogy egy gombnyomással végre lehessen hajtani őket. Egy integrált fejlesztői környezet mindenképpen tartalmaz

- ◆ **egy szövegszerkesztő** a forráskód megírására. Ez a szövegszerkesztő **tiszta karakteres állományt állít elő**, és lehetőség szerint **segíti a strukturált programírást** (például van benne „beljebb kezdés” (indent) mód, vagyis az Enter leütésére a kurzor mindenkor az előző sor alá áll).
- ◆ **egy fordítót**, amely egy gombnyomásra listát készít a forráskód szintaktikai hibáiról és kérésre a forráskód hibás sorára áll.
- ◆ **egy futtatót**, amely gombnyomásra futtatja a lefordított kódot.

Egy jó integrált környezetben megvannak a következő funkciók is:

- ◆ **Szintaktikai kiemelés** (syntax highlighting): más-más színben jelennek meg a forrás-kód szintaktikailag elkülöníthető szavai (pl. a kulcsszavak kékek, a szövegek zöldek stb.).
- ◆ **Nyomkövetés** (debugger): A program futását lépésről lépére hajthatjuk végre, figyelve közben a program alakulását.
- ◆ **Segítség** (help): A programfejlesztés különböző szakaszaiban magyarázatot kérhetünk a különféle fogalmakról, kulcsszavakról, sőt a környezet gépelés közben felkínálja az éppen beírható adatok és metódusok listáját!



9.16. ábra. JCreator – integrált fejlesztői környezet

Példák integrált Java fejlesztői környezetre:

- ◆ JCreator (Xinox Software, www.jcreator.com)
- ◆ JPadPro (ModelWorks Software, www.modelworks.com)
- ◆ TextPad (Helios Software Solution, www.textpad.com)
- ◆ JBuilder (Borland, www.borland.com)
- ◆ JDeveloper (Oracle, www.oracle.com)
- ◆ VisualAge for Java (IBM, www.ibm.com)
- ◆ Forte for Java (Sun Microsystems, Inc., www.sun.com)

A 9.16. ábra a JCreator integrált fejlesztői környezetet mutatja.

Vannak JDK-ra épülő, és vannak önálló fejlesztői környezetek:

- ◆ A JDK-ra épülő környezet csak akkor használható, ha telepítük a JDK-t és kapcsolatot teremtünk közte meg a környezet között. Az ilyen környezet közvetlenül ráépül a JDK-ra, használja annak programjait és osztálykönyvtárát (az API-t). Ezek a környezetek általában egyszerűek, csak a szövegszerkesztési és integrálási feladatokat látják el: gombnyomásra meghívják a JDK javac.exe fordítóját az aktuális forrásállománnyal, illetve a java.exe futtatóját az aktuálisan lefordított class állománnyal. JDK-ra épülő környezet a felsoroltak közül a JCreator, a JPadPro és a TextPad.
- ◆ Az önálló fejlesztői környezetbe a gyártó cégtől saját (feltehetően gyorsabb) fordító és futtató programokat épít bele, és a JDK aktuális verzióját is belefoglalja. Az önálló környezetek általában rengeteg specialitással szolgálnak, és sokkal drágábbak, mint a JDK-ra épülő környezetek. Ilyen önálló környezet a JBuilder, a JDeveloper, a VisualAge for Java és a Forte for Java. E négy környezet egyben vizuális fejlesztőeszköz is.

Egy integrált fejlesztői környezetbe (Integrated Development Environment, IDE) különböző funkciókat építenek össze a programozó munkájának könnyítése céljából. Egy integrált fejlesztői környezet mindenkorban tartalmaz szövegszerkesztőt, fordítót és futtatót. Az igazán kényelmes környezetben megtalálhatók az olyasfajta extrák is, mint a nyomkövetés, a segítségadás, a szintaktikai kiemelés stb. Vannak JDK-ra épülő, és vannak önálló integrált fejlesztői környezetek.

Tesztkérdések

- 9.1. Mi igaz egy JBuilder projekttel kapcsolatban? Jelölje meg az összes igaz választ!
 - a) A projekt csak class állományokat tartalmaz.
 - b) A projektállomány kiterjesztése jpb.
 - c) A projekthez tetszőleges kiterjesztésű állomány tartozhat.
 - d) A projekt logikailag összefogja állományait.
- 9.2. Mi igaz az alkalmazásböngészővel kapcsolatban? Jelölje meg az összes igaz választ!
 - a) A főmenü része az alkalmazásböngészőnek.
 - b) A struktúrapanel a projekt könyvtárstruktúráját jeleníti meg.
 - c) Az alkalmazásböngésző státuszsora mutatja, hogy a szövegszerkesztőben hányadik sorban van a kurzor.
 - d) Az üzenetpanelre a rendszer üzen, arra a programozó nem írhat.
- 9.3. Jelölje meg az összes igaz állítást a következők közül!
 - a) A JBuilderben egyszerre több projekt is nyitva lehet.
 - b) A JBuilder tartalompanelje képes megjeleníteni egy HTML fájlt.

- c) A JBuilder tartalompanelje képes „megjeleníteni” egy hangfájlt.
 - d) Ha a projektfáról leveszünk egy fájlt a Remove parancsal, akkor a fájl törlődik a lemezről.
- 9.4. Mely funkcióval lehet egy JBuilder projekt összes forráskódját feltétel nélkül lefordítani? Jelölje meg az összes igaz választ!
- a) *Project/Make Project "[Projektfájl]"*
 - b) *Project/Rebuild Project "[Projektfájl]"*
 - c) *Project/Compile Project "[Projektfájl]"*
 - d) *Project/Debug Project "[Projektfájl]"*
- 9.5. Mely funkcióval lehet egy tetszőleges Java forráskódot futtatni, ha azt hozzáadjuk a javaproj projekthez? Jelölje meg az összes igaz választ!
- a) *Project/Make Project "[Projektfájl]"*
 - b) *Project/Rebuild Project "[Projektfájl]"*
 - c) *Run/Run "[Forrásfájl]"*
 - d) *Jobb egérgomb/Run using "javaproj"*
- 9.6. Jelölje meg az összes igaz állítást a következők közül!
- a) A JBuilder első indítása után bármely projekt futtatható. minden beállításért a projekt felel.
 - b) Egy projekt forráskódjainak ajánlott helye az src könyvtár.
 - c) A JBuilder szövegszerkesztőjében van automatikus metódusfelkínálás.
 - d) A JBuilderben szövegszerkesztőjében van szintaktikai kiemelés.
- 9.7. Jelölje meg az összes igaz állítást a következők közül!
- a) A JDK többek között tartalmaz egy szövegszerkesztőt.
 - b) Az API egyfajta alkalmazáskészítő eszköz.
 - c) A JDK tartalmaz egy bin könyvtárat; abban vannak a fejlesztői programok.
 - d) A jre a JDK futtatási környezete.
- 9.8. Jelölje meg az összes igaz állítást a következők közül!
- a) Egy programot a JDK-ban a java.exe programmal fordíthatunk.
 - b) A program JDK-val való fordítása és futtatása előtt a CLASSPATH környezeti változóban meg kell adnunk az API könyvtárát.
 - c) Az API a szabványos Java osztályokat tartalmazza bájkód formában.
 - d) A Java osztályok könyvtárhierarchiája tükrözi a csomaghierarchiát.
- 9.9. Jelölje meg az összes igaz állítást a következők közül!
- a) Ha egy Java programban használni akarjuk a util csomagot, akkor azt importálni kell a programba.
 - b) A java.lang csomagot mindenkorral importálni kell, mert abban alapvető osztályok vannak.
 - c) Egy Java fordítási egység legalább két osztályból áll.
 - d) minden Java osztályban szerepelnie kell egy main metódusnak.

- 9.10. Az alábbiak közül mely API csomag tartalmazza a konténerosztályokat? Jelölje meg az egyetlen jó választ!
- a) java.lang
 - b) java.util
 - c) java.io
 - d) java.awt
- 9.11. Jelölje meg az összes igaz állítást a következők közül!
- a) Egy Java fordítási egységben pontosan egy osztály szerepel.
 - b) A fordítóprogram mindegyik fordítási egységből készít egy `class` kiterjesztésű állományt.
 - c) Bármely `class` kiterjesztésű állományt lehet futtatni.
 - d) Futtatáskor a program használja a `jre`-t.
- 9.12. Jelölje meg az összes igaz állítást a következők közül!
- a) Egy integrált fejlesztori környezet mindenkorban tartalmaz szövegszerkesztőt, fordítót és futtatót.
 - b) A JDK egyfajta integrált fejlesztori környezet.
 - c) A JDK-t akkor is külön kell telepítenünk a gépre, ha önálló fejlesztori környezetet használunk.
 - d) A Java forrásprogram tartalmazhat formázó karaktereket is.

Feladatok

- 9.1. (A) Változtassa meg a 9.8. pontban található Hurra osztályt úgy, hogy a program kiírja a következő szöveget: "Nem csüggédék, nemsokára elvezni fogom a programozást!". Futtassa a programot! (*Hurra1.java*)
- 9.2. (A) Változtassa meg a Hurra osztályt úgy, hogy a program kiírja családtagjainak nevét a konzolra, mégpedig kétszer! Futtassa a programot! (*Hurra2.java*)

10. A Java nyelvről

A fejezet pontjai:

1. Az OO programozási nyelvek térhódítása
 2. A Java nyelv története
 3. Az Internet, a World Wide Web és a Java
 4. A Java nyelv jellemzői
-

E fejezetben szó lesz az objektumorientált nyelvek kialakulásáról, majd a Java nyelvről: annak viharos sebességű térhódításáról, az Internettel való kapcsolatáról és különféle jellemzőiről.

10.1. Az OO programozási nyelvek térhódítása

Egy objektumorientált eszköz csak segít az OO elvek megvalósításában. Az OO nyelv és eszköz használata nem jelenti feltétlenül azt, hogy OO elveket alkalmazunk!

Az objektumorientált elvek akkor is érvényesek, ha nem a programozásban alkalmazzuk őket. Programkészítéshez azonban programozási nyelvre van szükség, mégpedig olyan nyelvre, amelynek segítségével az OO elveket könnyen valóra válthatjuk.

Az objektumorientált programozás elmélete meglehetősen régi keletű. Az 1960-as években megalkotott Simula-67 programnyelv már tartalmazza a főbb OO elveket. Ennek hatására az 1970-es évek elején a XEROX kutatóközpontja, a PARC (Palo Alto Research Center) létrehozta a Smalltalk-72 nyelvet, majd folyamatosan továbbfejlesztette, és sorban megszületett a Smalltalk-74, 76, 78, majd a Smalltalk-80 nyelv. Bár a szoftverkrízis már a 60-as években felütötte a fejét, a világ sajnos nem figyelt fel igazán az új programozási paradigmára. Ez talán azzal indokolható, hogy az új paradigma egészen más szemléletet követelt meg, mint a strukturált programozási módszer.

Az áttörést 1981 augusztusa hozta, amikor a Byte magazin szinte teljes terjedelmében a Smalltalk-80-ról áradozott. Sorban megjelentek az erről szóló könyvek, majd a Digitalk az addigiaktól függetlenül kidolgozta a Smalltalk/V nyelvet. 1986-ban az ACM (Association of Computing Machinery) elindított egy konferenciasorozatot, az OOPSLA-t (Object-Oriented Programming Systems, Languages, and Applications), s annak 1990-ben már több mint 2000

résztervezője volt. Elindult tehát a lavina, s úgy tűnik, egyhamar nem áll meg. Ma már szinte nincs is olyan programozási nyelv, amelyben ne foglalták volna bele valamennyire az OO paradigmával kínált lehetőségeket.

A programnyelveket csoportosítani szokás szerint, hogy milyen követik az OO elvet. Az olyan programnyelveket, amelyek kényszerítik a programozót az OO elvek betartására, **tiszta objektumorientált nyelveknek** (pure OO language) nevezzük – ilyen például a Java vagy a Smalltalk. Sok programozási nyelvbe később építették bele az OO lehetőségeket, így az ilyen nyelvekben továbbra is lehet hagyományos, strukturált módon programokat írni – ezek az ún. vegyes, **hibrid nyelvek** (hybrid OO language), mint például a C++ vagy a Turbo Pascal. Az olyan nyelvet, amely lehetővé teszi az objektumazonosság, az osztályozás és a bezárás használatát, de nem engedi az öröklést, **objektum alapú nyelvnek** (object-based language) nevezzük – ilyen például a Basic.

Tiszta objektumorientált nyelv: Olyan programozási nyelv, amely kikényszeríti az OO elvek betartását.

Hibrid nyelv: Olyan programozási nyelv, amelyben lehet objektumorientált és strukturált programokat is írni.

Objektum alapú nyelv: Olyan programozási nyelv, amely támogatja az objektumazonos-ságot, az osztályozást és a bezárást, de nem támogatja az öröklést.

Nézzük most át a legelterjedtebb OO nyelveket:

◆ **C++**

Hibrid nyelv. Az AT&T Bell Laboratories fejlesztette ki a C nyelvből az 1980-as évek elején, Bjarne Stroustrup tervei alapján. Napjaink OO rendszereinek jókora hányadát C++ nyelven írják. A C++ legnépszerűbb változata a Borland C++ és a Microsoft C++.

◆ **Java**

Tiszta OO nyelv. A Sun Microsystems egy csoportja, a JavaSoft fejlesztette ki James Gosling vezetésével. A Java a C++ átdolgozott, leegyszerűsített változata; az a legnagyobb erőssége, hogy alkalmas az Internet programozására. A fejlesztői környezet és Java osztálykönyvtár magja a szabványos JDK (Java Development Kit); ennek alapján több cég is gyárt fordító, illetve futtató programot.

◆ **Pascal**

Hibrid nyelv. Objektumorientált változatát először az Apple Computer fejlesztette ki 1986-ban Niklaus Wirth-tel, a Pascal tervezőjével egyetértésben. Legnépszerűbb változata az Object Pascal, a Delphi programozási nyelве (Borland International, Inc.).

◆ **Smalltalk**

Tiszta OO nyelv, legnépszerűbb változatai a Smalltalk/V (Digitalk, Inc.), a Smalltalk-80 (ParcPlace Systems, Inc.) és az IBM Smalltalk (IBM Corporation).

- ◆ **Eiffel**

Tiszta OO nyelv, Bertrand Meyer fejlesztette ki az 1980-as években (Interactive Software Engineering, Inc.).

OO vizuális fejlesztőeszközök

Egy vizuális fejlesztőeszköz birtokában a fejlesztő a program nagy részét nem forráskód írásával készíti, hanem interaktív módon, a fejlesztőrendszer eszköztára által felkínált vizuális elemek kiválasztásával, azok tulajdonságainak beállításával. A forráskód vázát a fejlesztőeszköz generálja, majd a fejlesztő kiegészíti.

A vizuális fejlesztőeszköz erősen megkönnyíti a programozó munkáját. Az éppen fejlesztett program elemeit (például egy dialógusdobozt és az azon levő beviteli sorokat és nyomógombokat) nem programnyelvi utasításokkal állítjuk elő, hanem a képernyön, interaktív módon adjuk meg: az elemeket egy eszköztárból „cipeljük le” a tervezett helyre, és addig igazítjuk őket, amíg jók és tetszetősek nem lesznek. Az elemek tulajdonságait is menüből választhatjuk. A megfelelő adatbázis-kezelést végző komponenst is interaktív módon helyezzük programunkba. A vizuális szerkesztés alapján a fejlesztőeszköz automatikusan elkészít egy forráskódvázat, ezt aztán a programozó kiegészíti. De sokszor ez a „kiegészítés” a program lelke, s ezt nem lehet kattintgatással helyettesíteni. Az alkalmazás logikáját továbbra is a fejlesztőnek kell kigondolnia! **Bármennyire kínálkozik is egy vizuális fejlesztőeszköz az azonnali programkészítésre, a programozást minden esetben tervezésnek kell megelőznie!**

Vizuális eszközök alkalmazásával könnyen készíthetünk a felhasználónak **prototípust** (programmintát) – így a fejlesztő kevésbé árul „zsákbamacskát”.

Elterjedt vizuális fejlesztőeszközök a következők:

- ◆ Java alapú: JBuilder (Borland), JDeveloper (Oracle), IBM VisualAge for Java, Visual J++ (Microsoft)
- ◆ C++ alapú: Borland C++ Builder, Microsoft Visual C++
- ◆ Object Pascal alapú: Delphi (Borland)
- ◆ Smalltalk alapú: IBM VisualAge for Smalltalk, Visual Smalltalk (ParcPlace-Digitalk)

OO adatbázis-kezelők

OO adatbázis-kezelő rendszerek segítségével az **objektumokat** adataikkal és viselkedéseivel együtt **perzisztens** módon (maradandóan, a programfutás idején túl is) el lehet tárolni és különböző szempontok szerint visszakeresni.

Az objektumorientált adatbázis-kezelő rendszerek csak a 90-es években kezdtek életre kelni, és még mindig nem igazán népszerűek. Talán azért nem tudnak betörni a piacra, mert egyelőre bonyolultak, lassúak, és nincs meg a megfelelő matematikai hátterük. Ezzel szemben egy

relációs adatbázis-kezelő rendszer megbízható, használata egyszerű; nem beszélve arról, hogy szabványos a lekérdező nyelve, az SQL (Structured Query Language). Átmeneti megoldásként sorban jönnek létre az ún. hibrid adatbázis-kezelő rendszerek, melyekbe OO funkcionálitást építenek be, így az adatbázis használója „OO szemüvegen” át látja a relációs adatbázist.

OO adatbázis-kezelők például:

- ◆ ObjectStore (Object Design Inc.)
- ◆ Versant (Versant Object Technology)

10.2. A Java nyelv története

A Java története 1991-re nyúlik vissza, amikor a Sun MicroSystems egy csoporthoz Patrick Naughton és James Gosling vezetésével egy beágyazott, számítógépes mini nyelvet tervezett olyan kommunikációs eszközök programozására, mint például egy kábel-TV kapcsoló doboza. Mivel az ilyen eszközök nem túl gyorsak, és memoriájuk is szűkös, a nyelvnek nagyon kicsinek, a lefordított kódnak pedig roppant hatékonyak kellett lennie. És közrejátszott még egy fontos szempont: mivel valósínűsíthető volt, hogy a különböző gyártók különböző típusú processzorokat fognak választani, ezért a nyelvet általánosra kellett megalkotni, nem volt szabad azt egy megadott architektúrára szűkíteni. Ez volt a „Green” projekt.

Kapóra jött a csapatnak az az ötlet, amelyet Niklaus Wirth már az UCSD Pascalban kidolgozott. Eszerint olyan hordozható nyelvet kell készíteni, amely egy virtuális (hipotetikus, elképzelt, nem létező) gépre közbenső kódot generál – innen ered a virtuális gép, vagyis a Java Virtual Machine (JVM) elnevezés. Ez a közbenső bájtkód aztán használható minden olyan gépen, amelyen megvan a megfelelő interpreter. James Gosling és társai a C++ nyelvet egyszerűsítették le a kívánt cél érdekében, hiszen unixos közegből jöttek, és addig C++-ban dolgoztak. Az új nyelvnek eredetileg az Oak (tölgy) nevet adták, mint mondják, azért, mert a Sunnál Gosling ablaka előtt egy gyönyörűszép tölgyfa állt. Később azonban a Sun emberei felfedezték, hogy Oak nevű programnyelv már létezik. És miközben azon töprengtek, mi is legyen az új név, nagy élvezettel fogyasztották a gőzölgő kávéjukat. **E finom, történelmi kávé a Java nevet viselte, utalva ezzel származási helyére...**

A Java nyelv nagyon jóra sikerült, s ezt érezték az alkotói. Terméket el szerették volna adni. 1992-től több év házalás következett, de szerződést senki sem kötött velük. A legérdekesebb, hogy még Jim Clark sem látta meg a Javában a jövőt, az az ember, aki később mint a Netscape vezetője leginkább támogatta a nyelvet.

Míg a szerzők házaltak, az Internet és a Web (lásd 10.3 pont) óriási sebességgel fejlődött. Mivel a webtechnika kulcskérdése a hatékony browser (böngésző) volt, a Java fejlesztőinek az a gondolatuk támadt, hogy új nyelvük kiválóan alkalmas lenne az „igazi” böngésző elkészítésére. Patrick Naughton és Jonathan Payne 1994-ben el is készítette a HotJava böngészőt. Belecsempészték a böngészőbe a Java futtatórendszerét, a JVM-et, s ezzel alkalmassá tették azt a

Java fordító által generált bájtkód futtatására. A honlapok életre keltek: már nem csak mozdulatlan információkat jeleníthettek meg, hanem mozgó ábrákat, animációkat és videófilmeket is. Egyszóval a távoli felhasználó honlapján futottak a Java programok! A szerzők 1995. május 23-án bemutatták terméküket, és ezzel a Java elindult diadalújtán: 1996 januárjában a Netscape 2.0 már Java-képes (Java enabled) volt, s ezután minden gyártó beállt a sorba: ma már nincs böngésző, amely ne futtná a Java appleteket (a böngészőben futó Java alkalmazást appletnek, „alkalmazáská”-nak nevezzük). A világ fejlesztői egyre mohóból „itták közös csészéjükön a gőzölgő kávét”. Az „a cup of Java” (egy csésze Java) egyre népszerűbbé vált.

A **Java nyelvet** a Sun MicroSystems egy csoportja fejlesztette ki, eredetileg kommunikációs eszközök programozására alkalmas, C++ alapú nyelvként. Alapvető cél volt a kisméretű nyelv, a hatékony kód és a hordozhatóság. A Java nyelv akkor lett igazán sikeres, amikor 1995-től kezdve futatórendszerét beépítették a böngészőkbe, s ezzel a böngészők Java-képessé váltak.

10.3. Az Internet, a World Wide Web és a Java

Mivel a Java nyelv elsősorban az Internet programozási nyelve, azért most néhány szóban összefoglaljuk a szorosan ide kapcsolódó fogalmakat.

Az Internet és a World Wide Web

Az Internet: számítógépekkel álló világháló. A számítógépek a szabványos TCP/IP (Transmission Control Protocol/Internet Protocol) technológiával kapcsolódnak egymáshoz. Az Internet segítségével bármely két, az Internetbe bekapcsolt számítógép „beszélgethet” egymással. Internet már a 60-as években is létezett, de csak 1995-től vált ilyen népszerűvé, a World Wide Web jóvoltából (World Wide Web = világméretű háló, WWW vagy egyszerűen Web). **A WWW elektronikus információs tárház**, amely az Interneten keresztül a világ bármely pontjáról elérhető. A lehetőségek korlátlanok: a hálózatra kötött bármely számítógépről szállodai szobát vagy repülőjegyet foglalhatunk, lekérdezhetjük a világ tetszőleges városának térképét, moziműsorát vagy időjárásjelentését, otthonról intézhetjük vásárlásainkat, kezelhetjük a bankszámlánkat stb. Az információs tárház természetesen sohasem teljes – azon bármi fent lehet, és bármi hiányozhat is belőle, hiszen ki-ki azt tesz számítógépének „kirakata”, amit akar. Az Internet és a WWW fogalma lassan összemossódik.

Az Internetet nemcsak a számítástechnikához értő szakember használja – az nélkülözhetetlen segítőtársa a XXI. emberének.

URL (Uniform Resource Locator)

Egy távoli gépen levő erőforrás (például dokumentum) egyértelmű azonosítója. Például:
<http://www.gdf.hu/angster/ook/info.htm>

Az erőforrás azonosítása három részből áll:

- ◆ Protokollmegadás. Itt: <http://> (HyperText Transfer Protokol)
- ◆ Az erőforrás helye, vagyis a szerver, a távoli gép neve. Itt: www.gdf.hu
- ◆ A dokumentum neve teljes útvonalallal. Itt: [/angster/ook/info.htm](#)

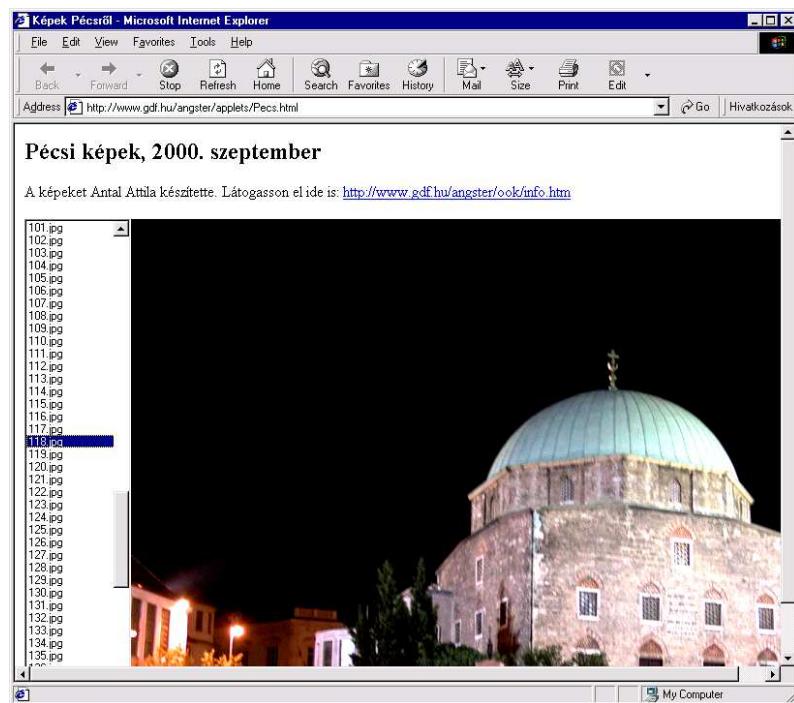
Browser, hypertext, applet

Az információs tárházban böngészőkkel (angolul browser, például a Netscape Navigator vagy az Internet Explorer) böngészhetünk. A **böngésző hiperszövegeket jelenít meg**. A hiperszöveg (hypertext) olyan dokumentum, amelynek bizonyos részeire egérrel rákattintva újabb dokumentumokat, dokumentumrészeket érhetünk el. A szövegnek ezek az érzékeny pontjai aláhúzással vagy más módon vannak megkülönböztetve. Az érzékeny pontokhoz hiperhivatkozások (hyperlink) tartoznak. A 10.1. ábrán látható, hogy az Internet Explorer a <http://www.gdf.hu/angster/applets/Pecs.html> lapot jelenítette meg. Ez a lap a Gábor Dénes Főiskola (GDF) szervevének /angster/applets könyvtárában található, Pecs.html néven.

Egy böngésző „Java-képes” (Java enabled), ha futtatja a Java appleteket. A böngészőtől függetlenül futó programot alkalmazásnak nevezik (application, applikáció), a böngészőben futó programot pedig appletnek (alkalmazáska). Egy applet biztonsági okok miatt sokkal kevesebb dologra képes, mint egy applikáció: mások gépen például egyáltalán nem tudja kezelní az állományokat. Böngészéskor nem kell tehát attól tartanunk, hogy a számítógépünkre letöltött applet letörli, módosítja, netán elolvassa állományainkat.

Hiperszövegek megadására egy egyszerű hiperszöveg-szerkesztő nyelvet fejlesztettek ki, a HTML-t (HyperText Markup Language). A **hiperszöveg egyszerű szöveges állomány, melyben HTML parancsok találhatók**. A 10.2. ábrán látható a Pecs.html állomány: ez jeleníti meg a 10.1. ábrán látható képet. Láthatók benne a HTML parancsok; egy ilyen parancsot a <PARANCS> vezet be, és a </PARANCS> zár le. Például:

| | |
|---|--|
| <HTML> ... </HTML> | HTML dokumentum |
| <HEAD> ... </HEAD> | A böngésző ablakának beállításai |
| <TITLE> ... </TITLE> | A böngésző ablakának címe |
| <BODY> ... </BODY> | A lap (hiperszöveg) tartalma |
| <H2> ... </H2> | Fejléc (header) |
| <P> ... </P> | Paragrafus |
| <U> ... </U> | Aláhúzás (underline) |
| <A> ... | Hiperhivatkozás megadása (anchor = horgony) |
| <APPLET code=... width=... height=... > </APPLET> | Beágyazott applet (code: a futtatandó applet bájkódja, width, height: az applet szélessége és magassága) |



10.1. ábra: Böngésző, s benne egy applet

```

<HTML>
<HEAD>
<TITLE>Képek Pécsről</TITLE>
</HEAD>
<BODY>
<h2 align = middle>Pécsi képek, 2000. szeptember</h2>
<P>A képeket Antal Attila készítette. Látogasson el ide is:
<A HREF="http://www.gdf.hu/angster/ook/info.htm">
<U>http://www.gdf.hu/angster/ook/info.htm</U></A></P>
<APPLET code = "ViewPictures.class"
        width = 800
        height = 500
>
</APPLET>
</BODY>
</HTML>

```

10.2. ábra: A HTML forrásszöveg: ez állítja elő az előbbi képet

A hiperszövegeket szokás HTML dokumentumoknak is nevezni; állománynevük HTML vagy HTM kiterjesztésű. Ha egy ilyen kiterjesztésű file-ra kettőt rakkattintunk, akkor az várhatóan megjelenik a böngészőben. Bár a HTML dokumentumok egyszerű szövegszerkesztőkkel is szerkeszthetők, bonyolultabb hiperszövegek szerkesztésére külön HTML szerkesztő szoftvereket fejlesztettek ki (például a FrontPage Editort).

10.4. A Java nyelv jellemzői

A Java nyelv fejlesztői hivatalos kiadványban (white paper) tették közzé tervezési céljaikat és eredményeiket. Ez a kiadvány 11 szóval jellemzi a Java nyelvet:

- ◆ **Egyszerű** (simple): A Java nyelv a C++ leegyszerűsített változata. A C++ a Java előtt a legnépszerűbb programozási nyelv volt, de kötetlensége és túlzottan sok lehetősége miatt bonyolult és nehézkes volt használni. A Java sokkal kevesebb nyelvi eszközt, lehetőséget kínál, és sokkal nagyobbak benne a kötöttségek. Leegyszerűsödött a szintaktika: eltüntek a mutatók, automatikus lett a memória felszabadítása (már nem a programozónak kell róla gondoskodnia). Emiatt egy Java programot írni vagy olvasni sokkal egyszerűbb, mint egy C++ programot. Némelyek a Javát – igencsak szellemesen – így hívják: C++-- (CPluszPluszMínuszMínusz).
- ◆ **Objektumorientált** (object-oriented): A Java OO szellemű, tiszta objektumorientált nyelv. Egy Java applikációt osztályok készítésével és újrafelhasználásával építünk összszere.
- ◆ **Elosztott** (distributed): Egy Java applikáció képes az Internet bármely pontján található, URL-lel azonosított objektumot elérni és feldolgozni.
- ◆ **Robusztus** (robust): Más néven: hibatűrő, megbízható. A szoftver hibatűrő, ha a lehető ségekhez képest normálisan működik az abnormális esetekben is. A nyelv tervezői nagy gondot fordítottak a hibák korai, még a fordítás idején való kiszűrésére, hogy elejét vegyék a lehetséges futási hibáknak.
- ◆ **Biztonságos** (secure): A Java nyelvet elsősorban internetes, elosztott környezetben való működésre terveztek. Ezért jelentős biztonsági intézkedéseket kellett bevezetni, nehogy egy ilyen program kárt tehessen a másik felhasználó rendszerében. A Java nyelv használatával olyan programokat állítunk elő, amelyek nem érhetnek el védett eszközöket és állományokat.
- ◆ **Architektúrasemleges** (architecture neutral): más szóval: gépfüggetlen. A fordítóprogram géptől független bájtkódot (.class file) generál, s az különböző gépek processzorain futtatható. A futtatás feltétele a futtató környezet (JVM, Java Virtual Machine) jelenléte. A bájtkód unikód formátumú és szabványos, számítógépes környezettől és processzortól független instrukciókból áll. A bájtkódot az adott gép futás közben értelmezi a virtuális gép segítségével. A JVM által véglegesre lefordított kód a natív kód, s az már ténylegesen fut a gépen. A bájtkód használatának pillanatnyilag az a hátránya, hogy az értel-

mező utasításonként fordít, és emiatt a program meglehetősen lassú. A teljesítményt a JIT (Just In Time) fordítók jócskán megnövelik, mert megjegyzik az egyszer már lefordított kódot, és a legközelebbi hivatkozáskor már nem fordítják újra.

- ◆ **Hordozható** (portable): A nyelvnek nincsenek implementációfüggő elemei, azaz nem fordulhat elő olyan eset, hogy egy nyelvi elem vagy osztály az egyik környezetben más-képpen legyen specifikálva, mint a másikban. Az int (integer = egész) adattípus például a szabvány szerint egy 32 bites előjeles egész, bármilyen környezetről (számítógép, operációs rendszer, Java fordító) van is szó.
- ◆ **Interpretált** (interpreted): A célgépen futó natív kódot az értelmező hozza létre utasításonként a bájtkód értelmezésével. Ha egy célgépen installálnak egy Java értelmezőt, akkor az bármilyen Java bájtkódot értelmezhet.
- ◆ **Nagy teljesítményű** (high performance): A Java magas teljesítménye még elérődő cél. A Javát ezen a téren éri a legtöbb támadás: sok fejlesztő szerint a fordítás és a futtatás is idegesítően lassú. A következő tendenciák azonban reményt keltenek:
 - a processzorok gyors ütemben fejlődnek, egyre gyorsabbak lesznek;
 - a Java fejlesztői óriási összegeket költenek ez irányú kutatásokra;
 - a JIT (Just In Time) fordítók már 10-20-szor gyorsabbak az értelmezőknél.
- ◆ **Többszálú** (multithreaded): A többszálú programozás lényegében azt jelenti, hogy ugyanabban az időben több programrész futhat egymással párhuzamosan, több szalon. Igaz, hogy egyszerre csak egy processzor dolgozik, de az mindenki szál munkáját igazságosan, „egyszerre” végzi, például:
 - amíg az egyik szál bevitelre vár, addig a másik nyugodtan dolgozik;
 - a program egy videófilm lejátszása közben például nyugodtan elvégezhet bizonyos számításokat, hiszen a képeket úgysem lehet egy adott sebességnél gyorsabban mutogatni.
- ◆ **Dinamikus** (dynamic): A Javát úgy terveztek, hogy könnyedén tovább lehessen fejleszteni. Az osztálykönyvtárak szabadon bővíthetők anélkül, hogy azok hatással lennének az öket használó kliensekre.

Tesztkérdések

- 10.1. Ki fejlesztette ki a Java nyelvet? Jelölje meg az egyetlen jó választ!
 - a) az AT & T Bell Laboratories
 - b) a Sun Microsystems
 - c) a Smalltalk
 - d) az IBM
- 10.2. Jelölje meg azokat a pontokat, amelyekben az összes nyelv tiszta objektumorientált!
 - a) Java, Object Pascal, C++
 - b) Eiffel, Smalltalk
 - c) Turbo Pascal, Java, Smalltalk
 - d) Eiffel, C++, Java

- 10.3. Jelölje meg az összes igaz állítást a következők közül!
- a) A vizuális fejlesztőeszközt használó programozó csak viszonylagosan adja meg a program tulajdonságait, forráskódot nem kell írnia.
 - b) Az OO adatbázis-kezelő rendszerek az objektumoknak csak az adatait tárolja el, a viselkedését nem.
 - c) A Java nyelv egy ősi játékról kapta a nevét.
 - d) A Java akkor lett igazán sikeres, amikor futtató rendszerét beépítették a böngészőkbe.
- 10.4. Jelölje meg az összes igaz állítást a következők közül!
- a) Az Internet nem más, mint számítógépekből álló vilagháló.
 - b) Egy távoli gépen levő dokumentumot az URL-je azonosít.
 - c) A WWW a World, World, World! rövidítése
 - d) a bohoc@where.hu szintaktikailag helyes URL.
- 10.5. Melyik volt az első olyan programozási nyelv, amely már tartalmazott OO elveket?
Jelölje meg az egyetlen jó választ!
- a) Smalltalk
 - b) C++
 - c) Simula
 - d) ACM
- 10.6. Melyik évben építették be a böngészőkbe a JVM-et? Jelölje meg az egyetlen jó választ!
- a) 1980-ben
 - b) 1985-ben
 - c) 1990-ben
 - d) 1995-ben
- 10.7. Jelölje meg az összes igaz állítást a következők közül!
- a) A böngészővel hiperszöveget lehet készíteni.
 - b) Az, hogy a böngésző Java-képes, azt jelenti, hogy alkalmaz Java appletek futtatására.
 - c) A HTML egyfajta nyelv.
 - d) A Java nyelvnek az az egyik jellemzője, hogy elosztott alkalmazásokat lehet benne írni.
- 10.8. Mely célokat tüzték ki a Java nyelv fejlesztői? Jelölje meg az összes jó választ!
- A Java legyen:
- a) Biztonságos
 - b) Felhasználóbarát
 - c) Robusztus
 - d) Egyszerű

I. BEVEZETÉS A PROGRAMOZÁSBA

1. A számítógép és a szoftver
2. Adat, algoritmus
3. A szoftver fejlesztése

II. OBJEKTUMORIENTÁLT PARADIGMA

4. Mitől objektumorientált egy program?
5. Objektum, osztály
6. Társítási kapcsolatok
7. Öröklődés
8. Egyszerű OO terv – Esettanulmány

III. JAVA KÖRNYEZET

9. Fejlesztési környezet – Első programunk
10. A Java nyelvről



IV.

IV. JAVA PROGRAMOZÁSI ALAPOK

11. Alapfogalmak
12. Kifejezések, értékkadás
13. Szelekciók
14. Iterációk
15. Metódusok írása

V. OSZTÁLYOK HASZNÁLATA, KÉSZÍTÉSE

16. Objektumok, karakterláncok, csomagolók
17. Osztály készítése

VI. KONTÉNEREK

18. Tömbök
19. Rendezés, keresés, karbantartás
20. A Vector és a Collections osztály

FÜGGELÉK

- A tesztkérdések megoldásai
Irodalomjegyzék
Tárgymutató

11. Alapfogalmak

A fejezet pontjai:

1. Mintaprogram – Krumpli
 2. ASCII és unikód karakterek
 3. A program alkotóelemei
 4. Változó, típus
 5. Primitív típusok
 6. A forrásprogram szerkezete
 7. Metódushívás (üzenet)
 8. Értékkedő utasítás
 9. Adatok bevitel a konzolról
 10. Megjelenítés a konzolon
-

A fejezetben a Java program alapvető fogalmait és szintaktikai szabályait tisztázzuk. Az első pontban bemutatunk egy mintaprogramot, melyre az egész fejezetben hivatkozni fogunk.

11.1. Mintaprogram – Krumpli

Feladat – Krumpli

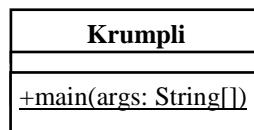
Egy cserkésztáborban készül a vacsora, de a konyhafőnök nem tudja, mennyi krumplit vegyen. A fejadagot ismeri, az 0.4 kg, de a szükséges mennyiséget számítógép segítségével szeretné kiszámolni az aktuális létszámtól függően. Írjuk meg a programot a konyhafőnöknek! A program kérje be a létszámot, majd írja ki a szükséges mennyiséget a következőképpen:

Létszám? 20

A szükséges mennyiség: $20 * 0.4 \text{ kg} = 8.0 \text{ kg}$

A `Krumpli.java` mintaprogramunk egyetlen osztályból, az pedig egyetlen metódusból áll. A program összes feladatát a statikus `main` metódus fogja elvégezni, a `Krumpli` osztályból példányt nem hozunk létre.

A mintaprogram terve a következő:



Forráskód

```
/* A program a cserkésztábor konyhafőnöke részére készült.
 * A program bekéri a létszámot, majd kiírja a krumpli
 * szükséges mennyiségett.
 */
import extra.*; // az extra egy saját csomag, nem API

public class Krumpli {
    public static void main(String[] args) {
        int letszam;
        double fejadag=0.4, osszesen;

        // A letszam változó bekérése konzolról:
        letszam = Console.readInt("Létszám? ");

        // A szükséges mennyiség kiszámítása és kiírása:
        osszesen = fejadag*letszam;
        System.out.print("A szükséges mennyiség: ");
        System.out.println(letszam+ " * "+fejadag+" kg = "
            +Format.left(osszesen,0,2)+" kg");
    }
}
```

A program egy lehetséges futása

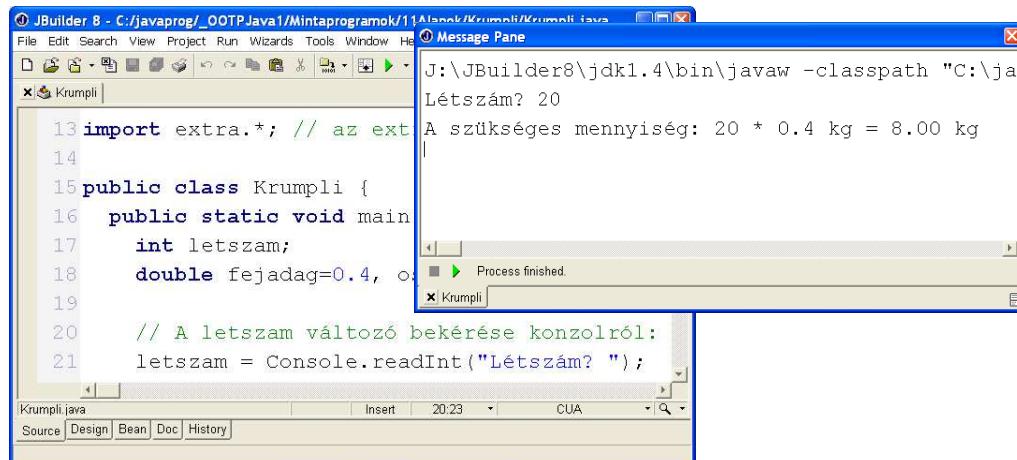
```
| Létszám? 20
| A szükséges mennyiség: 20 * 0.4 kg = 8.00 kg
```

Nyissa meg a javaprojektet, és adja hozzá a Krumpli.java forráskódot! A forráskód mappája: C:\javaproj_OOTPJava1\Mintaprogramok\11Alapok\Krumpli. Fordítsa és futtassa a programot a 9. fejezetben leírtak szerint!

Mielőtt megpróbálná lefordítani a programot, fontos megjegyezni a következőt: A program működéséhez szükség van az extra.Console és az extra.Format osztályra, mely nem része a Javának (a program importálja az extra csomag összes osztályát). Ehhez konfigurálja a javalib könyvtárat, ha ezt még nem tette meg! A javalib könyvtár struktúrája a következő:

```
javalib      // az extra csomag könyvtára (mappa vagy jar fájl)
extra        // extra csomag
Console     // Console osztály
Format       // Format osztály
...
...
```

A program futását a JBuilder alkalmazásböngészőjének üzenetpanelén a 11.1. ábra mutatja (az üzenetpanel külön ablakká varázsolható a bal alsó sarkában levő ikonnal).



11.1. ábra. A Krumpli program futása

A `Console` osztály többek között tartalmazza a `readInt()` statikus metódust (a Javában előbb írjuk a típust, aztán a változót):

- `int readInt(String str)`
A metódus segítségével konzolról be lehet olvasni egy egész számot. A konzolon megjelenik a paraméterben megadott információs szöveg. A kurzor a szöveg után villog, jelezve, hogy a program bevitelre vár. A program addig nem folytatódik, amíg egy egész értéket be nem tűnök. Ekkor a függvény visszatérési értéke a bevitt egész szám lesz.

A mintaprogramban a `letszam` változó értékét kértük be a felhasználótól:

```
letszam = Console.readInt("Létszám? ");
```

11.2. ASCII és unikód karakterek

Mindenekelőtt tisztázzuk az ASCII és az unikód karakterábrázolás fogalmát, mert a továbbiakban alkalmazni fogjuk őket. Karaktereknek nevezzük együttesen a betűket (A, B..., a, b...), a számjegyeket (0, 1, 2...), a speciális karaktereket (*, +, !...) és a vezérlőkaraktereket (Enter, Backspace, PageDown, Shift-F1...). Ezeket a karaktereket a számítógépnek valahogy meg kell különböztetnie egymástól. Kétféle karakterábrázolás terjedt el: az ASCII és az Unikód.

ASCII karakterkészlet

Az ASCII (American Standard Code for Information Interchange, az információcsere amerikai szabványos kódjai) kódtábla egy karaktert egy bájton ábrázol, vagyis összesen 256 karaktert különböztet meg. Az ASCII kódtáblát az amerikaiak 1963-ban rögzítették.

Az ASCII karakterek között vannak **megjeleníthető karakterek**, és vannak **vezérlő**, más szóval **kontroll** vagy **escape karakterek**. A megjeleníthető karaktereket szimpla aposztrófok közé szokás írni, például 'A', '\$'. A vezérlőkarakter elnevezés onnan ered, hogy ha egy ilyen karaktert a képernyőre vagy printerre küldünk, akkor az nem jelenik ott meg, hanem valamelyen vezérlő (kontroll) funkciót lát el, például sort emel, lapot dob stb. A vezérlőkarakterek az ASCII kódtábla 1.-26. karakterei. Jelölésük: Control-A, Control-B... vagy másképpen ^A, ^B ... az ábécé betűinek sorrendjében.

Az ASCII kódtábla megjeleníthető elemei:

- ◆ a számjegyek (10 db: 0,1,2,...9), ASCII kódjuk 48-tól 57-ig
- ◆ az angol ábécé nagybetűi (26 db: A,B,C,...), ASCII kódjuk 65-től 90-ig
- ◆ az angol ábécé kisbetűi (26 db: a,b,c,...), ASCII kódjuk 97-től 122-ig
- ◆ speciális karakterek (mintegy 40 db, pl. !,\$,%,...)
- ◆ grafikus jelek (pl. ¶, ■, ☺)

Az ASCII kódtábla a 256 karakterből tulajdonképpen csak az első 128-at adja meg, a tábla második felével szabadon lehet gazdálkodni. Ezekre a karakterekre az ANSI (American National Standards Institute) külön ajánlást adott, mi, magyarok pedig ide tehetjük a nekünk fontos "ő" betűt és társait.

A hatvanas években még senki sem gondolt arra, hogy az 1 bájt mennyire kevés lesz, micsoda gondot okoz majd a különböző nemzetek ékezes betűinek tárolása.

Unikód karakterkészlet

A számítástechnika fejlődésével és elterjedésével az ASCII kódtábla igencsak szükségesnek bizonyult. A világ különböző népei saját nyelvükön szeretnék használni számítógépüköt, szövegeiket saját nyelvükön szeretnék beütni és látni. E probléma megoldására dolgozták ki az **unikód karakterkészletet**, mely már 2 bájton ábrázolja a lehetséges karaktereket. A lehetőségek száma így 256-ról 65536-ra emelkedett, és ez böven elegendő az angol, magyar, orosz, kínai, szanszkrit vagy hottentotta betűk megkülönböztetésére is (pillanatnyilag mintegy 35000-féle karaktert használnak a világ számítógépein).

Az unikód karaktereket leggyakrabban hexadecimális kódjukkal adják meg \uhhhh formában, ahol a \u (backslash u) az unikód ábrázolást bevezető vezérlőkarakter, hhhh pedig négy hexadecimális szám (1..9, a..f/A..F). Az unikód karakterek a \u0000 karaktertől a \uffff karakterig terjednek. Az első 256 unikód karakter megegyezik az ASCII/ANSI karakterkészlettel, például:

| Unikód | ASCII |
|--------|-----------|
| \u001a | Control-Z |
| \u0041 | A |
| \u0031 | 1 |

Bizonyos vezérlőkarakterek jelölésére a Java speciális vezérlő szekvenciákat engedélyez. Ilyenkor a \ jelet egyetlen, u-tól különböző karakter követi:

| Escape szekvencia | Név | Unikód érték |
|-------------------|--------------------------------|--------------|
| \b | backspace (visszatörlés) | \u0008 |
| \t | tab (tabulátor) | \u0009 |
| \n | line feed (soremelés) | \u000a |
| \f | form feed (lapdobás) | \u000c |
| \r | carriage return (kocsi vissza) | \u000d |
| \" | dupla aposztróf | \u0022 |
| \' | szimpla aposztróf | \u0027 |
| \\" | backslash (\ jel) | \u005c |

Megjegyzés: A Javában egy szöveget dupla aposztrófok között adunk meg ("Hello"). Ezért magában a szövegben nem szerepelhetne a " karakter, ha azt a vezérlőkarakteres trükkkel nem szúrhatnánk mégiscsak be ("De \"Trükkös\" vagy! ").

Az unikód karaktereket a Unicode Consortium felügyeli és tartja karban. Bővebb információ a www.unicode.org Internet címen található.

ASCII karakterkészlet: Szabványos kódrendszer, mely a karaktereket 1 bájton ábrázolja.

Unikód karakterkészlet: Szabványos kódrendszer, mely a karaktereket 2 bájton ábrázolja. Jelölése: \uhhhh, ahol hhhh hexadecimális számok. Az unikód karakterek értéke \u0000 és \uffff közé esik.

11.3. A program alkotóelemei

Elemezzük most a mintaprogramot szintaktikailag! Egy Java forrásprogram unikód karakterből áll, és megkülönbözteti a kis- és nagybetűket.

A **Java program** szintaktikailag elkülöníthető **alkotóelemei** a következők:

- Fehér szóköz
- Megjegyzés
- Azonosító
- Kulcsszó
- Literál
- Szeparátor
- Operátor

A program alapelemei

Vegyük most sorra ezeket az alkotóelemeket!

Fehér szóköz

A fehér szóköz (white space) elválasztja a program alapelemeit. Fehér szóköznek nevezük a szóközt, a tabulátort, a sor- és lapvégjeleket:

| Fehér szóköz | Angol név | Rövidítés | Unikód | Esc. szekv. |
|----------------|-----------------|-----------|--------|-------------|
| Szóköz | Space | SP | \u0020 | |
| Vízszintes Tab | Horizontal Tab | HT | \u0009 | \t |
| Sor eleje | Carriage Return | CR | \u000d | \r |
| Új sor | Line Feed | LF | \u000a | \n |
| Lapvég | Form Feed | FF | \u000c | \f |

Mintaprogramunkban a `public` és a `class` kulcsszavakat például szóköz választja el. Ha a szóközt elhagynánk, a `public class` egyetlen alkotóelem lenne.

A fehér szóközök a program szövegében csupán „üres helyeknek” látszanak. A szintaktika szerint, ahol egy fehér szóköz van, ott lehet több is. A megjegyzés, az operátor és a szeparátor automatikusan elválasztó szerepet tölt be. Például a `fejadar` és a `letszam` azonosítót a `*` (szorzás) operátor már elválasztja, további szóközre tehát nincs szükség. Ugyanígy a sorvégi `;` (pontosvessző) után sem kellene sort emelnünk. Szépen, strukturáltan azonban csak úgy tudunk kódolni, ha a kód megfelelő helyeire fehér szóközöket szűrünk be, de ez természetesen felesleges a fordítóprogram számára.

Megjegyzés: Mintaprogramunk, mint minden Java program, egyszerű szöveges állomány. Ha a szöveg szerkesztésekor lenyomjuk a Tab vagy az Enter billentyűt, akkor a memóriában a megfelelő megjeleníthető karakterek között egy HT, illetve egy LF karakter jelenik meg. Amikor a szövegszerkesztő konzolra írja a szöveget, akkor a HT karakter hatására tabulálás, az LF hatására pedig soraelés következik be.

Például a `"{\n\t//Proba\n}"` szöveg a képernyón így fest:

```
{\n    //Proba\n}
```

Megjegyzés

A megjegyzés (comment) emlékeztető vagy dokumentáló célokat szolgáló szöveg. A megjegyzésekkel a Java fordító nem veszi figyelembe. A jó megjegyzésekkel megtüzdelt forrásprogram olvashatóbb és érthetőbb, mint a megjegyzésekkel mellőzött kód.

A Java programban alapvetően kétféle megjegyzést alkalmazhatunk:

- ◆ Egysoros megjegyzés: a `//` karakterpár után lévő szöveg az aktuális sor végéig. A következő sor már nem tartozik a megjegyzéshez.
- ◆ Többsoros megjegyzés: a `/*` és `*/` karakterpárok közé zárt szöveg. Ebben az esetben a megjegyzés akármilyen hosszú lehet, akár több soros is.

A megjegyzések nem ágyazhatók egymásba. A `/*` és `*/` jelek közé tehát hiába illesztünk `//` jeleket, azt a fordító nem veszi figyelembe.

● Ha elkezdett egy megjegyzést, akkor ne felejtse el bezárni! Lehet, hogy csak azért nem működik a programja, mert a fordító nem vesz tudomást a program utasításairól!

Mintaprogramunkban egy darab háromsoros és három darab egysoros megjegyzés van.

Megjegyzések:

- A megjegyzés alkalmazása egy jó forráskód elengedhetetlen velejárója ugyan, de nem helyettesíti a „beszédes” azonosítókat. A forráskód legyen öndokumentáló!
- Van még egy harmadik fajta megjegyzés is, amelyet a `/**` és `*/` karakterSORozatok közé teszünk. Ezt a fajta megjegyzést a `javadoc.exe` dokumentációkészítő szoftver használja fel a program `html` dokumentációjának elkészítéséhez.

Azonosító

A program bizonyos elemeit (osztályait, objektumait, változóit, metódusait stb.) azonosítani kell, vagyis azoknak nevet kell adni. Vannak olyan elemek, melyeket már a Java írói azonosítottak, és nekünk csak hivatkozni kell rájuk – ilyen például a `String`, a `System`, az `out`, és a `println`. Vannak elemek, melyeket a programozó azonosít – ilyen például a `Krumpli`, a `letszam`, a `fejadag`, és az `osszesen`. Az azonosítónak egy adott programszinten (csomagban, illetve blokkban) egyedinek kell lennie: ha például a fő osztály neve `Krumpli`, akkor egy másik osztálynak már nem lehet `Krumpli` a neve.

Azonosítóképzési szabályok

- ◆ Az azonosító bármilyen hosszú unikód karaktersorozat, melyben a karakterek a következők lehetnek:
 - Java betűk: az unikód táblában szereplő betűk, köztük a latin ABC kis- és nagybetűi;
 - Java számjegyek: az unikód táblában szereplő számjegyek, köztük a `0..9` számjegyek;
 - `_` (aláhúzás) karakter és a valutaszimbólumok (pl. `$`).
- ◆ Egy azonosító nem kezdődhet számjeggyel, nem lehet kulcsszó, és nem lehet a nyelv előre definiált konstansa (`true`, `false` vagy `null`).
- ◆ Két azonosító csak akkor egyezik meg, ha unikód karaktereik rendre megegyeznek.

◆ Vigyázat! Az azonosítók esetében megkülönböztetjük a kis- és a nagybetűket. Ha system helyett a system szót gépeljük be, akkor az szintaktikai hibát fog okozni.

Megjegyzések:

- Hogy egy unikód karakter betű-e, az lekérdezhető a Character.isLetter(kar) függvénnyel; hogy számjegy-e, lekérdezhető a Character.isDigit(kar) függvénnyel.
- A Java nyelv megengedi az ékezesek karakterek használatát, de ez a szolgáltatás az operációs rendszerben különleges beállítást igényelhet. Az ékezesek azonosítókat inkább mellözzük.

Például:

- ◆ Szintaktikailag helyes azonosítók:
pi_pi, Csapda22, _kakadu, \$nezzIde\$, PI, ÁlljMeg, grüß, IOJ
- ◆ Szintaktikailag helytelen azonosítók:
true, 1Program, Ez/az, Is=Is, no-Please, indulj!, default

Azonosító elnevezési konvenciók

- Törekedjünk arra, hogy jól olvasható, informatív, beszédes azonosítókat válasszunk!
- Az osztályok azonosítóját nagybetűvel kezdjük. (Az osztályokon kívül csak a konstans kezdődik nagybetűvel.)
- A változók és metódusok azonosítóját kisbetűvel kezdjük.
- A konstanst csupa nagybetűvel írjuk.
- Az azonosítókat lehetőség szerint úgy tagoljuk, hogy az összetett szavak kezdőbetűit nagybetűvel, a többöt kisbetűvel írjuk (pl. isLetterOrDigit, ButtonGroup).

Például:

- ◆ Beszédes azonosítók:
maxSebesseg, betukSzama, TerepAsztal
- ◆ Értelmetlen, csúnya azonosítók:
a, haidekerulavezelerlestuttihogyelszallagep, AKARMI, adsfd

A mintaprogram azonosítói:

- ◆ extra: csomagazonosító
- ◆ Console, Format, Krumpli, String, System: osztályazonosítók
- ◆ args, out: objektumazonosítók
- ◆ letszam, fejadag, osszesen: változóazonosítók
- ◆ main, print, println, readInt: metódusazonosítók

Kulcsszó

A **kulcsszó (keyword)** a rendszer részére fenntartott szó, azt másra nem lehet használni. Szokás fenntartott, foglalt vagy kötött szónak is nevezni. minden kulcsszónak megvan a maga kulcsszerepe. A mintaprogram kulcsszavai a következők:

- ◆ `import`: külső csomag (az abban levő osztályok) importálása
- ◆ `public`: publikus, azt kívülről is lehet látni
- ◆ `class`: osztály
- ◆ `static`: statikus, azaz osztálytag
- ◆ `void`: semleges. A metódus nem függvény, hanem eljárás (nincs visszatérési értéke).
- ◆ `int`: egész típus (integer).
- ◆ `double`: dupla pontosságú valós típus.

A Java kulcsszavai a következők:

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

A `const` és `goto` szintén foglalt szó, de egyik sem használatos.

Literál

A literál olyan állandó érték, amely beépül a program kódjába, és a továbbiakban már nem változtatható meg. A literálok fajtái:

- **egész**: egy pozitív vagy negatív egész szám, vagy nulla.
- **valós**: egy tizedesekkel leírható szám.
- **logikai**: két logikai (boolean) konstans létezik: a `true` (igaz) és a `false` (hamis).
- **karakter**: egy unikód karakter (szimpla aposztrófok közé tesszük).
- **szöveg**: akármilyen hosszú, unikód karakterekből álló sorozat (idézőjelek közé tesszük).
- **null**

Nézzük most sorra a különböző literálokat!

Egész literál

Egy egész (integer) literál egy pozitív vagy negatív egész szám, vagy nulla. Negatív szám esetén a szám elé írunk egy – (mínusz) jelet. A nulla elé írt + vagy – jel nem változtatja meg a nulla értéket, és a pozitív szám értéke sem változik, ha + jelet írunk elé. Az egész literálokat három formában adhatjuk meg:

- ◆ Decimális forma. Például: +238, -5, 2147483647
- ◆ Hexadecimális forma. A hexadecimális (0..9, a..f, A..F) karaktersorozat elé egy 0x (nulla és x) jelet teszünk. Például: 0x2a1 (decimális értéke $2 \cdot 256 + 10 \cdot 16 + 1 = 673$)
- ◆ Oktális forma. Az oktális (0..7) karakterek elé egy nullát teszünk. Például: 021 (decimális értéke $2 \cdot 8 + 1 = 17$)

Egy egész literál automatikusan int típusú, de ha a végére teszünk egy L vagy l betűt, akkor típusa long lesz, például: 400L.

❖ Az egész literál elé ne tegyen nullát, mert az ilyen szám oktális számrendszerbeli minősül; a végére pedig ne tegyen kis L betűt, mert 1 betű és az 1 számjegy összetéveszthető!

Valós literál

Valós vagy lebegőpontos (floating-point) literáloknak nevezük a tizedesekkel leírható számokat. A valós szám két formában adható meg:

- ◆ Tizedes forma: az egész és az esetleges tizedes rész között egy tizedespont van (pl. 9.12, 105, 105.0, 3345.6666, 0.00566, vagy 100000.0).
- ◆ Lebegőpontos forma: <mantissa>E/e<exponens>, a valós szám értéke $m \cdot 10^e$:
 - mantissa: tetszőleges előjeles tizedes szám;
 - E/e: az exponens jele, a két betű közül valamelyik.
 - exponens: előjeles egész szám. Azt jelzi, hogy 10-nek milyen hatványával kell a mantissát megszorozni, hogy a kívánt értéket kapjuk, vagyis hogy hány helytel kell a tizedespontot eltolni – jobbra, ha pozitív, balra, ha negatív.

Valós számok például:

| | |
|--------|-------------------------------------|
| 12.999 | { 12.999*10 ⁰ = 12.999 } |
| 1E0 | { 1*10 ⁰ = 1 } |
| 1e9 | { 1*10 ⁹ = 1000000000 } |
| -0.006 | { -0.006*10 ⁰ = -0.006 } |
| 12E2 | { 12*10 ² = 1200 } |
| 5E-4 | { 5*10 ⁻⁴ = 0.0005 } |
| -6.2E3 | { -6.2*10 ³ = -6200 } |

Egy valós literál automatikusan double típusú, de ha a végére teszünk egy F vagy f betűt, akkor típusa float lesz, például: 21.3F, 0.2f, 60F. (A double és a float lebegőpontos számokat tároló Java típusok. A double dupla pontosságú.)

Karakterliterál

A karakter (character) literál pontosan egy darab unikód karakter, melyet szimpla aposztrófok közé teszünk. Egy karakter megadható

- ◆ unikódjával: '\u0041', '\u0009'
- ◆ megjeleníthető karakter formájával: 'A', '?', 'É', '#'
- ◆ escape szekvenciával: '\t', '\''

Unikóddal minden karakter megadható. Vannak ún. megjeleníthető karakterek, ezeket olyan formában adjuk meg, ahogy a konzolon is megjelennek (pl. 'K', '9', '='). Vannak azonban olyan karakterek, amelyeknek nincs képük, és ezért csak escape szekvenciával adhatók meg: ilyen például a HT karakter: '\t', vagy a szimpla aposztróf: '\''. (Az escape menekülést jelent, s utal arra, hogy ezek a karakterek csak kerülő úton adhatók meg.) Az 'A' ugyanaz, mint a '\u0041', mert az A betű kódja: \u0041. A '\t' ugyanaz, mint a '\u0009', mert a HT kódja \u0009.

Karakterliterálok például:

```
'A', '\u0041', '!', '\'', '\t', '\\', '¤', '♥', 'Ë', '\u0a5f'
```

Szövegliterál

A szöveg (string) literál egy akármilyen hosszú, unikód karakterekből álló sorozat. A literál tartalmazhat escape szekvenciákat, és közvetlen unikódokat is. A sorozatot idézőjelek (dupla aposztrófok) közé tesszük. Például:

- ◆ A karaktersorozat: Hello
Megadása String literálban: "Hello"
- ◆ A karaktersorozat: "Nono", ezt ne csináld!
Megadása String literálban: "\"Nono\"", ezt ne csináld!"
- ◆ A karaktersorozat: c:\jdk1.4\bin
Megadása String literálban: "c:\\jdk1.4\\bin"
- ◆ A karaktersorozat: Most jön egy tabHT, még még egyHTVége
Megadása String literálban: "Most jön egy tab\t, még még egy\u0009Vége"
A println ezt jeleníti meg: Most jön egy tab , még még egy Vége

Szeparátor

A szeparátorok speciális jelentésű elválasztó elemek (() { } [] ; ..), némelyikük párosával fejt ki hatását. Jelentésük:

- ◆ A metódusok azonosítóit az különbözteti meg az egyéb azonosítóktól, hogy az azonosító után egy () karakterpár szerepel, benne az esetleges paramétekkel. Például: println(letszam)

- ◆ Az osztály feje után következő osztályblokkot, a metódus feje utáni metódusblokkot és az összes más blokkot is a {} karakterpár határolja.
- ◆ A [] karakterpár sokszoroz, tömböt képez. Például: a String[] egy String-eket tartalmazó tömböt jelent.
- ◆ A program utasításait és deklarációit a ; (pontosvessző) szeparátor zárja le. Például:
int letszam;
- ◆ A , (vessző) szeparátor felsorolást jelent. Például: fejadag, osszesen
- ◆ A . (pont) szeparátor minősítésre használatos. Például: System.out.println (a System osztály out objektumának println metódusa)

Operátor

Az operátor vagy műveleti jel a kifejezés része (például fejadag*letszam).

Java operátorok például:

* / % + - < <= > >= ++ != && || = +=

Az operátorokról részletesen a 12., Kifejezések, értékadás című fejezetben lesz szó.

11.4. Változó, típus

A programok adatokon manipulálnak. Egy gépi kódú programnak ki kell jelölnie a memoriában az általa használt adatterületeket, magas szintű nyelvben azonban nincs szükség erre. A programozónak csak meg kell mondania, mennyi és milyen memóriaterületre van szüksége a program futtatásához. Egy-egy ilyen memóriaterületet egyszer **azonosítani kell** (nevet kell adni neki), másrészt **meg kell adni a típusát**, vagyis azt, hogy milyen értékek kerülnek majd bele. Emlékezzünk vissza, hogy a memória nem más, mint egy címezhető bájtsorozat, és a bájtok értéke 0 és 255 közé eshet. Egy bájt sok mindenhet: lehet egy nagy szám elsőnek vagy utolsónak tárolt bájtja, lehet egy szöveg valamelyik karaktere, vagy jelenthet egy true értéket. A program mindezt magától nem tudhatja. A memóriaterületnek a programozó ad értelmet azzal, hogy megadja a típusát. A program aztán a típusnak megfelelően bánik a memóriaterülettel. **A változó típusa meghatározza a változónak adható értékek tartományát, valamint a rajta végezhető műveleteket.**

Mintaprogramunkban három változónak foglaltunk le helyet. Az elsőnek letszam az azonosítója és int típusú, a másik kettőnek fejadag, illetve osszesen a neve és double a típusa:

int letszam

20

double fejadag

0.4

double osszesen

8.0

Az int azt jelenti, hogy egy ilyen típusú memóriaterületen egy 4 bájtos előjeles egész számot lehet tárolni: az egész szám értéke +0x7fffffff (decimális 2147483647) és -0x80000000 (decimális -2147483648) között mozoghat. A memóriaterület aktuális értéke a program futása során állandóan változhat: felvehet pl. 0, 5, -9, 888 stb. értékeket, ami csak „belefér”. A double típusú változó 8 bájtot foglal le, és benne egy lebegőpontos szám „lakik”.

A letszam változónak beolvasással adunk értéket (`letszam=Console.readInt()`) a fejadag változónak kezdeti értéket adunk (`fejadag=0.4`), az osszesen változónak pedig a program ad értéket az előbbi két változó aktuális értékeitől függően (`osszesen=fejadag * letszam`). Az értékeket a futás során természetesen többször is megváltoztathatnánk, sőt a felhasználótól bármikor bekérhetnénk az aktuális értéket.

A programozónak általában nemigen kell töröndie azzal, hogy pontosan mekkora a különféle típusokhoz tartozó területfoglalás, de azért fontos, hogy felmérje: beleférnek-e akkora értékek a változóba, amelyek a feladat megoldása során felmerülhetnek. Ha nem így van, akkor előfordulhat, hogy a program rosszul működik, vagy futási hibával leáll.

- Ne tévesszen meg senkit az, ha egy változónak az azonosítója is szöveg, meg a tartalma is. Ezek a dolgok teljesen függetlenek egymástól. A szöveg nevű dobozban „lakhat” akár szöveg, akár másSzöveg is:



A változó deklarálása

Javában a program által használt összes változót deklarálni kell, vagyis meg kell adni a nevét és a típusát. Először megadunk egy típust, majd felsorolunk egy vagy több azonosítót, vesszővel elválasztva. A deklaráló utasítást a ; zárja. A felsorolt változók minden a megadott típusúak lesznek.

Mintaprogramunkban:

```
int letszam;
double fejadag, osszesen;
```

A deklarált változóra aztán a programozó a megadott néven hivatkozhat, és típusának megfelelő szabályok szerint használhatja azt. A változó neve azonosítja a változót, vagyis egyértelművé teszi a rá való hivatkozást.

Megjegyzés: Vannak olyan programnyelvek, melyekben a változódeklaráció automatikus. Ez azt jelenti, hogy ha egy olyan változót használunk, amelyet még nem deklaráltunk, akkor azt a fordító automatikusan deklarálja nekünk. Az automatikus változódeklaráció azonban rendeteg rejtvényt hozhat, amelyeket a Java megköveteli a változók deklarálását.

A változó inicializálása

Amikor egy változónak kezdeti, induló értéket adunk, akkor a változót **inicializáljuk**. Inicializálatlan változó értékének lekérdezése szintaktikai hibát okoz, azt már a fordító kiszűri! Egy változót inicializálhatunk deklaráláskor vagy később is. Fontos, hogy a változó értéke a forrás-kódban jól követhető legyen!

Egy változónak már rögtön deklaráláskor kezdőértéket adhatunk. Tegyük az azonosító után egy egyenlőségjelet, majd adjuk meg a kívánt kezdőértéket megadó kifejezést. Ha egy deklarációban több változót sorolunk fel, akkor azok az inicializálás szempontjából külön életet élnek, egymástól függetlenül kapnak (vagy nem kapnak) kezdőértéket:

Mintapéldánkban a fejadag kap egy kezdőértéket, az osszesen kezdőérték nélkül marad:

```
double fejadag=0.4, osszesen;
```

Konstans, a változtathatatlan változó

Megtehetjük, hogy egy változót egész életére „befagyaszunk”, vagyis nem engedjük megváltoztatni. Ezt úgy tehetjük meg, hogy a deklaráció elé tesszük a final módosítót (final = véleges). A konstansokat a kódolási konvenció szerint csupa nagybetűvel szokás írni:

```
final int EVSZAM=1999;
```

Mivel EVSZAM változtathatatlan, az EVSZAM=2000; értékkadó utasítás például fordítási hibát eredményezne.

Megjegyzés: A konstans nem tévesztendő össze a literállal! A konstans egy változó, melyre hivatkozni lehet a nevével; a literál a programkódba van „beégetve”, és máshonnan nem lehet rá hivatkozni.

A Java típusok osztályozása

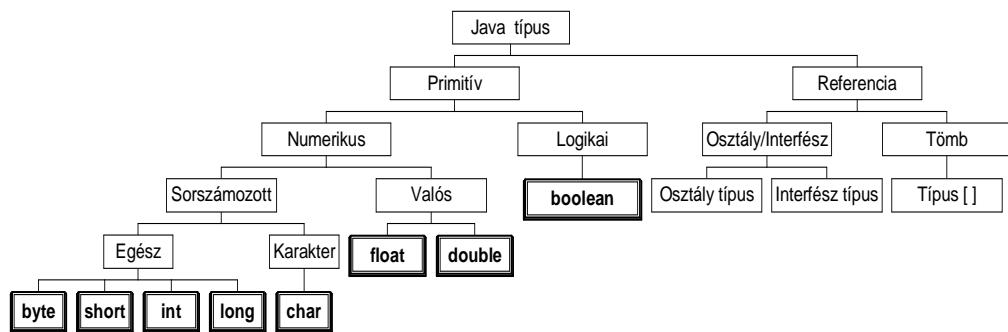
A Java típusok logikai csoportosítása a 11.2. ábrán látható. Kétféle típus létezik:

- ◆ A **primitív típusú változó** memóriaterülete oszthatatlan, az egy „élettelen”, viselkedés nélküli tulajdonságot tárol. A Javában minden összes 8 primitív típus létezik, ezek a byte, short, int, long, char, float, double és boolean (az ábrán vastagon vannak szedve).
- ◆ A **referencia típusú változó** olyan mutató, mely egy objektum hivatkozását tartalmazza. Az objektum összetett memóriaterület, több primitív típusú változót, valamint további referenciakat (hivatkozásokat) is tartalmazhat.

A primitív típus lehet numerikus vagy logikai típus; a numerikus típus lehet sorszámozott vagy valós típus; a sorszámozott típus pedig lehet egész vagy karakter típus.

Sorszámozott típus (integral type): olyan típus, amelyben minden lehetséges értékhez egy egyértelmű sorszám van rendelve.

A primitív típusokat a következő pontban részletesen tárgyaljuk.



11.2. ábra. A Java típusok osztályozása

Változó: Egy névvel azonosított memóriaterület. Egy változónak minden van azonosítója (neve), adott méretű memória helye, típusa és aktuális értéke.

Típus: A változó típusa meghatározza a változónak adható értékek tartományát, valamint a rajta végezhető műveleteket. A Javában kétfélé típus létezik: primitív és referencia típus. A primitív típusok a byte, short, int, long, char, float, double és boolean.

Deklarálás, inicializálás: A Javában minden változót deklarálni kell! Deklaráláskor meg kell adni a változó azonosítóját és típusát, és az esetleges kezdőértéket is (a változó inicializálható). A deklaráló utasítás formája (a nagy zárójelben lévő részeket nem kötelező megadni):

<típus> <változól> [= <kifejezés1>] [, <változó2> = <kifejezés2> ...] ;

Konstans: Változtathatatlan változó. Módosítója final.

11.5. Primitív típusok

Minden egyes típushoz megadjuk a nevét, az általa lefoglalt memória méretét, valamint a tárolható értékeket. Az alkalmazható operátorokat a 12., Kifejezések, értékadás című fejezetben tárgyaljuk.

Egész típusok

Négyfélé egész típus van, mindenek lehet negatív értéke is:

| Típus neve | Foglalt memória | Legkisebb érték | Legnagyobb érték |
|------------|-----------------|-------------------------|---------------------------|
| byte | 1 bájt (8 bit) | $-2^7 = -128$ | $2^7 - 1 = 127$ |
| short | 2 bájt (16 bit) | $-2^{15} = -32768$ | $2^{15} - 1 = 32767$ |
| int | 4 bájt (32 bit) | $-2^{31} = -2147483648$ | $2^{31} - 1 = 2147483647$ |
| long | 8 bájt (64 bit) | $-2^{63} \sim -10^{19}$ | $2^{63} - 1 \sim 10^{19}$ |

Valós típusok

A valós típusok számábrázolását az IEEE 754 szabvány írja le. A pontosság itt azt jelenti, hogy a program maximum hány számjegyig terjedő tizedest tárol. Ennél nagyobb pontosságú számot nem tudunk betenni a valós típusú változóba. Kétféle valós típust definiáltak:

| Típus neve | Foglalt memória | Legkisebb érték | Pontosság |
|------------|-----------------|--|------------|
| float | 4 bájt (32 bit) | 1.40129846432481707e-45 3.40282346638528860e+38 | 6-7 jegy |
| double | 8 bájt (64 bit) | 4.94065645841246544e-324 1.79769313486231570e+308 | 14-15 jegy |

A float típus sokkal nagyobb számot képes tárolni (~3E+38), mint a long típus (~1E19). Egy long számnak azonban az összes jegye szignifikáns (számít), a float szám végén sok 0 szerepel (a szám utolsó jegyei elhanyagolódnak). Ha az 1234567890123456789L számot például egy float változóba tesszük, akkor az érték 1234567894000000000L-re csonkul, ahogy ezt a következő kis program mutatja:

```
// LongToFloat.java
public class LongToFloat {
    public static void main(String[] args) {
        long lon = 1234567890123456789L;
        float f;
        f = lon;
        System.out.println(f);      // 1.234567894E18
    }
}
```

A kiírt érték tehát 1.234567894E18 lesz, s ez egész formában: 1234567894000000000.

Megjegyzés: A java.math csomagban lévő BigInteger és BigDecimal osztály alkalmazásával tetszőleges pontosság érhető el.

Karakter típus

A karakter típusú változóba egy karakterliterált tehetünk bele.

| Típus neve | Lefoglalt memória | Legkisebb érték | Legnagyobb érték |
|------------|-------------------|-----------------|------------------|
| char | 2 bájt (16 bit) | '\u0000' (0) | '\uffff' (65535) |

Példaként a c változóba egy 'A' értéket teszünk, majd a változó tartalmát kiírjuk a konzolra:

```
// Karakter.java
public class Karakter {
    public static void main(String[] args) {
        char c = 'A';
        System.out.println(c); // --> A
    }
}
```

Logikai típus

Egy logikai (boolean) típusú változónak egy boolean (logikai) literált adhatunk értékül, vagyis a true vagy false érték valamelyikét. E típus névadója George Boole XIX. századi angol matematikus: öt tekintik a szimbolikus logika megalkotójának.

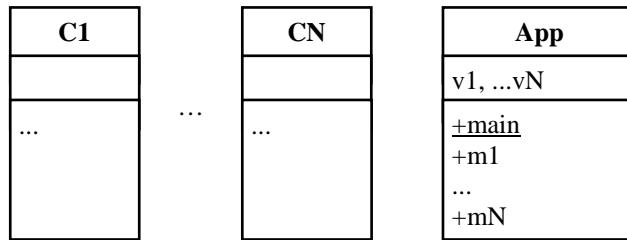
| Típus neve | Foglalt memória | Igaz érték | Hamis érték |
|------------|-----------------|------------|-------------|
| boolean | nem definiált | true | false |

Ha két érték összehasonlítható, akkor összehasonlításuk eredménye vagy igaz, vagy hamis. Ha például letszam egy int típusú változó, akkor a letszam>20 lehet igaz is, meg lehet hamis is – ez a letszam pillanatnyi értékétől függ. Mivel az összehasonlítás eredménye a program folyamán változhat, van értelme a mostani eredményt egy változóban eltárolni:

```
boolean tulnagy;
tulnagy = letszam>20;
```

11.6. A forrásprogram szerkezete

Egy Java fordítási egység (forrásállomány) egy vagy több osztályból állhat. Legyenek ezek az osztályok most a C1, C2 ... CN és az App (11.3. ábra). Egyelőre csak olyan programokkal foglalkozunk, amelyeknek az osztályait egyetlen fordítási egységen kódoljuk. A forrásállomány neve a main metódust tartalmazó publikus osztály neve lesz: App.java. A forráskód fordításának eredményeként (javac App.java) annyi class kiterjesztésű állomány keletkezik, ahány osztályt deklártunk (App.class, C1.class ... CN.class). A Javában minden egy class kiterjesztésű állományt futtatunk (java App). Csak olyan osztályt lehet futtatni, amelyben van egy main metódus. Indításkor a main automatikusan elkezd futni.



11.3. ábra. A Java forrásprogram szerkezete

Egy Java forrásprogram szerkezete a következő (részletek a 11.3. ábrán található osztályok forráskódjából):

```

// App.java forrásprogramja
import java.util.*; ← Csomagok importálása
import extra.*;

// Egy vagy több osztály deklarálása (a sorrend mindegy):
class C1 {
    ...
}

class CN {
    ...
}

// A fordítási egység egyetlen publikus osztálya.
// Ebben van a main:
public class App { ← osztály feje
    v1, ... vN      // App változóinak deklarációi
    // main metódus:
    public static void main(String[] args) { ← main metódus
        // metódus blokkja
        // változók deklarációi, utasítások
    }
    // App osztály egyéb metódusai:
    void m1() {
        ...
    }
    ...
    void mN() { ← a metódus feje
        ...
    } ← a metódus blokkja
}

```

Importálás

A **fordítási egység** elején **importálnunk** kell azokat az osztályokat, amelyekre a fordítási egységen hivatkozni szeretnénk. A * azt jelenti, hogy a csomagban szereplő összes osztályt importáljuk; ekkor a csomag bármelyik osztályára hivatkozhatunk. A `java.lang` csomagot nem kell importálnunk, mert azt a fordító automatikusan megteszzi.

A `java.lang`-ban van többek között a `System` osztály. A `java.util` egy API csomag, az `extra` egy saját osztályokat tartalmazó csomag.

Osztálydeklaráció

Az **osztály deklarációja** egy fejből és egy blokkból áll:

- Az **osztály fejében** a `class` kulcsszó utal arra, hogy osztálydeklarációról van szó. A `class` kulcsszó előtt szerepel az esetleges láthatósági módosító, utána pedig az osztály neve. Egy fordítási egységen csak egy publikus osztály lehet, s annak neve meg kell, hogy egyezzen az állomány nevével.
- Az **osztály blokkjában** lehet akárhány változó- és metódusdeklaráció. Bármely metódus blokkjából hivatkozhatunk bármely változóra, illetve metódusra.

Metódusdeklaráció (a main metódus)

Minden metódusnak van egy feje és egy blokkja (ha a metódus nem absztrakt). Egyelőre csak a `main` metódusba írunk kódot. A `main` metódus fejének részei és azok jelentése a következő:

- ◆ `public`: a metódus láthatósága nyilvános.
- ◆ `static`: a metódus osztálymetódus (nem példánymetódus).
- ◆ `void`: a metódus nem ad vissza értéket, vagyis eljárás.
- ◆ `main`: a metódus neve, azonosítója.
- ◆ `(String[] args)`: a metódus paramétere.

Blokk

Láthattuk, hogy az osztálynak és a metódusnak is van blokkja. Később látni fogjuk, hogy bizonyos utasításoknak is lehet blokkjuk.

Egy **blokkot** a `{ }` karakterpár határol. A blokk összefogja a benne található utasításokat. Mindaz, amit abban a blokkban adunk meg, pontosan arra a blokkra érvényes. Ahol egy utasítást megadhatunk, ott megadhatunk blokkot is. A blokkot szokás **összetett utasításnak** is nevezni.

Utasítás

Egy blokk **utasításokból** áll. Az utasítások fajtái a következők:

- **Deklaráló utasítás.** Memória helyek foglalása. Például: int letszam;
- **Értékadó utasítás.** Például: letszam=45;
- **Postfix és prefix növelő és csökkentő utasítás.** Például: letszam++;
- **Metódushívás.** Például: System.out.println("Hello");
- **Példányosítás.** Új objektum létrehozása.
Például: aDatum = new Datum(2000,11,05);
- **Programvezérlő utasítás.** Például: if (i==0) ...
- **Üres utasítás:** ; (csak egy pontosvessző)

Minden utasítást pontosvessző zár le.

✿ Figyelem! A blokk után nem szabad pontosvesszöt tenni!

Kivételek

Ha egy utasítás (például metódushívás) belsőjében valamilyen rendellenesség, hiba történik, akkor az utasítás **kivételek** (exception) **dobhat** (bocsáthat ki). Ekkor – ha csak nem kezeli valamilyen módon ezt a kivételelt, – a program hibaüzenet kíséretében leáll, más szóval futási hibával „elszáll”. A hibaüzenet tartalmazza

- a kivétel típusát,
- a hiba okát,
- az osztály és a metódus nevét, amelyben a hiba keletkezett, valamint
- a hibás Java állomány nevét és abban a hibás sor számát.

Ilyenkor javítsuk ki a hibát, majd fordítsuk és futtassuk újra a programot!

A következő kis program a nullával való osztás miatt „száll el”, ha a felhasználó a „Hányan vagytok? ” kérdésre nullával válaszol:

```
// Kivetel.java //1
import extra.*; //2

//3
public class Kivetel { //4
    public static void main(String[] args) { //5
        int narancsokSzama = 56; //6
        System.out.println(narancsokSzama+" narancs van."); //7
        int letszam = Console.readInt("Hányan vagytok? "); //8
        System.out.println("Fejadag:"+narancsokSzama/letszam); //9
        System.out.println("Finom a narancs!"); //10
    } //11
} //12
```

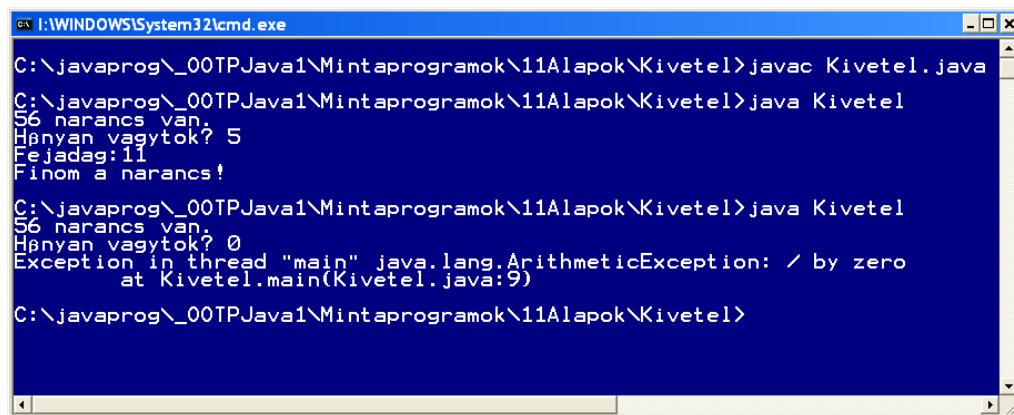
Amint beütik a nullát, a konzolon a következő üzenet jelenik meg:

```
| java.lang.ArithmetricException: / by zero
|         at Kivetel.main(Kivetel.java:9)
| Exception in thread "main"
```

Az üzenet értelmezése:

- ◆ a kivétel típusa: `java.lang.ArithmetricException`;
- ◆ a hiba oka: `/ by zero` (osztás nullával);
- ◆ az osztály és a metódus neve, amelyben a hiba keletkezett: `Kivetel.main`;
- ◆ a hibás Java állomány neve és abban a hiba helyének sorszáma: `Kivetel.java:9`

A programot kétszer futtatjuk (ezúttal JDK-ban): a beütött érték először 5, majd 0. A második esetben a fejadag és a "Finom a narancs!" már nem íródik ki. A program futásának konzolablakát a 11.4. ábra mutatja. A futási hibáért természetesen a program felelős: meg kellett volna vizsgálnia, és el kellett volna utasítania a beütött nulla értéket. A kivételek kezelésével a tankönyv 2. kötete foglalkozik majd.



```
C:\javaprogs\_00TPJava1\Mintaprogramok\11Alapok\Kivetel>javac Kivetel.java
C:\javaprogs\_00TPJava1\Mintaprogramok\11Alapok\Kivetel>java Kivetel
56 narancs van.
Hányan vagytok? 5
Fejedag:11
Finom a narancs!

C:\javaprogs\_00TPJava1\Mintaprogramok\11Alapok\Kivetel>java Kivetel
56 narancs van.
Hányan vagytok? 0
Exception in thread "main" java.lang.ArithmetricException: / by zero
        at Kivetel.main(Kivetel.java:9)

C:\javaprogs\_00TPJava1\Mintaprogramok\11Alapok\Kivetel>
```

11.4. ábra. Futási hiba (kivétel) keletkezése a Java programban

11.7. Metódushívás (üzenet)

Egy osztály metódusának végrehajtását a metódus névére való hivatkozással kérhetjük. A metódus neve után a () karakterpár szerepel, benne az esetleges paraméterekkel. A metódus neve előtt megadható az osztály vagy objektum neve attól függően, hogy mit akarunk megszólítani:

- osztálymetódus hívása: `Osztály.metódus(paraméterek)`
- példánymetódus hívása: `objektum.metódus(paraméterek)`
- ha a metódust a saját osztályából hívjuk, akkor nem kell minősítenünk, csak a nevét kell megadnunk: `metódus(paraméterek)`

Egy metódus lehet eljárás vagy függvény aszerint, hogy van-e visszatérési értéke:

- **Eljárás:** visszatérési értéke nincs, vagyis `void` (semleges). A visszatérési érték nélküli metódusokat utasításként hívjuk meg. Ekkor a metódus blokkja végrehajtódik az aktuális paraméterek függvényében, majd visszakerül a vezérlés a metódushívást követő utasításra. A metódushívó utasítást a `:` (pontosvessző) zárja le.
- **Függvény:** Ha egy metódusnak van visszatérési értéke, akkor azt kifejezésben szokás használni – ekkor a visszatérési érték behelyettesítődik a kifejezésbe. Függvényt használhatunk például értékkádó utasítás jobb oldalán, metódusok paramétereiben stb.

Eljárás például:

```
System.out.println("Ezt a println eljárás írta ki");
```

Függvények például:

```
java.lang.Math.sin(szogRadian), visszatérési értéke double típusú  
extra.Console.readInt(), visszatérési értéke int típusú
```

Megjegyzés: A visszatérési értékkel rendelkező metódus (függvény) meghívható eljárásnéként is; ilyenkor a program nem használja fel a visszatérési értéket.

A Java osztálykönyvtára rengeteg metódust kínál fel a programozónak. Mielőtt azonban alkalmaznánk egy metódust, meg kell néznünk, mi a metódus feladata (mit várhatunk tőle), és hogyan lehet azt meghívni. A metódus leírásában a következő dolgok szerepelnek:

- ◆ láthatóság;
- ◆ a metódus neve;
- ◆ a metódus feladata;
- ◆ hány paramétere van, és azoknak egyenként mi a típusuk;
- ◆ a visszatérési érték típusa.

Ebben a könyvben szinte kivétel nélkül `public` láthatóságú API metódusokról lesz szó. Ezért az áttekinthetőség érdekében csak az ettől eltérő láthatóságot adjuk meg.

Példaként nézzük meg a `java.lang` csomag `Math` osztályában levő statikus `cos` metódust:

- `static double cos(double a)`
 - a metódus neve: `cos`
 - feladata: Visszaadja az a szög koszinuszát.
 - a paraméterek száma: 1, típusa `double`
 - a visszatérési érték típusa: `double`

A jellemzők egyértelműen kiolvashatók a metódus fejéből, így bőven elegendő megadni a metódus fejét és feladatát:

- `static double cos(double a)`
Visszaadja az a szög koszinuszát.

Metódus szignatúrája, túlterhelés (overloading)

A Javában vannak olyan metódusok, amelyeket többféle paraméterezővel lehet meghívni. Ez azért lehetséges, mert **egy metódust a szignatúrája** (a metódus neve, a paraméterek száma és rendre a paraméterek típusai együtt) **azonosít**. Az ugyanolyan nevű, de más paraméterező metódus nem ugyanaz a metódus – azt külön kódban adták meg, más implementációt takar. Egy metódust mindenkor a paramétereivel együtt azonosítunk. A metódusok e tulajdonságát **túlterhelésnek** (overloading) nevezzük – a metódust túlterheljük többféle paraméterezővel.

Például a `println()` metódus paramétere lehet `long`, `double` vagy `String` is. Ez a metódus annyiszor szerepel a `PrintStream` osztályban, ahány különböző paraméterezője van.

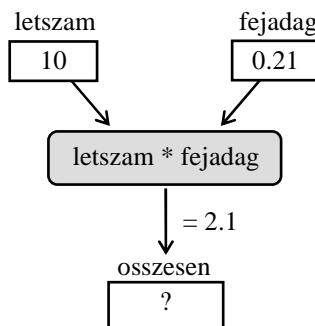
11.8. Értékkadó utasítás

Az értékkadó utasítás segítségével értéket adhatunk egy változónak:

`<változó> = <kifejezés>;`

Az értékkadó operátor (= legyen egyenlő) bal oldalán levő változó fel fogja venni a jobb oldalon szereplő kifejezés értékét.

A jobb oldali kifejezésnek jól meghatározott típusa van, amely a kifejezés alkotóelemeitől függ. Az értékkadás első lépéseként a program kiértékeli a kifejezést, azaz kiszámítja a kifejezés aktuális értékét a bal oldaltól függetlenül. Ezt követően a kapott értéket megpróbálja betenni a bal oldali változóba. A betétel nem minden esetben sikerül: fontos, hogy **a kifejezés típusa értékkadás szerint kompatibilis legyen a változó típusával**: egy `long` típusú érték például nem fér bele „csak úgy” egy `int` típusú változóba. Az értékkadás „mechanizmusát” a 11.5. ábra szemlélteti. A kifejezések típusáról és kiértékeléséről a 12., Kifejezések, értékkadás című fejezetben lesz szó.



11.5. ábra. Értékkadó utasítás

Például:

```
letszam = 10; fejadag = 0.21;  
osszesen = fejadag*letszam; // osszesen értéke 2.1 lesz
```

A Javában egyszerre több változónak is értéket adhatunk a következőképpen:

```
változó1 = változó2 = kifejezés;  
változó1 = változó2 = változó3 = ... kifejezés;
```

A következő példában letszam és fejadag változó értéke egyaránt 2:

```
letszam = fejadag = 2;
```

11.9. Adatok bevitеле a konzolról

extra.Console osztály

Egy modern program grafikus interfész (ablakozó rendszer) segítségével kommunikál a felhasználóval. A Java programozó külön API csomagokat használhat a GUI (Graphical User Interface, grafikus felhasználói felület) beprogramozására. Sajnos az AWT és a Swing használata egy kezdő programozó számára nehéz. Ezért a kezdeti lépések megtételéhez feltétlenül szükséges egy olyan metódusgyűjtemény, amelynek segítségével a konzolról egyszerűen bekérhetünk primitív típusú értékeket. A Javában azonban a konzolról való beolvasás igencsak bonyolult procedúra. Mivel a Java készítői nem gondoltak a kezdő programozókra, ezért kénytelenek vagyunk egy saját osztályt felhasználni a konzolról való beolvasásokhoz: ez az extra csomag Console osztálya (11.6. ábra).

| extra::Console |
|---|
| <u>+readInt(str: String): int</u> |
| <u>+readInt(): int</u> |
| <u>+readLong(str: String): long</u> |
| <u>+readLong(): long</u> |
| <u>+readDouble(str: String): double</u> |
| <u>+readDouble(): double</u> |
| <u>+readChar(str: String): char</u> |
| <u>+readChar(): char</u> |
| <u>+readLine(str: String): String</u> |
| <u>+readLine(): String</u> |
| <u>+pressEnter()</u> |

11.6. ábra. Az extra.Console osztály

Megjegyzés: Az extra csomag és a könyv forráskód melléklete **letölthető** a könyv hátán megjelölt Internet címről!

A 11.6. ábrán látható, hogy az UML-ben néhány jelölés eltér a Java szintaktikától:

- ◆ a csomag és az osztálynév közé az UML-ben két kettőspontot teszünk, a Javában csak pontot;
- ◆ az adatok és a paraméterek UML-beli deklarációjában előbb szerepel az azonosító, azután a típus; Javában ugyanez fordítva van. UML-ben a metódus visszatérési értékét a függvény deklarációjának végén adjuk meg, Javában az elején. Ezt a szabályt az UML nem veszi nagyon szigorúan – megengedi a nyelvspecifikus elemeket is. Az a lényeg, hogy a terv mindenkor jól olvasható legyen.

A `Console` osztály statikus metódusai (a `static` kulcsszót nem írjuk ki, mert zavaró lenne):

- ▶ `int readInt(String str)`
- ▶ `int readInt()`

Mindkét függvény addig vár, amíg a felhasználó be nem üt egy egész számot. Az Enter leütése után a függvény a beütött `int` típusú számmal tér vissza. Az `str` paraméterben megadható a tényleges bekérés előtt egy prompt, vagyis egy információs szöveg.

- ▶ `long readLong(String str)`
- ▶ `long readLong()`

Egy `long` típusú érték bekérése konzolról promptos és prompt nélküli változatban.

- ▶ `double readDouble(String str)`
- ▶ `double readDouble()`

Egy `double` típusú érték bekérése konzolról promptos és prompt nélküli változatban.

- ▶ `char readChar(String str)`
- ▶ `char readChar()`

Egy `char` típusú érték bekérése konzolról promptos és prompt nélküli változatban.

- ▶ `String readLine(String str)`
- ▶ `String readLine()`

Szöveg bekérése egy `String` típusú változóba a konzolról promptos és prompt nélküli változatban. `str` tartalma a teljes begépelt sor lesz az Enter leütésig. A következő kis programrészlet például bekéri a felhasználó nevét, majd duplázva kiírja azt:

```
String nev = Console.readLine("Mi a neved? ");
System.out.println(nev+ " , "+nev+" ! ");
```

A Javában a szöveget (karakterláncot) objektum tárolja. A `String` típusú objektumnak értékül adható egy szövegliterál. A `String` típusú objektum állapota nem változtatható, abból csak függvényekkel lehet információt kérni. A karakterláncokra nem lehet alkalmazni az összehasonlító operátorokat! A karakterláncokkal a 16. fejezet foglalkozik.

► `void pressEnter()`

Az eljárás vár az Enter billentyű leütésére. Bizonyos fejlesztői környezetekben a program lefutása után a konzolablak automatikusan becsukódik – a `pressEnter()` utasítással feltartóztathatjuk ezt a becsukódást, és elolvashatjuk a program futási képernyőjét.

11.10. Megjelenítés a konzolon

Konzolra a következő metódusokkal írhatunk:

- `System.out.print(paraméter)`
- `System.out.println(paraméter)`
- `System.out.println()`

A `System` osztály a `java.lang` csomagban található (ezt a csomagot nem kell importálni). Az `out` a `System` osztály egy objektuma (osztályváltozója). Az `out` a `PrintStream` osztályból való, s az a `java.io` csomagba tartozik (az `io` csomagot nem nekünk kell importálnunk). A `PrintStream` osztálynak van egy `println()` metódusa, amely képernyőre írja a paramétereiben megadott szöveget.

A metódusoknak minden össze egyetlen paraméterük lehet, de az többféle típusú: `int`, `double`, `String` stb. Ha egyszerre több dolgot is ki szeretnénk írni (például egy szöveget és egy `int` típusú számot), akkor össze kell adnunk őket (például "Létszám"+letszam). A Java minden szöveggé tud konvertálni, és ezért megengedi, hogy különböző típusú kifejezéseket összeadjunk. A végeredmény egy `String` típusú érték lesz – ezt a szöveget fogja a `print()`, illetve a `println()` kiírni a konzolra. A `println()` metódus minden össze annyiban tér el a `print()` metódustól, hogy a kiírás végén sort emel (print line).

Szabályok

- ◆ A primitív típusú kifejezés értéke szöveggé konvertálódik, és ez íródik ki:

| | |
|--|-------------|
| <code>print(123);</code> | → 123 |
| <code>print(123.4E3);</code> | → 123400.0 |
| <code>print(0.4588877);</code> | → 0.4588877 |
| <code>print('B');</code> | → B |
| <code>boolean b=true; print(b);</code> | → true |
| <code>int a=1; print((a+12)*3);</code> | → 39 |

- ◆ A `print` metódus a paraméterében megkapott szöveget változatlan formában írja ki a konzolra. Ha a szövegben escape szekvenciák is vannak, akkor azok kiváltják a megfelelő hatást, mint tabulálás, soremelés stb.

| | | | |
|--------------------------------|-----|---|---|
| <code>print("a\tb\tc");</code> | → a | b | c |
|--------------------------------|-----|---|---|

- ◆ Szövegeket és primitív típusú kifejezéseket tetszőleges hosszúságban összeadhatunk.

Ekkor a nem szöveg típusú kifejezéseket a metódus szöveggé konvertálja:

```
print(letszam+ " * "+fejadag+ " kg = "+összesen+ " kg");  
→ 3 * 2 kg = 6 kg
```

- ❖ Vigyázat! Ha egy kifejezésben `String` is van, akkor a fordító a nem zárójelzett kifejezéseket egyenként karakterláncá konvertálja!

```
int a=1, b=2; System.out.println("a+b="+a+b); → a+b=12  
(nem a+b=3!)
```

Megjegyzés: A `java.lang` csomag `System` egységében van egy statikus objektum: neve `out`, osztálya `PrintStream`. A `PrintStream` osztályban szerepelnek a `print()` és `println()` példánymetódusok.

extra.Format osztály

Sajnos a `print()` metódus a lebegőpontos számokat barátságtalan formában ontja a konzolra, és igazításra sem lehet rábírni. A saját `extra.Format` osztály metódusaival a kiírandó adatokat (számokat, szövegeket) igazítani tudjuk. Elérhetjük például, hogy egy `double` típusú szám ne lebegőpontos formában kerüljön a konzolra, hanem a megadott hosszúságban, a megadott tizedesszámmal.

Az `extra.Format` osztály metódusai:

- `String left(long num, int len)`
- `String right(long num, int len)`
Visszaad egy `len` hosszúságú szöveget, melyben a `num` egész szám balra (left), illetve jobbra (right) van igazítva. A visszaadott szöveg `len`-től függetlenül legalább olyan hosszú, hogy a megadott szám elférjen, vagyis a szám semmiképpen sem csonkul.
- `String left(double num, int len, int frac)`
- `String right(double num, int len, int frac)`
Visszaad egy `len` hosszúságú szöveget, melyben a `num` valós szám balra (left), illetve jobbra (right) van igazítva `frac` darab tizedessel, kerekítve. A többi tizedes tehát nem íródik ki. A visszaadott szöveg `len`-től függetlenül legalább olyan hosszú, hogy a megadott szám elférjen, vagyis a szám egész része semmiképpen sem csonkul.
- `String left(String str, int len)`
- `String right(String str, int len)`
Visszaad egy `len` hosszú szöveget, melyben az `str` szöveg balra (left), illetve jobbra (right) van igazítva. A szöveget `len`-től függően levágja vagy szóközökkel egészíti ki.

Például:

```

int i = 45;           → Format.right(i,5) == "    45"
int i = 45;           → Format.left(i,5)  == "45    "
double d = 59.9477;  → Format.right(d,6,2) == " 59.95"
double d = 126.1;    → Format.right(d,0,3) == "126.100"
String str = "Hapci"; → Format.left(str,3) == "Hap"
                        → Format.left(str,8) == "Hapci   "
                        → Format.right(str,2) == "Ha"

```

Megjegyzés: A konzolon való balra igazított kiíráshoz használhatja a TAB karaktert is.

Feladat – Henger

Kérjük be konzolról egy henger sugarát és magasságát cm-ben, majd

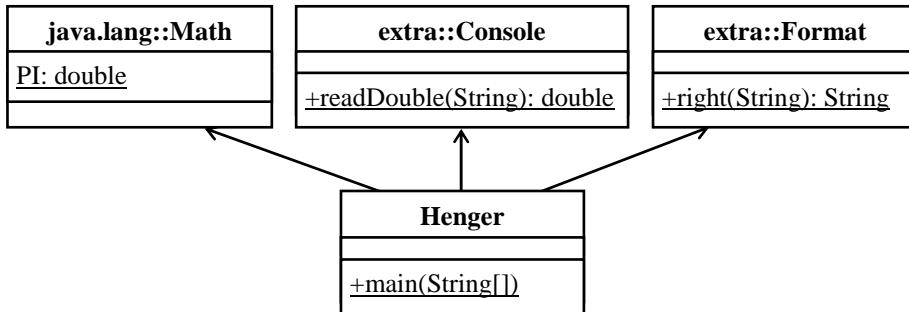
– írjuk ki a henger térfogatát!

– írjuk ki a henger súlyát, ha ez tömör vashenger, és ha fahenger!

A kiírásokban a számokat igazítsuk egymás alá, és kerekítsünk 2 tizedesre!

A program terve

Ezt a tervet csak a tisztánlátás végett adjuk meg: ekkora program tervét nem szokás papírra vetni. A tervből az olvasható ki, hogy a `main` metódus három osztály statikus adatait, illetve metódusait használja:



Megjegyzés: Nem kell a terven feltétlenül minden osztályt szerepelteni. Az a fontos, hogy a terv alapján a program könnyen kódolható legyen. A `Math`, `Console` és `Format` osztályokat például tipikusan nem szokás feltüntetni a programterven. Ha a program csak egy `main` metódusból áll, nem is szoktunk osztálydiagramot tervezni.

Forráskód

```

import extra.*;
public class Henger {
    public static void main(String[] args) {
        final float VAS_SURUSEG = 7.8F; // g/cm3
        final float FA_SURUSEG = 0.7F; // g/cm3
        double terfogat;
        double sugar, magassag;
        sugar = Console.readDouble("Sugár (cm)? ");
        magassag = Console.readDouble("Magasság (cm)? ");
        terfogat = sugar*sugar*Math.PI*magassag;
        System.out.println("Térfogat : "+
            Format.right(terfogat,8,2)+" cm3");
        System.out.println("Vashenger : "+
            Format.right(terfogat*VAS_SURUSEG,8,2)+" g");
        System.out.println("Fahenger : "+
            Format.right(terfogat*FA_SURUSEG,8,2)+" g");
    }
}

```

A program egy lehetséges futása

| |
|-----------------------------------|
| Sugár (cm)? 3 |
| Magasság (cm)? 4.5 |
| Térfogat : 127.23 cm ³ |
| Vashenger: 992.43 g |
| Fahenger : 89.06 g |

A program (main metódus) elemzése

A VAS_SURUSEG és a FA_SURUSEG konstans, vagyis olyan változó, amelynek nem lehet megváltoztatni az értékét. Miután bekértük a felhasználótól a sugár és a magasság értékét, //1-ben kiszámítjuk a térfogatot. A Math.PI egy konstans, a Math osztály statikus változója. A kiírásokban minden változót, illetve kifejezést 8 hosszon írunk ki, ebből a tizedesek száma 2.

Tesztkérdések

- 11.1. Mely állítások igazak az unikód karakterábrázolásra vonatkozóan? Jelölje meg az összes igaz állítást!
- a) A maximálisan ábrázolható karakterek száma 32768.
 - b) A maximálisan ábrázolható karakterek száma 65536.
 - c) Hexadecimális formája \uhhhh
 - d) Hexadecimális formája //hhhh

- 11.2. Mik tartoznak a program szintaktikailag elkülöníthető alkotóelemei közé? Jelölje meg az összes jó választ!
- a) azonosító
 - b) metódusfej
 - c) szeparátor
 - d) változó
- 11.3. Jelölje meg az összes igaz állítást a következők közül!
- a) Az ASCII kódkészlet a karaktereket 2 bájton ábrázolja.
 - b) Egy azonosító számjeggyel is kezdődhet.
 - c) A megjegyzést a fordító nem veszi figyelembe.
 - d) Az azonosítók elnevezési konvenciója szerint egy osztály azonosítóját nagybetűvel kezdjük.
- 11.4. Mely azonosítók helyesek szintaktikailag? Jelölje meg az összes jó választ!
- a) for
 - b) 56kezd
 - c) Logika
 - d) x/y
- 11.5. Mely változódeklarációk helyesek szintaktikailag? Jelölje meg az összes jó választ!
- a) int tripleSec;
 - b) int nagy.egesz;
 - c) char betu2;
 - d) integer i;
- 11.6. Mely változódeklarációk helyesek szintaktikailag? Jelölje meg az összes jó választ!
- a) double d=49;
 - b) boolean egenlo=5;
 - c) int a=256, b;
 - d) double d, e;
- 11.7. Jelölje meg az összes igaz állítást a következők közül!
- a) Egy karakterliterál idézőjelek (kettős aposztrófok) között adható meg, például "=".
 - b) A szövegliterál csak akkor tartalmazhat idézőjelet, ha azt kétszer leírjuk:
"EZ ""nem"" szép dolog!"
 - c) A valós literál megadható akár tizedes, akár lebegőpontos formában.
 - d) Az egész literál megadható hexadecimális formában is.
- 11.8. Jelölje meg az összes igaz állítást a következők közül!
- a) A Javában összesen 4-féle primitív típus létezik: int, double, char és boolean.
 - b) A java.lang csomagot minden Java programban importálni kell.
 - c) Egy osztály több metódust is tartalmazhat.
 - d) minden osztályban kell lennie egy main metódusnak.

- 11.9. Mi jelenik meg a képernyőn a következő utasítások végrehajtása után? Jelölje meg az egyetlen jó választ!

```
byte a=5, b=2;  
System.out.println("a+b= " +a+b);
```

- a) a+b= 7
- b) a+b= 52
- c) a+b= 5 2
- d) Semmi, mert a kódrészlet szintaktikailag hibás.

Feladatok

- 11.1. **(A)** Jelenítse meg a konzolon a „Holnap „jó” leszek” szövegkonstanst! (*JoLeszek.java*)
- 11.2. **(A)** Kérje be konzolról a felhasználó nevét, majd írja ki a következő jókívánságot:
Kedves X! Sikeres Java programozást! (*Jokivansag.java*)
- 11.3. **(B)** Kérje be konzolról egy hasáb három élének hosszúságát, majd írja ki a hasáb felszínét és térfogatát! (*Hasab.java*)
- 11.4. **(C)** Kérjen be konzolról két valós számot (az összeadandókat)! Ezután írja ki a számokat egymás alá, két tizedesre igazítva! Végül húzzon egy vonalat, és írja ki a számok összegét szintén két tizedesre igazítva! Tegyük fel, hogy a számok egész jegyeinek száma maximum 7, vagyis 2 tizedessel kiírva 10 karakteren elférnek. (*Osszead.java*)

12. Kifejezések, értékadás

A fejezet pontjai:

1. A kifejezés alkotóelemei
 2. Operátorok
 3. Típuskonverziók
 4. Értékadás
 5. Kifejezések kiértékelése – példák
 6. Feltétel
 7. Paraméterátadás, túlterhelt metódusok
 8. java.lang.Math osztály
-

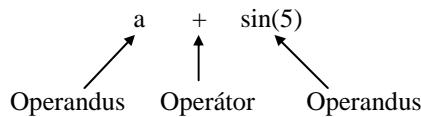
A kifejezések kiértékelését, illetve az értékadást erősen bonyolítja az a tény, hogy az egyes változóknak típusuktól függően más és más a „befogadóképességük”. Egy long típusú értéket például, bármilyen kicsiny legyen az, nem lehet egyszerűen értékül adni egy int típusú változónak, mert adatvesztés következhet be. A Java nyelv szigorú típusegyeztetési szabályokat állít fel az adatvesztési hibák kiszűrésére. Ebben a fejezetben a kifejezések kiértékelésével és az értékadással kapcsolatos szabályokat, az implicit és explicit típuskonverziókat tárgyaljuk.

12.1. A kifejezés alkotóelemei

Egy kifejezés operandusokból és operátorokból (műveletekből) áll (12.1. ábra). A kifejezésben szerepelhet egy vagy több operandus, és bármelyik operandus lehet maga is egy kifejezés. Vannak egy-, illetve kétoperandusú operátorok.

Az operandus lehet:

- Változó
- Literál
- Függvényérték
- Kifejezés (zárójelben vagy zárójel nélkül)



12.1. ábra. Kifejezés

Kifejezések például:

```

alfa
Math.sin(alfa) + 5
-98E4
(j<5) || (j>10)
!(Math.sqrt((n+3)/2)==0)
  
```

A kifejezések **kiértékelési sorrendjét** a zárójelek és az operátorok határozzák meg. A kiértékelés a következő szabályok szerint történik:

- Először a zárójelekben található kifejezések értékelődnek ki (belülről kifelé).
- Ezen belül előbb minden a nagyobb prioritású művelet hajtódik végre.
- Ha a műveletek prioritása egyforma, akkor a művelet asszociativitásától függően jobbról balra (\leftarrow) vagy balról jobbra (\rightarrow) történik a kiértékelés.

Az operandus típusa

A Java nyelvben minden operandusnak van egy jól meghatározott típusa. minden operátorhoz tartozik egy szabály arra vonatkozólag, hogy milyen típusú operandus állhat annak bal, illetve jobb oldalán. Az adott művelettől és annak operandusaitól függően a művelet eredményének is jól meghatározott típusa lesz. Először nézzük meg, hogyan tudjuk egyértelműen megállapítani egy operandus típusát, aztán vegyük sorra a műveleteket és a kiértékeléssel kapcsolatos szabályokat.

Az operandus típusát a következőképpen tudjuk megállapítani:

- Egy változó típusa a deklaráláskor megadott típus;
- Egy függvény típusa annak visszatérési értéke;
- Egy egész literál automatikusan `int` típusú, ha csak nem teszünk a szám mögé egy `L` betűt – ekkor a literál `long` típusú lesz (pl. `0L`, `236L`). Egy literál típusa nem lehet sem `byte`, sem `short`.
- Egy valós literál automatikusan `double` típusú, ha csak nem teszünk a szám mögé egy `F` betűt – ekkor a literál `float` típusú lesz (pl. `2F`, `3.1F`).
- A logikai, a karakter és a szövegliterálok típusai rendre `boolean`, `char` és `String`. A `String` nem primitív típus, hanem osztály – a `String` változókról később lesz szó.

12.2. Operátorok

A 12.2. ábrán felsoroljuk a Java összes operátorát. A műveletek prioritása fentről lefelé csökken, az egy sorban szereplő operátorok prioritása egyenlő. Az asszociativitás a kiértékelés irányát adja meg (bal asszociativitás: balról jobbra; jobb asszociativitás: jobbról balra).

Egy operátor lehet:

- ◆ **unáris**, melynek egyetlen operandusa van. Ilyen például a negatív képzés: `-i`.
Az unáris operátor lehet
 - **prefix**, mely az operandus előtt szerepel. Ilyen a kiértékelés előtti léptetés: `++i`
 - **postfix**, mely az operandus után szerepel. Ilyen a kiértékelés utáni léptetés: `i--`
- ◆ **bináris**, melynek két operandusa van. Ilyen például az összeadás: `i+j`



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

12.2. ábra. Java operátorok

Figyelje meg, hogy az értékadó operátoroknak a legalacsonyabb a prioritása. Értékadáskor ezért először kiértékelődik a teljes jobb oldal, amely azután bekerül a bal oldali változóba.

A táblázat műveleteit példák segítségével magyarázzuk meg. minden esetben megadjuk a kiértékelt kifejezés típusát és az eredményt. Tegyük fel, hogy az egész fejezetben érvényesek a következő deklarációk (a változók nevéből könnyen lehet következtetni típusukra):

```
boolean bool; char c;
byte b; short s; int i; long lon; float f; double d;
```

Unáris postfix és prefix operátorok

- ◆ [] : tömbképző operátor (lásd a 18., Tömbök fejezetet).
- ◆ . (pont) : minősítő operátor.
- ◆ (<param>) : metódusképző operátor (lásd a 15., Metódusok írása fejezetet).
- ◆ ~ és ! : (lásd ezt a fejezetet, Logikai operátorok és Bitenkénti operátorok).
- ◆ new : példányosító operátor (lásd a 16., Karakterláncok, csomagolók fejezetet)
- ◆ (<típus>)<kif> : típuskényszerítő operátor (lásd ezt a fejezetet, Típuskonverziók)

+ (plusz), - (mínusz)

Előjel operátorok: a prefix – (mínusz) előjel operátor egy egész vagy valós kifejezést a -1-szeresére változtat. A + (plusz) előjel operátor a kifejezést változatlanul hagyja.

++ (plussz plussz), -- (mínusz mínusz)

Léptető operátorok: a prefix, illetve a postfix léptető operátor eggyel növeli (++), vagy csökkeneti (--) a hozzá tartozó numerikus változót. A különbség akkor jelentkezik, ha a léptetést egy kifejezésen belül használjuk. A prefix léptető megnöveli a változót, és aztán használja fel a változó értékét a kifejezésben, a postfix változat pedig előbb használja fel az értéket, és csak utólag növeli a változót.

Példák az előjel és léptető operátorokra:

| Utasítások | Eredmény |
|------------------|----------------|
| i=3; i=-i; | i== -3 |
| i=3; i++; | i== 4 |
| d=1.1; d--; | d== 0.1 |
| s=-2; ++s; | s== -1 |
| i=3; lon=3+i++; | lon== 6, i== 4 |
| i=3; lon=3+ ++i; | lon== 7, i== 4 |

Megjegyzés: Kerüljük a bonyolult, olvashatatlan utasításokat! Bár a

lon=4; i=3; lon=lon++i; az eredmény: lon==7, i==3.

értékkadás-sorozat teljesen szabályos, ne alkalmazzuk! A + jeleket mindenkorban válasszuk szét (lon=lon++ +i;)! Ez az eset ráadásul megtévesztő is, hiszen lon léptetése teljesen felesleges (felülírjuk értékét).

Mutiplikatív (szorzó) operátorok

Ide tartoznak a * (szorzás), / (osztás), és a % (maradékképzés). A maradékképzés a valós számokra is alkalmazható. Ha az operandusok egészek, akkor az eredmény is egész, ha legalább az egyik operandus valós, akkor az eredmény is valós.

Példák:

| Utasítások | Eredmény |
|----------------|----------|
| d=1.0; d=d*6; | d==6.0 |
| i=22; i=i/5; | i==4 |
| d=20.5; d=d/5; | d==4.1 |
| i=22%5; | i==2 |
| f=20.5F%5; | f==0.5 |

Additív (összeadó) operátorok

Ide tartoznak a + (összeadás) és a - (kivonás).

Példák:

| Utasítások | Eredmény |
|---------------|----------|
| i=3; f=i+600; | f==603.0 |
| i=3; i=7-i; | i==4 |

Hasonlító és egyenlőségvizsgáló operátorok

Ezek az operátorok összehasonlítják a jobb és bal oldal értékét. Az eredmény egy boolean típusú érték lesz, true vagy false.

- ◆ Hasonlító operátorok: < (kisebb-e), <= (kisebb vagy egyenlő-e), > (nagyobb-e), >= (nagyobb vagy egyenlő-e).
- ◆ Egyenlőségvizsgáló operátorok: == (egyenlő-e), != (nem egyenlő-e).

Példák:

| Utasítások | Eredmény |
|-----------------|-------------|
| i=3; bool=i==3; | bool==true |
| i=3; bool=i>=5; | bool==false |

Logikai operátorok

A logikai kifejezésekre alkalmazhatók a ! (nem), az & (és), és a | (vagy) műveletek. A ! unáris, a & és | műveletek binárisak. A logikai műveletek eredményeit a megfelelő

igazságtáblák szemléltetik, amelyek a b1 (és b2) logikai változók minden lehetséges érték-kombinációjára megadják a kifejezés értékét.

```
boolean b1, b2;
```

! (NEM)

A logikai tagadás (nem=not) az értéket egyszerűen megfordítja:

| b1 | !b1 |
|-------|-------|
| true | false |
| false | true |

&, && (ÉS)

A logikai szorzás (és=and) eredménye true, ha minden operandus true. minden más esetben false értéket ad:

| b1 | b2 | b1 & b2 |
|-------|-------|---------|
| true | true | true |
| true | false | false |
| false | true | false |
| false | false | false |

Az **&& művelet a rövid kiértékelésű** (short circuit) változat. Ez azt jelenti, hogy csak addig történik kiértékelés, amíg a program el nem tudja dönten a végeredményt. Az `i==3 && f !=4` kifejezésben például az `f !=4` már nem értékelődik ki, ha `i` értéke nem 3. A rövid kiértékelés gyorsítja a programot, arra azonban vigyázni kell, hogy a kifejezés nem kiértékel részének lehet olyan mellékhatása, amely fontos az algoritmus helyessége szempontjából.

|, || (VAGY)

A logikai összeadás (vagy=or) eredménye false, ha minden operandus false. minden más esetben true értéket ad:

| b1 | b2 | b1 b2 |
|-------|-------|---------|
| true | true | true |
| true | false | true |
| false | true | true |
| false | false | false |

A **|| művelet a rövid kiértékelésű** (short circuit) változat. A következő program arra példa, hogy a rövid és a teljes kiértékelés más végrehajtást eredményez:

Feladat – Egy vagy két A betű

A program kérjen be egy karaktert. Ha az nem az A betű, akkor kérjen be még egyet. Végül írja ki, hogy az egyik betű A betű volt-e.

```

import extra.*;
public class Egyketa {
    public static void main(String[] args) {
        boolean vanA;
        vanA = Console.readChar("Írjon be egy betűt: ")=='A' ||
            Console.readChar("Még egyet: ")=='A';
        System.out.println("Volt benne A betű: "+vanA);
    }
}

```

A program egy lehetséges futása

| |
|-------------------------|
| Írjon be egy betűt: A |
| Volt benne A betű: true |

Futtassa a programot úgy, hogy a `||` műveletet kicseréli a `|` műveletre! Ekkor mindenkorban két karakterbeolvasás megy végbe függetlenül attól, hogy az első alkalommal a betűt ütöttek-e be, vagy sem.

`^ (KIZÁRÓ VAGY)`

A logikai kizárá vagy (exclusive or) eredménye `true`, ha a két operandus közül az egyik `true`, a másik `false`. minden más esetben `false` értéket ad:

| b1 | b2 | <code>b1 ^ b2</code> |
|--------------------|--------------------|---------------------------|
| <code>true</code> | <code>true</code> | <code>false</code> |
| <code>true</code> | <code>false</code> | <code>true</code> |
| <code>false</code> | <code>true</code> | <code>true</code> |
| <code>false</code> | <code>false</code> | <code>false</code> |

Példák:

| Utasítások | Eredmény |
|---|--------------------------|
| <code>i=3; s=-2; bool = (i<4) & (s!=6);</code> | <code>bool==true</code> |
| <code>b=1; s=-2; bool=true; bool = b==s !bool;</code> | <code>bool==false</code> |
| <code>i=3; d=1.0; bool = i==5 d!=4;</code> | <code>bool==true</code> |

Megjegyzés: Az `(i<4) & (s!=6)` kifejezésben nem kellene zárójeleket alkalmaznunk, hiszen az `&` gyengébb művelet, mint a hasonlítás. Sokszor azonban a zárójelek használata javítja az olvashatóságot.

Bitenkénti operátorok

A logikai műveletek bitekre is alkalmazhatók. A bitenkénti `~` (NEM), `&` (ÉS), `|` (VAGY) és `^` (KIZÁRÓ VAGY) műveletek bitenként végezik el a logikai műveleteket úgy, hogy az 1 `true`, a 0 pedig `false` értéknek számít. A műveletek minden az egymásnak megfeleltethető biteken hajtódnak végre. `b<<n` (signed shift-left) b-ben n-nel balra tolja a biteket úgy, hogy jobbról nullák „jönnek be”. `b>>n` (signed shift-right) b-ben n-nel jobbra tolja a biteket úgy, hogy balról

nullák (ha b pozitív) vagy egyesek (ha b negatív) „jönnek be”. A `b>>>n` (shift-right-with-zero-fill) b-ben n-nel jobbra tolja a biteket úgy, hogy balról mindenkorábban nullák „jönnek be”.

A következő példában a művelet bal oldalán mindenkorábban `b=124` van. Az egyes sorok végén megjegyzésbe tettük a konzolon megjelenő eredményt, majd zárójelben a szám két számrendszerbeli alakjának utolsó 8 bitjét:

```
public class BitMuveletek {
    public static void main(String[] args) {
        int b=124; // b=124 (01111100)
        System.out.println(~b = "+~b); // ~b = -125 (10000011)
        System.out.println(b|1 = "+(b|1)); // b|1 = 125 (01111101)
        System.out.println(b&4 = "+(b&4)); // b&4 = 4 (00000100)
        System.out.println(b^2 = "+(b^2)); // b^2 = 126 (01111110)
        System.out.println(b<<1 = "+(b<<1)); // b<<1 = 248 (11111000)
        System.out.println(b>>1 = "+(b>>1)); // b>>1 = 62 (00111110)
        System.out.println(b>>>3 = "+(b>>>3)); // b>>>3=15 (00001111)
    }
}
```

Feltételes kiértékelés

A **feltételes kiértékelés** szintaktikai formája a következő:

```
(feltétel) ? kifejezés1 : kifejezés2;
```

Ha a `feltétel` értéke `true`, akkor `kifejezés1` kerül kiértékelésre, egyébként `kifejezés2`.

Például:

```
int teljesitmeny = 60;
int premium = (teljesitmeny>50)?10000:0; // -> 10000
```

Értékadó operátorok

Az értékadó művelet egy változónak ad értéket:

```
<változó> = <kifejezés>;
```

Az = értékadó operátor beteszí a bal oldali változóba a jobb oldali kifejezés értékét. A bal oldal kizárolag változó lehet. A jobb oldal tetszőleges kifejezés, amely lehet változó, vagy literál is.

Ha egy változó értékét egy adott értékkel növelni/csökkenteni, szorozni/osztani stb. akarjuk, akkor használhatjuk az ún. kiterjesztett aritmetikai operátorokat:

- += hozzáadja a változóhoz a jobb oldal értékét: `a+=b;`
- -= levonja a változóból a jobb oldal értékét: `a-=b;`
- *= megszorozza a változót a jobb oldal értékével: `a*=b;`
- /= elosztja a változót a jobb oldal értékével: `a/=b;`
- %= a jobb oldallal való osztás maradékát teszi be a változóba: `a%=b;`

Példák:

| Utasítások | Eredmény |
|----------------|----------|
| i=32000; | i==32000 |
| s=-2; s+=2; | s==0 |
| lon=4; lon-=7; | lon== -3 |
| d=1; d/=2; | d==0.5 |

✿ Vigyázat! Az értékkadás jele nem tévesztendő össze az egyenlőségvizsgálat (==) jelével. Az a=b eredménye nem igaz vagy hamis, hanem az a értéke b lesz, még akkor is, ha ugyanazon kifejezésen belül használjuk!

12.3. Típuskonverziók

A Java **erősen típusos nyelv**. Ez azt jelenti, hogy a fordító a típus kompatibilitásának ellenőrzését minden lehetséges esetben elvégzi. Egy típus **kompatibilis** egy másik típussal, ha az adott környezetben helyettesíthető vele.

Bizonyos műveletek végrehajtása előtt az operandus típusát konvertálni kell egy másik típussá.

A típuskonverziót jellegét tekintve két szempontból is osztályozhatjuk:

- **automatikus (implicit) vagy kényszerített (explicit) konverzió:** első esetben a konverziót a fordító automatikusan elvégzi, másodikban a programozónak kell azt kikényszeríteni;
- **konverzió irány:** eszerint megkülönböztetünk **szűkítő** vagy **bővítő** típuskonverziót.

Implicit és explicit típuskonverzió

A típuskonverzió **implicit**, ha azt a fordító automatikusan elvégzi, és **explicit**, ha azt a programozónak kell elvégeznie. Ez utóbbi esetben **típuskényszerítésről** beszélünk, ilyenkor a programozó egy kifejezés értékére „ráerőltet” egy típust:

(<típus>)<kifejezés>

A kifejezés előtt kell tehát írnunk zárójelben azt a típust, amit „rá szeretnénk húzni” az értékre. A típuskényszerítés angol neve type cast, ezért az értéket, illetve változót „castolni (kásztolni)” is szokták.

Például:

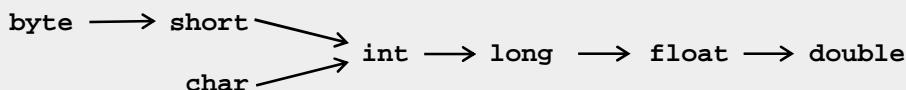
```
double d; int i = 5;
d = i; // int konvertálása double típusúá implicit módon. Bővítés.
```

```
d = 79.4;
i = (int)d; // double konvertálása int típusúvá explicit módon. Szűkítés. ->79
// i = d;    // Szintaktikai hiba!!!
```

Szűkítő és bővítő konverziók

Primitív típusok esetén egy szűkebb adattípus értéke konvertálható egy bővebb adattípus értékébe információvesztés nélkül. Az ilyen konverziót **bővítő konverzió**nak nevezzük. A bővítő konverziók irányát a 12.3. ábra mutatja. A konverzió tranzitív.

Egy `int` típusú értékre például minden további nélkül ráhúzható a `long` típus, hiszen az érték bőven belefér egy `long` típusú „dobozba”, még „lötyög” is benne. A konverzió tranzitív: például ha egy `int` átkonvertálható `long` típussá, az pedig `float`-tá, akkor az `int` átkonvertálható `float` típussá is.



12.3. ábra. A bővítő konverzió iránya

Szűkítő konverzióról akkor beszélünk, ha egy bővebb típusból szűkebb típusba konvertálunk. A szűkítő konverzió információvesztéssel járhat.

A `long` típusú érték például csak akkor fér bele egy `int` típusú változóba, ha az nem nagyobb az `int`-ben ábrázolható maximális, és nem kisebb az `int`-ben ábrázolható minimális értéknél.

Látható, hogy a `char` típus `int` típusúvá bővíthető: az egész érték a karakter unikódja lesz. A többi esetben egyszerű másolás megy végbe, kivéve a `long`→`float` esetet, amikor a pontoság csökkenhet, hiszen a `float` típus lebegőpontosan tárolja a számokat, és csak 6-7 jegy pontosságú (1234567890123456789L→1.23456794E18F).

A `boolean` típusú értéket nem lehet konvertálni egyetlen másik primitív típusba sem, és egyik érték sem konvertálható `boolean` típusúvá.

Például:

```
int i = 'A';           // implicit bővítő konverzió, char → int (i==65)
char c = (char)65;     // explicit szűkítő konverzió, int → char (c=='A')
```

Numerikus operátorok típuskonverziói

Egy numerikus operátornak csak bizonyos típusú operandusai lehetnek. A szorzás bal-, illetve jobb oldalán állhat például numerikus, de nem állhat logikai típusú kifejezés. Emlékezzünk vissza arra, hogy az egész literál automatikusan `int`, a valós literál pedig automatikusan `double` típusú.

A Java a numerikus operátorok operandusait kiértékelés előtt a következő szabályok szerint **automatikusan konvertálja**:

- Unáris (egy operandusú) műveletek esetén: Ha az operandus típusa `int`-nél szűkebb, akkor az operandust `int` típusúvá konvertálja, egyébként nem konvertálja.
- Bináris (két operandusú) műveletek esetén: Mindkét operandust a kettő közül a bővebb, de minimum `int` típussá konvertálja.

Az eredmény típusa a közös, konvertált típus lesz.

Például:

| Kifejezés | Eredeti típusok | Konvertált típusok | Kifejezés típusa |
|----------------------|--------------------------|----------------------------|---------------------|
| <code>b*s</code> | <code>byte*short</code> | <code>int*int</code> | <code>int</code> |
| <code>i/lon</code> | <code>int/long</code> | <code>long/long</code> | <code>long</code> |
| <code>f+3</code> | <code>float+int</code> | <code>float+float</code> | <code>float</code> |
| <code>12L-s</code> | <code>long-short</code> | <code>long-long</code> | <code>long</code> |
| <code>lon*1.0</code> | <code>long*double</code> | <code>double*double</code> | <code>double</code> |
| <code>c/b</code> | <code>char/byte</code> | <code>int/int</code> | <code>int</code> |

A Javában ha az `a` és `b` egész típusú kifejezések, akkor az `a/b` is egész! Az eredmény nem kerekítéssel, hanem levágással keletkezik. Például: `5/6==0, 1/2==0, 25/6==4`.

12.4. Értékadás, értékadási kompatibilitás

Az értékadó operátor bal oldalán egy változó, jobb oldalán egy kifejezés áll:

`<változó> = <kifejezés>`

Az értékadó operátor precedenciája a legkisebb, így mindenkor a jobb oldali kifejezés értékkelődik ki először, és csak ezután kerül be az eredmény a bal oldali változóba. Az értékadó operátor a művelet eredményeképpen visszaadja a jobb oldali kifejezés értékét. Ezért a Javában a többszörös értékadás is értelmezhető:

`<változó1> = <változó2> = ... <változón> = <kifejezés>`

Az értékadó operátor jobb asszociativitású, először tehát a jobb értékadás kerül kiértékelésre (`<változón>=<kifejezés>`), majd a kifejezés sorban bekerül a többi változóba is.

A deklaráló utasításban egy változónak értékkadással kezdőérték adható:

```
<típus> <változó> = <kifejezés>;
```

A következő példában a //1 és //2 sorok egyenértékűek:

```
int n1, n2;
n2 = 5; n1 = n2;           //1
n1 = n2 = 5;              //2
```

A terfogat változónak deklaráláskor adunk értéket:

```
double terfogat = 15*5*Math.PI*4/1E6;
```

A következő példa azt mutatja, hogy a terulet=a*b értéke maga a terulet – ezt írjuk ki:

```
int terulet; int a=2, b=3;
System.out.println(terulet=a*b); // -> 6
System.out.println(terulet*2);   // -> 12
```

A kifejezés típusának **értékkadás szerint kompatibilisnek** kell lennie a változó típusával!

Értékkadás előtt a jobb oldal kiértékelődik, tehát minden oldal típusa adott. Az értékkadás véghajtásakor a Java a következő szabályok szerint jár el:

- Ha a bal és jobb oldal típusai megegyeznek, akkor az értékkadás típuskonverzió nélkül véghajtódik.

```
int i, j;
i = 5;           // minden oldal int típusú
j = i;           // minden oldal int típusú
byte b = (byte)128; // 128 nem fér be, ezért kényszerítjük → -128!!
```

- Ha a jobb oldal szükebb, mint a bal oldal, akkor implicit bővítő konverzió következik be (a jobb oldalt a bal oldal típusára konvertálja).

```
long lon; int i=50; double d;
lon = i;          // implicit konverzió: int → long
d = 2;            // implicit konverzió: int → double
```

- Implicit szűkítő konverzió történik, ha a bal oldal típusa byte, short vagy char, és a fordító el tudja dönteneni, hogy az int típusú jobb oldal belefér-e a bal oldalba. Ezt a fordító természetesen csak akkor tudja eldönteneni, ha a jobb oldali kifejezés kizárolag literálok tartalmaz, vagyis nincsen benne hivatkozás (változó/függvény).

```
byte b = 127;      // 127 értéke belefér a byte-ba
char c = 65;       // belefér, van ilyen karakter
```

- minden más esetben (amikor a fordító nem képes implicit típuskonverzióval a két oldalt kompatibilissá tenni) fordítási hiba keletkezik.

- A fordítási hiba típuskényszerítéssel elkerülhető, de a vele járó esetleges logikai hiba nem!
A típuskényszerítést tehát csak nagy körültekintéssel szabad alkalmazni.

Tekintsük a következő programrészletet:

```
int i= 32000;
short s;
s = (short)i;
System.out.println(i+" , "+s);
```

A program jól működik: kétszer kiírja a 32000 értéket. De ha i kezdeti értéke 33000, akkor s értéke torzul, és a képernyőn ez jelenik meg: 32000 , -32536. A 33000 ugyanis már nem fér bele a short „dobozába”. Ezt a préselést a rendszer automatikusan nem is tenné meg, de mivel mi a leghatározottabban kértük a betételt, hát megtette. A 33000-ből így lett -32536. A fordítóprogram ezért persze nem vállalt garanciát.

Nézzünk meg néhány helyes és helytelen értékadást:

| Helyes | Nem helyes |
|----------------------------|------------|
| d=f; | b=s; |
| f=lon; //lehet adatvesztés | s=i; |
| lon=i; | lon=f; |
| d=b; | c=i; |
| i=c; | c=f; |
| f=c; // f==65.0 | c=s; |
| s=32000; | b=256; |

Kiterjesztett aritmetikai értékadások (*=, /=, %=, +=, -=)

A kiterjesztett értékadó operátor szintaktikai formája a következő:

<változó> <operátor> = <kifejezés> (ahol operátor: *, /, %, +, -)

Szemantikája pedig a következő:

<változó> = (<típus>)<változó> <operátor> <kifejezés>

ahol típus a változó típusa, amelyet rákényszerítünk a jobb oldalra. A változót tehát meg-szorozzuk a kifejezéssel, rákényszerítjük saját típusát, és ez lesz a változó új értéke.

- Vigyázni kell, mert a jobb oldali eredményt a bal oldal csonkíthatja!

Példaként megadunk kiterjesztett értékadó utasításokat, és mindegyikhez megadjuk annak kiterjesztés nélküli pájrát. minden értékadás előtt i értéke legyen 5:

```
int i=5;
```

| Kiterjesztett értékkadás | Kiterjesztés nélküli forma | Eredmény |
|--------------------------|----------------------------|----------|
| i += 4; | i = (int)(i+4); | -> i==9 |
| i *= 6; | i = (int)(i*6); | -> i==30 |
| i *= 1.1; | i = (int)(i*1.1); | -> i==5 |

Az utolsó esetben $5 * 1.1 == 5.5$, ami int-té alakítva 5.

12.5. Kifejezések kiértékelése - példák

Összefoglalásként nézzük meg néhány kifejezés kiértékelését! minden egyes esetben megvizsgáljuk, hogy mi a kifejezés típusa és értéke, az eredményről programrészlettel győződünk meg. A vizsgálandó kifejezéseket vastagon szedtük:

1. példa: Vigyázat, a $d*i/2$ nem ugyanaz, mint $i/2*d$!

```
int i=5; double d=1;
System.out.println(i/2*d);           // 1 → 2.0
System.out.println(d*i/2);           // 2 → 2.5

// 1-ben először az i/2 értékelődik ki: e részkifejezés int típusú, értéke 2; itt véss el
// tehát a pontosság. A végeredmény már double típusú lesz, értéke  $2 * 1.0 == 2.0$ .
// 2-ben először d*i kerül kiértékelésre: e részkifejezés double típusú, értéke 2.5. A
// végeredmény is double típusú lesz, értéke  $2.5 * 1.0 == 2.5$ .
```

2. példa:

```
byte b=1; int i=3;
boolean ok = (b+2*i++)%2==0|i==3;
System.out.println(ok);           // → false
```

1. lépés: $i++$ (a kifejezés típusa int, értéke 3, de: i értéke 4!). **2. lépés:** $2*i++$ (típusa int, értéke 6). **3. lépés:** $b+2*i++$ (típusa int, értéke 7). **4. lépés:** $(b+2*i++) \% 2$ (típusa int, értéke 1). **5. lépés:** $(b+2*i++) \% 2 == 0$ (típusa boolean, értéke false). **6. lépés:** $i==3$ (típusa boolean, értéke false). **7. lépés:** $(b+2*i++) \% 2 == 0 | i==3$ (típusa boolean, értéke false)

3. példa: Vigyázat, túlcordulás!

```
double d; byte b=-2; short s=30000; int i=5000000; float f=5F;
d = f*-b+s*i;
System.out.println(d);           // → -3.2385536E8

Az teljes kifejezés típusa float. Azonban az s*i típusa int, amibe a  $30000 * 5000000$  már nem fér bele, ezért ezen a ponton túlcordulás következik be.
```

3. példa:

```
short s=32; char c=' ';
int i = c/s;
System.out.println(i);           // → 1
```

A char és a short egyaránt int-té konvertálódik, és az eredmény int lesz. Az eredmény tehát $32/32=1$ lesz.

4. példa:

```
char c=65;
double d = c/10;
System.out.println(d);           // → 6.0
```

A c változóba az int típusú 65 belefér, az értékkadás tehát legális. A c/10 kifejezésben c int-té konvertálódik, és az egész osztás eredményeképpen a jobb oldal értéke 6 lesz. Ez kerül bele a double típusú bal oldalba.

5. példa:

Vigyázat, a $b=b+10;$ és a $b+=10;$ értékkadó utasítások nem „egyformák”!

```
byte b=110;
// b=b+10;                      //1 fordítási hiba!
b += 10;                         //2 b==120, még belefér
b += 10;                         //3 b==126, túlcsordulás!
```

//1 fordítási hibát eredményez. Ez azért van, mert a jobb oldal int típusú, amely nem kompatibilis a byte-tal. //2-ben a jobb oldal byte típusú, és az eredmény belefér a bal oldalba. //3-ban az eredmény túlcsordul.

12.6. Feltétel

Feltételnek nevezzük a boolean típusú kifejezést.

Például:

```
a==b & a%2==0 | !a+b<10
```

A feltételek és különösen az összetett feltételek megfogalmazása nem minden egyszerű feladat.

- ◆ Először meg kell fogalmaznunk az egyszerű feltételeket.
- ◆ Ezután az egyszerű feltételeket összekapcsoljuk logikai műveletekkel – a prioritási szabályok figyelembevételével: legerősebb a `!`, aztán az `&` és a `|`, végül az `&&`, és a `||`, egyébként a balról jobbra szabály érvényes.
- ◆ Megpróbáljuk a feltételt minél egyszerűbb formára hozni.
- ◆ A feltételt száraz teszt alá vetjük, vagyis több konkrét esetre kipróbáljuk, helyes-e az eredmény.

Példaként fogalmazzunk meg néhány feltételt! Legyenek a, b, c és d int típusú változók:

- ◆ a és b közül egyik sem nulla:
- a nem nulla: `a!=0`
- b nem nulla: `b!=0`

Teljesüljön minden feltétel egyszerre: $a != 0 \ \&& \ b != 0$

Tesztelejük a feltételt: Legyen például $a = 0$, és $b = 5$. Ekkor az $a != 0$ feltétel `false`, a $b != 0$ feltétel `true`. A `false & true` pedig `false` eredményt ad. Az egyik sem nulla feltétel ebben az esetben tehát hamis, ahogy azt vártuk is.

- ◆ a 25-nél nagyobb és páros, b pedig (a határokat is beleérte) 1000 és 2000 közé eső, nulla végződő szám:
 $(a > 25 \ \&& \ a \% 2 == 0) \ \&& \ (b >= 1000 \ \&& \ b <= 2000 \ \&& \ b \% 10 == 0)$
- ◆ a és b közül legalább az egyik nagyobb, mint 100.
 $a > 100 \ | \ b > 100$
- ◆ a és b közül pontosan az egyik nagyobb, mint 100.
 $a > 100 \ ^ \ b > 100$
- ◆ a és b közül, valamint c és d közül legalább az egyik nagyobb, mint 100.
 $(a > 100 \ | \ b > 100) \ \& \ (c > 100 \ | \ d > 100)$
Itt a zárójel használata fontos, mert az és művelet erősebb, mint a vagy.
- ◆ Vagy az a és b közül, vagy c és d közül minden feltétel nagyobb, mint 100.
 $a > 100 \ \& \ b > 100 \ | \ c > 100 \ \& \ d > 100$
Itt nem kell zárójelet használnunk.

A feltétel tagadása

Amikor egy feltételt tagadunk, akkor annak éppen az ellenkezőjét állítjuk. Például: **Nem igaz az, hogy** a és b közül legalább az egyik nagyobb, mint 100. Ilyenkor a legegyszerűbb, ha megfogalmazzuk a „**hogyan**” utáni feltételt, azt zárójelbe tesszük, és elő írjuk a **nem** műveletet:

$!(a > 100 \ | \ b > 100)$

A zárójel fontos, hiszen a `!` minden erősebb művelet. Rövidebb és hamarabb kiértékelhető eredményt kapunk, ha a fenti logikai kifejezést leegyszerűsítjük. Ehhez ismerni kell a tagadás szabályait, az ún. DeMorgan azonosságokat:

DeMorgan azonosságok

$$\begin{aligned} !(a \ \& \ b) &== !a \ | \ !b \\ !(a \ | \ b) &== !a \ \& \ !b \end{aligned}$$

Vagyis az egyszerű feltételeket külön-külön letagadjuk, majd az és-t vagy-ra, a vagy-ot pedig és-re cseréljük. Példánkban:

$$!(a > 100 \ | \ b > 100) = !(a > 100) \ \& \ !(b > 100)$$

Az egyszerű feltételeket ez esetben könnyű letagadni. A végső változat tehát:

$$a <= 100 \ \& \ b <= 100$$

Bizonyítás igazságítáblával

Előfordulhat, hogy két logikai kifejezés azonosságáról szeretnénk meggyőződni. Az azonosról a legegyszerűbben igazságítáblázat felállításával győződhetünk meg. Az igazságítábláról a logikai operátoroknál szó volt. Ha a logikai kifejezések igazságítáblázatai megegyeznek, akkor a kifejezések azonosak. Íme, annak bizonyítása, hogy $!(a|b) == !a \& !b$. Az igazságítábla két utolsó oszlopa megegyezik:

| a | b | $!(a b)$ | $!a \& !b$ |
|-------|-------|----------|------------|
| true | true | false | false |
| true | false | false | false |
| false | true | false | false |
| false | false | true | true |

12.7. Paraméterátadás, túlterhelt metódusok

Programunkban rendszeresen meghívunk metódusokat (eljárásokat, függvényeket). A hívott metódusok mindegyikét egy konkrét osztályban deklarálták publikus láthatósággal. Példánymetódust egy objektum megszólításával, statikus metódust pedig általában a metódus osztályának megszólításával hívhatunk meg. Például:

```
int y; double d;
d = Math.cos(1.3);
y = Console.readInt("y=");
y = Math.min(2,y+3);
d = Console.readDouble("d=");
System.out.println("Ez egy szöveg paraméter");
```

A `cos` és `min` a `java.lang.Math` osztály statikus függvényei, a `readInt` az `extra.Console` osztály szintén statikus függvénye (a `Math` és `Console` osztályokból nem is lehet példányt létrehozni). A `println` metódust a `java.io.PrintStream` osztályban deklarálták példánymetódusként – a `java.lang.System` osztály `out` objektumát szólítjuk meg vele.

Primitív típusú paraméterátadás

Amikor meghívunk egy metódust, annak a legtöbb esetben paramétereit is át kell adni. Az átadott paramétereket **aktuális paramétereknek** nevezzük, hiszen a metódus sokféle értékkel képes lenne futni, de most aktuálisan azokkal az értékekkel fut, amelyeket megkap. Például:

```
int kisebb = Math.min(3,9); // a min függvény aktuális paraméterei: 3 és 9
double x1 = d*Math.cos(3.2); // a cos függvény aktuális paramétere: 3.2
double x2 = d*Math.cos(2); // a cos függvény aktuális paramétere: 2
```

Az aktuális paramétert a metódus **formális paramétere** kapja meg. Ahhoz, hogy tudjuk, milyen aktuális paraméterek küldhetők a metódusnak, ismerni kell a metódus formális para-

métereit, a paraméterek deklarációit. Ezek hasonlók a változódéklárációkhöz: meg van adva az összes paraméter neve és típusa. A metódus használójának csak a paraméterek számát kell ismernie, és rendre az összes paraméter típusát. A metódus írói ezenkívül még megadták a metódus visszatérési értékének típusát, ezt a metódus neve elő írták. A `java.lang.Math` osztály `min` és `cos` függvényeit például így adták meg:

```

    visszatérési érték
    ↘
    ◀ int min(int a, int b) // két int értéket fogad, egy int értéket ad vissza
    ◀ double cos(double a) // egy double értéket fogad, egy double értéket ad vissza
    ↙ formális paraméterek
  
```

A metódus aktuális paraméterei rendre átadódnak a formális paramétereknek. A paraméterek számának egyeznie kell, és az aktuális paramétereknek rendre értékadás szerint kompatibiliseknek kell lenniük a formális paraméterekkel.

Például:

```

int min(int a, int b)           // metódus feje
int m = Math.min(x+2, 3)        // metódus hívása (a=x+2, b=3)
  
```

Itt `x+2`-nek `int`-nél szükebb, vagy `int` típusúnak kell lennie, azaz `x` csak `byte`, `short`, `int` vagy `char` típusú lehet.

Metódusok szignatúrája és túlterhelése

Egy metódusnak többféle paraméterezése is lehetséges. A `min` függvénynek például négy változatát is megtalálhatjuk a `Math` osztályban. Ezek mind különböző függvények, amelyeknek egyenként írták meg a blokkjukat:

```

int min(int a, int b)           //1
long min(long a, long b)        //2
float min(float a, float b)     //3
double min(double a, double b)   //4
  
```

A Javában egy metódust annak **szignatúrája** azonosít, vagyis a metódus neve, a paraméterek száma, és sorban a paraméterek típusai (például `min,int,int`). Az ugyanolyan néven, de más paraméterezással megadott metódusokat **túlterhelt (overloaded)** metódusoknak nevezünk (a `min` neve túl van terhelve).

Túlterhelés esetén a metódus hívásakor már a fordító kiválasztja a metódus „fajtáját”: amilyen típusú értékekkel hívták a metódust, azt a paraméterező metódust fogja kiválasztani. A `min(x, 3)` esetén például, ha az `x` paraméter típusa `int` vagy annál szükebb, akkor a //1 metódus kerül végrehajtásra, ha `x` `float` típusú, akkor a //3. A `min(x, 3.0)` a //4 metódus végrehajtását eredményezi.

A Javában csak érték szerinti paraméterátadás létezik, és ez azt jelenti, hogy paramétereken keresztül nem kaphatunk vissza értéket. Az aktuális paramétereket csak használja a metódus, az átadott változó például biztosan érintetlen marad, azt a metódus nem tudja megváltoztatni. Értéket csak visszatérési értéken keresztül kaphatunk a metódustól. Ebből az is következik, hogy egy metódustól egyszerre csak egy értéket kaphatunk, hiszen a metódusnak csak egy visszatérési értéke lehet. Ha a visszatérési érték típusa `void`, az azt jelenti, hogy nincs visszatérési érték. Ekkor a metódus csak eljárásként hívható. Megjegyezzük, hogy a visszatérési értékkel rendelkező metódus is hívható eljárásként, ekkor a visszatérési érték elvész, az nem kerül felhasználásra. Ennek persze csak akkor van értelme, ha a függvény az érték kiszámításán kívül más hasznosat is tesz.

12.8. java.lang.Math osztály

A `Math` osztály rengeteg hasznos matematikai függvényt és konstanst tartalmaz. Az osztályt nem lehet példányosítani (nincs publikus konstruktora), az kizárolag statikus deklarációkat tartalmaz.

A `java.lang.Math` számunkra lényegesebb adatait és metódusait a 12.4. ábra (táblázat) foglalja össze. Az osztály a `java.lang` csomagban van, azt nem kell importálnunk. A szögek radiánban értendők! $2 * \text{PI}$ radián = 360 fok, vagyis 1 radián = $180/\text{PI}$ fok ~57.3 fok.

Adatok (konstansok)

| | |
|------------------------------|--------------------|
| <code>final double E</code> | a természetes alap |
| <code>final double PI</code> | Pi értéke |

Metódusok

| | |
|---|--|
| <code>double abs(double a)</code> | visszaadja a double szám abszolút értékét |
| <code>float abs(float a)</code> | visszaadja a float szám abszolút értékét |
| <code>int abs(int a)</code> | visszaadja az int szám abszolút értékét |
| <code>long abs(long a)</code> | visszaadja a long szám abszolút értékét |
| <code>double acos(double a)</code> | visszaadja a arcus koszinuszát radiánban ($0..\text{PI}$) |
| <code>double asin(double a)</code> | visszaadja a arcus szinuszt ($-\text{PI}/2..\text{PI}/2$) |
| <code>double atan(double a)</code> | visszaadja a arcus tangensét ($-\text{PI}/2..\text{PI}/2$) |
| <code>double ceil(double a)</code> | a legközelebbi egész, amely nem kisebb nála |
| <code>double cos(double a)</code> | az a szög (radián) koszinuszát adja vissza |
| <code>double exp(double a)</code> | a visszaadott érték e^a (e a természetes alap) |
| <code>double floor(double a)</code> | a legközelebbi egész, amely nem nagyobb nála |
| <code>double log(double a)</code> | természetes alapú logaritmus függvény |
| <code>double max(double a, double b)</code> | visszaadja a nagyobbik double számot |
| <code>float max(float a, float b)</code> | visszaadja a nagyobbik float számot |
| <code>int max(int a, int b)</code> | visszaadja a nagyobbik int számot |
| <code>long max(long a, long b)</code> | visszaadja a nagyobbik long számot |

| | | |
|--------|---------------------------------|--|
| double | min(double a, double b) | visszaadja a kisebbik double számot |
| float | min(float a, float b) | visszaadja a kisebbik float számot |
| int | min(int a, int b) | visszaadja a kisebbik int számot |
| long | min(long a, long b) | visszaadja a kisebbik long számot |
| double | pow(double a, double b) | hatványozás. Visszaadja a^b értékét |
| double | random() | ad egy véletlen számot, mely $>=0.0$ és <1.0 . |
| double | rint(double a) | a legközelebbi egész |
| long | round(double a) | a legközelebbi long érték |
| int | round(float a) | a legközelebbi int érték |
| double | sin(double a) | az a szög (radián) szinuszát adja vissza |
| double | sqrt(double a) | visszaadja a négyzetgyökét |
| double | tan(double a) | az a szög (radián) tangensét adja vissza |
| double | toDegrees(double radian) | a radiánban megadott szöget átváltja fokra |
| double | toRadians(double degree) | a fokban megadott szöget átváltja radiánra |

12.4. ábra. A java.lang.Math osztály metódusai

Feladat – Kamat

Ha beteszünk a bankba egy adott összeget, adott éves kamatszázalékra, adott hónapra, mennyi pénzt vehetünk majd fel az idő lejártakor?

Három adatot fogunk bekérni tehát: az összeget, a kamatot százalékban, és a hónapok számát – ezekből az adatokból kell kiszámítani az új összeget. Mivel évi kamatos kamattal számolunk, annyiszor kell szoroznunk az összeget az $1+kamatSzázalék/100$ szorzóval, ahány évig az kamatozik, vagyis a szorzót az évvel ($hónap/12$) hatványozni kell. Bonyolítja a helyzetet, hogy az év tört szám is lehet:

$$\text{újÖsszeg} = \text{összeg} * (1+\text{kamatSzázalék}/100)^{\text{év}}.$$

Forráskód

```
import extra.*;
public class Kamat {
    public static void main(String[] args) {
        double osszeg, ujOsszeg;
        double kamatSzazalek;
        int honap;
        osszeg = Console.readDouble("Összeg (Ft)? ");
        kamatSzazalek = Console.readDouble("Kamat%? ");
        honap = Console.readInt("Hány hónapra köti le? ");
        ujOsszeg = osszeg *
            Math.pow(1+kamatSzazalek/100,honap/12.0); //1
        System.out.println(honap+" hónap múlva "+
            Format.left(ujOsszeg,0,2)+" Ft-ot vehet ki"); //2
    }
}
```

A program futása

```
| Összeg (Ft)? 5000
| Kamat%? 12.5
| Hány hónapra köti le? 5
| 5 hónap műlva 5251,50 Ft-ot vehet ki
```

A program elemzése

Két sor érdemel külön figyelmet. //1-ben fontos, hogy a hónapot double típusú értékkel osszuk, mert az egész osztás torz eredményhez vezetne. //2-ben a Format.left() metódus második paramétere 0; így az eredmény pontosan annyi karakter hosszú, ahány karakteren a szám elfér.

Tesztkérdések

- 12.1. Egészítse ki a következő igazságátáblát! Melyik pont tartalmazza a hiányzó oszlop értékeit? Az oszlop elemeinek felsorolása fentről lefelé történik. Jelölje meg az egyetlen jó választ!

| a | b | a&b a |
|-------|-------|---------|
| true | true | ... |
| true | false | ... |
| false | true | ... |
| false | false | ... |

a) true false true false
b) true true false false
c) false false false false
d) true false false false

- 12.2. Mely kifejezés szerepelhet az igazságátábla fejlécében? Jelölje meg az egyetlen jó választ!

| a | b | ... |
|-------|-------|-------|
| true | true | true |
| true | false | false |
| false | true | false |
| false | false | false |

a) a | b
b) a & b
c) a | b & a
d) !b

- 12.3. A megadott deklarációk mellett mi a típusa a következő kifejezésnek? Jelölje meg az egyetlen jó választ!

```
boolean a=true;
byte b=1;
short c=21;
```

(c%5!=b) | !a

- a) true
b) false
c) 1
d) szintaktikai hiba
- 12.4. Mely pontra igaz, hogy a felsorolt típusok szigorúan bővülő sorrendben vannak? Jelölje meg az összes jó választ!
- a) byte, short, int, long, float, double, char, boolean
b) byte, short, int, long, float, double
c) byte, short, int, char, long, float, double
d) char, int, long, double
- 12.5. Jelölje meg az összes igaz állítást a következők közül!
- a) Az implicit típuskonverziót a program magától elvégzi.
b) A prefix operátor az operandus előtt szerepel.
c) Értékadáskor a jobb oldal típusának minden bővebbnek kell lennie a bal oldal típusánál.
d) Értékadáskor a bal oldal típusa egyértelműen meghatározza a jobb oldal típusát.
- 12.6. Jelölje meg az összes igaz állítást a következők közül!
- a) A Java minden aritmetikai műveletet legalább int típusba konvertálva végez.
b) A boolean típus a memóriában 1 bitet foglal el, mert a lehetséges két érték tárolására elegendő ennyi hely.
c) Egy char típusú kifejezés minden értékül adható egy int típusú változónak, kényszerítés nélkül.
d) Egy int típusú kifejezés minden értékül adható egy char típusú változónak, kényszerítés nélkül.
- 12.7. Mit ír ki a következő programrészlet? Jelölje meg az egyetlen jó választ!
- ```
int a=2, b=5;
System.out.print(++a*4+b+ " " +b++);
```
- a) 17 6  
b) 17 5  
c) 13 5  
d) Egyik sem
- 12.8. Jelölje meg az összes igaz állítást a következők közül!
- a) A / és a % operátorok ugyanolyan precedenciával rendelkeznek.  
b) A >= és a != operátorok ugyanolyan precedenciával rendelkeznek.  
c) A + és – kétoperandusú operátorok magasabb precedenciával rendelkeznek, mint a \* és / operátorok.  
d) A || operátor magasabb precedenciával rendelkezik, mint a && operátor.

- 12.9. A megadott deklarációk mellett mi a típusa a következő kifejezésnek? Jelölje meg az egyetlen jó választ!

```
byte b; int i;
```

```
(byte)Math.sqrt(i)+3*b
```

- a) double
- b) int
- c) byte
- d) A kifejezés szintaktikailag hibás

- 12.10. A megadott deklarációk mellett a következő kifejezések közül melyikre igaz: szintaktikailag helyes, és típusa int? Jelölje meg az összes jó választ!

```
byte b; int i; double d;
```

- a) (b+i)/2
- b) (i/2+5)/2
- c) b\*i\*55.0%2
- d) 55\*b\*Math.pow(5, 5)

## Feladatok

- 12.1. a, b és c valós típusú változók. Írja le azt a logikai kifejezést, amely igaz, ha:

- a) (A) a értéke 0 és 1 közé esik (a határokat beleértjük)
- b) (A) a értéke 5 és 10, b értéke pedig 15 és 20 közé esik (a határokat beleértjük)
- c) (A) nem igaz az előző állítás
- d) (A) sem a, sem b, sem c nem 0
- e) (B) a egész
- f) (B) a, b és c közül legalább 2 szám egész
- g) (C) a és b közül az egyik pozitív, a másik negatív
- h) (C) a, b és c közül legfeljebb kettő pozitív

(Kifejezesek1.java)

- 12.2. (B) a és b egész típusú változók. Írja le azt a logikai kifejezést, mely igaz, ha:

- a) a és b közül legfeljebb az egyik páros
- b) a és b közül pontosan az egyik páros

(Kifejezesek2.java)

- 12.3. (A) Tárolja konstansokban a krumpli, a hagyma és a padlizsán egységárát! Írjon olyan programot, amely bekéri, miből mennyit óhajt a vásárló, majd készítsen egy számlát a következő formában:

|                                   |         |
|-----------------------------------|---------|
| krumpli : 2.5 kg * 70 Ft/kg =     | 175 Ft  |
| hagyma : 3.0 kg * 98 Ft/kg =      | 294 Ft  |
| padlizsán : 10.0 kg * 200 Ft/kg = | 2000 Ft |
| <hr/>                             |         |
| Összesen                          | 2469 Ft |

Az egységárák egészek, a mennyiség valós is lehet. A vásárolt adagok értékeit kerekít-sük egészre! (*Szamla.java*)

- 12.4. (A) A piacon minden áru olcsóbb, mint a szomszédunkban levő boltban. Kérje be, mennyibe kerül egy kg eper a piacon, valamint a szomszéd boltban. Ezt követően kérje be, hány kg epret szeretnénk befőzni. Írja ki, hány Ft veszteség ér bennünket, ha a szomszéd boltban vesszük az epret. (*Piac.java*)
- 12.5. (A) Kérje be a gömb sugarát, majd írja ki a gömb felszínét és térfogatát! (*Gomb.java*)
- 12.6. (B) Ha a számla ÁFA összege a számla nettó értékének egy adott százaléka, akkor hány százalék ÁFÁ-t tartalmaz a számla bruttó összege? Készítsen a problémára egy kisegítő programot! Például: 25%-os ÁFA esetén a számla 20% ÁFÁ-t tartalmaz, 12%-os ÁFA esetén a számla ÁFA tartalma 10.71 százalék. (*AfaTartalom.java*)
- 12.7. (C) Feri pénzt kap. Hogy mennyit, azt kérje be a program. A kifizetéshez 5000, 1000, 500 és 100 Ft-os címletek állnak rendelkezésre – a maradékot Feri nem kapja meg. Feltételezzük, hogy minden címletből van elég, és a lehető legkevesebb számú pénz kerül kiosztásra. Milyen címletből hányat kapott Feri, és mennyi pénzt hagyott ott ajándékba? (*Penzvaltas.java*)

Vigyázzon a jó változónevek megválasztására és a program helyes strukturálására!

## 13. Szelekciók

---

A fejezet pontjai:

1. Egyágú szelekció – if
  2. Kétágú szelekció – if..else
  3. Egymásba ágyazott szelekciók
  4. Többágú szelekciók – else if és switch
  5. Független feltételek vizsgálata
- 

A szelekció válogatást jelent. Előre elgondoljuk, hogy milyen esetek következhetnek be, és minden esetre megadunk egy tevékenységet vagy tevékenységsorozatot, melyet a programnak végre kell hajtania. A végrehajtás után a program újra „összefolyik”, s egy közös tevékenységgel folytatódik. Többféle szelekció létezik. Az esetek száma szerint megkülönböztetünk egyágú, kétágú, illetve többágú szelekciót. A szelekciók egymásba ágyazhatók. Ebben a fejezetben sorra vesszük a szelekciók különböző fajtait.

### 13.1. Egyágú szelekció – if

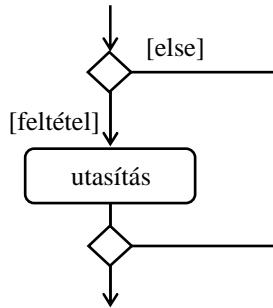
Az **egyágú szelekció** azt jelenti, hogy ha igaz egy adott feltétel, akkor a hozzá kapcsolódó utasítást (tevékenységet) végrehajtjuk, egyébként pedig kikerüljük, és a programot az azt követő közös utasítással folytatjuk. Ha több utasítást szeretnénk az igaz ágban végrehajtani, az utasításokat blokkba zárjuk. Az egyágú szelekció tevékenységsdiagramját a 13.1. ábra mutatja, Java kódja a következő:

```
if (feltétel) {
 utasítás1;
 utasítás2;
 ...
 utasításN;
}
```

Az **if** magyar jelentése: ha. Az **if** utáni feltételt (boolean típusú kifejezést) zárójelbe kell tenni, a feltétel után pedig megadjuk az utasítást, vagy a több utasítást tartalmazó blokkot.

Kódolási konvenciótól függően a blokk kezdő zárójelét tehetjük új sorba is:

```
if (feltétel)
{
 ...
}
```



13.1. ábra. Az egyágú szelekció tevékenységsdiagramja

• Vigyázat! Az `if (feltétel)` után ne tegyen pontosvesszőt, mert ezzel lezárja az `if` utasítást úgy, hogy az igaz ágon minden összetevő üres utasítás áll, és a pontosvessző utáni utasítás az `if` utasítást szekvenciálisan követi!

```
int n=10;
if (n==5); // Ne tegyen pontosvesszőt!!!
 n*=2; // A kód strukturálása így helytelen!!!
// n értéke 20!!!
```

Általános szabályok a Javában:

- minden **feltételezett zárójelbe** kell tenni;
- az utasítások blokkba tehetők. A blokk összfogja a benne található utasításokat.

### A program strukturálása (tördelezése)

Figyelje meg, hogy azok az utasítások, amelyeknek végrehajtása feltételektől függ, a forráskódban két karakterhellyel jobbra tolva szerepelnek!

A forráskódban a vezérlőszerkezetek jelenlétéit a forrásszöveg tördelésével, a sorok vízszintes eltolásával fejezzük ki. Ha egy tevékenység beljebb kezdődik, mint egy valamelyik felette álló elem, az minden esetben egy vezérlőszerkezet jelenlétéit mutatja. **A vezérlőszerkezetek belső alarendelt egységeit a forráskódban jobbra toljuk.**

- A szekvenciálisan egymást követő utasítások pontosan egymás alatt szerepelnek;
- Az az utasítás vagy blokk, amelynek végrehajtása feltételektől függ, két vagy három karakterhellyel jobbra tolva szerepel a forráskódban.

Fontos, hogy egy **olyan kódolási konvenciót alakítsunk ki, amely eleget tesz az általános szabályoknak**. A szép, strukturált forráskód olvashatóvá teszi a programot. A strukturálás szabályai a programozóra nézve kötelezők!

Ezt a kódolási konvenciót aztán az egész programon át kövessük, ellenkező esetben programunk áttekinthetetlenné válik. A szép program írása pusztán szokás dolga, semmiféle plusz megerőltetést nem kíván. Az utólagos szépítgetés mindig veszteségekkel jár. A kifejezetten program írására készített szövegszerkesztőkben megvan a lehetőségünk arra, hogy programunkat **Indent** módban írjuk. Ilyenkor az **Enter** lenyomására a kurzor a most beírt sor első karaktere alá áll, s ez nagyban elősegíti a szekvenciák kódolását. Ha ilyenkor lenyomjuk a **Backspace** billentyűt, akkor a kurzor az aktuális sor felett levő első, tőle balra „kiálló” sor alá áll, ez pedig az egyéb struktúrák programozásában nyújt segítséget (például egy szelekció vagy iteráció befejezésében). A kódolási konvenciókat folyamatosan tárgyaljuk a különböző struktúráknál. Tartsa szem előtt a következőket:

- ◆ Az a jó, ha az eltolás egysége jól látható (legalább két karakternyi), és nem túl sok (maximum három karakternyi), hogy a mélyebben strukturált program se csússzon le a képernyöről, illetve a papírról.
- ◆ A beljebb toláshoz lehetőleg ne használjuk a tabuláltort, inkább üssük le kétszer a szóközt! Ha ragaszkodunk a **Tab** karakterhez, akkor állítsuk be a környezetben a **Tab méretét** 2-re, valamint állítsuk be a **Szóközök Tab helyett** lehetőséget (így minden Tab leütésekor 2 szóköz kerül a szövegbe)! Az „igazi” Tab karakter és a szóköz váltakozó használata a legrosszabb, mert így a forráskód teljesen „széteshet” a különböző szövegszerkesztőkben!
- ◆ Ha ez az áttekinthetőséget nem zavarja, akkor kivételesen megengedhető, hogy egy sorba két utasítást írunk.
- ◆ Ha egy utasítás csak két sorban fér ki, akkor a sort áttekinthetően tagoljuk, és a második sort egy egységgel beljebb tolva kezdjük!

### Feladat – Fizetés

Kérjük be konzolról egy alkalmazott fizetését! Ha ez a fizetés 100000 forintnál nem nagyobb, akkor emeljük meg 25%-kal. Végül írjuk ki az alkalmazott fizetését!

### Forráskód

```
import extra.*;

public class Fizetes {
 public static void main(String[] args) {
 int fizetes = Console.readInt("Fizetés? ");
 if (fizetes < 100000)
 fizetes *= 1.25;
 System.out.println("Az új fizetés: "+fizetes+" Ft");
 }
}
```

### A program egy lehetséges futása:

|                          |
|--------------------------|
| Fizetés? 90000           |
| Az új fizetés: 112500 Ft |

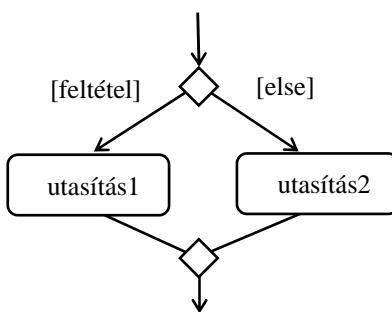
**Feladat – Következő karakter**

Kérjünk be konzolról egy karaktert! Ha a karakter 'A' és 'Z' közé esik, akkor írjuk ki az unikód táblában öt követő karaktert, egyébként ne írunk ki semmit!

**Forráskód**

```
import extra.*;
public class KovKarakter {
 public static void main(String[] args) {
 char karakter = Console.readChar("Karakter? ");
 if (karakter >= 'A' && karakter <= 'Z') {
 karakter++;
 System.out.println(karakter);
 }
 }
}
```

**Más megoldások:** A `karakter++;` helyett használhattuk volna a következő utasításokat is:  
`karakter+=1;` vagy `karakter=(char)(karakter+1);`

**13.2. Kétágú szelekció – if..else**

13.2. ábra. A kétágú szelekció tevékenységsdiagramja

A **kétágú szelekció** azt jelenti, hogy ha igaz egy adott feltétel, akkor a hozzá kapcsolódó utasítást (tevékenységet) hajtjuk végre, egyébként egy másikat. Az elágazás után a program „összefolyik”, vagyis egy közös utasítás következik. Ha több utasítást szeretnénk valamelyik ágban végrehajtani, akkor ott az utasításokat blokkba zártuk. Az `else` magyar jelentése: egyébként. A kétágú szelekció tevékenységsdiagramját a 13.2. ábra mutatja, Java kódja a következő:

```

 if (feltétel) {
 utasítás;
 ...
 }
 else {
 utasítás;
 ...
 }
 }
}

```

**Feladat – Jó szám**

Kérjünk be konzolról egy valós számot! A szám akkor jó, ha 1000 és 2000 közötti páros egész (a határokat is beleértve). Írjuk ki, hogy a szám jó, vagy nem jó!

**Forráskód**

```

import extra.*;
public class JoSzam {
 public static void main(String[] args) {
 double szam = Console.readDouble("Kérek egy számot: ");
 if (szam>=1000 && szam<=2000 && szam%2==0)
 System.out.println("Jó szám");
 else
 System.out.println("Nem jó szám");
 }
}

```

**13.3. Egymásba ágyazott szelekciók**

Az `if` utasítás megfelelő ágán akármilyen utasítás állhat, akár egy újabb `if` utasítás. Ilyenkor azonban különösen vigyázni kell a tiszta programszerkezetre, mert könnyen elkeveredhetünk. **Mindig egyértelműen látni kell a program struktúráját!**

**Feladat – Legnagyobb**

Kérjünk be három számot, majd írjuk ki ezek közül a legnagyobbat!

**Forráskód**

```

import extra.*;
public class Legnagyobb {
 public static void main(String[] args) {
 int szam1, szam2, szam3;
 int legnagyobb;

 szam1 = Console.readInt("1. szám: ");
 szam2 = Console.readInt("2. szám: ");
 szam3 = Console.readInt("3. szám: ");
 }
}

```

```
if (szam1 > szam2)
 if (szam1>szam3)
 legnagyobb = szam1;
 else
 legnagyobb = szam3;
else
 if (szam2>szam3)
 legnagyobb = szam2;
 else
 legnagyobb = szam3;

 System.out.println("A legnagyobb szám: " + legnagyobb);
}
}
```

### A program egy lehetséges futása

```
1. szám: 33
2. szám: 29460
3. szám: 2
A legnagyobb szám: 29460
```

### A program elemzése

A három szám beolvasása után megnézzük az első két számot, melyik a nagyobb (//1). Ha a szam1 a nagyobb, akkor a legnagyobb szám csak szam1 és szam3 közül kerülhet ki (//2). Ha szam2 a nagyobb, akkor szam2 és szam3 közül választhatjuk ki a legnagyobbat (//3).

### Az if és else párosítása

Az if utasítások egymásba ágyazhatók. Ilyenkor a fordító az **else kulcsszavakat mindenkor a legutóbbi if kulcsszóval párosítja**, ha csak ezt a szabályt blokkok képzésével felül nem bíráljuk.

Összetett, nehezen áttekinthető szelekció írásakor érdemes az egyes ágakat blokkba foglalni még akkor is, ha abban csupán egyetlen utasítás van. Ez különösen akkor hasznos, ha a szelekció nem szimmetrikus.

A következő példa mutatja, mennyire megtévesztő lehet egy rossz kódolás. Egy kétágú szelekció igaz ágába szeretnénk egy egyágú szelekciót illeszteni:

```
if (f1)
 if (f2) //1
 u1;
else //2 Vigyázat, rossz kódolás!
 u2;
```

A fordító sajnos nem olvassa ki a gondolatainkat. Ő a //2 sor else-ét az //1 sor if-ével állítja párba, és a következő kódolás szerint fog eljárni:

```
if (f1)
 if (f2)
 u1;
 else
 u2;
```

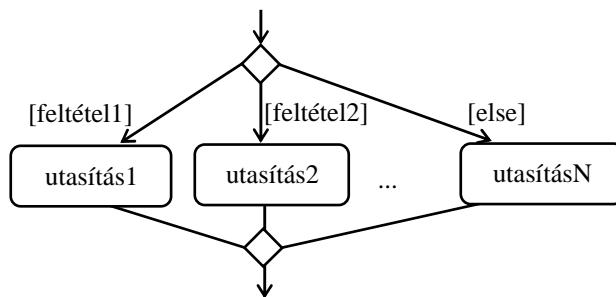
**Jobban járunk tehát, ha úgy strukturálunk, ahogy azt a fordító is értelmezi.** Ha ragaszkodunk eredeti elképzelésünkhez, akkor a helyes kód így fest:

```
if (f1) {
 if (f2)
 u1;
}
else
 u2;
```

### 13.4. Többágú szelekciók – else if és switch

**A szelekciótól több ága is lehetséges.** A többágú szelekció tevékenysége diagramját a 13.3. ábra mutatja. Ilyenkor a **feltételek közül legfeljebb egy teljesülhet**, vagyis ha az egyik feltétel teljesül, a többi már szóba sem jöhet. A feltételek tehát ilyen értelemben egymástól függenek, a kiértékelés a tevékenységsdiagram szerint balról jobbra történik. Utolsó ágként megadható egy **egyébként (else)** ág, amelyre akkor kerül a vezérlés, ha egyik feltétel sem volt igaz. Ha nincs egyébként ág, akkor előfordulhat, hogy egyetlen ág sem kerül végrehajtásra. Az egy- és a kétágú szelekció a többágú szelekció speciális esetei.

A Javában a többágú szelekciót kétféle vezérlőszerkezzel valósíthatjuk meg: ezek az `else if` és a `switch` szerkezetek.



13.3. ábra. A többágú szelekció tevékenységsdiagramja

Mivel a számítógép egyszerre csak egy feltétel igaz vagy hamis voltát tudja megállapítani, a több esetből történő válogatás szükségszerűen egymásba ágyazott `if` utasítások sorozata. Ha az első feltétel nem teljesül, akkor a második feltétel vizsgálata következik; ha az sem teljesül, akkor a harmadiké stb. egészen addig, amíg valamelyik feltétel végül mégiscsak teljesül, vagy nincs több feltétel. A vezérlés legfeljebb egy igaz ágon fog áthaladni. Végső esetben az `else` ágra kerül a vezérlés, ha van `else` ág. Ha valamely feltétel esetén nem akarunk tevékenységet végrehajtani, akkor azon az ágon nem adunk meg tevékenységet (üresen hagyjuk).

A feladatok többségében a többágú szelekció esetei elvileg egyenrangúak. A szintaktika azonban ekkor is megköveteli a feltételek sorrendjének megadását. Előfordulhat, hogy a feltételek sorrendje a program működése szempontjából nem mindegy; de ha mindegy is, ajánlatos egy logikus sorrendet felállítani a program olvashatosága érdekében. Végig kell gondolnunk:

- ◆ Van-e átfedés a feltételek között? Figyelembe kell venni, hogy csak az első igaz feltételhez tartozó utasítás hajtódkik végre!
- ◆ Melyik feltétel teljesülése a gyakoribb? Bizonyos esetekben, amikor számít, hogy a gép mennyit dolgozik, és fel tudjuk mérni az esetek előfordulásainak gyakoriságát, akkor a gyakoribb eseteket előrébb tesszük, hiszen a feltétel vizsgálata is gépidőt vesz igénybe.
- ◆ Milyen sorrend megadása mellett olvashatóbb a program kódja?

### **else if**

Az `else if` szerkezzel bármilyen többágú szelekció lekódolható. Szintaktikája:

```
if (feltétel1)
 utasítás1;
else if (feltétel2)
 utasítás2;
...
else
 utasításN;
```

Az utasítások helyén blokkok is állhatnak. Az `else if` szerkezet tulajdonképpen egymásba ágyazott `if` utasításoknak egy trükkös, látványban eltorzított, de az ember számára mégis természetesebb változata. E struktúra kihangsúlyozza az esetek egyenrangúságát.

#### **Feladat – Kor**

Olvassunk be egy nem negatív egész számot, valakinek az életkorát. Kortól függően írjuk ki a megfelelő szöveget:

|             |            |
|-------------|------------|
| 0-13 évig:  | Gyerek     |
| 14-17 évig: | Fiatalkorú |
| 18-23 évig: | Ifjú       |
| 24-59 évig: | Felnőtt    |
| 60 évtől :  | Idős       |

### Forráskód

```

import extra.*;
public class Kor {
 public static void main(String[] args) {
 int kor = Console.readInt("Hány éves? ");
 if (kor < 14) // kor = ..13
 System.out.println("Gyerek");
 else if (kor < 18) // kor = 14..17
 System.out.println("Fiatalkorú");
 else if (kor < 24) // kor = 18..24
 System.out.println("Ifjú");
 else if (kor < 60) // kor = 25..59
 System.out.println("Felnőtt");
 else // kor = 60..
 System.out.println("Idős");
 }
}

```

#### Feladat – Mi igaz?

Kérjen be egy tetszőleges egész számot! Állapítsa meg, hogy a következő két állítás közül melyik igaz: 1. állítás: 50 és 150 közé esik; 2. állítás: Páratlan, és a szám 5. hatványa kisebb, mint 1 millió. Ennek megfelelően írja ki a következőket:

- Csak az 1. állítás igaz
- Csak a 2. állítás igaz
- Mindkét állítás igaz
- Egyik állítás sem igaz

### Forráskód

```

import extra.*;

public class MiIgaz {
 public static void main(String[] args) {
 int szam = Console.readInt("Szám= ");
 boolean allitas1 = szam >=50 && szam <= 150;
 boolean allitas2 = szam%2==1 && Math.pow(szam,5)<1E6;

 if (allitas1 && !allitas2)
 System.out.println("Csak az 1. állítás igaz");
 else if (!allitas1 && allitas2)
 System.out.println("Csak a 2. állítás igaz");
 else if (allitas1 && allitas2)
 System.out.println("Mindket állítás igaz");
 else
 System.out.println("Egyik állítás sem igaz");
 }
}

```

## switch

A switch szerkezet is többágú szelekció kódolására használatos, de csak akkor alkalmazható, ha egy kifejezés jól meghatározott, különálló értékeire szeretnénk bizonyos utasításokat végre hajtani. Az utasítás szerkezete a következő:

```
switch (kifejezés) {
 case érték1: utasítások; break;
 case érték2: utasítások; break;
 ...
 default: utasítások;
}
```

Jelentések: switch: kapcsol; case: eset; default: alapértelmezés; break: megtörés.

A switch utasítás az utasítás fejéből és egy blokkból áll. A fej tartalmazza a switch kulcsszót, és zárójelben egy kifejezést. A blokk akárhány esetet tartalmazhat. Az eseteket a case kulcsszó vezeti be, utolsó esetként megadható egy default, alapértelmezett eset.

Szabályok:

- **kifejezés:** tetszőleges kifejezés, de visszatérési típusa csak primitív lehet, mégpedig byte, short, int vagy char (nem lehet tehát sem long, sem boolean, sem valós).
- **értékI:** egyetlen érték, mely csak konstans kifejezés lehet – olyan kifejezés, amelyet a fordító ki tud értékelni. Egy konstans kifejezés csak literált és konstanst (final változót) tartalmazhat, metódushívás nem szerepelhet benne. minden eset után pontosan egy érték adható meg, és ennek egyenlőnek kell lennie a kifejezéssel.
- Az egyes esetekben, vagyis a case érték: után akárhány utasítás megadható. Az eset lehet „üres” is – nem kötelező tehát megadni semmilyen utasítást, így a break-et sem.
- Az esetek sorban egymás után kerülnek végre hajtásra, ezt csak a break utasítás bírálja felül. A break hatására a switch blokk végére kerül a vezérlés. Ha nem teszünk a case végére break utasítást, akkor a vezérlés a következő case-re kerül! Ha egyik eset sem „igaz”, akkor a default eset hajtódik végre. Nem kötelező default esetet adni.
- Két egyforma eset megadását a szintaktika nem engedi meg.
- Értéktartományt több egymás utáni case leírásával adhatunk meg.

### Feladat – Kocka

Dobjunk egyet a kockával! A dobás értékétől függően írjuk ki a következő üzenetek egyikét:

```
1..3: Gyenge!
 4: Nem rossz!
 5: Egész jó!
 6: Fantasztikus!
```

Ha a dobás nem 1 és 6 között van, írjuk ki, hogy Érvénytelen dobás!

### Forráskód

```
import extra.*;

public class Kocka {
 public static void main(String[] args) {
 int dobas = Console.readInt("Dobj! 1-6: ");

 switch (doras) {
 case 1:
 case 2:
 case 3:
 System.out.println("Gyenge!");
 break;
 case 4:
 System.out.println("Nem rossz!");
 break;
 case 5:
 System.out.println("Egész jó!");
 break;
 case 6:
 System.out.println("Fantasztikus!");
 break;
 default:
 System.out.println("Érvénytelen dobás!");
 }
 }
}
```

*Megjegyzés:* A switch nem alkalmas feltételek megfogalmazására. Az előző, Mi igaz? feladatot például nem lehet megoldani switch szerkezzel.

## 13.5. Független feltételek vizsgálata

Előfordul, hogy egymás után több feltételt is meg akarunk vizsgálni, de ezek mindegyikét külön-külön, egymástól függetlenül. A feltételeket persze valamilyen sorrendbe kell állítani, de függetlenül attól, hogy valamelyik feltétel teljesül-e, meg akarjuk vizsgálni a többit is.

**Független feltételek vizsgálata** esetén nem többágú szelekcióról van szó, hanem **egyágú szelekciókból álló szekvenciáról**. A feltételek egymástól függetlenek, azokat egymás után meg kell vizsgálni, a hozzájuk tartozó tevékenységeket a feltétel teljesülése esetén végre kell hajtani:

```
if (feltétel1)
 utasítás1;
if (feltétel2)
 utasítás2;
...
```

A szelekciók sorrendjét alaposan át kell gondolni, mert a már végrehajtott utasítás hatással lehet a következő szelekció feltételére!

**Feladat – Osztható?**

Kérjünk be egy számot! Írjuk ki, hogy az osztható-e 2-vel, 3-mal, illetve 5-tel (külön-külön)!

A feltételek megfogalmazása megtévesztő: ezek a feltételek teljesen függetlenek egymástól.

**Forráskód**

```
import extra.*;

public class Oszthato {
 public static void main(String[] args) {
 int szam = Console.readInt("Adj meg egy egész szamot: ");
 if (szam%2==0)
 System.out.println("Osztható 2-vel");
 if (szam%3==0)
 System.out.println("Osztható 3-mal");
 if (szam%5==0)
 System.out.println("Osztható 5-tel");
 }
}
```

*Megjegyzés:* Speciális szelekció még a kivételkezelés try-catch-finally szerkezete is. Erről a tankönyv 2. kötetében lesz szó.

**Tesztkérdések**

13.1. Mi igaz az alábbi strukturálthatlan kódrészletre? Jelölje meg az összes igaz állítást!

```
if (f1)
if (f2)
u1;
else
u2;
```

- a) Az első if egyágú szelekció.
- b) Az első if kétágú szelekció.
- c) A második if egyágú szelekció.
- d) A második if kétágú szelekció.

13.2. A forráskód strukturálásának szabályait szem előtt tartva mely állítások helyesek?

Jelölje meg az összes helyes állítást!

- a) A programot nem kötelező strukturálni, hiszen a fordító úgysem veszi figyelembe az eltolásokat.
- b) Ha a forráskódban egy utasítás beljebb kezdődik, mint a felette álló sor, az a program olvasója számára egy feltételes végrehajtást sugall.
- c) Ha két utasítás a forráskódban egymás alatt szerepel, akkor azok szekvenciálisan követik egymást.
- d) Kétágú szelekció esetén az if kulcsszó else párját ugyanabban az oszlopbán ( pontosan egymás alatt) kell kezdeni.

13.3. Mit ír ki a következő programrészlet? Jelölje meg az egyetlen jó választ!

```
int honap=5, nap=3, d=0;
if (honap>=4 && honap!=10) {
 d++;
 if (nap<2 | nap>5)
 d++;
 else if (nap>0)
 d++;
 else
 d++;
}
System.out.println(d);
```

- a) 1
- b) 2
- c) 3
- d) Semmit, mert a kód szintaktikailag hibás.

13.4. Jelölje meg az összes igaz állítást a következők közül!

- a) Az if utasítás feltétele tartalmazhat metódushívásokat.
- b) A switch utasítás fejében a kifejezés tartalmazhat metódushívásokat.
- c) Az if utasítás feltételeként bármilyen boolean típusú kifejezés állhat.
- d) A szintaktika megköveteli, hogy az if utasítás minden ágát blokkba foglaljuk.

13.5. Adott a következő switch utasítás:

```
int a=48;
switch (kifejezés) {
 case 1: utasítások; break;
 case 2: utasítások; break;
 ...
}
```

Mely kifejezés helyettesíthető be a switch utasítás fejébe? Jelölje meg az összes jó választ!

- a) 6
- b) (char)a
- c) Math.sin(a)
- d) a+3

## Feladatok

- 13.1. (A) Kérje be egy telek oldalait méterben! Írja ki a telek területét négyzetméternél (1 m<sup>2</sup>= 3.6m<sup>2</sup>). Ha a telek 100 négyzetméternél kisebb, akkor írja ki, hogy túl kicsi! (*Telek.java*)
- 13.2. (B) Van egy henger alakú hordónk, melybe nem tudjuk, hogy belefér-e a rendelkezésre álló bor. Kérje be a bor mennyiségét literben, majd a hordó összes szükséges adatát cm-ben. Adjon tájékoztatást, hogy hány literes a hordó, és hogy belefér-e a hordóba a bor! Ha belefér, akkor adja meg, hogy mennyi férne még bele! Írja ki százalékosan is a telítettséget! Az adatokat egészre kerekítve írja ki! (*Hordo.java*)
- 13.3. (A) Kérjen be egy évszámot! Ha a beütött szám negatív, adjon hibajelzést, ha nem, állapítsa meg, hogy az évszám osztható-e 17-tel, vagy nem! (*EvszamOszthato.java*)
- 13.4. Kérjen be egy karaktert!
  - a) (A) Írja ki, hogy a karakter nagybetű, kisbetű, szám vagy egyéb-e! minden esetben írja ki a karakter unikódját!
  - b) (C) Ha a karakter nagybetű, akkor írja ki a kisbetűs alakját, ha kisbetű, akkor a nagybetűs alakját! (*Karakter.java*)
- 13.5. (A) Kérje be a főnöke fizetését, aztán a sajátját. Hasonlítsa össze a két fizetést: Írjon ki egy-egy megjegyzést, ha a főnök fizetése a nagyobb, ha a sajátja a nagyobb, illetve ha egyenlő! (*FonokFizetese.java*)
- 13.6. (C) Kérje be Zsófi, Kati és Juli születési évét. Írja ki a neveket udvariassági sorrendben (előre az idősebbeket...)! (*Udvariassag.java*)
- 13.7. (B) Kérjen be egy jegyű, nem negatív számot! Írja ki a szám szöveges formáját (1 = egy, 2 = kettő stb.) (*SzamSzoveggel.java*)
- 13.8. (B) Kérjen be egy egész óra értéket. Ha a szám nem 0 és 24 óra között van, akkor adjon hibaüzenetet, egyébként köszönjön el a program a napszaknak megfelelően! 4-9: Jó reggelt!, 10-17: Jó napot!, 18-21: Jó estét!, 22-3: Jó éjszakát! (*Koszones.java*)
- 13.9. (B) Egy dolgozatra annak pontszámától függően a következő osztályzatot adják:

|           |     |   |       |
|-----------|-----|---|-------|
| elégtelen | (1) | : | 0-29  |
| elégséges | (2) | : | 30-37 |
| közepes   | (3) | : | 38-43 |
| jó        | (4) | : | 44-49 |
| jeles     | (5) | : | 50-55 |

Kérje be a dolgozat pontszámát, majd írja ki az osztályzatot számmal és betűvel! (*Dolgozat.java*)

## 14. Iterációk

---

A fejezet pontjai:

1. Elöltesztelő ciklus – while
  2. Hátultesztelő ciklus – do while
  3. Léptető ciklus – for
  4. Ciklusok egymásba ágyazása, kiugrás a ciklusból
  5. Adatok feldolgozása végig
  6. Megszámlálás
  7. Összegzés, átlagszámítás
  8. Minimum- és maximumkiválasztás
  9. Menükészítés
- 

A program legfontosabb tulajdonságai közé tartozik, hogy képes utasításokat ismételten végrehajtani. Az iteráció ismétlést jelent, ilyenkor egy vagy több utasítás újra és újra végrehajtódik. A Java három mechanizmust kínál az iterációk megvalósítására:

- ◆ **while:** Elöltesztelő ciklus, melynek a belépési feltételét fogalmazzuk meg.
- ◆ **do while:** Hátultesztelő ciklus, melynek szintén a belépési feltételét kell megadni.
- ◆ **for:** Léptető ciklus, melyben a ciklusváltozó minden egyes ciklus végrehajtásakor automatikusan lép egyet.

A feladatokat többféleképpen is meg lehet oldani. Vannak azonban ún. típusfeladatok, amelyek esetében érdemes az adott, sokak által elfogadott és tesztelt megvalósítási eljárást követni. A fejezet a Java által felkínált iterációs lehetőségeken kívül néhány típusfeladatot is bemutat.

### 14.1. Elöltesztelő ciklus – while

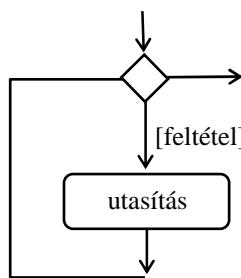
**Elöltesztelő ciklus** esetén a program még a ciklusba való belépés előtt megvizsgál egy feltételt – ezt belépési feltételnek nevezzük –, és ha ez teljesül, akkor a ciklusmag végrehajtásra kerül, egyébként nem. A ciklusmag ismétlten végrehajtódik, amíg a belépési feltétel teljesül. Ha már nem teljesül, akkor a program a ciklus utáni utasítással folytatódik. Az

elöltesztelő ciklust a Javában a `while` utasítás valósítja meg – tevékenységsdiagramját a 14.1. ábra mutatja; Java kódja a következő:

```
while (feltétel) {
 utasítás1;
 utasítás2;
 ...
 utasításN;
}
```

Szabályok:

- A `while` utasítás fejében a feltételt zárójelbe kell tenni.
- Ha a ciklusmagban több utasítás van, akkor azokat utasításblokkba zárjuk.



14.1. ábra. A `while` utasítás tevékenységsdiagramja

❖ Ha a feltétel után `;`-t teszünk, ott befejeződik a `while` utasítás, és a ciklusmag egy üres utasítás lesz! Például az `int i=1; while (i<5); i++;` végtelen ciklust eredményez.

❖ Ha nem teremtjük meg annak lehetőségét, hogy a vezérlés kikerülhessen a ciklusból, végtelen ciklust kapunk! Végtelen ciklus esetén előfordul, hogy a program „lemerevedik”, vagy elkezd „rángatózni”. Ilyenkor az operációs rendszer vagy a fejleszteri környezet által nyújtott lehetőségek valamelyikével meg kell szakítanunk a program futását (például leütjük a `Ctrl-C` billentyűkombinációt).

A `while` ciklus tipikusan olyan feladatok megoldására javasolható, amelyekben az induló feltételek határozzák meg a ciklusmag végrehajtásának szükségességét. Például:

**Feladat – Bank**

Van egy kis megtakarított pénzem. Arra vagyok kíváncsi, hány hónap múlva éri el ez az összeg a bankban a 100 000 Ft-ot, ha havi 2%-os kamatos kamattal számolhatok?

**A feladat elemzése**

A problémát úgy oldjuk meg, hogy az induló egyenleget addig szorozzuk a havi kamatos kamattal, amíg az összeg el nem éri a 100 000 Ft-ot. A hónapok számát tehát 0-ról indítjuk, és a ciklusmagban az egyenleg szorzásával egyidejűleg mindenüiszor 1-gyel növeljük az értékét. Ahányszor el kell végezni a szorzást, annyi hónapig kell bent tartani a bankban a pénzt. Előre természetesen nem tudhatjuk, hogy a ciklust hányszor kell végrehajtani, hiszen ha tudnánk, nem írnánk programot. A következő sarkalatos kérdés, hogy megengedhető-e az üres ciklus, vagyis létezhet-e olyan eset, mikor a ciklus egyetlenegyszer sem hajtódiák végre. Ilyen eset akkor lehetséges, ha az induló egyenleg eléri a 100 000 Ft-ot. Most mondhatná a kedves Olvasó, hogy ha az induló összeg ilyen nagy, akkor szintén nem írnánk programot. A program használója azonban beüthet bármekkora értéket, meg kell engednünk tehát az üres ciklust.

**Forráskód**

```
import extra.*;
public class Bank {
 public static void main (String args[]) {
 final int ALOMEGYENLEG = 100000;
 double egyenleg = Console.readInt("Mennyi pénzed van (Ft)?");
 short honap = 0;
 while (egenleg < ALOMEGYENLEG) {
 honap++;
 egyenleg *=1.02;
 }
 System.out.println("Tartsd bent "+honap+" hónapig!");
 System.out.println("Utána kivehetsz "+
 Format.right(egenleg,0,0)+" Ft-ot.");
 }
}
```

**A program egy lehetséges futása**

|                               |
|-------------------------------|
| Mennyi pénzed van (Ft)? 84000 |
| Tartsd bent 9 hónapig!        |
| Utána kivehetsz 100388 Ft-ot. |

**14.2. Hátultesztelő ciklus – do while**

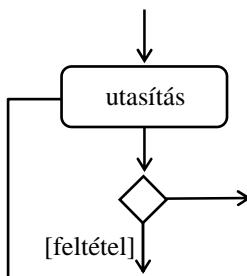
**Hátultesztelő ciklus** esetén a ciklus magja egyszer mindenkorban végrehajtódik, majd a ciklus végén, hátul következik egy feltételvizsgálat, amely eldönti, bent maradunk-e a ciklusban, vagy kilépünk. Ha a feltétel igaz, akkor a ciklusmag újból végrehajtódik. A

hátultesztelő ciklust a Javában a `do while` utasítás valósítja meg. Tevékenységsdiagramja a 14.2. ábrán látható; Java kódja a következő:

```
do {
 utasítás;
 while (feltétel);
 ...
 utasításN;
} while (feltétel);
```

Szabályok:

- A while utáni feltételt zárójelbe kell tenni.
- Ha a ciklusmagban több utasítás van, akkor azokat utasításblokkba zárjuk.



14.2. ábra. A `do while` utasítás tevékenységsdiagramja

A hátultesztelő ciklust nagy elővigyázattal kell alkalmazni, hiszen a program a ciklusba ész nélkül belemegy. Egy hasonlattal élve: az ember kétféleképpen mehet be egy helyiségbe. Vagy megáll az ajtó előtt, kopog, és csak akkor lép be, ha erre engedélyt kapott (elöltesztelő ciklus), vagy kérdezés nélkül benyit, és ott esetleg kiderülhet, hogy nincsen keresnivalója (hátultesztelő ciklus). Gondolkodás nélkül csak oda megyek be, ahol van keresnivalóm, és ezt biztosan tudom, hiszen előre átgondoltam.

Tipikusan hátultesztelő feladat, ha a program használójától egy elfogadható bevitelt várunk:

#### **Feladat – Számbekérés**

Kérjünk be terminálról egy számot. A számot csak akkor fogadjuk el, ha az egy naptári nap értéke, vagyis 1 és 31 közé esik! Írjuk ki minden esetben, ha a szám nem jó. (Feltételezzük, hogy a felhasználó számot üt be.)

### Forráskód

```
import extra.Console;
public class SzamBeker {
 public static void main (String args[]) {
 int szam;
 boolean ok;
 System.out.println("Szám (1..31)");
 do {
 szam = Console.readInt();
 // A szám vizsgálata:
 ok = szam>=1 && szam<=31; // a szám vizsgálata
 if (!ok)
 System.out.println("Nem jó, mégegyszer!");
 } while (!ok);
 System.out.println("Jó. Vége.");
 }
}
```

### A program egy lehetséges futása

|                     |
|---------------------|
| Szám (1..31)        |
| 221                 |
| Nem jó, mégegyszer! |
| 31                  |
| Jó. Vége.           |

### Feladat – Jegyek száma

Kérjünk be terminálról egy számot! Írjuk ki a szám jegyeinek a számát!

Osszuk el a számot 10-zel, az eredményt megint osszuk el 10-zel, egészen addig, amíg a szám nulla nem lesz. Közben számoljuk az osztások számát, ez lesz a szám jegyeinek a száma. Hogy az eredeti számot ne rontsuk el, az osztogatást egy segédváltozóban végezzük el!

### Forráskód

```
import extra.Console;
public class JegyekSzama {
 public static void main (String args[]) {
 int szam = Console.readInt("Szám: ");
 int seged = szam;
 int jegySzam = 0;
 do {
 seged /= 10;
 jegySzam++;
 } while (seged!=0);
 System.out.println(szam+ " jegyeinek száma: "+jegySzam);
 }
}
```

### A program egy lehetséges futása

|                           |
|---------------------------|
| Szám: 300998              |
| 300998 jegyeinek száma: 6 |

### 14.3. Léptető ciklus – for

A **léptető (növekményes, számláló) ciklusban** egy vagy több ciklusváltozó szerepelhet, melyek minden egyes ciklus végrehajtásakor automatikusan lépnek egyet. A `for` utasítás tevékenyséendiagramját a 14.3. ábra mutatja; Java kódja a következő:

```
for (inicializálás; feltétel; léptetés)
 utasítás;

for (inicializálás; feltétel; léptetés) {
 utasítás1;
 utasítás2;
 ...
 utasításN;
}
```

A `for` utasításnak van egy feje és egy blokkja. A fej zárójeles elemének három része van, melyeket pontosvesszők választanak el egymástól (Pontosan két darab). A részek a következők:

- **Inicializálás:** Megadható egy vagy több utasítás, vesszővel elválasztva. Az itt megadott utasítás(ok) az utasítás elején, egyszer kerül(nek) végrehajtásra. Itt lehet például ciklusváltozót deklarálni, és annak kezdeti értéket adni. A ciklusváltozó típusa tetszőleges, és az egész `for` utasítás alatt él (a fejben és a blokkban). Csak ugyanolyan típusú ciklusváltozók adhatók meg. Például:

```
int i=0, j=10
```

- **Feltétel** (belépési): Megadható egy feltétel (boolean típusú kifejezés). Ez a feltétel minden ciklus elején kiértékelődik, és általában a ciklusváltozót vizsgálja. Ha a feltétel teljesül, a ciklus újból végrehajtódik, egyébként a vezérlés a `for` utáni utasításra kerül. Összetett feltétel is megadható. A feltétel alapértelmezés szerinti értéke `true`, vagyis ha nem adunk meg feltételt, akkor a ciklus minden esetben újból végrehajtódik. Például:

```
i<10 && j>20
```

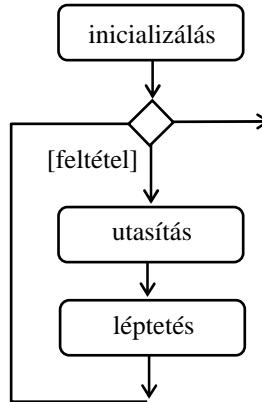
- **Léptetés:** Megadható egy vagy több utasítás, vesszővel elválasztva. Valamennyi léptető utasítás a ciklusmag végén minden alkalommal automatikusan végrehajtódik. A léptető utasítás „előrébb viszi” a ciklust. Általában a ciklusváltozót szokás itt megváltoztatni, elősegítve ezzel a valamikori kilépést a ciklusból. Például:

```
i++, j++
```

Egyéb szabályok:

- Ha a ciklusmagban több utasítás van, akkor azokat utasításblokkba zárjuk.
- A `for` fejének tagjai hagyhatók üresen is, de az elválasztó pontosvesszőket akkor is kell tenni!: `for (;;)...`

- A léptető ciklust egy előtesztelő ciklus valósítja meg. Ha a feltétel az első ciklus végrehajtása előtt nem igaz, akkor a ciklus egyszer sem hajtódi végre.



14.3. ábra. A for utasítás tevékenységsdiagramja

Növekményes ciklust leggyakrabban akkor alkalmazzuk, ha a ciklusok számát előre tudjuk. A ciklusok számát nem közvetlenül adjuk meg: a ciklus a ciklusváltozó egymás utáni értékeire hajtódi végre egy kezdeti értéktől valamilyen feltétel bekövetkeztéig.

- <sup>\*</sup> Ha a fej után `:t` teszünk, akkor ott befejeződik a `for` utasítás, vagyis a `for (....);` ciklusmagja üres. Az ezt követő utasítás a `for`-t szekvenciálisan követi!
- <sup>\*</sup> Vigyázat! Fönnáll a végtelen ciklus veszélye! Ha nem léptetjük a ciklusváltozót vagy a feltételt nem változtatjuk, lehet, hogy a ciklus sohasem fejeződik be!

### Feladat – Csillag

Írunk ki a konzolra 10 darab csillagot egy sorba!

### Forráskód

```

public class Csillag {
 public static void main(String[] args) {
 for (int i=0; i<10; i++)
 System.out.print("*");
 System.out.println();
 }
}

```

### A program elemzése

A kiírás az `i` ciklusváltozó 0, 1, 2, ... 9 értékeire hajtódik végre, vagyis összesen 10-szer. Ugyanezt a hatást értük volna el, ha a `for` utasítás ciklusváltozója 1-től 10-ig (vagy akár 33-tól 42-ig) halad, a ciklusmagban ugyanis nem használjuk `i` értékét:

```
for (int i=1; i<=10; i++) System.out.print("*");
for (int i=33; i<=42; i++) System.out.print("*");
```

• Fontos, hogy minden esetben olyan határokat adjunk meg, amelyek olvashatók, és utalnak a feladatra! 10 darab csillag kiírásához a 33-tól 42-ig haladó ciklus teljesen félrevezető!

Ha a feladat az lenne, hogy írjuk ki 1-től 10-ig a számokat, akkor már nem variálhatnánk a ciklusváltozó kezdeti értékét, hiszen azt a ciklusmag felhasználja:

```
for (int i=1; i<=10; i++)
 System.out.print(i+" ");
```

A ciklusváltozót nem szükséges a `for` utasítás fejében deklarálni. Elvileg megtehetjük ezt a metódus blokkjában másol, még a `for` utasítás előtt (ebben az esetben `i` a `for` utasítás után is használható):

```
int i;
for (i=1; i<=10; i++)
 System.out.print(i+" ");
```

### Feladat – Fizetés

Most 2001-et írunk. Írjuk ki, hogy mostantól 2006-ig melyik évben mennyi lesz József fizetése, ha az évenként 12%-kal növekszik! József fizetését konzolról kérjük be!

### Forráskód

```
import extra.Console;

public class Fizetes {
 public static void main (String args[]) {
 final int ELSOEV = 2001;
 final int UTOLSOEV = 2006;
 int fizetes = Console.readInt("Fizetés 2001-ben? ");
 for (int ev=ELSOEV+1; ev<=UTOLSOEV; ev++) {
 fizetes *= 1.12;
 System.out.println(ev+-ben: "+fizetes);
 }
 }
}
```

### A program futása

```
| Fizetés 2001-ben? 100000
| 2002-ben: 112000
| 2003-ben: 125440
| 2004-ben: 140492
| 2005-ben: 157351
| 2006-ben: 176233
```

A ciklusváltozó bármilyen típusú lehet, így karakter is:

### Feladat – Angol ábécé

Írjuk ki az angol **ábécé** nagybetűit a konzolra szóközökkel elválasztva!

#### Forráskód

```
public class AngolABC {
 public static void main (String args[]) {
 for (char betu='A'; betu<='Z'; betu++)
 System.out.print(betu+ " ");
 }
}
```

#### A program futása

```
| A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

### Feladat – Első öt

Írjuk ki 200-tól kezdve 17-esével az első 5 számot!

#### Forráskód

```
public class ElsoOt {
 public static void main (String args[]) {
 for (int i=0, szam=200; i<5; i++, szam+=17)
 System.out.print(szam+ " ");
 }
}
```

#### A program futása

```
| 200 217 234 251 268
```

### Feladat – Kis és nagy ábécé

Írjuk ki konzolra az angol **ábécé** nagybetűit és kisbetűit a következő formában:

aA bB cC dD ...zz

#### Forráskód

```
import extra.Console;
public class KisNagyABC {
 public static void main (String args[]) {
 // A feltételt elég az egyik határra megfogalmazni:
 for (char kis='a', nagy='A'; kis<='z'; kis++, nagy++)
 System.out.print(kis+" "+nagy+ " ");
 }
}
```

**A ciklusok szerkezete legyen tiszta, könnyen áttekinthető!** Ellenkező esetben könnyen elkövethetünk valamilyen szemantikai hibát!

## 14.4. Ciklusok egymásba ágyazása, kiugrás a ciklusból

A ciklusokat egymásba is ágyazhatjuk.

A következő feladat külső ciklusa hátultesztelő, mert az adott műveletet mindenkorban végre-hajtjuk, majd a felhasználóra bízzuk annak eldöntését, kéri-e még egyszer a műveletet:

### Feladat – Rendszám

Egy rendszámtábla betűkből (A..Z) és számokból (0..9) áll, bármelyik helyen bármi állhat. Az autók száma alapján tervezzük meg a rendszámtábla hosszúságát!

Kérjük be a számot, majd írjuk ki a rendszámtábla szükséges hosszát! Ezután kérdezzük meg a felhasználótól, akarja-e folytatni a játékot. Ha igen, újból kérjük be a számot, ha nem, fejezzük be a programot!

### A feladat analizálása

A külső ciklus egy olyan do while ciklus, amelynek végén minden felkínáljuk a folytatási lehetőséget. minden egyes ciklusban beolvassuk a kérdéses számot, melyhez ki kell számolnunk a szükséges jelek számát. Nézzük meg először, hogy adott darab jelből hány rendszámot tudunk kirakni: A rendszámtábla bármelyik helyére 36 jelből válogathatunk (26 betű és 10 szám). Ha a rendszámtáblára 2 jelet tehetünk, akkor az első helyen álló akármelyik jel mellé 36 jelből válogathatunk, tehát összesen  $36 \times 36$  különböző rendszámtábla készülhet. 3 jel esetén ez a szám  $36 \times 36 \times 36$ , és így tovább, az  $n$  jelből álló rendszámtáblák maximális száma  $36^n$ .

A megoldáshoz egy belső ciklus is szükséges, melyben folyamatosan képezzük 36 hatványait. Abban a pillanatban, hogy ez a hatványszám eléri vagy lehagyja a kérdéses számot, a ciklusból kiléünk – ahányszor végrehajtottuk a ciklust, annyi jelre van szükségünk a rendszámtáblák előállításához. A 0 esetet külön kezeljük, hiszen ebben az esetben nem is kell rendszámtábla; de ha legalább egy kell, akkor már 36 rendszámtáblánk is lehetne – ezért az ennyiLehet értékét 36-ról indítjuk.

### Forráskód

```
import extra.Console;
public class Rendszam {
 public static void main (String args[]) {
 char valasz;
 do {
 byte rendszamHossz;
 long ennyiKell=Console.readInt("Hány rendszámtábla kell? ");
 if (ennyiKell<=0)
 System.out.println("Nem kell rendszámtábla.");
 else {
 long ennyiLehet = 36;
 for (rendszamHossz=1;ennyiLehet<ennyiKell;rendszamHossz++)
 ennyiLehet *= 36;
 System.out.println(rendszamHossz+" hosszú lesz a tábla.");
 }
 valasz = Console.readChar("Folytatod? I/N ");
 } while ((valasz!='N') & (valasz!='n'));
 }
}
```

```

 } // main
} // Rendszam

```

### A program egy lehetséges futása

```

Hány rendszámtábla kell? 875000
4 hosszú lesz a tábla.
Folytatod? I/N i
Hány rendszámtábla kell? 1000000000
6 hosszú lesz a tábla.
Folytatod? I/N n

```

A következő példában for ciklusokat ágyazunk egymásba:

#### Feladat – Szorzótábla

Készítsük el a következő szorzótáblát:

Szorzótábla

|   |   |    |    |    |    |    |    |    |    |
|---|---|----|----|----|----|----|----|----|----|
| 1 | 1 | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  |
| 2 | 2 | 4  | 6  | 8  | 10 | 12 | 14 | 16 | 18 |
| 3 | 3 | 6  | 9  | 12 | 15 | 18 | 21 | 24 | 27 |
| 4 | 4 | 8  | 12 | 16 | 20 | 24 | 28 | 32 | 36 |
| 5 | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 |
| 6 | 6 | 12 | 18 | 24 | 30 | 36 | 42 | 48 | 54 |
| 7 | 7 | 14 | 21 | 28 | 35 | 42 | 49 | 56 | 63 |
| 8 | 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 |
| 9 | 9 | 18 | 27 | 36 | 45 | 54 | 63 | 72 | 81 |

#### A feladat analizálása

Először kiírjuk a fejlécet, majd a táblázatot. A külső ciklusban i értéke (az  $i*j$  bal oldali tényezője) fut 1-től 9-ig. A belső ciklusban fut j értéke (a jobb oldali tényező) úgy, hogy i minden egyes rögzített értékére végigfut 1-től 9-ig. Az egyes szorzatokat egymás után, igazítva írjuk ki, és a belső ciklus után a külső ciklus utolsó utasításaként sort emelünk.

#### Forráskód

```

import extra.Format;
public class SzorzoTabla {
 public static void main (String args[]) {
 System.out.println(" Szorzótábla");
 System.out.println("-----");
 for (int i=1; i<=9; i++) {
 // Egy sor írása:
 System.out.print(i+" | ");
 for (int j=1; j<=9; j++)
 System.out.print(Format.right(i*j,3)); // igazítunk
 System.out.println();
 }
 }
}

```

## A break és continue utasítások

A ciklusból való kiugrásra, illetve a ciklusmag átlépésére a Java két lehetőséget kínál:

**A break utasítás az aktuális utasításblokkból való azonnali kiugrást eredményezi.**

Hatására a vezérlés a blokk utáni utasításra kerül. Érvényes a while, a do, a for és a switch utasítások blokkjaiban. Például:

```
for (int i=0; i<10; i++) {
 for (int j=0; j<20; j++) {
 if (ok)
 break; // kiugrás a belső ciklusból
 ...
 }
 System.out.println("Vége a belső ciklusnak");
}
System.out.println("Vége a külső ciklusnak");
```

Megvan arra is a lehetőség, hogy több blokkból is kiugorunk egyszerre. Bármely utasítás megcímkézhető (<címke>:<utasítás>), és ez a címke a break utasítás után megadható. Ebben az esetben a vezérlés a címkézett utasításra kerül. Például:

```
for (int i=0; i<10; i++) {
 for (int j=0; j<20; j++) {
 if (ok)
 break tovabb; // kiugrás a tovább címkére
 ...
 }
 System.out.println("Vége a belső ciklusnak");
}
tovabb:
System.out.println("Vége a külső ciklusnak");
```

**A continue utasítás hatására a vezérlés az utasításblokk (ciklus) végére kerül.** A ciklus tehát tovább folytatódik, csak az aktuális ciklusmag (blokk) ettől kezdve nem kerül végre-hajtáshoz. Érvényes a while, a do, és a for utasítások blokkjaiban. Például:

```
for (int i=0; i<10; i++) {
 for (int j=0; j<20; j++) {
 if (ok)
 continue; // folytatás a ciklusváltozó következő értékével
 ...
 }
 System.out.println("Vége a belső ciklusnak");
}
System.out.println("Vége a külső ciklusnak");
```

A continue esetén is megadható címke.

**A ciklusból való kiugrást csak nagy elővigyázatossággal szabad alkalmazni!** Fontos, hogy a program továbbra is áttekinthető legyen!

*Megjegyzés:* A return az egész metódusblokkot elhagyja (lásd 15., Metódusok írása fejezet).

A break utasítást akkor szokás alkalmazni, ha egy bizonyos probléma a ciklus közben megoldódik, és már nincs értelme a további végrehajtásnak. Végső (nem optimális) esetben a ciklust egy adott feltétel bekövetkeztéig kell végrehajtani. Például:

#### Feladat – Első osztható

Két szám között határozzuk meg az első olyan számot, amelyik osztható egy megadott számmal!

A két szám között egyesével lépegetünk, és minden számról megvizsgáljuk, hogy osztója-e a megadott szám. Ha valamelyik szám ilyen, akkor elhagyjuk a ciklust, de előfordulhat, hogy a ciklus végére érünk úgy, hogy egyetlen ilyen számot sem találunk:

#### Forráskód

```
import extra.Console;
public class ElsoOszthato {
 public static void main (String args[]) {
 int kezd=5000, veg=7000, oszto=1797, szam;
 boolean van = false;
 for (szam=kezd; szam<=veg; szam++) {
 if (szam%oszto==0) {
 van = true;
 break;
 }
 if (van)
 System.out.println("Első "+oszto+"-vel osztható szám: "+szam);
 else
 System.out.println("Nincs ilyen");
 }
}
```

A continue utasítást akkor érdemes alkalmazni, ha a ciklus magját csak rendhagyó esetben nem kell végrehajtani. Ekkor a lényegi utasítássorozatot nem kell if blokkba tenni, és így a kód áttekinthetőbbé válik. Például:

#### Feladat – Szökőév

Írjuk ki a megadott két évszám közötti összes szökőévet, majd a szökőévek számát!

Egy szökőév (366 napos év, amelyben a február 29 napos) 4-gyel osztható, de: a 100-zal osztható évek nem szökőévek, csak a 400-zal oszthatók. A for ciklus tehát az első 4-gyel osztható évről indul, és négyesével megy (/1). A //2-ben megfogalmazott feltételnek eleget tevő évek nem szökőévek (ezek korrekciós évek), azokat egyszerűen átugorjuk.

### Forráskód

```

import extra.Console;

public class Szokoev {
 public static void main (String args[]) {
 int tolEv = 1979, igEv = 2020;
 int szokoSzam = 0;

 System.out.print("Szökőévek (366 naposak):");
 for (int ev=((tolEv-1)/4+1)*4; ev<igEv; ev+=4) { //1
 if (ev%100==0 && ev%400!=0) //2
 continue;
 // Ez egy szökőév:
 System.out.print(ev+" ");
 szokoSzam++;
 }
 System.out.println("\nSzökőévek száma: "+szokoSzam);
 }
}

```

## 14.5. Adatok feldolgozása végjelig

### Feladat – Kör kerülete

A felhasználó adott sugarú körök kerületére kívánCSI. Amikor ő beüt egy sugarat, mi kiírjuk a kör kerületét. Ha már nem kívánCSI több eredményre, akkor a kör sugaránál nullát (végjelet) kell ütnie

### A feladat analizálása

Első gondolatunk az lehetne, hogy a ciklusmagban beolvassuk a kör sugarát, majd kiírjuk a kör kerületét. Az utasításoknak ez a sorrendje azonban helytelen, mert így a végjelet is egy kör sugarának tekintenén. A végjelet sosem szabad feldolgozni, még akkor sem, ha ezzel történetesen nem működnék hibásan a program. Feladatunkban a felhasználó a végjelet azért üti be, mert nem kívánCSI több kerületre. A 0 sugarú kör kiírása tehát hibás és zavart keltő művelet. A végjelet hasznos konstansként felvenni, mert így könnyebb azt alkalmadtán kicserélni. Arra is ügyelni kell, hogy a végjel ne essék a feldolgozandó elemek közé. Ha például a feladatban pozitív számokat dolgozunk fel, akkor a végjel lehet 0. De ha napi középhőméréséket dolgozunk fel, akkor a nulla végjelként való alkalmazása nemcsak, hogy kizárná a nulla fok bevitelét adatként, hanem meg is szaktaná a bevitelt. Ha a sorozat elemei elvileg bármilyen számok lehetnek, akkor ki kell találni valamilyen abszurd számot, például egy adott típus legnagyobb vagy legkisebb értékét. Az ilyen végjel bevitelle persze rendkívül kényelmetlen. Tanulmányaink előrehaladtával a végjelig való feldolgozást egyre „felhasználóbarátibb” módon fogjuk majd megoldani (például Esc-et is lehet ütni).

**Az adatok végjelig való feldolgozására** többféle megoldás is kínálkozik. E megoldások egy dologban hasonlók: az adatbevitelt mindenkorban egy vizsgálatnak kell követnie, amely meghatározza, hogy a bevitt adat végjel-e vagy sem.

- Az első megoldásban a ciklus előtt és a ciklusmag végén kérünk be adatot úgy, hogy a beolvasást közvetlenül követi a while ciklus feltételének vizsgálata. Így a ciklusmagban csak érvényes adatot dolgozunk fel. Az adatokat előolvassuk a tényleges feldolgozáshoz.
- A második megoldásban a beolvasást egy huszárvágással betesszük a ciklus feltételébe, így elkerüljük a redundanciát, a kétszeri beolvasást.
- A harmadik megoldásban egy végtelen ciklusban olvassuk be és dolgozzuk fel az adatokat. Az adat bekérése után itt is vizsgálat következik, és ha a bekért adat végjel, akkor elhagyjuk a ciklust.

### Forráskód (három megoldás)

```
import extra.Console;

public class KorKerulet {
 public static void main (String args[]) {
 final int VEGJEL = 0;
 double sugar;
 long kerulet;

 // 1. megoldás:
 sugar = Console.readDouble("Sugár: ") ;
 while (sugar != VEGJEL) {
 kerulet = Math.round(sugar*2*Math.PI);
 System.out.println("Kerület: "+kerulet) ;
 sugar = Console.readDouble("Sugár: ") ;
 }

 // 2. megoldás:
 while ((sugar = Console.readDouble("Sugár: ")) != VEGJEL) {
 kerulet = Math.round(sugar*2*Math.PI);
 System.out.println("Kerület: "+kerulet) ;
 }

 // 3. megoldás:
 while (true) {
 sugar = Console.readDouble("Sugár: ") ;
 if (sugar == VEGJEL)
 break;
 kerulet = Math.round(sugar*2*Math.PI);
 System.out.println("Kerület: "+kerulet) ;
 }
 }
}
```

## 14.6. Megszámlálás

Ebben a feladattípusban **megszámoljuk** egy sorozat valamelyen adott tulajdonsággyal rendelkező elemeit. A számlálót ( $n$ -et) kezdetben nullára állítjuk, majd az adott tulajdon-ságú elem feldolgozásakor eggyel megnöveljük.

A legegyszerűbb eset, amikor minden elemet megszámolunk:

### Feladat – Megszámol

Kérjünk be számokat a felhasználótól 0 végig. Írjuk ki a bevitt számok számát!

#### Forráskód

```
import extra.Console;
public class Megszamol {
 public static void main (String args[]) {
 final int VEGJEL = 0;
 int n = 0;
 int szam;
 while ((szam = Console.readInt("Szám: ")) != VEGJEL) {
 n++;
 }
 System.out.println("A bevitt számok száma: "+n);
 }
}
```

Most nézzünk egy olyan esetet, amelyben válogatunk is, vagyis csak az adott tulajdonságú elemeket számoljuk:

### Feladat – Prímek

Adott két szám között hány darab prímszám van?

Egy szám prímszám, ha csak eggyel és önmagával osztható. Ha egyéb osztót is találunk, akkor a szám nem prím. A vizsgálatot természetesen elegendő a szám négyzetgyökéig elvégezni.

#### Forráskód

```
public class Primek {
 public static void main (String args[]) {
 int tol=2000, ig=2100;
 int primekSzama=0;
 for (int n=tol; n<=ig; n++) {
 boolean prim=true;
 for (int i=2; prim && (i<=Math.sqrt(n)); i++)
 prim = n%i!=0;
 if (prim) {
 System.out.print(n+" ");
 primekSzama++;
 }
 }
 }
}
```

```

 System.out.println("\nPrímek száma: " +primekSzama);
 }
}

```

### A program futása

|                                                                                              |
|----------------------------------------------------------------------------------------------|
| 2003 2011 2017 2027 2029 2039 2053 2063 2069 2081 2083 2087<br>2089 2099<br>Prímek száma: 14 |
|----------------------------------------------------------------------------------------------|

## 14.7. Összegzés, átlagszámítás

Az olyan feladatokat, melyekben a sorozat elemeit valamilyen módon gyűjteni kell, összegzéses feladatoknak nevezzük. Ebbe a feladatcsoportba sorolható a különbség-, illetve szorzatképzés is.

Az **átlag kiszámításakor** egyszerre két dolgot is végeünk – összegzünk, s közben számlálunk. Végül a két gyűjtött érték hányadosát kell képezni, de csak akkor, ha volt átlagolandó adat. Ellenkező esetben nullával osztanánk, és ez futási hibát eredményezne.

### Feladat – Átlag

Olvassunk be számokat nulla végig, majd írjuk ki ezek összegét, darabszámát és átlagát!

### Forráskód

```

import extra.*;
public class Atlag {
 public static void main (String args[]) {
 final int VEGJEL = 0;
 long osszeg = 0;
 int db = 0, szam;
 while ((szam = Console.readInt("Szám: ")) != VEGJEL) {
 db++;
 osszeg += szam;
 }
 if (db!=0) {
 System.out.println("Összeg= "+osszeg);
 System.out.println("Darab = "+db);
 System.out.println("Átlag = "+
 Format.right(osszeg*1.0/db,0,2));
 }
 else
 System.out.println("Nincs beolvasott szám!");
 }
}

```

### A program egy lehetséges futása

|                    |
|--------------------|
| Szám: 5<br>Szám: 9 |
|--------------------|

```

| Szám: 3
| Szám: 0
| Összeg= 17
| Darab = 3
| Átlag = 5.67

```

## 14.8. Minimum- és maximumkiválasztás

**Maximumkiválasztás** esetén a sorozat legnagyobb, **minimumkiválasztás** esetén a sorozat legkisebb elemét kell meghatároznunk.

Az algoritmus lényege, hogy a sorozat elemeit sorban megvizsgálva minden megjegyezzük az addigi maximális/minimális elemet. Ha egy annál nagyobb/kisebb elem érkezik, akkor a megjegyzett elemet ezzel kicseréljük. Induló értékként vagy az első elemet adjuk meg, vagy egy olyan szélsőértéket (nagyon kicsi/nagyon nagy értéket), amelyet az összes lehetőséges érkező elem biztosan „lehagy”. A feldolgozás végén – amennyiben volt a sorozatnak legalább egy eleme – a megjegyzett elem az összes szám maximuma/minimuma lesz.

Legkönnyebb dolgunk akkor van, ha előre tudjuk a sorozat elemeinek számát, és ez nem nulla. Ilyen a következő feladat:

### Feladat – Minimális és maximális kilométer

Kérjük be a hónap minden napjára, hogy autónkkal hány km-t tettünk meg aznap! Írjuk ki, melyik nap mentünk a legtöbbet, illetve a legkevesebbet, és mennyit! A hónap 31 napos. Két egyforma teljesítményű nap esetén az elsőt írjuk ki!

### Forráskód

```

import extra.Console;

public class MinMaxKm {
 public static void main (String args[]) {
 int km = Console.readInt(1+. napi km: ");
 int minKm=km, maxKm=km;
 int minNap=1; int maxNap=1;
 for (int nap=2; nap<=31; nap++) {
 km = Console.readInt(nap+. napi km: ");
 if (km < minKm) {
 minNap = nap;
 minKm = km;
 }
 else if (km > maxKm) {
 maxNap = nap;
 maxKm = km;
 }
 }
 }
}

```

```

 System.out.println("Legkevesebb a "+minNap+
 ". napon "+minKm+" km");
 System.out.println("Legtöbb a "+maxNap+
 ". napon "+maxKm+" km");
 }
}

```

A következő feladatban maximumot számítunk úgy, hogy nem tudjuk az érkező elemek számát, viszont tudjuk, hogy az érkező elemek csak pozitívak lehetnek. Ezért a végjel is lehet nulla, és a kezdeti maximumot is vehetjük nullának. Ha a végén a maximum még mindig nulla, az azt jelenti, hogy nem volt bevitel.

### Feladat – Maximális számla

Kérjünk be a felhasználótól számlaoszzegeket. A bevitel befejeződik, ha az összeg-nél nullát írunk. Írjuk ki a legnagyobb összegű számla sorszámát és összegét! Nulla összegű számla nem lehetséges.

### Forráskód

```

import extra.*;

public class MaxSzamla {
 public static void main (String args[]) {
 double maxOsszeg = 0, osszeg;
 int maxSorszam=0, sorszam=0;
 System.out.println("Összegek (vége=0)");

 while ((osszeg=Console.readDouble(" ? ")) != 0) {
 sorszam++;
 if (osszeg > maxOsszeg) {
 maxOsszeg = osszeg;
 maxSorszam = sorszam;
 }
 }
 if (maxOsszeg == 0)
 System.out.println("Nincs számla!");
 else {
 System.out.print("A maximális számla összege: "+
 Format.left(maxOsszeg,0,2));
 System.out.println(", sorszáma: "+maxSorszam);
 }
 }
}

```

### A program egy lehetséges futása

|                                               |
|-----------------------------------------------|
| Összegek (vége=0)                             |
| ? 5020                                        |
| ? 6400                                        |
| ? 700                                         |
| ? 0                                           |
| A maximális számla összege: 6400, sorszáma: 2 |

## 14.9. Menükészítés

A menükészítésnek most egy egyszerű változatát programozzuk be. Hogy hogyan lehet egy profi menüt készíteni, arról a következő kötetben lesz majd szó a grafikus felhasználói felület programozása kapcsán.

Amikor egy **menüt készítünk**, választási lehetőségeket kínálunk fel a felhasználónak, majd várjuk az erre való reagálását. A lehetőségeket ismételten felkínáljuk egészen addig, amíg a felhasználó úgy nem dönt, hogy nem akar több lehetőséget igénybe venni, vissza akar lépni a programnak egy előző szintjére, vagy be akarja fejezni a programot. Ha a felhasználó nem a kilépést választotta, akkor válogatni kell a programrészek között a felhasználó válaszának megfelelően. A menükészítés tipikusan háltaltesztelő ciklussal megoldandó feladat.

### Feladat – Menü

Készítsünk egy menüt! Három dologból lehet választani: egyik funkció, másik funkció és kilépés.

Jelenítsük meg a választási lehetőségeket: E(gyik) / M(ásik) / V(ége)?

Az Egyik, illetve a Másik funkció választására ismételten jelenjék meg egy-egy szöveget, a Vége választására pedig legyen vége a programnak.

### Forráskód

```
import extra.Console;
public class Menu {
 public static void main (String args[]) {
 char valasz;

 do {
 // Választási lehetőségek kiírása:
 System.out.print("E(gyik) / M(ásik) / V(ége)? ");

 // A felhasználó választ:
 valasz = Console.readChar();
 // Nagybetűre alakítás:
 valasz = Character.toUpperCase(valasz);

 // A kiválasztott funkció végrehajtása:
 switch (valasz) {
 case 'E':
 System.out.println("Egyik funkció végrehajtása\n");
 break;
 case 'M':
 System.out.println("Másik funkció végrehajtása\n");
 break;
 }
 } while (valasz != 'V');
 }
}
```

### Futás

```

E(gyik) / M(ásik) / V(ége)? e
Egyik funkció végrehajtása

E(gyik) / M(ásik) / V(ége)? m
Másik funkció végrehajtása

E(gyik) / M(ásik) / V(ége)? e
Egyik funkció végrehajtása

E(gyik) / M(ásik) / V(ége)? v

```

### Tesztkérdések

- 14.1. Jelölje meg az összes igaz állítást a következők közül!
- A while ciklusban a feltétel teljesülése esetén végrehajtódik a ciklusmag.
  - A do while ciklusban a ciklusmag után egy kilépési feltétel van.
  - A do és while kulcsszavak közötti egyetlen utasítást nem kell blokkba tenni.
  - A while ciklus feltételét zárójelbe kell tenni.
- 14.2. Jelölje meg az összes igaz állítást a következők közül!
- A while utasítás elől tesztel, és egyszer mindenkorban végrehajtódik.
  - A do while utasítás hátul tesztel, és egyszer mindenkorban végrehajtódik.
  - A for utasítás fejében kötelező ciklusváltozót deklarálni.
  - A for utasítás fejében a középső részben két feltétel is megadható, vesszővel elválasztva.
- 14.3. Melyik for ciklus feje helyes szintaktikailag? Jelölje meg az összes jó választ!
- `for(int i=0;i==8;++i)` utasítás;
  - `for(int i=0;false;i++)` utasítás;
  - `for (int i=0;i++)` utasítás;
  - `for(;;)` utasítás;
- 14.4. Mit ír ki a következő kódrészlet? Jelölje meg az egyetlen jó választ!
- ```

for (int k=0;k<=6;k++) {
    int s=k%5;
    s*=2;
}
System.out.print(s+" ");

```
- 0 2 4 6 8 0 2
 - Fordítási hiba, mert a kiíró utasításban az s ismeretlen.
 - 2
 - 0 1 2 3 4 0 1

- 14.5. Melyik ciklus hajtódik végre legalább egyszer a következő deklaráció mellett? Jelölje meg az összes jó választ!

int a=10; b=12;

- a) while ($a \geq 0 \ \& \ b \neq 0$) { ... }
- b) do {
...
} while ($a == b \ | \ b < 0$);
- c) for (int i=5; i<0; i--) { ... }
- d) while ($a < 0 \ | \ (a+b) \% 5 == 2$) { ... }

- 14.6. Mi igaz a következő kódrészletre? Jelölje meg az összes igaz állítást!

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

- a) A kódrészlet szintaktikailag hibás, mert a külső ciklusváltozó sohasem érheti el a 10-et.
- b) A kódrészlet szintaktikailag hibás, mert az i változó kétszer van deklarálva.
- c) A kódrészlet szintaktikailag hibás, mert a második for ciklusban nincs feltétel.
- d) A kódrészlet szintaktikailag helyes.

- 14.7. Jelölje meg az összes igaz állítást a következők közül!

- a) A break utasítás hatására a vezérlés az adott utasításblokk végére kerül.
- b) A break utasítás hatására a vezérlés a break-et tartalmazó összes ciklust elhagyja.
- c) A continue hatására a vezérlés az adott utasításblokk végére kerül.
- d) A continue utasítás a ciklusváltozót automatikusan egygyel megnöveli.

- 14.8. Jelölje meg az összes igaz állítást a következők közül!

- a) Végjelig való feldolgozás esetén a végjel feldolgozása a sorozat elemeként logikai hibát eredményez.
- b) A megszámlálás algoritmusában egy sorozat valamelyen adott tulajdonságú elemeit számoljuk meg.
- c) A minimumszámítás algoritmusában, ha a sorozat elemeit minden egy addigi minimumhoz hasonlítjuk, akkor vagy az első számáról, vagy egy olyan kicsi számáról kell indulnunk, amilyen biztosan nincs a sorozatban.
- d) A maximumszámítás algoritmusában, ha a sorozat elemeit minden egy addigi maximumhoz hasonlítjuk, akkor vagy az első számáról, vagy egy olyan kicsi számáról kell indulnunk, amilyen biztosan nincs a sorozatban.

Feladatok

- 14.1. (A) Szüretkor sorban lemérík a puttonyokban levő szőlő súlyát. Készítsünk programot, mely segítségével ezek az értékek feldolgozhatók! Kíváncaik vagyunk a nap végén, hogy összesen hány puttonnyal és hány kg szőlőt szüreltek. Természetesen a puttonyok számát előre nem tudjuk. (*Szuret1.java*)

- 14.2. **(B)** Egészítsük ki az előző feladatot: A szőlőt 1000 kg teherbírású gépkocsikkal szállítják el a helyszínről. minden esetben, amikor a szőlő mennyisége elérte (meghaladta) az 1000 kg-ot, akkor a program figyelmezzen: Mehét a kocsi! (*Szuret2.java*)
- 14.3. **(A)** Munkásokat veszünk fel egy adott munkára. A szükséges létszámot a program elején megkérdezzük. Az emberek csoportosan jelentkeznek; üssük be sorban a jelentkező csoportok létszámait! Ha megvan a szükséges létszám, akkor írjuk ki, hogy „A létszám betelt!”, valamint azt, hogy hány főre van szükség az utolsó csoportból! (*MunkasFelvetel.java*)
- 14.4. **(A)** Kérjük be, hogy a héten mennyi kalóriát fogyasztottunk az egyes napokon! Ezután írjuk ki az összes kalóriafogyasztásunkat, valamint a napi átlag kalóriafogyasztást! (*Kaloria.java*)
- 14.5. **(A)** Kérjen be egy egész számot, majd írja ki 0-tól n-ig (n-et is beleértve)
a) az összes egész számot!
b) az összes páros számot!
c) az összes hárommal osztható számot!
(*SzamSor.java*)
- 14.6. **(A)** Készítsen egy díszítősort! A díszítősor egy adott karaktersorozat ismétlése. Kérje be a felhasználótól a karaktersorozatot és az ismétlések számát, majd jelenítse meg a díszítősort a konzolon! A díszítősor több soros is lehet. (*Diszitosor.java*)
- 14.7. **(B)** Kérje be n értékét, majd készítse el a következő n soros háromszöget (*Haromszog.java*):
1
1 2
1 2 3
⋮ ⋮ ⋮
1 2 3 . . . n
- 14.8. **(B)** Kérjen be egy egész számot, és állapítsa meg, hogy hány 0 jegy szerepel benne! (*NullaJegyek.java*)
- 14.9. Hány darab négyzetszám van ezerig? Írja is ki a négyzetszámokat
a) **(B)** sorban, egymás után!
b) **(C)** úgy, hogy egy sorban csak öt szám szerepeljen, és a számok 6 hosszon legyenek kiírva, jobbra igazítva! (*NegyzetSzamok.java*)
- 14.10. Kérjen be két karaktert, majd írja ki az összes olyan karaktert, amely az unikód táblában a megadott karakterek közé esik!
a) **(A)** Tegyük fel, hogy előbb az unikód táblában előrébb szerepelő karaktert adják meg.
b) **(B)** Figyeljünk a következőre: a felhasználó nem ismeri az unikód táblát, ezért nem biztos, hogy először a kisebb karaktert üti be!
(*KettoKozott.java*)

- 14.11. (B) Kérdezze meg a felhasználótól, hogy mennyi pénze van, és milyen évi kamatozással tette azt be a bankba! Ezután a program ismételten tegye lehetővé, hogy az illető többször megkérdezhesse, melyik évben mennyi pénze lesz? (*MennyiPenz.java*)
- 14.12. (A) Kérjünk be terminálról karaktereket ' - ' végjelig! Végül írjuk ki, hány karaktert ütöttek be A és Z, a és z, valamint 0 és 9 között! Írjuk ki azt is, hány ezektől eltérő karaktert ütöttek! (*KarSzamol.java*)
- 14.13. (B) Van egy bizonyos összegünk, amelyből számlákat szeretnénk kifizetni, de maximum tízet. Kérje be a rendelkezésre álló összeget, majd sorban a számlaoösszegeket! Ha a számlák száma eléri a tízet, vagy az adott számlára már nincs pénz, adjon a program egy figyelmeztetést! Végül írja ki, hány számlát sikerült kifizetni, és mennyi a ténylegesen kifizetendő összeg! (*SzamlaKifizetes.java*)
- 14.14. (C) Kérjen be egy határszámot, majd írja ki eddig a számig az összes prímszámot! (*Primek.java*)
- 14.15. (C) Sheran hindu király meg akarta jutalmazni a sakkjáték feltalálóját, és rábítta, hogy Ő maga válassza meg a jutalmát. A bölcs ember így válaszolt: A sakktábla első mezőjéért csak egy búzaszemet adj uram! A másodikért 2-t, a harmadikért 4-et stb., minden mezőért kétszer annyit, mint az előzőért. Számoljunk utána, mennyi is volt pontosan a jutalom?
- A sakktábla vízszintes koordinátái: A és H, függőleges koordinátái pedig: 1 és 8 közötti értékek. A tábla (A,1) mezőjére egy darab búzaszem helyezünk, a (B,1) helyre ennek kétszeresét. A sakktábla összes mezőjére sorfoltonosan mindenig az előző mezőre tett búzamennyiség kétszeresét tesszük.
- Melyik mezőn lesz már legalább 1 millió búzaszem? Írja ki az oszlopot és sort is (pl. E3 mezőn 1048576 szem búza)!
 - Hány búzaszem lesz az egyes mezőkön? Ciklusban kérjük be a felhasználótól, milyen koordinátára kíváncsi, és minden esetben írjuk ki az eredményt! A programnak akkor legyen vége, ha a beviteli vízszintes koordinátánál a * karaktert ütik be!
- (*Buzaszemek.java*)
- 14.16. (A) Kérdezzük meg a felhasználótól, hogy minek a területét óhajtja kiszámítani: téglalapét, körét vagy körcikkét? Kérjük be a szükséges adatokat és írjuk ki az eredményt! Ezután újra kínáljuk fel a lehetőségeket. Tegyük lehetővé azt is, hogy a felhasználó kiszálljon a programból! (*Terulet.java*)
- 14.17. Kérjük be, hogy a héten mennyi kalóriát fogyasztottunk az egyes napokon! Ezután írjuk ki, hogy hányszádik napon fogyasztottuk a legtöbb, illetve a legkevesebb kalóriát!
- (A) Ha több ilyen nap van, akkor az első ilyent írjuk ki! (*LegKaloria1.java*)
 - (B) Ha több ilyen nap van, akkor írjuk ki az összeset! Tipp: A maximális és minimális értékeket egy karakterláncban gyűjtsük! (*LegKaloria2.java*)

14.18. **(B)** Kérje be egymás után különböző henger alakú hordók adatait (átmérő, magasság)!

Ha a hordó első adatánál 0-t ütnek be, akkor vége a bevitelnek. Írja ki, hányadik hordóba fér bele a legkevesebb, illetve a legtöbb bor! Írja ki ezen hordók adatait is! (*MinMaxHenger.java*)

14.19. **(C)** Készítsen egy pénzbedobó automatát! Először kérje be a bedobandó összeget. Az automatába csak 10, 20, 50 és 100 Ft-ost lehet bedobni, ha más pénzt dobnak bele, azt visszaadja. Addig kérje az automata a pénzt, amíg a bedobott összeg el nem éri a kívánt összeget. Közben folyamatosan tájékoztassuk a bedobót, hogy mennyit kell még bedobnia! Végül, ha a bedobó több pénzt dobott be, akkor a visszajáró összeget az automata visszaadja. (*Automata.java*)

15. Metódusok írása

A fejezet pontjai:

1. A metódus fogalma, szintaktikája
 2. Paraméterátadás
 3. Visszatérés a metódusból
 4. Metódusok túlterhelése
 5. Lokális változók
 6. Néhány példa
 7. Hogyan tervezzük meg metódusainkat?
-

Metódusok írásával az objektum, illetve osztály feladatait részekre bonthatjuk, azokat külön-külön megnevezhetővé tehetjük. Amikor egy objektumnak (vagy osztálynak) üzenetet küldünk, akkor egy olyan metódus kerül végrehajtásra, amely az üzenetnek egyértelműen megfeleltethető. Az osztályban deklarált metódusok egymást is hívhatják. A fejezet vezeti az Olvasót a metódusírás rejtelmeibe.

15.1. A metódus fogalma, szintaktikája

A metódusírás alapvető felépítését és szintaktikáját egy mintafeladaton keresztül mutatjuk be:

Feladat – Metódusminta

Kérjünk be számokat a konzolról 0 végig! minden egyes számról állapítsuk meg, hogy az hányszám jegyű! A program elején és végén, valamint az egyes eredmények kiírása után húzzunk egy-egy 20 hosszúságú vonalat!

A nagyvonalú megoldás terve (pszeudokód)

```
adatok: szám, jegyek száma
vonalthúzás
in: szám
while szám != 0
    jegyek számának kiszámítása és kiírása
    vonalthúzás
    in: szám
end while
vonalthúzás
```

E durva elképzelésnek most meg kell írnunk a Java kódját. A program úgy is „működőképes” lenne, ha az összes utasítást bepréselnénk a main metódusba. Ez azonban a programot teljesen áttekinthetetlenné tenné, hiszen

- ◆ a vonal húzását háromszor kellene kódolnunk, és ha a vonalhúzó algoritmuson változtatni szeretnénk, azt három helyen kellene átvezetnünk;
- ◆ a feladatok részekre bontása az áttekinthetőség elengedhetetlen feltétele. Gondoljuk meg, hogy a Java kód a tervnek többszöröse lesz.

A metódus utasítások (tevékenységek) összessége, melyet meghívhatunk a metódus nevére való hivatkozással.

Írunk egy-egy metódust a vonal húzására és a jegyek számának kiszámítására; ezeket aztán a megfelelő helyekről meghívhatjuk. Nézzük a teljes programot:

Forráskód

```
import extra.*;

public class MetodusMinta { //1

    public static void main(String[] args) { //2
        vonalhuz(); //3
        int szam; //4

        while ((szam=Console.readInt("Szám: ")) != 0) { //5
            System.out.println("Jegyek száma: "+ //6
                jegyekSzama(szam)); //7
            vonalhuz(); //8
        } //9
        vonalhuz(); //10
    }

    static void vonalhuz() { //11
        for (int i=1; i<=20; i++) //12
            System.out.print('-'); //13
        System.out.println(); //14
    } //15

    static int jegyekSzama(int n) { //16
        int result = 0; //17
        do { //18
            n /= 10; //19
            result++; //20
        } while (n !=0); //21
        return result; //22
    } //23
} //24
```

A program egy lehetséges futása

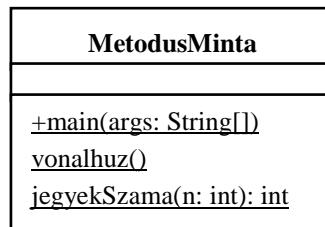
```

-----
Szám: 3
Jegyek száma: 1
-----
Szám: 6289
Jegyek száma: 4
-----
Szám: 0
-----
```

A program elemzése

A program egyetlen osztálya a MetodusMinta; az osztály UML diagramja a 15.1. ábrán látható. Az osztály három metódust tartalmaz: main, vonalhuz és jegyekSzama. A main az osztály kötelező formájú metódusa, amely minden Java program belépési pontja. A program futása során a Java futtató rendszer a main utasításait sorban végrehajtja. Általában, amikor a vezérlés egy metódushíváshoz ér, a rendszer megjegyzí, hol tartott, „elszalad” végrehajtani a meghívott metódust, majd visszatér oda, ahonnan a hívás történt. Programunkban a main a következő metódusokat hívja meg:

- ◆ a vonalhuz metódus eljárás, amelynek feladata a 20 karakter hosszúságú vonal meg-húzása. Az eljárás kódja a //11 és //15 sorok között van, és azt a //3, //7 és //9 sorokból hívjuk meg.
- ◆ a jegyekSzama metódus függvény, amely azért felelős, hogy kiszámítsa egy adott szám jegyeinek a számát. A függvény kódja a //16 és //23 sorok között van, és azt a //6 sorból hívjuk meg.



15.1. ábra. MetodusMinta osztály

Az osztály metódusainak deklarálási sorrendje tetszőleges, de szem előtt kell tartani az osztály áttekinthetőségét. A main metódust mindenkorábban az osztály első vagy utolsó metódusaként célszerű megadni.

Egy Java program megírása minden esetben osztályok fejlesztésével történik. Programírás közben hivatkozhatunk akár az API osztályaiban deklarált metódusokra, akár a saját osztályainkban megírtakra. Hivatkozáskor vagy egy objektumot, vagy egy osztályt szólítunk meg. Egy metódust az öt tartalmazó csomag(ok), azon belül osztály(ok) neveivel **minősítünk** – például a `java.lang.Math.random()` a `java` csomag `lang` alcsomagjában levő `Math` osztály `random()` metódusát azonosítja. A minősítés nélküli metódushívás annak az osztálynak a metódusát hívja meg, amelyből a hívás történt. A `MetodusMinta` osztályban például felesleges a `vonalhuz();` helyett a `MetodusMinta.vonalhuz();` minősített nevet használnunk, hiszen a hívás a `MetodusMinta` egy másik metódusából történt.

Eljárás, függvény

Egy metódust attól függően, hogy van-e visszatérési értéke, vagy nincs, eljárásnak vagy függvénynek szokás nevezni:

- Az **eljárás** utasítások összessége, melyet egyszerűen végrehajtunk az eljárás nevére való hivatkozással (metódushívó utasítással). Végrehajtás után a program azzal az utasítással folytatódik, amelyik követi a metódushívó utasítást. Az eljárás visszatérési típusa `void` (`void=semleges`), ami azt jelenti, hogy a visszatérési típus nincs definiálva. Például:

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

- A **függvény** is utasítások összessége, melyet a nevére való hivatkozással hajtunk végre, de egy lényeges dologban eltér az eljárástól: a függvény visszaad egy értéket, melyet a függvény neve képvisel. A visszatérési érték típusa a függvény visszatérési típusa. A függvény értékét olyan kifejezésekben használhatjuk, ahol ez a visszatérési típus meg van engedve. Például:

```
double d=Math.cos(2);
int jegySzam=jegyekSzama(65);
```

Az eljárás és a függvény deklarációja a Javában formailag azonos, és mindenkorral metódusnak nevezzük.

Az eljárás visszatérési értéke semleges, és nem hívható meg függvényként. A **függvény** ezzel szemben **eljárásként is meghívható**, ekkor a visszatérési érték a „szemébe” kerül. A függvény eljárásként történő hívásának természetesen csak akkor van értelme, ha a függvénynek más feladata is van a visszatérési érték meghatározásán kívül. A `Math.sin(0);` utasítás például szintaktikailag helyes, de semmi értelme sincs.

Fontos tudni: a fordító nem engedi meg, hogy statikus metódusból (osztálymetódusból) példánymetódust hívunk, hiszen a statikus metódus futása nem követeli meg a példány létezését, a példánymetódus ezzel szemben csak példányon tud dolgozni. **Statikus metódusból kizárolag az osztály statikus elemeire (adataira és metódusaira) hivatkozhatunk!**

A metódus általános szintakszisa

A metódus általános szintakszisa és az egyes részek jelentése a következő:

```
[<módosítók>] <visszatérési típus> <metódus neve>
    ( [<formális paraméterlista> ] [ <throws ...> ] )
{
    <metódusblokk>
}
```

Metódusfej

A nagy zárójelek közötti részeket nem kötelező megadni.

Példák metódusfejek megadására

| Módosítók | Visszat. típus | Név | Paraméterek |
|---------------|----------------|---------|-----------------------------|
| public | void | print | (float f) |
| public static | int | min | (int a, int b) |
| | void | vonahuz | () |
| private | int | minChar | (char c1, char c2, char c3) |

A példákból jól látható, hogy módosítókat nem feltétlenül kell adni. Visszatérési típusa és neve azonban mindegyik metódusnak van. Kötelező továbbá kiírni a formális paraméterlista nyitó és csukó zárójelét, mert ez azt jelzi, hogy metódusról és nem változódéklárációról van szó.

Módosítók

Módosítót nem kötelező adni, és a sorrend elvileg kötetlen. A metódus módosítói a következők lehetnek:

- ◆ public, protected vagy private: a **metódus láthatósága**. Legfeljebb egy adható meg. Ha nem adjuk meg, akkor alapértelmezés szerint a deklaráció kizárolag a saját csomagjából látható (csomagszintű láthatóság).
- ◆ abstract: **absztrakt metódus**, vagyis üres, és azt egy utód osztályban kell majd kifejteni.
- ◆ final: **véleges metódus**, nem lehet az utód osztályokban módosítani. Ha nem adjuk meg, akkor a metódus elvileg később felülírható. (Az abstract és final módosítók nem adhatók meg egyszerre.)
- ◆ static: **statikus, vagyis osztálymetódus**. Ha nem adjuk meg, akkor a metódus példánymetódus.
- ◆ Egyéb módosítók: native, synchronized

A felsorolt módosítók közül egyelőre csak a `public` és `static` módosítókat alkalmazzuk:

- `public`: Publikus láthatóságú metódus. Ezt a módosítót pillanatnyilag csak a `main` metódus elő fontos kiírnunk, hiszen a fő osztályt nem akarjuk és nem is tudjuk kívülről más metódussal megszólítani.
- `static`: Statikus, vagyis osztálymetódus. Egy metódus alapértelmezés szerint példánymetódus (elé tehát nem írjuk ki a `static` kulcsszót).

Megjegyzés: Ebben a fejezetben „kényszerűségből” mindegyik metódusunk statikus lesz, mert még nem tudunk osztályt deklarálni és példányosítani.

Visszatérési típus

Minden metódusnak kötelezően definiálni kell a visszatérési típusát, illetve közölni kell azt is, ha nincsen. Az eljárásnak nincsen visszatérési típusa, erre utal a `void` (semleges) kulcsszó. Függvény esetében a visszatérési típus a függvény által visszaadott érték típusa, amely lehet:

- ◆ bármelyik primitív típus vagy
- ◆ egy referencia (objektum) típus.

A metódus neve

Minden metódusnak kötelezően meg kell adni a nevét, erre az azonosító képzési szabályai érvényesek. A metódus szignatúrájának az osztályban egyedinek kell lennie (név+a paraméterek típusa).

Formális paraméterlista

A formális paraméterlistában adjuk meg az esetleges formális paraméterek nevét és típusát. Ha nincs a metódusnak paramétere, a nyitó és csukó zárójeleket akkor is kötelező megadni, mert a Java így különbözteti meg a metódusokat az egyéb deklarációktól. A vonalhuz metódusnak nincsen paramétere, míg a `jegyekSzama` metódusnak egyetlen, `int` típusú paramétere van, az n. Több paraméter esetén a paramétereket vesszővel választjuk el egymástól, és minden egyes paraméternek külön-külön megadjuk a típusát. Például: `min(int a, int b)`.

throws ...

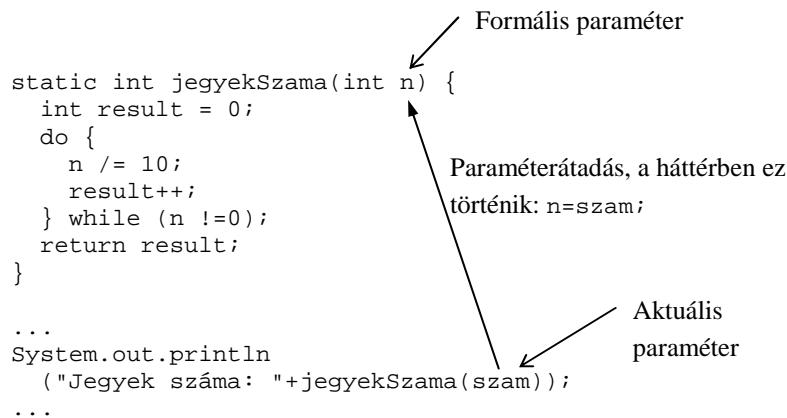
Nem kötelező megadni. E kulcsszó után kivétel- (exception) típusokat sorolhatunk fel: az ilyen kivételekkel a metódus nem kíván foglalkozni, azokat továbbítja az őt hívó eljárásnak. A kivételezelésről a könyv 2. kötetében lesz szó.

Metódusblokk

A metódus blokkja utasításokat tartalmaz. A metódus blokkját nyitó és csukó kapcsos zárójelek közé írjuk. A metódus blokkjában nem lehet újabb metódust deklarálni.

15.2. Paraméterátadás

A metódusoknak paramétereik adhatók át. A `jegyekSzama(szam)` függvényt egyetlen paraméterrel hívjuk meg. A metódus mindenkor az aktuális paraméter értékétől függően „fut le”. Nézzük meg, hogyan kell felkészítenünk egy metódust a paraméterek fogadására!



15.2. ábra. Paraméterátadás

Formális és aktuális paraméterek

A paraméterátadás technikáját a 15.2. ábra mutatja. A `jegyekSzama` metódus a futáskor kapott `szam` aktuális értékéből számítja ki a jegyek számát.

A metódus fejében deklarált paramétert **formális paraméternek** nevezzük, hiszen ez egészben addig csak formális, míg az eljárást meg nem hívjuk valamilyen **aktuális paraméterrel**. A formális paramétert ugyanúgy kell deklarálni, mint egy változót – meg kell adni nevét és típusát. Egy metódusnak akárhány (nulla, egy vagy több) formális paramétere lehet. A **formális paraméterlista formális paraméterdeklarációk vesszővel elválasztott sorozata**.

Az aktuális paraméter bármilyen kifejezés lehet, amely a formális paraméterrel értékkadás szerint kompatibilis. Ez azt jelenti, hogy ha leírhatjuk a `formPar=kifejezés; értékkadó` utasítást, akkor a `kifejezés` paraméterként is átadható `formPar`-nak.

Példánkban leírhatjuk az `n=szam;` értékkadó utasítást, tehát a paraméterátadás szintaktikailag helyes. Az aktuális paraméter tetszőleges kifejezés lehet. Az `int` típusú `n` formális paraméternek például a következő kifejezések is átadhatók aktuális paraméterként:

```
jegyekSzama(szam+20)
jegyekSzama(102) // egy operandusú kifejezés
jegyekSzama(szam+(int)Math.round(Math.random()*10))
```

Változtassuk meg most a MetodusMinta feladatot úgy, hogy a program elején és végén 50 hosszúságú vonalat húzunk, közben pedig változatlanul 20 hosszúságút. Ezt úgy tudjuk legszebbet megoldani, ha a vonalhuz eljárásnak egy hossz paramétert definiálunk. A következő metódus minden hosszúságú vonalat húzza:

```
static void vonalhuz (int hossz) {
    for (int i=1; i<=hossz; i++)
        System.out.print('-');
    System.out.println(); 20 helyett a hossz most minden más!
}
```

A metódust most a programból kétféleképpen fogjuk meghívni:

```
vonahuz(50);
vonahuz(20);
```

Egy olyan vonalat, melynek hosszúsága 1 és 80 közötti véletlen érték, így húzhatunk:

```
vonahuz((int)(Math.random()*80)+1);
```

Egy eljárásnak nemcsak egy paramétere lehet, hanem akárhány. A következő metódus egy adott hosszúságú vonalat húz, adott karakterrel:

```
static void vonalhuz(int hossz, char ch) {
    for (int i=1; i<=hossz; i++)
        System.out.print(ch);
    System.out.println(); Már két változó függ az aktuális értéktől!
}
```

A következő hívás hatására a ***** vonal jelenik meg a konzolon.

```
vonahuz(5, '*');
```

Megjegyzés: A Javában csak érték szerinti paraméterátadás van. Ez azt jelenti, hogy az aktuális paraméter értékét a metódus felhasználja, de nem változtathatja meg. Aktuális paraméterként átadható egy bármilyen, értékkedás szerint kompatibilis kifejezés értéke, de ha történetesen egy pusztá változót adunk át, azt a metódus nem fogja és nem is tudja megváltoztatni. Objektum referencia átadása esetén az objektum megváltoztatható ugyan, de maga a referencia nem.

Más programozási nyelvekből ismert a cím szerinti paraméterátadás: ilyen esetben aktuális paraméterként egy címmel azonosított változót adunk át, melyet a metódus megváltoztathat. Javában nem írható meg például a szoroz(a, 2) eljárás, amely az a értékét megkétszerezi.

15.3. Visszatérés a metódusból

Függvény esetén kötelezően meg kell adnunk a visszatérési értéket a `return` (visszatérés) utasítással, még a blokkból való kilépés előtt:

```
return kifejezés;
```

A `return` hatására a vezérlés azonnal a blokk végére kerül, és a metódus futása befejeződik. A `return` után kötelező megadni egy kifejezést, és annak értékadás szerint kompatibilisnek kell lennie a függvény visszatérési típusával. A függvényhívás a kifejezés aktuális értékét fogja felvenni. A 15.3. ábrán látható, hogy a `jegyekSzama` függvény értéke a lokálisan kiszámított `result` lesz.

```
static int jegyekSzama(int n) {
    int result = 0;
    do {
        n /= 10;
        result++;
    } while (n!=0);
    return result;
}

...
System.out.println
    ("Jegyek száma: "+jegyekSzama(szam));
...
```

15.3. ábra. A függvény visszatérési értéke

A fordító figyeli, hogy a függvénynek nincs-e olyan ága, amelyikból kikerülhet a vezérlés `return` nélkül. **Fordítási hibát eredményez, ha a függvénynek van `return` nélküli ága.**

A következő példában `n` negatív értéke mellett a függvénynek nincs visszatérési értéke (visszatérés azért van, hiszen a blokk végére kerülve mindenképpen befejeződik a metódus futása):

```
static int kovetkezo(int n) {
    if (n>=0)
        return n+1;
} // -> Fordítási hiba!!!
```

Eljárás esetén, vagyis ha a metódus visszatérési típusa `void`, nem kell `return` utasítással elhagynunk a blokkot. Ha mégis ki szeretnénk igrani a blokkból, akkor ezt a pusztta `return` utasítással tehetjük meg:

```
return;
```

Megjegyzés: Egy metódust önmagából újra meghívhatunk, mielőtt a vezérlés a blokk végére érne. Ezt a jelenséget **rekurzió**nak nevezzük. Ilyen esetben a hívásnak egy az előzőtől teljesen független példánya születik meg. A különböző hívások minden különálló, saját lokális változókkal dolgoznak. Rekurzió esetén ügyelni kell arra, hogy ne keveredjünk végtelen ciklusba. A rekurziók kell, hogy legyen egy leállító feltétele! A rekurzióról a könyv 2. kötetében lesz szó.

Feladat – Következő

Írunk egy függvényt, mely egyelőre nagyobb számot ad vissza, mint a paraméterként átadott szám!

Forráskód

```
public class Kovetkezo {  
  
    static int kovetkezo(int n) {  
        return n+1;  
    }  
  
    public static void main (String args[]) {  
        System.out.println(kovetkezo(555)); // -> 556  
    }  
}
```

15.4. Metódusok túlterhelése

Egy metódust nem a neve, hanem a szignatúrája azonosít, amely a nevén kívül tartalmazza a metódus összes paramétereinek típusát a deklarálás sorrendjében. (A szignatúrához a visszatérési típus nem tartozik hozzá.) Ezért megtehetjük, hogy ugyanolyan néven különböző paraméterezésű metódusokat is deklarálunk. Ezt a jelenséget a **metódusok túlterhelésének** nevezzük. Híváskor a fordító megkeresi azt a metódust, amely minden névben, minden paraméterezésben „ráhúzható” a hívásra. Az aktuális hívástól függően az a metódus fog futni, amelynek formális paramétere

- megfelel a hívásnak, vagyis ahol az összes aktuális paraméter értékadás szerint kompatibilis a megfelelő formális paraméterrel; és azok közül is, amelyik
- legközelebb áll az aktuális paraméterekhez.

A következő, MetodusMinta2 feladatban a vonalhuz() metódust háromféle paraméterezzel is megterheljük:

Feladat – Metódusminta2

Kérjünk be számokat a konzolról 0 végig! minden egyes számról állapítsuk meg, hogy az hány jegyű! Húzzunk

- a program elején egy 50 hosszú, @ karakterekből álló vonalat,
- az egyes eredmények kiírása után egy-egy 20 hosszúságú, mínusz karakterekből álló vonalat, valamint
- a program végén egy 50 hosszú, nevető fejecskekből álló vonalat!

Forráskód

```
import extra.*;
public class MetodusMinta2 {

    static void vonalhuz() {
        vonalhuz(20, '-');
    }

    static void vonalhuz(int hossz) {
        vonalhuz(hossz, '-');
    }

    static void vonalhuz(int hossz, char ch) {
        for (int i=1; i<=hossz; i++)
            System.out.print(ch);
        System.out.println();
    }

    static int jegyekSzama(int n) {
        int result = 0;
        do {
            n /= 10;
            result++;
        } while (n !=0);
        return result;
    }

    public static void main(String[] args) {
        vonalhuz(50,'@');
        int szam;
        while ((szam=Console.readInt("Szám: ")) != 0) {
            System.out.println("Jegyek száma: "+jegyekSzama(szam));
            vonalhuz();
        }
        vonalhuz(50,'\u0002'); // nevető fejecskek
    }
}
```

Ebben a feladatban nem nehéz a megfelelő hívásokat „ráhúzni” a metódusokra: a paraméterek száma egyértelműen eldönti, hogy melyik legyen a végrehajtandó metódus. A paraméter nélküli vonalhuz() alapértelmezésben 20 hosszúságú, mínusz jelekből álló vonalat húz.

A következő példában a paraméterek típusa alapján dől el, melyik lesz az aktuálisan hívott metódus:

Feladat – Minimumok

Írunk három minimumfüggvényt: az egyik két `int`, a másik két `long`, a harmadik két `double` szám közül adja vissza a kisebbet, ugyanolyan típusú értékként!

Forráskód

```
public class Minimumok {

    static int min(int a, int b) {
        return (a<b)? a:b;
    }

    static long min(long a, long b) {
        return (a<b)? a:b;
    }

    static double min(double a, double b) {
        return (a<b)? a:b;
    }

    public static void main(String[] args) {
        int i = 53;
        double d = 66.4666;
        System.out.println(min(i,5));           //1 -> 5
        System.out.println(min(5.0,d));         //2 -> 5.0
        System.out.println(min(292,d));          //3 -> 66.4666
        System.out.println(min(i,5000000000L)); //4 -> 53
    }
}
```

A forráskód elemzése

A `min` metódus hívásakor:

- ◆ //1-ben minden aktuális paraméter `int` típusú, ezért a fordító a `min(int,int)` szignatúrájú metódust hajtja végre;
- ◆ //2-ben minden aktuális paraméter `double` típusú, ezért a fordító a `min(double,double)` szignatúrájú metódust hajtja végre;
- ◆ //3-ban az egyik aktuális paraméter `int`, a másik pedig `double` típusú, ezért a fordító a `min(double,double)` metódust hajthatja végre, hiszen a `double`-nak lehet `int` típusú értéket adni, de fordítva nem;
- ◆ //4-ben egy `int` és egy `long` típusú paraméter kerül átadásra (az 5 milliárd után kötelező kitenni az `L` betűt, mert az `int`-be ekkora érték már nem fér be). A meghívott metódus a `min(long,long)`.

Megjegyzés: Ezek a `min` metódusok megtalálhatók a `java.lang.Math` osztályban is.

15.5. Lokális változók

A metódus blokkjában az utasítások között lehetnek deklaráló utasítások is. A `main` metódusban már eddig is deklaráltunk lokális adatokat.

A metódus blokkjában deklarált változók a metódus lokális változói, azok csak az őt deklaráló metódusban ismertek.

Egy lokális változó a deklarálás pontjától az őt deklaráló blokk végéig él. Egy lokális változó minden egyes alkalommal, amikor a metódus meghívásra kerül, a deklarálás pontján megszületik (számára memória helyet ekkor foglal a rendszer), majd amikor a vezérlés a blokk végére kerül, ez a memóriahely felszabadul, arra hivatkozni a továbbiakban nem lehet.

A formális paraméterek abban különböznek a lokális változóktól, hogy azok születésükkor (a metódusba való belépéskor) kívülről értéket kapnak. **A lokális változók és a formális paraméterek** egy programszinten vannak, **azonosítóiknak egyedieknek kell lenniük**.

A konstans lokális változó módosítója `final`.

A `jegyekSzama` metódusban a `result` lokális változó, az `n` formális paraméter:

```
static int jegyekSzama(int n) {
    int result = 0;
    do {
        n /= 10;
        result++;
    } while (n != 0);
    return result;
}
```

A lokális változó kezdőértéke

A lokális változók nem kapnak kezdőértéket a blokkba való belépéskor, arról a programozónak kell gondoskodnia. **A nem inicializált lokális változó használata fordítási hibát eredményez.**

A metódus- és utasításblokk változói

Minden változó a deklarálás pillanatában születik meg, és az őt deklaráló blokk végéig él. Ez a szabály a metódusblokkra és az utasításblokkra egyaránt érvényes. Egy érvényben levő változóval azonos nevű másik változó nem deklarálható. Az inicializálatlan változóra való hivatkozás fordítási hibát eredményez.

Például:

```

static void metodus() {
    double d=1e3; // d a metódusblokk változója
    for (int i=0; i<10; i++) { // i csak a for ciklusban él!
        //double d = 3.5; // Szintaktikai hiba! d már létezik!
        int n=10, m; // n és m az utasításblokk változói
        //System.out.println(m); // Szintaktikai hiba! m nincs inicializálva!
        System.out.println(d+n+i); // Rendben, d itt is érvényes.
    }
    int i=5; // Rendben, a for i-je már meghalt.
    int m=10; // Rendben, a for m-je már meghalt.
    //System.out.println(n); // Szintaktikai hiba! n már nem él!
}

```

A main metódus változói is lokális változók (a mintaprogramban a szam). Azért élnek ezek mégis egészen a program végéig, mert a program futását maga a main metódus futása jelenti. A main változói akkor szabadulnak fel, amikor a main végére kerül a vezérlés – ekkor azonban a programnak is vége van.

Megjegyzés: Egy metódus lokális változóit csak az a metódus ismeri, amelyben a változót deklarálták. Ezeket a változókat más metódusok egyáltalán nem látják, így nem is manipulálhatják. Ha egy változót az osztály több metódusából is használni szeretnénk, akkor a változót az osztály adataként (osztályszinten) kell deklarálni. Osztályszintű adatok deklarálásáról a 17., Osztály készítése című fejezetben lesz szó.

15.6. Néhány példa

Feladat – Összeg

Írunk egy olyan metódust, mely visszaadja a paraméterként megkapott két, int típusú szám összegét! A metódus egy lehetséges hívása:

```
int osszeg, a=3; osszeg = sum(5,a+3);
```

Megjegyzés: Ezt a metódust csak a feladat kedvéért írjuk meg. Az összeadó operátor használata kézenfekvőbb!

Forráskód

```

public class Osszeg {
    static int sum(int a,int b) {
        return a+b;
    }

    public static void main(String[] args) {
        int osszeg, a=3;
        osszeg = sum(5,a+3);
        System.out.println(osszeg); // -> 11
    }
}

```

Feladat – Faktoriális

Írunk eljárást, amely kiírja két szám között a számokat és a hozzájuk tartozó faktoriális értékeket ($n!=1*2*3*...*n$) növekvő sorrendben! (Akkor is, ha az első paraméter kisebb, mint a második.)

Az eljárás egy lehetséges hívása:

```
faktKiir(5,12);
```

Az eljáráshoz készítünk egy segédfüggvényt, mely kiszámítja egy szám faktoriálisát!

Forráskód

```
import extra.*;
public class Faktorialis {

    static long fakt(int n) {
        if (n < 1)
            return 1;
        long f = 1;
        for (int i=1; i<=n; i++)
            f *= i;
        return f;
    }

    static void faktKiir (int a, int b) {
        // a>b esetén a két szám felcserélése:
        if (a>b) {
            int seged=a; a=b; b=seged;
        }
        for (int i=a; i<=b; i++)
            System.out.println(i+" faktoriálisa: "+fakt(i));
    }

    public static void main(String[] args) {
        faktKiir(7,5);
    }
}
```

A program futása

| |
|----------------------|
| 5 faktoriálisa: 120 |
| 6 faktoriálisa: 720 |
| 7 faktoriálisa: 5040 |

Feladat – Véletlen szám

Készítsünk egy függvényt, mely visszaad egy olyan véletlen egész számot, amelyik a paraméterként megadott két egész között van (a határokat is beleértve)!

A függvény egy lehetséges hívása:

```
int veletlen = random(3,12);
```

Generálunk 1000 darab véletlen számot, és számoljuk ki az átlagukat!

Forráskód

```

import extra.*;

public class VeletlenSzam {
    static int random(int also, int felso) {
        // a>b esetén a két szám felcserélése:
        if (also>felso) {
            int seged=also; also=felso; felso=segед;
        }
        return (int)(Math.random()*(felso-also+1))+also;
    }

    public static void main(String[] args) {
        final short DB = 1000;
        long osszeg = 0;
        for (int i=0; i<DB; i++)
            osszeg += random(3,12);
        System.out.println(osszeg/(float)DB); // például -> 7.541
    }
}

```

Feladat – Sinus

A Math osztályban levő `sin(double)` függvény paraméterében radiánban kéri a szöget. Írunk egy olyan `sin` függvényt, amely a szöget fokokban várja!

Forráskód

```

public class Sinus {
    static double sin(double fok) {
        // 360 fok = 2PI radián, 1 fok = 2PI/360 radián
        return Math.sin(fok * 2 * Math.PI / 360);
    }
    public static void main(String[] args) {
        System.out.println(sin(90));           // -> 1.0
        System.out.println(sin(60));           // -> 0.866 (√3/2)
    }
}

```

15.7. Hogyan tervezzük meg metódusainkat?

Amikor egy metódust (rutint, eljárást vagy függvényt) szeretne írni, **sose a kódolással kezdje!** Először gondolja át, mit is akar tulajdonképpen. Nagyon fontos, hogy a metódusokat kényelmesen használhassuk, ezért először azt próbálghassa, hogyan szeretné meghívni. Gondoljon arra, hogy „Jó lenne, ha lenne egy ilyen metódus...”

- Az első dolog, hogy **megfogalmazzuk a metódus feladatát**.
- Megvizsgáljuk, hogy a metódust kifejezésben akarjuk-e használni, vagy sem – azaz hogy **függvény legyen-e vagy eljárás**. Általában: ha igeként (vagy igéből képzett fönévként) természetes a metódus hívása, akkor a metódus eljárás lesz. Például olvas,

olvasás, átalakít, `setData`; ha főnévként vagy melléknévként természetes a hívás, akkor a metódus függvény lesz. Például összeg, kiadás. Melléknév esetén a függvény visszatérési értéke valószínűleg logikai, például páros, prim, szabad, hibás, `isNice`.

- Végiggondoljuk: **van-e már ilyen metódusunk** esetleg egy másik osztályban, amely az adott körülmények között alkalmazható. Ha megtaláltuk, már használhatjuk is a metódust.
- **Elgondolkodunk a metódus nevén:** Utaljon feladatára! Legyen könnyen megjegyezhető! Ha nem tudok jó nevet választani, akkor gyanús: talán még nem is tisztáztam pontosan, mit akarok.
- Aztán a paraméterek következnek. Milyen paramétereket lenne kényelmes átadni a metódusnak? **Gondoljuk végig az egyes paramétereket egyenként:** milyen típusú értékeket, változókat akarunk majd átadni? A formális paraméterek típusát úgy kell megválasztani, hogy a `formPar=aktPar`; minden esetben értékadás szerint kompatibilis legyen! Végül csoportosítjuk a paramétereket, és azok számára is kitalálunk jó neveket. Ha függvényről van szó, akkor meghatározzuk a visszatérési érték típusát is.
- **Megtervezzük a metódust!** Ehhez megállapítjuk a szükséges lokális adatokat, majd a feladat bonyolultságától függően fejben vagy pszeudokód, illetve tevékenységszabály segítségével megtervezzük az algoritmust.
- Most jöhet a **metódus kódolása és tesztelése**. A forráskódot megjegyzésekkel látjuk el, hogy később is eligazodjunk rajta. Ne feledkezzünk meg a módosítók megadásáról!
- Ha a metódus általános célú, akkor azt **elraktározzuk** egy alkalmas osztályban, **hogy később is felhasználhassuk**.

Figyeljünk a következőkre:

- **Tartsuk be az adatok elrejtésének elvét!** Lehetőség szerint minden változót abban a blokkban deklaráljuk, amelyikben használjuk!
- **Kerüljük a redundanciát!** Ha lehet, semmit se írunk le kétszer.
- **Az eljárás, illetve függvény ne végezzen se többet, se kevesebbet, mint ami a feladatleírásban szerepel.** Ha például a függvénynek az a feladata, hogy megállapítsa egy számról, hogy az pozitív-e, vagy sem, akkor adjon vissza egy boolean értéket, de ne írjon a képernyőre üzenetet. A függvény használója valószínűleg nem oda, nem olyan nyelven és nem azt az üzenetet akarja majd kiírni!

Tesztkérdések

- 15.1. Jelölje meg az összes igaz állítást a következők közül!
- a) minden metódusnak van visszatérési értéke.
 - b) Egy függvényt meg lehet hívni eljárásként is.
 - c) A metódus fejében legalább egy módosítót mindenkorban meg kell adni.
 - d) minden metódus blokkjában kötelezően szerepelnie kell a `return` utasításnak.
- 15.2. Jelölje meg az összes igaz állítást a következők közül!
- a) Futáskor a formális paraméterek adódnak át az aktuális paraméterekeknek.
 - b) A függvény visszatérési típusa csak primitív típus lehet.
 - c) Az osztály metódusainak deklarálási sorrendje tetszőleges.
 - d) `static` módosítóval ellátott metódusból csak `static` módosítóval ellátott metódus hívható.
- 15.3. Mely kulcsszavak szerepelnek egy visszatérési értékkel nem rendelkező, a leszámazott osztályban felülírható, védett példánymetódus fejében? Jelölje meg az összes jó választ!
- a) `static`
 - b) `final`
 - c) `void`
 - d) `public`
- 15.4. Jelölje meg az összes szintaktikailag helyes metódusfejet!
- a) `public static boolean ok(a int)`
 - b) `void kiir()`
 - c) `public mennye_valahova()`
 - d) `public double pozicio(byte x, byte y)`
- 15.5. Mit ír ki a következő programrészlet? Jelölje meg az egyetlen jó választ!
- ```
static double div(int x, int y) {return x/y;}
...
System.out.println(div(15,4));
```
- a) 3
  - b) 3.0
  - c) 3.75
  - d) Semmit, mert a metódus deklarációja szintaktikailag hibás.
- 15.6. Jelölje meg az összes igaz állítást a következők közül!
- a) A formális paraméterlista elemeit pontosvesszővel választjuk el egymástól.
  - b) Ha a metódusnak van egy `int szam` formális paramétere, akkor a metódus blokkjában nem vehető fel ilyen nevű változó.
  - c) Fordítási hibát eredményez, ha egy metódusnak, amelynek nem `void` a visszatérési típusa, van `return` nélküli ága.
  - d) Az aktuális paraméterek értékkedás szerint kompatibilisnek kell lennie a formális paraméterrel.

- 15.7. Mi lesz az eredménye a következő program fordítási és futtatási kísérletének? Jelölje meg az egyetlen jó választ!

```
public class Proba {
 public static boolean met(int a, float b, int c) {
 return a-b>c ;
 }
 public static void main(String[] args) {
 System.out.println(met(5,6,4));
 }
}
```

- a) Fordítási hiba, mert a metódus feje hibás.
- b) Fordítási hiba, mert a paraméterátadás helytelen.
- c) Sikeresen lefut, és ezt írja ki: false
- d) Sikeresen lefut, és ezt írja ki: true

- 15.8. Jelölje meg az összes igaz állítást a következők közül!

- a) A metódust a neve egyértelműen azonosítja, függetlenül a paramétereitől.
- b) A main metódus hivatkozhat bármely más metódus lokális változóira.
- c) Egy lokális változó a deklarálás pontjától az öt deklaráló blokk végéig él.
- d) Egy lokális int típusú változó alapértelmezés szerinti értéke 0.

- 15.9. Mi a helyes sorrendje a metódus írásakor a következő tevékenységeknek? Jelölje meg az egyetlen jó választ!

1) Meghatározzuk a metódus nevét; 2) Meghatározzuk a paramétereket; 3) Megtervezzük a metódust; 4) Megfogalmazzuk a metódus feladatát.

- a) 1, 2, 3, 4
- b) 4, 1, 2, 3
- c) 3, 4, 1, 2
- d) 4, 3, 1, 2

- 15.10. Mi lesz az eredménye a következő program fordítási és futtatási kísérletének? Jelölje meg az egyetlen jó választ!

```
public class Szorzo {
 static void szoroz(int mit, int mivel) {
 mit = mit*mivel;
 }
 public static void main(String[] args) {
 int a=5;
 szoroz(a,7);
 System.out.println(a);
 }
}
```

- a) Fordítási hiba, mert a metódus feje hibás.
- b) Futás során kivétel keletkezik, hibás paraméterátadás miatt.
- c) Sikeresen lefut, és ezt írja ki: 5
- d) Sikeresen lefut, és ezt írja ki: 35

15.11. Mi lesz az eredménye a következő program fordítási és futtatási kísérletének? Jelölje meg az egyetlen jó választ!

```
public class Oszt {
 static double oszt(int mit,int mivel) {
 if (mivel==0)
 return;
 return (double)mit/mivel;
 }
 public static void main(String[] args) {
 int a=5, b=2;
 System.out.println(oszt(a,b));
 }
}
```

- a) Sikeresen lefut, és ezt írja ki: 2.0
- b) Sikeresen lefut, és ezt írja ki: 2.5
- c) Fordítási hiba, mert a metódus feje hibás.
- d) Fordítási hiba, mert a metódus nem ad vissza minden esetben értéket.

## Feladatok

15.1. (A) Írjon eljárást, amelynek

- a) nincs paramétere, és 10-szer leírja, hogy "Ezentúl minden szépen strukturálom a programomat!"
- b) egy paramétere van, és 10-szer leírja a paraméterben megadott szöveget!
- c) két paramétere van, és a paraméterben megadott számszor leírja a szintén paraméterben megadott szöveget!

A három eljárásnak ugyanaz legyen a neve! (*SokszorKiir.java*)

15.2. (A) Írja meg a következő eljárásokat (*Eljarasok.java*):

- a) Konzolra írja a megadott értékhatarok közötti páros számokat!
- b) Kiír egy adott hosszságú, adott karakterekből álló sort!
- c) Egy adott szélességű és magasságú tömör téglalapot rajzol konzolra a megadott karakterekkel! Az egyes sorokat az elkészített metódus írja!

Például a *teglalap(11,2,'@')* ; hívásának eredménye:

```
@@@@@@@#@@@@
@@@@@@@#@@@@
```

15.3. (A) Írja meg a következő függvényeket (*Fuggvenyek.java*):

- a) Visszaadja egy szám kétszeresét!
- b) A kör sugarából kiszámolja a kör területét!
- c) A gömb sugarából kiszámolja a gömb térfogatát!
- d) Megadja, hogy a megadott szám pozitív-e!
- e) Megadja, hogy a paraméterben megadott karakter benne van-e a szintén paraméterként megadott tartományban!

A függvény hívása például: *if benne('a','z',kar) ...*

- f) Visszaadja két karakter közül a kisebbiket!
- Például a *ch = min('a','A')* ; értékdás után ch értéke 'A' lesz.
- g) Megadja két szám közötti összes szám összegét (a határokat is beleértve)!
  - h) A cm-ben megadott hosszságot inch-re alakítja!

- 15.4. **(B)** Próbálja megírni a `Math.round(double)` és a `Math.round(float)` függvényeket a `Math.floor` függvény ismeretében! (*Round.java*)
- 15.5. **(C)** Írja meg a következő prímszámokkal kapcsolatos metódusokat:
- a) Eldönti egy számról, hogy az prím-e!
  - b) Kap egy számot, és kiírja az összes nála kisebb prímszámot! minden prím előtt írjuk ki, hogy hánnyadik prím!
  - c) Kap egy számot, és visszaadja, hogy hány nála kisebb prímszám van!
  - d) Kap egy számot, és visszaadja az első nála nagyobb prímszámot!
- (*Primek.java*)



## I. BEVEZETÉS A PROGRAMOZÁSBA

1. A számítógép és a szoftver
2. Adat, algoritmus
3. A szoftver fejlesztése

## II. OBJEKTUMORIENTÁLT PARADIGMA

4. Mitől objektumorientált egy program?
5. Objektum, osztály
6. Társítási kapcsolatok
7. Öröklődés
8. Egyszerű OO terv – Esettanulmány

## III. JAVA KÖRNYEZET

9. Fejlesztési környezet – Első programunk
10. A Java nyelvről

## IV. JAVA PROGRAMOZÁSI ALAPOK

11. Alapfogalmak
12. Kifejezések, értékadás
13. Szelekciók
14. Iterációk
15. Metódusok írása



## V. OSZTÁLYOK HASZNÁLATA, KÉSZÍTÉSE

16. Objektumok, karakterláncok, csomagolók
17. Osztály készítése

**V.**

## VI. KONTÉNEREK

18. Tömbök
19. Rendezés, keresés, karbantartás
20. A Vector és a Collections osztály

## FÜGGELÉK

- A tesztkérdések megoldásai  
Irodalomjegyzék  
Tárgymutató



## 16. Objektumok, karakterláncok, csomagolók

---

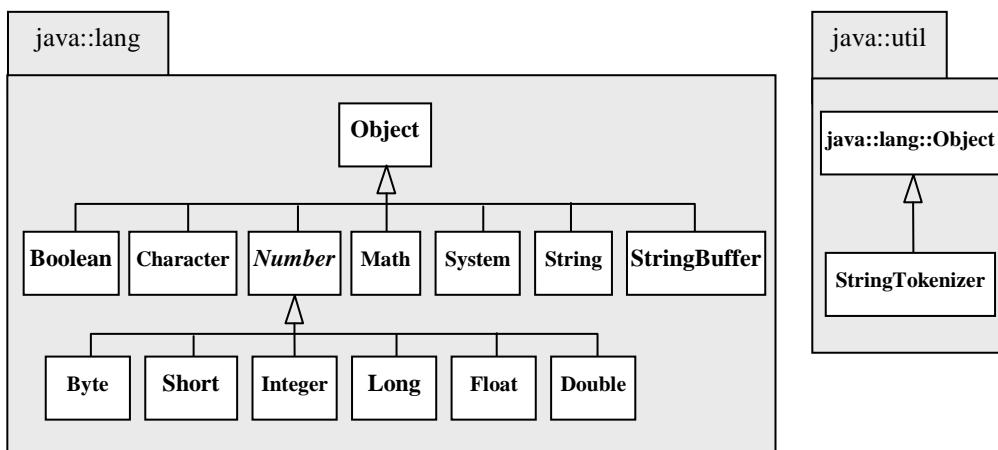
A fejezet pontjai:

1. A fejezet osztályainak rendszerezése
  2. Primitív típus – referenciatípus
  3. Objektum létrehozása, deklarálása
  4. Értékadás az objektumok körében
  5. Az objektum élete, automatikus szemétygyűjtés
  6. String osztály
  7. StringBuffer osztály
  8. Objektum átadása paraméterként
  9. Primitív típusok csomagolása
  10. StringTokenizer osztály
- 

A fejezet egyrészt megismerteti az Olvasót az objektumok létrehozásának és használatának technikájával, másrészről bemutat néhány alapvető Java (API) osztályt.

### 16.1. A fejezet osztályainak rendszerezése

Az API mintegy 2000 osztályt tartalmaz, amelyek csomagokba vannak szervezve. Egy programozó általában ezen osztályoknak csak igen kis hárnyadával dolgozik. E könyvnek sem az a célja, hogy minél több osztályt „használatba vegyen”, hanem felkészíteni a programozót arra, hogy a feladatához szükséges osztályokat szükség esetén minél könnyebben megtalálja, és azokat hatékonyan felhasználja. Vannak azonban az API-nak olyan alapvető osztályai, amelyeket minden programozónak alaposan ismernie kell – ezekből mutatunk be most néhányat. A fejezetben tárgyalt osztályok a `java.lang` és a `java.util` csomagokban kaptak helyet – a 16.1. ábra ezen osztályok öröklési hierarchiáját és az osztályokat tartalmazó csomagokat ábrázolja. **UML-ben a csomagot füles téglalappal jelöljük, a csomag nevét a fülre írjuk.** Egy csomagban idegen (más csomagban definiált) osztályt is ábrázolhatunk, de ekkor az osztály nevét minősítenünk kell az eredeti csomag nevével (a csomag és az osztály neve közé dupla kettőspontot teszünk). Az `Object` osztályt például a `java.util` csomagban így kell azonosítanunk: `java.lang:Object`.



16.1. ábra. A fejezetben tárgyalt API osztályok

### Az Object osztály – minden osztály űse

A Javában minden osztály az `Object` osztályból származik. Mondhatjuk tehát, hogy bármely osztályból létrehozott példány „az egy” objektum, így azt minden olyan feladatra meg lehet kérni, amely már az `Object` osztályban is definiálva van. Ahogy az a 16.1. ábráról leolvasható, az `Object` osztály a `java.lang` csomagban foglal helyet, a `java.util` csomagban már idegen osztályként szerepel (nevét saját csomagjával minősítettük: `java.lang.Object`). Az `Object` osztály olyan metódusokat tartalmaz, amelyek a Java minden objektumára jellemzők – ízelítőként felsorolunk néhányat:

- ▶ `boolean equals(Object obj)`

Összehasonlítja a megszólított objektumot egy másik (a paraméterben megadott) objektummal. A visszaadott érték `true`, ha a két objektum egyenlő. Ezt a metódust átszokás írni az utód osztályokban: meg lehet adni annak definícióját, hogy két objektum mikor egyenlő. Az eredeti, `Object.equals` metódus akkor ad vissza igaz értéket, ha a két objektum referenciajá ugyanaz.

- ▶ `String toString()`

Visszaadja az objektum szöveges reprezentációját. A metódust az utódokban felül átszokás írni. Ha nem írják felül, a metódus visszaad egy szöveget, amely az objektum osztályát és hasítókódját<sup>1</sup> tartalmazza. A `System.out.println(objektum)` minden

<sup>1</sup> hash code: egy objektumhoz rendelt szám, amely az objektum hasítótáblában való elhelyezéséhez kell.

esetben az `objektum.toString()` metódus által visszaadott szöveget írja ki a konzolra. Például:

```
Object obj = new Object(); // obj:Object objektum létrehozása
System.out.println(obj); //->java.lang.Object@11b1
```

► `Class getClass()`

Visszaadja az objektum osztályát. Például:

```
String str = new String("babaruha");
System.out.println(str.getClass()); //->Class java.lang.String
```

### Az Object osztály leszármazottai

A `java.lang` csomagban kaptak helyet az ún. csomagoló osztályok: `Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float` és `Double`. Ezek a megfelelő primitív típusokat csomagolják osztályokba, lehetővé téve ezzel a primitív értékek objektumként való használatát. Ahogy a 16.1. ábrán látható, a numerikus osztályoknak van egy közös absztrakt őse, a `Number`. Az absztrakt osztály csak örökölteti célokat szolgál, abból nem lehet példányt létrehozni. Az UML-ben az absztrakt osztályt dölt betűvel szedjük.

A `java.lang` csomagban levő `Math` és `System` nem örökíthető (`final`) és nem példányosítatható osztályok (nincs publikus konstruktork), ezekben csupa statikus deklaráció szerepel:

- ◆ A `Math` osztály matematikai konstansokat és függvényeket definiál. Ezt az osztályt már a 12., Kifejezések, értékadás fejezetben tárgyaltuk.
- ◆ A `System` osztályban a rendszer működésével kapcsolatos alapvető metódusok és objektumok kaptak otthont. Itt van például a rendszer azonnali leállását eredményező `exit()` metódus, és itt vannak az `in` és az `out` objektumok, amelyeket a konzolról történő beolvasáshoz, illetve az arra való kiíráshoz használunk. Például:

```
if (szam<=0)
 System.exit(0);
System.out.println("Ez egy pozitív szám");
```

A `java.lang` csomag `String` és `StringBuffer` osztályai szövegek (karakterláncok) tárolására alkalmasak. A két osztály közötti legnagyobb különbség az, hogy a `String` típusú objektum állapota nem változtatható (egész élete során ugyanaz a szöveg van benne), szemben a `StringBuffer` típusú objektummal, amelynek állapota változtatható, a benne lévő szöveg manipulálható.

A `java.util` csomag  `StringTokenizer` osztálya segítségével karakterláncot tudunk darabolni: megadjuk az elválasztójeleket, majd ennek alapján sorban visszakaphatjuk a karakterlánc részeit, például egy mondat szavait.

A karakterlánckezelő, a csomagoló és a karakterláncdaraboló osztályokat ebben a fejezetben tárgyaljuk.

## 16.2. Primitív típus – referenciatípus

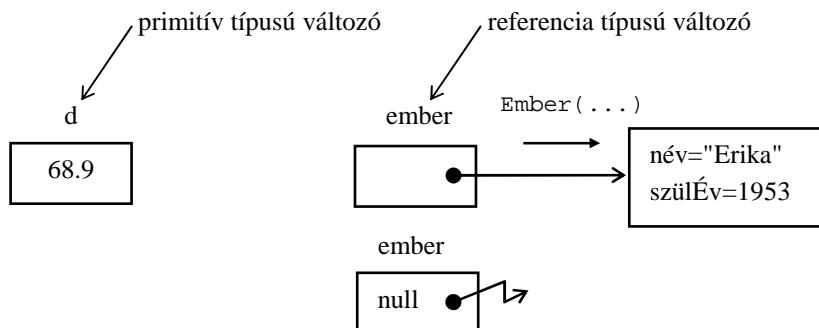
**A Javában kétféle típus létezik:** primitív típus és referencia típus (16.2. ábra):

- **Primitív típus:** Egy primitív típusú változó azonosítójával **közvetlenül hivatkozunk** a változó memóriahelyére. Ezt a helyet a rendszer a deklaráló utasítás végrehajtásakor foglalja le. A Javában 8 előre definiált primitív típus van: boolean, char, byte, short, int, long, float és double. A programozó nem definiálhat primitív típust (egy programban csak ez a 8 primitív típus szerepelhet).
- **Referenciatípus:** A Javában a referencia típusú változók objektumokra mutatnak. Egy referencia típusú változó azonosítójával az objektum memóriahelyére **közvetve hivatkozunk** egy referencián (hivatkozáson) keresztül.

A referencia típusú változók alkalmazása egyszerű, mert maga a hivatkozás rejtvé marad.

Például:

```
double d=68.9;
Ember ember;
ember = new Ember("Erika",1953);
ember.megy();
System.out.println(ember);
ember = null;
```



16.2. ábra. Primitív és referencia típusú változók

### A null referencia

A `null` referencia típusú konstans, foglalt szó (16.2. ábra). A `null` referencia értéke a nulla cím, és azt fejezi ki, hogy nem mutat semmire. Arra való, hogy megkülönböztessük azoktól a referenciáktól, amelyek egy adott objektumra hivatkoznak.

### 16.3. Objektum létrehozása, deklarálása

#### new operátor, konstruktur

Egy osztály példányait a **new** (új) operátorral hozhatjuk létre. A new után meg kell adni a létrehozandó objektum osztályát, ezt a konstruktor aktuális paraméterlistája követi:

```
new <OsztályAzonosító>(<aktuális paraméterlista>)
```

##### A new operátor feladatai:

- Létrehoz egy új, OsztályAzonosító osztályú objektumot; lefoglalja számára a szükséges memóriát;
- Meghívja az osztálynak azt a konstruktorát, amelynek szignatúrájára röházható az aktuális paraméterlista;
- Visszaadja az újdonsült objektum referenciáját (azonosítóját). Az objektum osztálya az lesz, amit a new után megadtunk.

A **konstruktor** beállítja az objektum kezdeti állapotát (kezdeti értéket ad az adatoknak és az esetleges kapcsolatoknak). A konstruktor neve megegyezik az osztály nevével. Egy osztálynak több konstruktora is lehetséges.

Az objektum egész élete során a new után megadott osztályhoz fog tartozni, osztályát megváltoztatni nem lehet.

Minden Java osztályban szerepelnie kell legalább egy konstruktornak. A konstruktort az osztály készítője írja meg, Ő a felelős a helyes konstruálásért. Ha az osztály készítője nem ír konstruktort, akkor egy alapértelmezés szerinti, paraméter nélküli, üres konstruktor fut le. Ha a konstruktort priváttá tesszük, letilthatjuk az osztály példányosítását. Ezt olyankor szokás például alkalmazni, amikor az osztály csupa statikus metódussal rendelkezik.

#### Objektum deklarálása

Minden referencia típusú változót (mint ahogyan a primitív típusúkat is) deklarálni kell!

##### Deklaráláskor csak a referencia részére következik be tárfoglalás:

```
<OsztályAzonosító> objektum;
```

Az objektum létrehozásáról a programozónak kell gondoskodnia. A new operátor által visszaadott referencia értékül adható a referencia típusú változónak:

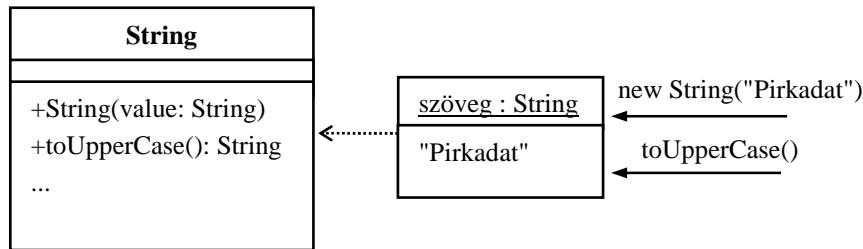
```
objektum = new <OsztályAzonosító>(<aktuális paraméterlista>);
```

vagy

```
<OsztályAzonosító> objektum =
 new <OsztályAzonosító> (<aktuális paraméterlista>);
```

Az objektumot a referencia típusú változón keresztül szólíthatjuk meg:

```
objektum.metódus()
```



16.3. ábra. Objektum létrehozása

Példaként hozunk létre egy `String` objektumot (16.3. ábra)!

```

String szoveg; //1
szoveg = new String("Pirkadat"); //2
System.out.println(szoveg.toUpperCase()); //3 -> PIRKADAT
System.out.println(szoveg); //4 -> Pirkadat

```

//1-ben deklaráltunk egy `szoveg` nevű referencia típusú változót. A `szoveg` értéke egyelőre definiálatlan, egy `String` objektumra mutatót fog majd tartalmazni. //2-ben létrehozunk egy `String` objektumot, melynek állapota "Pirkadat" lesz. A létrehozott objektum mutatóját rögtön értékül adjuk a `szoveg` referenciának, amely a `String` típusú (osztályú) objektumot egyértelműen azonosítja. Az objektumra így hivatkozunk: `szoveg`. Az objektumot //3-ban így szólítjuk meg: `szoveg.toUpperCase()` – ez egy függvény, mely visszaadja a szöveg nagybetűs változatát. //4-ben kiírjuk a `szoveg` objektum tartalmát.

Természetesen megtehetünk volna, hogy az objektumot rögtön, vagyis a deklarációkor hozzuk létre:

```
String szoveg = new String("Pirkadat");
```

Látható tehát, hogy a referencia típusú változóval a hivatkozás közvetett – tulajdonképpen a referencia által mutatott objektumot szólítjuk meg. A közvetettséget a programozó nem látja, és nem is szoktuk jelölni az objektumdiagramokon. A Javában minden objektum automatikusan referencia típusú változó.

## 16.4. Értékadás az objektumok körében

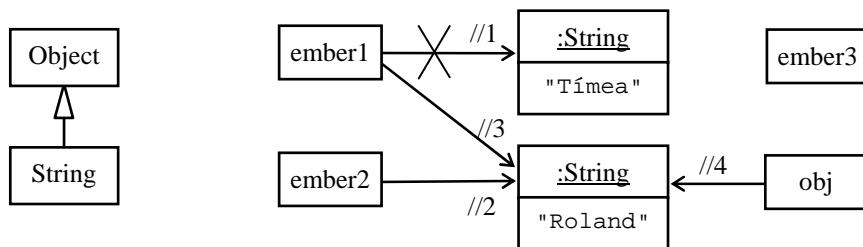
Objektumokat (referencia típusú változókat) értékül adhatunk egymásnak – ilyenkor maga a hivatkozás másolódik át az egyik változóból a másikba: **értékadás után a két változó ugyanazt az objektumot azonosítja.**

Az `obj2` referencia típusú változó **értékadás szerint kompatibilis** az `obj1` referencia típusú változóval, vagyis `obj2` értékül adható `obj1`-nek (`obj1=obj2`), ha

- `obj1` és `obj2` ugyanahhoz az osztályhoz tartoznak, vagy
- az `obj1` osztálya őse az `obj2` osztályának.

Az `obj1=obj2` értékadás után `obj1`, akármilyen objektumot azonosított is eddig, most ugyanazt fogja azonosítani, mint `obj2`. Ha `obj2` nem azonosított még objektumot, akkor fordítási hiba keletkezik.

Az értékadás szabályát így is megjegyezhetjük: Egy referencia csak abban az esetben mutathat egy objektumra, ha a mutatott objektum legalább olyan, mint ő. Képzeljük el a következő szituációt: Az Ember osztály leszármazottai a Hallgató és a Tanár osztályok. Egy teremben hallgatók és tanárok vannak. Ha mindenkihez akarok szólni, akkor csak mint emberekhez beszélhetek, hiszen a tanároknak küldött üzeneteket a hallgatók nem értik meg, és fordítva. A közös mutató típusának mindig meg kell keresni a legnagyobb közös osztályt, mert az abban levő üzeneteket még mindenki megérte – bár ugyanazokat az üzeneteket ki-ki a saját módján fogja értelmezni (polimorfizmus). Ugyanígy, mivel „a String az egy Object”, ezért egy Object mutató mutathat Stringre, de fordítva nem, mert „az Object az nem String”.



16.4. ábra. Objektumok közötti értékadás

A következő példában értékadások sorozatával próbáljuk érzékeltetni az objektumok közötti értékadás szabályait. Az egyes utasításokat a 16.4. ábra is érzékelte.

```

String ember1 = new String("Tímea"); //1
String ember2 = new String("Roland"); //2
Object obj;
String ember3;
ember1 = ember2; //3
obj = ember1; //4
// ember1 = obj; Szintaktikai hiba! obj általánosabb!
// ember1 = ember3; Szintaktikai hiba! ember3-nak nincs értéke!

```

Az értékkadások után már három hivatkozás is Rolandra mutat, Tímeára pedig egy sem. Hacsak nincs a programban egyéb referencia, amely Tímeára mutatna, Tímea megszólíthatatlanná válik, a külvilág számára egyszerűen elvész.

### Objektumok egyenlőségvizsgálata

Objektumok állapotát nem lehet a hasonlító operátorokkal (`==`, `!=`) összehasonlítani. Az `obj1==obj2` logikai kifejezés a két objektum referenciáját hasonlíta össze: azt adja meg, hogy a két referencia ugyanoda mutat-e, azaz a két objektum fizikailag azonos-e! Ha a két objektum azonos, akkor természetesen állapotuk is megegyezik, de két nem azonos objektumnak is lehet ugyanaz az állapota. Objektumok egyenlőségét az `equals` metódussal szokás megállapítani.

Például:

```

String s1 = new String("Hello"), s2 = new String("Hello");
System.out.println((s1==s2)?"Azonosak":"Nem azonosak");
System.out.println((s1.equals(s2))?"Egyenlők":"Nem egyenlők");

```

A programrész a következő szövegeket jeleníti meg:

|              |
|--------------|
| Nem azonosak |
| Egyenlők     |

### 16.5. Az objektum élete, automatikus szemétgyűjtés

Minden egyes `new` operátor létrehoz egy objektumot, amely természetesen memória foglalással jár. Amikor egy referencia típusú változó értékét megváltoztatjuk, az addig hivatkozott objektum még tovább él, foglalja a memóriát! Ha nincs egyetlen olyan referencia sem, amely az adott objektumot azonosítaná, az objektum a továbbiakban elérhetetlenné, megszólíthatatlanná válik. Az ilyen objektumokat a rendszer figyeli, és időnként „kisöpri” a memóriából – ezt a folyamatot nevezzük automatikus szemétgyűjtésnek. A Javában nincs lehetőség az objektumok egyesével történő megszüntetésére; az objektumok felszámolása a rendszer dolga.

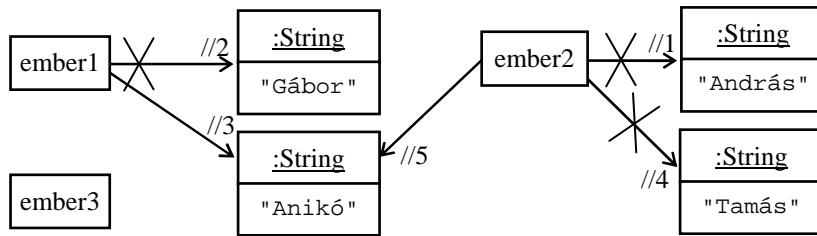
Tekintsük a következő példát, és kövessük nyomon a hivatkozásokat (16.5. ábra):

```

String ember1, ember2=new String("András"), ember3; //1
ember1 = new String("Gábor"); //2
ember1 = new String("Anikó"); //3

```

```
ember2 = new String("Tamás"); //4
ember2 = ember1; //5
```



16.5. ábra. Hivatkozások nyomon követése

//1-ben három darab referenciát deklaráltunk, ezek közül `ember2` a szöveg objektumra, Andrásra mutat, `ember1` és `ember3` még inicializálatlanok. //2-ben már `ember1` is egy létező objektumot azonosít, ö Gáborra mutat. //3-ban felülvírtuk Gábor mutatóját – `ember1` most már Anikót azonosítja. Ezzel Gábor elvész, hiszen nem jegyeztük meg a mutatóját. //4-ben András vész el, `ember2` új azonosítottja most már Tamás. Végül //5-ben `ember2`-t átirányítjuk `ember1`-re, így Tamás is az enyészet lesz, Anikóra viszont ketten mutatnak. `ember3` egyáltalán nem kapott értéket e tevékenységsorozatban. A kód részletben tehát összesen 4 objektum született, de az egészet minden összesen Anikó élte túl. A három fiúért alkalmasint jön a takarítónő, és a szemétygyűjtés áldozatai lesznek. Hogy ez mikor következik be, az a programozót nem érdekli egészen addig, amíg van memória az új objektumok számára. Ha fogytán a memória, a „takarítónő” úgy is akcióba lép.

**Automatikus szemetgyűjtés** (garbage collection): A rendszer bizonyos időnként automatikusan felszabadítja a hivatkozás nélküli objektumok memóriahelyeit.

## 16.6. String osztály

A **String osztály** olyan szöveg (karakterlánc) tárolására szolgál, amelynek értékét egész élete során nem akarjuk megváltoztatni. A `String` objektum szövege unikód karakterek sorozata, a karakterek sorszámozottak: ha a lánc `n` karaktert tartalmaz, vagyis a lánc hossza `n`, akkor a legelső karakter sorszáma (indexe) 0, az utolsó pedig `n-1`.

A 16.6. ábrán látható `"Kész!"` objektum egy 5 hosszú szöveg, a `K` betű indexe 0, az `é` indexe 1, a `!` karakteré pedig 4.

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| K | é | s | z | ! |

16.6. ábra. Karakterlánc indexelése

A szöveg elvileg olyan hosszú lehet, amilyent még az int típus ki tud fejezni. A String objektum állapota nem változtatható, vagyis a Kész! minden Kész! marad, amíg csak az objektum él.

### Létrehozás a new operátorral és anélkül

Mint minden más objektumot, a String objektumot is a new operátorral hozhatjuk létre:

```
String szoveg = new String("Ez egy szövegobjektum");
```

A Java megengedi, hogy egy String objektumot létrehozhassunk egy szöveg literál értékadásával is; ekkor a new feladatát a program automatikusan elvégzi:

```
String szoveg = "Ez is egy objektum, new nélkül";
```

Ez utóbbi esetben a rendszer egy optimalizációt végez el, miszerint a megegyező karakterláncokat a programban csak egyszer tárolja, így ezen szövegek referenciai minden ugyanarra az objektumra mutatnak. Mivel a szöveget a program során úgysem lehet megváltoztatni, a tárolás módját a programozó észre sem veszi. Ha azt szeretnénk, hogy biztosan történjék tárfoglalás, akkor a new operátort kell használnunk!

### A String osztály metódusai

A String osztály főként olyan példánymetódusokat (csak függvényeket) deklarál, amelyek a tárolt karakterláncról függően egy új karakterláncot vagy információt adnak vissza:

- a szöveg hosszát;
- a szöveg kisbetűs, illetve nagybetűs változatát;
- egy másik szöveggel való konkatenálását (összeadását);
- más karakterláncnal való összehasonlítás eredményét;
- adott pozícióon levő karakterét, illetve részláncát;
- egy adott karakter, illetve részlánc indexét;
- stb.

E függvények mindegyike az objektum tartalmát változatlanul hagyva egy új String típusú objektumot állít össze, és ennek az új objektumnak a referenciaját adja vissza.

A `String` osztályban vannak ezenkívül olyan statikus metódusok, amelyekkel különböző primitív értékek, illetve objektumok karakterlánc-változatait kaphatjuk meg.

Az egyes metódusok a következő kivételeket dobhatják:

- ◆ `StringIndexOutOfBoundsException`: Ez akkor következik be, ha a paraméterben megadott index a szövegen kívülre mutat.
- ◆ `NullPointerException`: Ez akkor következik be, ha a paraméterben megadott objektum referencia a null.

A `String` osztály lényegesebb, publikus konstruktőrait, metódusait a 16.7. ábra mutatja. Az osztályban először szerepelnek a konstruktőrök, majd ábécé rendben az egyéb metódusok. A metódusokat logikai csoportokba foglalva tárgyaljuk úgy, hogy közben egy-egy példát is bemutatunk.

| <b>String</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>String()</code><br><code>String(value: String)</code><br><code>String(buffer: StringBuffer)</code><br><code>charAt(index: int): char</code><br><code>compareTo(o: Object): int</code><br><code>compareTo(str: String): int</code><br><code>compareToIgnoreCase(str: String): int</code><br><code>concat(str: String): String</code><br><code>equals(anObject: Object): boolean</code><br><code>equalsIgnoreCase(anotherString: String): boolean</code><br><code>indexOf(ch: int): int</code><br><code>indexOf(ch: int, fromIndex: int): int</code><br><code>indexOf(str: String): int</code><br><code>indexOf(str: String, fromIndex: int): int</code><br><code>length(): int</code><br><code>replace(oldChar: char, newChar: char): String</code><br><u><code>valueOf(value: Type): String</code></u><br><u><code>valueOf(value : type): String</code></u><br><code>substring(beginIndex: int): String</code><br><code>substring(beginIndex: int, endIndex: int): String</code><br><code>toLowerCase(): String</code><br><code>toUpperCase(): String</code><br><code>toString(): String</code><br><code>trim(): String</code> |

16.7. ábra. String osztály

## Konstruktorok

- ▶ `String()`  
A létrehozott objektum az üres karakterláncot reprezentálja, illetve tartalmazza.
- ▶ `String(String value)`
- ▶ `String(StringBuffer buffer)`  
A létrehozott objektum a paraméterben megadott szöveget tartalmazza. A `StringBuffer` osztályról a következő pontban lesz szó. Például:  
`String szoveg = new String("Változtathatatlan szöveg!");`

## Hossz, index

- ▶ `int length()`  
Visszaadja a szöveg hosszát. Például:  
`String str = "Baromfi";  
int szam = str.length(); // -> szam==7`
- ▶ `char charAt(int index)`  
Visszaadja az index indexű karaktert. Például:  
`String str = "Baromfi";  
char ch = str.charAt(1); // -> ch=='a'`

## Manipulált karakterláncok

- ▶ `String toLowerCase()`  
Visszaad egy objektumot, amely az objektum szövegének csupa kisbetűs változata.
- ▶ `String toUpperCase()`  
Visszaad egy objektumot, amely az objektum szövegének csupa nagybetűs változata.
- ▶ `String toString()`  
Visszaadja saját magának a másolatát.
- ▶ `String trim()`  
Visszaad egy szöveget, amelynek elejéről és végéről kitörölte a fehér szóközöket.
- ▶ `String replace(char oldChar, char newChar)`  
A metódus visszaad egy karakterlánc-objektumot, melyben minden `oldChar` karaktert `newChar`-ra cserélt. Például:  
`String str1 = "Szip iz icipici niji cipi", str2;  
str2=str1.replace('i','a'); //str2=="Szap az acapaca naja capa"`
- ▶ `String substring(int beginIndex)`  
Visszaadja az objektum részláncát `beginIndex`-től kezdve végig. (A `substring`-ben a középső s kisbetű!) Például:  
`0123456  
String str = "Tehénke";  
System.out.println(str.substring(3)); // -> "énke"`
- ▶ `String substring(int beginIndex, int endIndex)`  
Visszaadja az objektum részláncát `beginIndex`-től `endIndex-1`-ig. `endIndex` már nem tartozik bele a részláncba, így a részlánc hossza `endIndex-beginIndex`. Például:  
`0123456  
String str = "Tehénke";  
System.out.println(str.substring(3,5)); // -> "én"`

**Feladat – StringTeszt**

Kérjünk be terminálról egy szöveget, majd:

- 1) Írjuk ki fordítva!
- 2) Írjuk ki csupa nagybetűvel, aztán csupa kisbetűvel!
- 3) Írjuk ki az első 9 karakterét és az utolsó 3 karakterét!
- 4) Cseréljük ki az összes szóközt kötőjelre!

**Forráskód**

```
import extra.*;
public class StringTeszt {
 public static void main(String[] args) {
 String szoveg = Console.readLine("Szöveg: ");
 // Kiírás fordítva //1
 for (int i=szoveg.length()-1; i>=0; i--)
 System.out.print(szoveg.charAt(i));
 System.out.println();
 // Nagybetűs, majd kisbetűs formák //2
 System.out.println(szoveg.toUpperCase());
 System.out.println(szoveg.toLowerCase());
 // Első 9 és utolsó 3 karakter kiírása //3
 if (szoveg.length()>=9) // egyébként futási hiba lenne
 System.out.println(szoveg.substring(0,9));
 if (szoveg.length()>=3) // egyébként futási hiba lenne
 System.out.println(szoveg.substring(szoveg.length()-3));
 // Az összes pont kicserélése kötőjelre //4
 System.out.println(szoveg.replace('.', '-'));
 }
}
```

**A program egy lehetséges futása**

```
Szöveg: Bolond Istok
kotsI dnoloB
BOLOND ISTOK
bolond istok
Bolond Is
tok
Bolond-Istok
```

**A program elemzése**

A szövegnek a `Console.readLine()` metódussal adunk értékét: a `readLine()` a konzolról bekér egy szöveget, ennek alapján összeállít egy `String` objektumot, majd visszaadja ennek az objektumnak a referenciáját. //1-ben a for ciklus i ciklusváltozója visszafelé halad: a szöveg karaktereit sorban kiolvassuk visszafelé a `charAt(i)` segítségével, és kiírjuk őket a konzolra. //2-ben kiírjuk a szöveg nagybetűs és kisbetűs formáit. //3-ban az első 9 karakter kiírásakor amennyiben a bevitt szöveg 9-nél rövidebb, nem írunk ki semmit. Megtehetünk volna, hogy egy üzenet kíséretében kiírjuk a 9-nél rövidebb szöveget. Ugyanez vonatkozik az utolsó 3 karakter kiírására is: ha a szöveg nincs legalább 3 karakternyi, nem írunk ki semmit. Figyelje meg, hogy

az eredeti szöveg objektum mindenkoráig megmarad, az egyes függvények minden esetben egy új objektumot készítenek!

### Egyenlőségvizsgálat, hasonlítás

- ▶ `boolean equals(Object anObject)`  
Összehasonlítja az objektumot a paraméterként megadott másik objektummal. A visszaadott érték `true`, ha a paraméterben megadott objektum osztálya `String`, és a szövegek karakterei rendre megegyeznek.
- ▶ `boolean equalsIgnoreCase(String str)`  
Összehasonlítja az objektumot a paraméterként megadott másik `String` objektummal. A visszaadott érték `true`, ha a két szöveg karakterei rendre megegyeznek úgy, hogy a nagy- és kisbetűk között nincs különbség.
- ▶ `int compareTo(Object o)`
- ▶ `int compareTo(String str)`
- ▶ `int compareToIgnoreCase(String str)`  
Összehasonlítják az objektumot a paraméterként megadott másik `String` objektummal. A harmadik esetben a kis- és nagybetűk közt nem tesz különbséget. A visszaadott érték 0, ha a két szöveg egyenlő; negatív, ha a szöveg lexikográfikusan kisebb, mint a paraméter szövege; és pozitív, ha a szöveg nagyobb, mint a paraméter szövege.

● Vigyázat! Az `==` operátor nem az egyenlőséget, hanem az azonosságot vizsgálja!

#### Feladat – Legelső

Kérjünk be terminálról szavakat "\*" végjelig. Csak az a szó számít a feldolgozásban, amelyik nem tartalmaz szóközt, és nem üres. Végül írjuk ki a bevitt szavak közül az ábécében első helyen állót!

#### Forráskód

```
import extra.*;
public class LegElso {
 public static void main(String[] args) {
 // A bevitt szavak ennél biztosan kisebbek lesznek:
 final String NAGY = "\uffff";
 String szo, elso = NAGY;
 while (!(szo=Console.readLine("Szó: ")).equals("*")) {
 // Nem jó, ha van benne szóköz vagy üres:
 if (szo.indexOf(" ")>=0 || szo.length()==0)
 System.out.println("Van benne szóköz, vagy üres!");
 else if (szo.compareTo(elso)<0) // szo < elso
 elso = szo;
 }
 if (elso.equals(NAGY))
 System.out.println("Nem volt jó szó bevitel!");
 else
 System.out.println("ABC szerint az első: "+elso);
 }
}
```

### A program egy lehetséges futása

```

Szó: durung
Szó: hegyes bot
Van benne szókoz, vagy üres!
Szó: Uborka
Szó: *
ABC szerint az első: Uborka

```

### A program elemzése

A feladatban a minimumkiválasztás algoritmusát alkalmazzuk. Kezdetben az ábécében legelső szónak egy olyan NAGY szót adunk, amelyiknél az ábécében biztosan kisebb minden bevitt szó. A szavakat a while ciklus fejében kérjük be, értéktől adjuk a szó változónak, és azonnal megvizsgáljuk, hogy értéke a \* végjel-e: ha igen, kilépünk a ciklusból, ha nem, akkor amennyiben a szó legális (nincs benne szóköz, és nem üres), megnézzük, hogy az eddigi első szót megelőzi-e. Ha igen, Ő lesz a pillanatnyi első. Végül kiírjuk az ábécé szerinti első szót, feltéve, hogy vittek be egyáltalán elfogadható szót. Megjegyezzük, hogy a nagybetűk unikódjai előrébb találhatók a kódtáblában, mint a kisbetűkéi – ezért van előbb az Uborka, mint a durung.

### Keresések

- ▶ int indexOf(int ch)
- ▶ int indexOf(int ch, int fromIndex)
- ▶ int indexOf(String str)
- ▶ int indexOf(String str, int fromIndex)

E négy metódussal karaktert (ch) vagy részláncot (str) kereshetünk a szövegben, előlről vagy egy adott indextől (fromIndex) kezdve. A visszaadott egész érték a szöveg azon indexe, ahol a keresendő elemet a metódus először találta meg. Ha nincs ilyen elem, akkor a visszaadott érték -1. Megjegyzendő, hogy a karakter típusa int, így a karaktert szám formában is megadhatjuk. Például:

```

String str = "KATONA";
System.out.println(str.indexOf('A')); // -> 1
System.out.println(str.indexOf(65)); // -> 1
System.out.println(str.indexOf('A', 2)); // -> 5

```

*Megjegyzés:* A lastIndexOf metódusok a karakterláncban visszafelé keresnek.

### Konkatenáció

- ▶ String concat(String str)

A metódus az objektum karakterláncához hozzáírja a paraméterben megadott karakterláncot, és ez lesz a metódus visszatérési értéke. Két String objektumot a + operátorral is össze lehet adni. A keletkezett új objektum szövege az operandusok konkatenációja.

A következő két utasítás ekvivalens egymással:

```

str = str1+str2;
str = str1.concat(str2);

```

## Karakterláncformák

- static String valueOf(<Type> value)
- static String valueOf(<type> value)

Type egy tetszőleges osztályt, type egy tetszőleges primitív típust jelöl. A statikus metódusok visszaadják value karakterláncformáját. Ha a paraméter objektum, akkor az objektum osztályának `toString()` metódusa határozza meg a visszaadott értéket; primitív típusú paraméter esetén a megfelelő csomagoló osztály `toString()` metódusa a mérvadó (a csomagoló osztályokról még ebben a fejezetben szó lesz). Például:

```
String dStr = String.valueOf(4.519);
String iStr = String.valueOf(71);
System.out.println(dStr+" "+iStr); // -> "4.519 71"
```

### Feladat – TizedespontCsere

Kérjünk be egy valós számot. A számot úgy írjuk ki a konzolra, hogy a tizedespont helyén tizedesvessző jelenjék meg!

### Forráskód

```
import extra.*;
public class TizedespontCsere {
 public static void main(String[] args) {
 double szam = Console.readDouble("Valós szám: "); //1
 String szamSt = String.valueOf(szam); //2
 System.out.println(szamSt.replace('.', ',')); //3
 }
}
```

### A program egy lehetséges futása

|                    |
|--------------------|
| Valós szám: 34.981 |
| 34,981             |

### A program elemzése

//1-ben bekérünk egy double típusú számot. //2-ben ezt a számot karakterláncá alakítjuk, mert a manipulációt csak szövegen tudjuk elvégezni. //3-ban a szöveg összes pont karakterét vesszőre cseréljük (mi tudjuk, hogy a szövegen a pontok száma csak 0 vagy 1 lehet).

## 16.7. StringBuffer osztály

A String osztály nem alkalmas arra, hogy egy karakterláncot megváltozzassunk, azon bonyolultabb műveleteket hajtsunk végre. Egy String objektumból nem tudunk egyetlen karaktert sem kitörölni vagy megváltoztatni, és nem tudunk egyetlen karaktert sem beszűrni.

A String osztálytalálal ellentében a **StringBuffer osztály** szövege manipulálható. Elvégezhetők például a következő műveletek:

- karakter, szöveg, valamint primitív típusú adatok hozzáadása a szöveghez;
- karakter, szöveg, valamint primitív típusú adatok beszűrása a szövegbe;
- adott pozíciójú karakter, illetve részlánc törlése;
- stb.

A StringBuffer által tartalmazott szöveg aktuális hossza futás közben változhat. A StringBuffer objektumnak van

- kapacitása (capacity), ez azt jelenti, hogy milyen hosszú szöveg tárolására van felkészülve. Az objektum kapacitását az egyes műveletek automatikusan állítják, de a programozónak is lehetősége van a kapacitás megváltoztatására hatékonyiségi célokban.
- aktuális hossza (length), vagyis az éppen tárolt szöveg hossza.

A StringBuffer osztály metódusait főleg eljárásként szokás meghívni, de a metódusok mintegy extra szolgáltatásként visszaadják a megváltoztatott objektum referenciáját.

A String és StringBuffer osztályok nincsenek öröklési viszonyban egymással.

A StringBuffer osztály lényegesebb, publikus konstruktőrait, metódusait a 16.8. ábra mutatja.

| <b>StringBuffer</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <pre> StringBuffer() StringBuffer(length: int) StringBuffer(str: String) append (value: &lt;Type&gt;): StringBuffer append (value: &lt;type&gt;): StringBuffer capacity():int charAt(index: int): char delete(start: int, end: int): StringBuffer deleteCharAt(index: int): StringBuffer ensureCapacity(minimumCapacity: int) insert (offset: int, value: &lt;Type&gt;): StringBuffer insert (offset: int, value: &lt;type&gt; value): StringBuffer length():int replace(start: int, end: int, str: String): StringBuffer reverse(): StringBuffer setCharAt(index: int, ch: char) setLength(newLength: int) substring(start: int): String substring(start: int, end: int): String toString(): String </pre> |

16.8. ábra. StringBuffer osztály

## Konstruktörök

- ▶ `StringBuffer()`  
A létrehozott objektum az üres karakterláncot tartalmazza, kezdeti kapacitása 16 karakter.
- ▶ `StringBuffer(int length)`  
A létrehozott objektum az üres karakterláncot tartalmazza, kezdeti kapacitása `length` karakter.
- ▶ `StringBuffer(String str)`  
A létrehozott objektum a paraméterben megadott szöveget tartalmazza. Kezdeti kapacitása a lánc hossza + 16 karakter. Például:  

```
StringBuffer szoveg = new StringBuffer("Do re mi");
```
- Vigyázat! A `StringBuffer szoveg = "Kezdeti érték";` szintaktikailag hibás. A jobb oldal String típusú, ez értékkedés szerint nem kompatibilis a `StringBuffer` típussal.

## Kapacitás, hossz, index

- ▶ `int capacity()`  
Megadja az aktuális kapacitást. Ennyi karakter fér az objektumba.
- ▶ `int length()`  
Megadja a szöveg aktuális hosszát.
- ▶ `void ensureCapacity(int minimumCapacity)`  
Bővíti az aktuális kapacitást. Az egyes metódusok a kapacitást szükség szerint automatikusan bővítik. A programozó ezzel a metódussal előre meghatározhatja a kapacitást a hatékonyság érdekében. Az új kapacitás a `minimumCapacity` és a `2*capacity() + 2` közül a nagyobbik lesz.
- ▶ `void setLength(int newLength)`  
A hossz módosítása. Ha `newLength` kisebb, mint az aktuális hossz, akkor a metódus levágja a szöveget, ha nagyobb, akkor feltölti a \u0000 karakterekkel. Szükség esetén növeli a kapacitást. A szöveg hossza `newLength` lesz. (A nulla karakterek a konzolon nem látszódnak.)
- ▶ `char charAt(int index)`  
Visszaadja az `index` indexű karaktert.

## Bővítés

- ▶ `StringBuffer append (<Type> value)`
- ▶ `StringBuffer append (<type> value)`  
Sok túlterhelt metódusról van szó: a `Type` egy osztály, a `type` pedig egy tetszőleges primitív típus. minden esetben a `value` karakterlánc változata adódik hozzá a szöveg végéhez. A karakterlánc változatot az osztály, illetve primitív típus esetén a csomagoló

osztály `toString()` metódusa határozza meg. A visszatérési érték a megváltoztatott szöveg. Például:

```
StringBuffer szoveg = new StringBuffer("Szám=");
float f = 34.6f;
szoveg.append(f);
szoveg.append("*");
System.out.println(szoveg); // Szam=34.6*
```

- `StringBuffer insert (int offset,<Type> value)`
- `StringBuffer insert (int offset,<type> value)`

Hasonlóképpen működik, mint az `append` metódus, azzal a különbséggel, hogy a `value`-ból átalakított karakterláncot nem a szöveg végére helyezi, hanem beszúrja a megadott `offset` pozíciótól kezdődően. A visszatérési érték a megváltoztatott szöveg. Például:

```
StringBuffer szoveg = new StringBuffer("Osszesen Ft");
float f = 34.6f;
szoveg.insert(9,f);
System.out.println(szoveg); // "Osszesen 34.6 Ft"
```

## Törlés

- `StringBuffer deleteCharAt(int index)`

Az adott indexű karakter törlése a szövegből. A szöveg aktuális hossza eggyel csökken. A visszatérési érték a megváltoztatott szöveg.

- `StringBuffer delete(int start, int end)`

Részlánc törlése a szövegből. Az első törlendő karakter indexe `start`, az utolsóé `end-1`. A szöveg hossza `end-start` értékkel csökken. A visszatérési érték a megváltoztatott szöveg. Például:

```
StringBuffer szoveg = new StringBuffer("Bolha");
szoveg.delete(2,4);
System.out.println(szoveg); // Boa
szoveg.deleteCharAt(0);
System.out.println(szoveg); // oa
```

## Egyéb

- `StringBuffer replace(int start, int end, String str)`

A `start` és `end-1` közötti (`end` már marad) részláncot kicseréli `str`-re. A visszatérési érték a megváltoztatott szöveg.

- `StringBuffer reverse()`

Megfordítja az objektum szövegét. A visszatérési érték a megváltoztatott szöveg.

- ▶ `String substring(int start, int end)`
- ▶ `String substring(int start)`

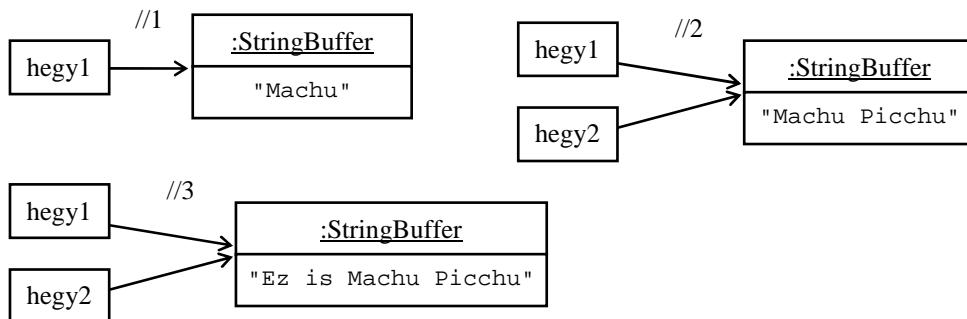
Visszaad egy új `String` objektumot, amely a `StringBuffer` objektum `start` és `end` közötti részlánca (az utolsó visszaadott karakter pozíciója `end-1`). A második esetben `end=length()`. Az eredeti objektum változatlan marad.

- ▶ `String toString()`

Visszaadja az objektum szövegét `String` formában.

A szöveget megváltoztató metódusok többségének van egy `StringBuffer` típusú visszatérési értéke. A visszaadott objektum maga a megváltoztatott szöveg, amelyet természetesen nem fontos felhasználnunk. Emlékezzünk vissza arra, hogy bármely függvény hívható eljárásoknál is – ekkor a visszatérési érték egyszerűen nem kerül felhasználásra. A `String` osztály metódusait nincs értelme eljárásoknál hívni, mivel ezek nem változtatják meg az objektumot, következésképpen csak a visszatérési érték lehet érdekes számunkra. A `StringBuffer` osztály metódusait azonban a legtöbb esetben eljárásoknál fogjuk meghívni. A következő példát a 16.9. ábra szemlélteti:

```
StringBuffer hegy1 = new StringBuffer("Machu"); //1
StringBuffer hegy2 = hegy1.append(" Picchu"); //2
hegy2.insert(0,"Ez is ");
System.out.println("Hegy1="+hegy1);
System.out.println("Hegy2="+hegy2);
```



16.9. ábra. A `StringBuffer` változásának nyomonkövetése

//1-ben létrehozzuk a `hegy1` objektumot, melynek tartalma "Machu". //2-ben a `hegy1` objektumhoz az `append` metódussal hozzáfűzzük a " Picchu" szöveget, és a megváltoztatott szöveget adjuk értékül a `hegy2` objektumnak. `hegy1` és `hegy2` tehát ugyanazt a "Machu Picchu" tartalmú objektumot azonosítják. //3-ban a `hegy2` objektumba beszúrjuk az

"Ez is " szöveget, ami természetesen hegy1-et is érinti. Végeredményképpen tehát hegy1 és hegy2 ugyanaz az objektum, és értékük "Ez is Machu Picchu".

Sajnos a String és a StringBuffer osztályok értékkadás szerint nem kompatibilis típusok. Van ugyan egy közös ősük, az Object, de ezen kívül semmi közük sincs egymáshoz. Egyik sem örökölődik a másikból, és nehezíti a használatot, hogy a gyakran használatos metódusok egy része (mint a keresés) a String osztályban, más része (mint a törlés és a beszúrás) a StringBuffer osztályban kapott helyet. A következő feladatban például a cél érdekében a szöveget ide-oda kell alakítgatnunk:

### Feladat – Csere

Kérjünk be egy szöveget, majd cseréljük ki az összes & jelet az and szóra!

### Forráskód

```
import extra.*;
public class Csere {

 public static void main(String[] args) {
 StringBuffer szoveg =
 new StringBuffer(Console.readLine("Szöveg: ")); //1
 int poz = szoveg.toString().indexOf('&'); //2
 while (poz!=-1) {
 szoveg.replace(poz, poz+1, "and"); //3
 poz = szoveg.toString().indexOf('&'); //4
 }
 System.out.println(szoveg); //5
 }
}
```

### A program egy lehetséges futása

|                     |
|---------------------|
| Szöveg: O & te & en |
| O and te and en     |

### A program elemzése

A feladatot úgy tudjuk megoldani, hogy rákeresünk az összes & jelre, majd azokat egyenként kicseréljük az and szóra. Ezt a cserét csak a StringBufferben tudjuk megoldani, ezért //1-ben a beolvasott szöveget egy StringBufferben helyezzük el. A Console.readLine() metódus egy Stringet ad vissza, azt nem lehetjük át közvetlen értékkadással a StringBufferbe, csak konstruktoron keresztül. A beolvasott szövegen most keresünk kell. Mivel a StringBufferben nincs erre alkalmas metódus, a szöveget //2-ben ideiglenesen átalakítjuk Stringgé a StringBuffer.toString() metódussal. A metódus visszatérési értékében végezzük el a keresést. Miután megtaláltuk az & pozíóját (poz), visszatérünk a StringBufferhez, hogy elvégezhessük a cserét (//3-ban kicseréljük a poz. karaktert az "and" láncra). Ezután újra keresünk (//4), és ezt a műveletsorozatot addig folytatjuk, amíg van

& jel. Ha már nincs (`poz=-1`), akkor kiírjuk a `StringBuffer` tartalmát (`//6`). A `print` metódus bármit ki tud írni.

❖ Vigyázat! A megoldás nem működik bármilyen cserével! Próbálja ki például úgy, hogy az & jelet `&&-re` cseréli! Mi történik? A program végtelen ciklusba fog esni. Mi a feladat általánosabb megoldása? Lásd a 16.5. feladatot!

**Megjegyzés:** A feladat megoldható csak `String` típusú objektumok segítségével is, de ebben az esetben rengeteg szemét keletkezik.

## 16.8. Objektum átadása paraméterként

Egy referencia típusú változó (objektum) akkor adható át aktuális paraméterként egy referencia típusú formális paraméternek, ha **az aktuális paraméter értékadás szerint kompatibilis a formális paraméterrel**. A metódus megkapja az objektum referenciáját. A metódus a mutatott objektumot megváltoztathatja, de nem változtathatja meg az eredeti mutatót.

Objektum paraméterként való átadásakor nem az egész objektum másolódik át, hanem csak a referencia. A formális és aktuális paraméterek tehát a metódusba való belépéskor ugyanarra az objektumra mutatnak, és az átmásolt referenciával elvileg éppúgy meg tudjuk változtatni az objektumot, mint az eredetivel. Megtehetjük, hogy a formális paraméter értékét a metódus futása közben megváltoztatjuk (átirányítjuk egy másik objektumra), de ettől az eredeti objektum érintetlen marad. Nézzünk erre két példát!

### Feladat – Objektum paraméter

Írunk eljárást, amely a paraméterként megkapott szöveget széthúzza, vagyis karakterei közé beszűr egy-egy szóközt!

#### Forráskód

```
import extra.*;

public class ObjektumParameter {

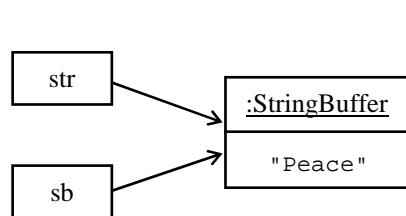
 static void szethuz(StringBuffer str) {
 for (int i=str.length()-1; i>0; i--)
 str.insert(i, ' ');
 str = null; // csak demonstrációs célú utasítás
 }

 public static void main(String[] args) {
 StringBuffer sb = new StringBuffer("Peace");
 szethuz(sb);
 System.out.println("*"+sb+"*"); // -> "*P e a c e*"
 }
}
```

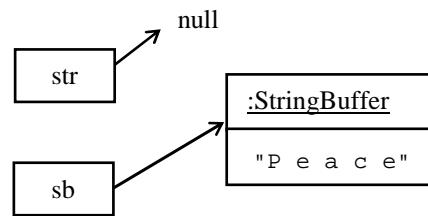
### A program elemzése

A main metódusban sb egy `StringBuffer` objektumot azonosít. sb-t paraméterként átadjuk a szethuz metódusnak, ott tehát str is ugyanezt az objektumot azonosítja. A metódusban az str által mutatott objektumot megváltoztatjuk (értéke `peace` helyett `p e a c e` lesz), de ez ugyanaz az objektum, mint amelyet a main metódusban sb azonosít. A metódus végén str értékét null-ra állítjuk, ez azonban nem befolyásolja az sb által mutatott objektumot. Az aktuális és formális objektum paraméter viselkedését a 16.10. ábra mutatja: a metódusba való belépéskor sb és str ugyanazt az objektumot azonosítja, str ezt a közös objektumot változtatja meg. A metódusból való kilépés előtt str-nek értékül adjuk a null referenciát, de ettől még sb továbbra is a megváltoztatott objektumot azonosítja. A metódusból való kilépéskor str elvész.

A metódusba való belépéskor:



A metódusból való kilépés előtt:



16.10. ábra. Az objektum paraméter viselkedése

### Feladat – Palindróma

Kérjünk be mondatokat az üres mondat végjelig! Állapítsuk meg minden egyes mondatról, hogy az palindróma-e, azaz a szöveg visszafelé olvasva ugyanaz-e!

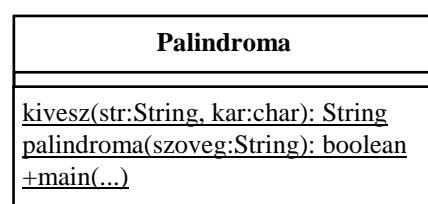
Feltételezzük, hogy a mondatban nincsenek írásjelek, és a vizsgálatot tegyük függetlenné a szóközöktől és a kis/nagybetűktől!

### A feladat nagyvonalú terve

```

in: szöveg
while szöveg != ""
 szóközök kivétele a szövegből
 nagybetrüre alakítás
 palindróma vizsgálat
 eredmény kiírása
 in: szöveg
end while

```



A szóközök kivételére írunk egy általánosan is használható `kivesz()` függvényt, mely a paraméterként megkapott szövegből kiveszi az összes paraméterként megkapott karaktert! A szöveg nagybetűre alakítására van a `String` osztályban egy metódus, a `toUpperCase()`. A palindróma vizsgálatára írunk egy `palindroma()` függvényt, amely bármely szövegről megállapítja, hogy az palindróma-e! Nézzük a programot:

### Forráskód

```
import extra.*;
public class Palindroma {

 // str-ból az összes kar törlése:
 static String kivesz(String str,char kar) {
 StringBuffer sb = new StringBuffer(str);
 int p;
 while ((p=sb.toString().indexOf(kar)) != -1)
 sb.deleteCharAt(p);
 return sb.toString();
 }

 // true, ha a szöveg palindróma:
 static boolean palindroma(String szoveg) {
 szoveg = kivesz(szoveg,' ').toUpperCase();
 StringBuffer sb = new StringBuffer(szoveg);
 return szoveg.equals(sb.reverse().toString());
 }

 public static void main (String args[]) {
 // Szövegek beolvasása és feldolgozása:
 String szoveg;
 while (!(szoveg=Console.readLine("Szöveg: ")).equals("")){
 // Palindróma vizsgálat:
 if (palindroma(szoveg))
 System.out.println("Palindróma");
 else
 System.out.println("Nem palindróma");
 }
 }
}
```

### A program egy lehetséges futása

|                        |
|------------------------|
| Szöveg: Kuka           |
| Nem palindróma         |
| Szöveg: Géza kék az ég |
| Palindróma             |
| Szöveg:                |

### A program elemzése

A `kivesz()` metódusnak két paramtere van: az `str`-ben található `kar` karaktereket kiveszi, és ez lesz a metódus visszatérési értéke. A karakterek kivételéhez a szöveget egy

`sb:StringBuffer` objektumba tesszük. Ahhoz, hogy megvizsgáljuk, van-e a szövegben karakter, átalakítjuk a szöveget `Stringgé`, mert csak a `String` osztályban van `indexOf()` függvény, és amelyik pozíciót találtunk karaktert, azt a pozíciójú karaktert kiveszük az `sb` objektumból. Ezt addig ismételjük, amíg az `sb`-ben még van a kérdéses karakterből. Végül `sb-t` átalakítjuk `Stringgé`, és ez lesz a függvény visszatérési értéke.

## 16.9. Primitív típusok csomagolása

A primitív típusú változók nem objektumok. Annak érdekében, hogy a primitív típusú adatokat objektumokként kezelhessük, a `java.lang` csomag ún. csomagoló osztályokat (wrapper classes) definiál: minden primitív típushoz tartozik egy osztály, mely a megfelelő típust becsomagolja. Ennek megfelelően a csomagoló osztályok a következők:

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

A `Character` és a `Boolean` osztályok közvetlenül az `Object`-ből származnak, a többi hat numerikus osztálynak van egy közös őse, az absztrakt `Number` osztály. A csomagoló osztályok példányai változtathatatlanok, vagyis állapotuk életük végéig az marad, amit inicializáláskor megadtunk.

A csomagoló osztályok segítségével lehetőségünk van arra, hogy

- a primitív típusú változókat objektumokként használjuk;
- a megfelelő primitív típusokkal kapcsolatban információt kérjünk.

A csomagoló osztályokban nagyon sok a hasonló elven működő metódus, ezért azokat nem osztályonként, hanem működés szerint csoportosítva tárgyaljuk. Az egyes metódusok leírásai osztályonként megtalálhatók az API dokumentációiban.

Az összes numerikus csomagoló osztályra vonatkozik, hogy ha egy metódusban egy szöveg nem alakítható át a megfelelő primitív számmá, akkor `NumberFormatException` kivétel keletkezik.

### Konstruktörök

► `<Type> (<type> value)`

Itt `Type` bármelyik csomagoló osztály lehet. minden csomagoló osztálynak van egy olyan konstruktora, amelynek paramétere a megfelelő primitív érték. A `type` a `Type`-nak megfelelő primitív típus. Például:

```
Character cObj = new Character('A');
Boolean bObj = new Boolean(true);
Integer iObj = new Integer(30);
Double dObj = new Double(6099.8);
```

► <Type> (String s)

Minden csomagoló osztálynak (kivéve a Characternek) van egy olyan konstruktora, amelynek paramétere a megfelelő primitív érték szöveges formája. Például:

```
Boolean b1Obj = new Boolean("True"); // kis/nagybetű mindegy
Boolean b2Obj = new Boolean("t"); // false (ha nem true)
Integer iObj = new Integer("30");
Double dObj = new Double("6099.8");
```

### Példánymetódusok

► String **toString()**

Ez minden csomagoló osztályban megtalálható. Az Object megfelelő metódusát írja felül úgy, hogy visszaadja az objektum szöveges formáját. Például:

```
Double dObj = new Double(6099.8);
String s = dObj.toString(); // s=="6099.8"
```

► <type> <type>Value()

Itt type egy primitív típust jelöl. minden csomagoló osztályban megvan a megfelelő metódus (Character-ben char **charValue()**, Integer-ben int **intValue()** stb.). A példánymetódus visszaadja az objektum primitív típusú értékét. Ezenkívül minden numerikus csomagoló osztály tartalmazza az összes többi primitív típusra átalakító metódust is: bármelyik numerikus objektumnak ki lehet tehát kérni annak típuskényszerített primitív értékét. Például:

```
Character cObj = new Character('A');
Integer iObj = new Integer(30);
char c = cObj.charValue(); // c=='A'
int i = iObj.intValue(); // i==30
double d = iObj.doubleValue(); // d==30.0
```

► boolean **equals(Object obj)**

Minden csomagoló osztály felülírja az Object osztály **equals** metódusát. Ez a metódus megvizsgálja, hogy az objektum egyenlő-e a paraméterben megadottal. A visszaadott érték csak akkor true, ha a két objektum osztálya ugyanaz, és az objektumok azonos állapotúak. Például:

```
Integer iObj = new Integer(30), jObj = new Integer(30);
if (iObj.equals(jObj)) ... // true
if (iObj.equals("30")) ... // false, nem ugyanaz az osztály
```

### Statikus metódusok

► static <Type> **valueOf(String s)**

A Type-nak megfelelő csomagoló osztályban van (kivéve a Characterben). A karakterláncból egy Type típusú objektumot gyárt.

```
Integer iObj, jobj;
iObj = Integer.valueOf("30");
jobj = Integer.valueOf("45D"); // NumberFormatException!
```

Most nézzük sorra, milyen egyéb deklarációk találhatók az egyes csomagoló osztályokban:

### Boolean osztály

- static final Boolean TRUE = new Boolean(true);
  - static final Boolean FALSE = new Boolean(false);
- Ez két Boolean típusú konstans objektum.

### Character osztály

- static final char MIN\_VALUE = '\u0000';
- static final char MAX\_VALUE = '\uffff';

A char típus által reprezentált legkisebb, illetve legnagyobb unikód értékű konstans.

- static boolean isUpperCase(char ch) // Nagybetű?
  - static boolean isLowerCase(char ch) // Kisbetű?
  - static boolean isDigit(char ch) // Szám?
  - static boolean isLetter(char ch) // Betű?
  - static boolean isLetterOrDigit(char ch) // Betű vagy szám?
  - static boolean isISOControl(char ch) // Vezérlő karakter?
  - static boolean isSpaceChar(char ch) // Szóköz
  - static boolean isWhitespace(char ch) // Fehér szóköz?
- Visszaadja, hogy a karakter nagybetű (upper case), kisbetű (lower case) stb.
- static char toUpperCase(char ch)
  - static char toLowerCase(char ch)

Visszaadja a karakter nagybetűs, illetve kisbetűs formáját.

#### Feladat – Milyen karakter?

Kérjünk be karaktereket '-' végig! Állapítsuk meg minden egyes karakterről, hogy az szám, betű (azon belül kisbetű vagy nagybetű), szóköz, fehér szóköz vagy egyéb!

### Forráskód

```
import extra.*;
public class MilyenKarakter {
 public static void main(String[] args) {
 char kar = Console.readChar("Karakter:");
 while (kar != '-') {
 if (Character.isDigit(kar))
 System.out.println("Szám");
 else if (Character.isLetter(kar)) {
 if (Character.isUpperCase(kar))
 System.out.println("Nagybetű");
 else
 System.out.println("Kisbetű");
 }
 }
 }
}
```

```

 else if (Character.isSpaceChar(kar))
 System.out.println("Szóköz");
 else if (Character.isWhitespace(kar))
 System.out.println("Fehér szóköz");
 else
 System.out.println("Egyéb");
 kar = Console.readChar("Karakter:");
 }
}
}

```

### A program egy lehetséges futása

```

Karakter:!
Egyéb
Karakter:A
Nagybetű
Karakter:-

```

### Numerikus csomagoló osztályok

- ▶ static final <type> MIN\_VALUE = <a típusnak megfelelő érték>;
- ▶ static final <type> MAX\_VALUE = <a típusnak megfelelő érték>;

A megfelelő típus által reprezentált legkisebb, illetve legnagyobb szám értékű konstans:

```

System.out.println("Legkisebb byte: "+Byte.MIN_VALUE); // -128
System.out.println("Legnagyobb int: "+Integer.MAX_VALUE);
// -> 2147483647

```

- ▶ static <type> parse<Type>(String s)

E statikus metódus megfelelő változata minden egyik numerikus csomagoló osztályban szerepel. Visszaadja a szöveg által reprezentált primitív típusú értéket (kivétel: int esetén a metódus neve nem parseInteger hanem parseInt). Például:

```

int b = Byte.parseByte("3"); // b==3
int i = Integer.parseInt("50"); // i==50
double d = Double.parseDouble("3.14"); // d==3.14

```

### Feladat – Összegzés

Kérjünk be egy összeget szöveg formájában, ahol a valós számok + jelekkel vannak elválasztva! Írjuk ki egymás alá az egyes összeadandókat 8 hosszon, két tizedessel jobbra igazítva, majd az egészet aláhúzva írjuk ki az összeget is!

### Forráskód

```

import extra.*;

public class Osszegzes {
 public static void main(String[] args) {
 String osszegStr = Console.readLine
 ("Írjon be egy összeget 9.9+9.9+9.9 formában:\n");
 // Így könnyebb lesz feldolgozni az utolsó számjegyet:
 osszegStr = osszegStr+"";
 double osszeg = 0, szam;
 }
}

```

```

int kezd = 0; //2
int p = osszegStr.indexOf('+'); //3
while (p >= 0) {
 szam = Double.parseDouble
 (osszegStr.substring(kezd,p)); //4
 System.out.println(Format.right(szam,8,2)); //5
 osszeg += szam;
 kezd = p+1; //6
 p = osszegStr.indexOf('+',kezd); //7
}
System.out.println("-----");
System.out.println(Format.right(osszeg,8,2));
}
}

```

### A program egy lehetséges futása

Írjon be egy összeget 9.9+9.9+9.9 formában:  
 123.5+1+99+5.134  
 123.50  
 1.00  
 99.00  
 5.13  
 -----  
 228.63

### A program elemzése

A bekért szöveg végére teszünk egy + jelet, így a feldolgozás addig folytatódhat, amíg van + jel (//1). A következő szám részlánca mindenkor a kezdő pozíció (kezd) és a következő + jel pozíciója (p) közötti rész lesz, ezeket az értékeket a ciklus előtt (//2, //3), valamint a ciklus végén (//7, //8) mindenkor számoljuk. A ciklusban kivesszük a soron következő részláncot, és azt rögtön double értékké alakítjuk (//4), majd igazítva kiírjuk az értéket (//5), és elvégezzük az összegzést (//6).

## 16.10. StringTokenizer osztály

A StringTokenizer osztály segítségével egy szöveg könnyen egységekre (részekre, tokenekre) bontható. Az egyes egységeket egy vagy több elválasztó karakter (elválasztó jel, delimiter) különíti el egymástól. Ezek az elválasztó karakterek alapértelmezésben a fehér szóközök (szóköz, TAB, CR, LF és FF karakterek), de az elválasztó karakterek halmazát mi magunk is megadhatjuk.

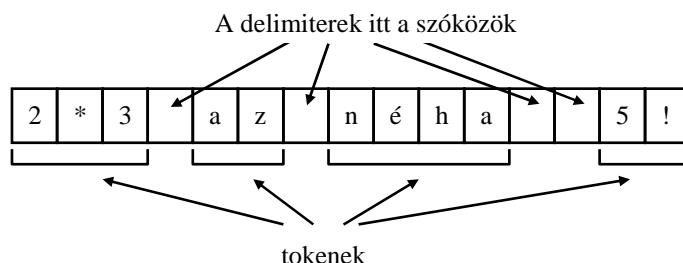
A String darabolását a 16.11. ábra mutatja.

Példák:

- ◆ Szöveg:  $2 * 3$  az néha  $5!$ 
  - Ha az elválasztó karakter: szóköz, akkor az eredmény 4 egység:  **$2 * 3$ , az, néha** és  **$az\ 5!$**  (16.11. ábra)
  - Ha az elválasztó karakter:  $*$ , akkor az eredmény 2 egység:  **$2$**  és a  **$3\ az\ néha\ 5!$**
- ◆ Szöveg:  $1+34+567*3-5$ 
  - Ha az elválasztó karakterek:  $+-* /$ , akkor az eredmény 5 egység:  **$1, 34, 567, 3$**  és  **$5$**

Az alapértelmezés szerinti elválasztókkal a mondat szavakra bomlik.

Egy szöveg felbontását úgy tudjuk elvégezni, hogy létrehozunk egy StringTokenizer típusú objektumot, melyet a felbontandó szöveggel inicializálunk. A szöveg következő részét minden a nextToken() metódus adja meg. A szöveg végéhez érve a hasMoreTokens() függvény false értékű, ilyenkor a nextToken() újból meghívása NoSuchElementException Exception kivételt vált ki. A szövegen csak egyszer lehet végigmenni, ezért a szöveg többszöri felbontásához minden új daraboló objektumot kell létrehozni. Az elválasztó karaktereket minden közben meg lehet változtatni.



16.11. ábra. String darabolás

## Konstruktörök

- `StringTokenizer(String str, String delim, boolean returnTokens)`  
Létrehoz egy olyan objektumot, mely az str szöveget fogja majd felbontani. A delim által tartalmazott karakterek bármilyen sorozata elválasztja az egységeket. Ha returnTokens értéke true, akkor a felbontó eljárás az elválasztójeleket is visszaadja (egyesével) mint egységet. Ha returnTokens értéke false, akkor a nextToken() minden csak az „értelmes” egységeket adj a vissza (a szöveg nem tartalmaz elválasztó karaktert).

- ▶ `StringTokenizer(String str, String delim)`  
Létrehoz egy olyan objektumot, mely az `str` szöveget fogja majd felbontani. A `delim` által tartalmazott karakterek sorozata elválasztja az egységeket. `returnTokens` értéke `false`.
- ▶ `StringTokenizer(String str)`  
Létrehoz egy olyan objektumot, mely az `str` szöveget fogja majd felbontani. `delim` értéke "`\t\n\r\f`" (fehér szóközök), `returnTokens` értéke `false`.

## Metódusok

- ▶ `boolean hasMoreTokens()`  
true-t ad vissza, ha van még rész (egység).
- ▶ `String nextToken()`  
Visszaadja a következő egységet. Ha nincs több egység, `NoSuchElementException` kivétel keletkezik.
- ▶ `String nextToken(String delim)`  
Először átvált a `delim` elválasztójel-készletre, majd ennek megfelelően visszaadja a következő egységet, mint ahogyan azt a `nextToken()` teszi. A továbbiakban ezek az elválasztók lesznek érvényben.
- ▶ `int countTokens()`  
Visszaadja, hogy hány rész van még a szövegen.

### Feladat – Szavak

Kérjünk be egy mondatot! Írjuk ki, hány szó van benne, majd írjuk ki a szavakat is természetes sorszámmal ellátva (1-től), mindegyiket külön sorba!

### Forráskód

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

public class Szavak {

 public static void main(String[] args) {
 String mondat = Console.readLine("Mondat: ");
 StringTokenizer st = new StringTokenizer(mondat);

 System.out.println("Szavak száma: " + st.countTokens());
 int sorszam = 0;
 while (st.hasMoreTokens())
 System.out.println(++sorszam + ": " + st.nextToken());
 }
}
```

### A program egy lehetséges futása

```
Mondat: Remélem, jól szétbontod ezt a mondatot!
Szavak száma: 6
1: Remélem,
2: jól
3: szétbontod
4: ezt
5: a
6: mondatot!
```

### Feladat – Dátumbontás

Az `eeee.hh.nn.` formátumú szöveg egy dátumot tartalmaz. Határozzuk meg az évet, a hónapot és a napot egész értékként!

Most egyetlen elválasztójel van, a pont. A részek szövegek, melyeket az `Integer` osztály `parseInt` függvényével primitív típusú `int` értékekékké alakítunk.

### Forráskód

```
import java.util.*;

public class DatumBontas {
 public static void main(String[] args) {
 String datumStr = "1978.03.15.";
 StringTokenizer dt = new StringTokenizer(datumStr, ".");
 int ev = Integer.parseInt(dt.nextToken());
 int ho = Integer.parseInt(dt.nextToken());
 int nap = Integer.parseInt(dt.nextToken());
 System.out.println("Év: " + ev + " Hó: " + ho + " Nap: " + nap);
 }
}
```

### A program futása

```
| Év: 1978 Hó: 3 Nap: 15
```

### Tesztkérdések

- 16.1. Mely állítások igazak a `toString()` metódusra vonatkozóan? Jelölje meg az összes igaz állítást!
  - a) minden objektumot meg lehet szólítani a `toString()` metódussal függetlenül attól, hogy az adott osztályban azt definiálták-e, vagy sem.
  - b) A `toString()` minden esetben az objektum adatait adja vissza szöveges formában.
  - c) minden osztályban kötelezően szerepelnie kell egy saját `toString()` metódusnak.
  - d) A `System.out.println()` metódus paraméterében bármilyen objektum szerephet.

- 16.2. Jelölje meg az összes helyes állítást a következők közül!
- a) A `null` referencia egy speciális, üres objektumra mutat.
  - b) A referencia típusú változó azonosítójával az objektum memóriaheleyre nem közvetlenül, hanem közvetve hivatkozunk.
  - c) Egy objektum deklarálásakor a rendszer az objektum számára memóriaheleyet foglal.
  - d) Az objektum osztálya futás közben megváltoztatható.
- 16.3. Jelölje meg az összes helyes állítást a következők közül!
- a) Az inicializálást végző metódust konstruktornak nevezzük.
  - b) A konstruktornak elvileg többféle paramétere zése is lehet.
  - c) A Javában a konstruktur neve nem az osztály nevével egyezik meg, hanem a létrehozandó objektuméval.
  - d) Az objektum adatainak a konstruktornal kezdeti értékeket adhatunk.
- 16.4. Jelölje meg az összes helyes állítást a következők közül!
- a) Két objektum azonos, ha állapotaik megegyeznek.
  - b) A `new` operátorral egy objektumot hozhatunk létre.
  - c) Egy metódus meg tudja megváltoztatni a paraméterben megkapott objektumot.
  - d) A hivatkozás nélküli objektumokat az automatikus szemétyűjtő időnként megsemisíti.
- 16.5. Adott a következő deklaráció:
- ```
String szo="Virágot";
```
- Melyik kifejezés adja vissza a "rág" részláncot a szo-ból? Jelölje meg az egyetlen jó választ!
- a) `szo.substring(3,3)`
 - b) `szo.substring(2,3)`
 - c) `szo.substring(2,4)`
 - d) `szo.substring(2,5)`
- 16.6. Jelölje meg az összes helyes állítást a következők közül!
- a) A `Character` osztály közvetlenül az `Object` osztályból származik.
 - b) Az `Integer` osztály közvetlenül az `Object` osztályból származik.
 - c) A `Boolean` primitív típus.
 - d) A Javában az objektumok referencia típusú változók.

Feladatok

String osztály

- 16.1. Kérjen be konzolról egy szöveget! Írja ki a szöveget
- a) (A) csupa kisbetűvel!
 - b) (A) tízszer egymás után, a „+” jellel elválasztva!
 - c) (A) úgy, hogy az összes „_” karakter helyett kötőjel szerepel!
 - d) (B) úgy, hogy a szöveg első és második fele fel van cserélve (középső karakter marad)! Például: Buldozer → ozerBuld

- e) (B) úgy, hogy az első szóköz előtti és utáni rész fel van cserélve!
 Például: Ady Endre → Endre Ady
(SzovegAlakit.java)
- 16.2. Kérjen be konzolról szövegeket egy adott végjelig, majd írja ki közülük a leghosszabbat!
 a) (A) Több egyenlő hosszúságú szöveg esetén az utolsó leghosszabbát írja ki!
 b) (B) Több egyenlő hosszúságú szöveg esetén írja ki mindeneket! Tipp: Gyűjtse az egyenlő hosszúságú szövegeket úgy, hogy azokat koncatenálja szóközökkel is beiktatva! Ha ezeknél talál egy rövidebbet, akkor kezdje előlről a gyűjtést!
(Leghosszabb.java)
- 16.3. Írjon metódust, amely megszámolja, hogy a paraméterben megadott szövegben
 a) (A) hány szóköz van!
 b) (B) hány „and” szó van! Nem számít, hogy előtte, illetve utána van-e szóköz.
 c) (B) hány, második paraméterként megadott szó szerepel! (Előző feladat általánosítása.)
(Megszamol.java)
- 16.4. (C) Kérjen be egy szöveget, majd cserélje le benne az összes ékezetes betűt ékezet nélkülire! (*Ekezetcsere.java*)

StringBuffer osztály, objektum átadása paraméterként

- 16.5. (B) Írjon egy olyan metódust, amely a paraméterként megadott karakterláncban bizonyos karakterláncokat másikra cserél! A kicserélendő és beszúrandó karakterláncokat is paraméterként adjuk meg! A metódus bármilyen két láncra működjön!
(Szocserje.java)
- 16.6. (B) Írjon egy olyan függvényt, amely a paraméterként megadott karakterláncból kiveszi a második paraméterként megadott karakterláncban található összes karaktert! A függvény az eredeti láncot hagyja változatlanul, és adja vissza a megváltoztatott láncot!
(Kivesz.java)
- 16.7. (B) Írjon egy olyan metódust, amely egy szöveget a paraméterként megadott hosszúságúra alakít: ha kell, levágja, ha kell, kiegészíti szóközökkel! Hívása pl.:
`szoveg=lenString(szoveg, 6);` (*LenString.java*)
- 16.8. (C) Írjon egy olyan metódust, amely visszaadja egy szám bitképét karakterlánc formájában! A bevezető nullák ne szerepeljenek a láncban (például 5 == "101")! Egy másik metódus adja vissza a bitkép által reprezentált számot! (*BitKep.java*)
- 16.9. (B) A gyerekek körében jól ismert a „Tuvudsza ívígy beveszévelni?” típusú beszélgetés. Tanítsa meg a számítógépet így beszélni! A beolvasott mondatot alakítsa át úgy,

hogy minden magánhangzó után szúrjon be egy v betű és az előző magánhangzót!
(*Tuvudsz.java*)

16.10. (C) Írjon egy olyan metódust, amely kódol egy szöveget: fordítsa meg minden karakter 3. és 4. bitjét: a 0-ból legyen 1-es és fordítva! Készítsen megfejtő metódust is!
(*Kodolas.java*)

16.11. (C) Kérjen be egy szöveget, majd vegye ki belőle az összes zárójeles megjegyzést! Például: Ez a kutya (házörzö) szép (ápolt). → Ez a kutya szép. (*MegjegyzesKivesz.java*)

Primitív típusok csomagolása

16.12. (A) Kérjen be egy szöveget! Írja ki, hogy a szövegben összesen hány darab betű, hány szám, hány fehér szóköz, illetve hány egyéb karakter szerepel! (*KarEloszlas.java*)

16.13. (C) Kérjen be egy szöveget! A szövegben szereplő összes dollárösszeget váltsa át forintra (például ha 1 \$=280 Ft, akkor a 21 \$ helyett 5880 Ft szerepeljen)! Tegyük fel, hogy a dollárösszeg formája: egy egész szám, egy szóköz és a \$ jel. Írja ki az átalakított szöveget! (*Atvaltas.java*)

16.14. (C) Kérjen be egy nevet, mely akárhány résznévből állhat. Csak akkor fogadja el a nevet, ha az legalább két résznévből áll (a részneveket legalább egy szóköz választja el)! Vegye le a szóközöket a név elejéről és végéről, a résznevek között pedig csak egy szóközt hagyjon! A nevet alakítsa át úgy, hogy minden résznév nagybetűvel kezdődjön, a többi pedig kisbetű legyen! (*NevAlakit.java*)

StringTokenizer osztály

16.15. (A) Írjon egy függvényt, mely megszámolja, hány szó van a paraméterként megadott szövegben! (A szavakat egy vagy több fehér szóköz választja el egymástól.) (*SzavakSzama.java*)

16.16. (B) Bontsuk mondatokra egy paragrafus tartalmát! Egy mondatot a pont, a felkiáltójel vagy a kérdőjel zárja le (együtt nem alkalmazhatók). Írjuk ki egyenként a mondatokat és utána zárójelben a benne levő szavak számát! (*Paragrafus.java*)

16.17. (C) Egy szövegsor egy hallgató nevét és Programozás érdemjegyeit tartalmazza. A hallgató neve után egy kettőspont áll, majd szóközzel elválasztva jön akárhány darab 1 és 5 közötti érdemjegy. Kérjük be ezt a sort, majd írjuk ki a hallgató nevét, és jegyeinek átlagát! (*JegyekAtлага.java*)

17. Osztály készítése

A fejezet pontjai:

1. OO paradigmá – Emlékeztető
 2. Első mintaprogram – Raktárprogram
 3. Második mintaprogram – Bank és a „Jószerencse”
 4. Az osztály felépítése, az osztály deklarációi
 5. Osztálytag, példánytag
 6. Azonosító, hivatkozási kör, takarás
 7. Változók alapértelmezés szerinti kezdeti értékei
 8. A this objektumreferencia
 9. Konstruktörök
 10. Inicializálók
-

Az előző fejezetben az API kész osztályait használtuk; most mi magunk fogunk osztályt készíteni. A megírt osztályt aztán ugyanúgy példányosíthatjuk, mint bármely más API osztályt. Az osztálykészítés szabályait két mintaprogram segítségével szemléltetjük: az első program bevezető jellegű – ott a vezérlő objektum egyetlen, osztálytagokat nem tartalmazó objektummal áll kapcsolatban; a második programban a vezérlő már két objektumnak üzen, és az üzeneteket fogadó objektum osztálytagokat is tartalmaz. A fejezet összes pontjában e két mintaprogramra fogunk hivatkozni.

17.1. OO paradigmá – Emlékeztető

Először is összefoglaljuk a II. rész, OO paradigmában tárgyalt, ide tartozó fontosabb definíciókat:

Példánydeklaráció: A példányonként (objektumonként) helyet foglaló változók a **példányváltozók**, más néven példányadatok. Az osztály azon metódusait, amelyek példányonkon (példányváltozókon) dolgoznak, **példánymetódusoknak** nevezzük. (Lásd 5.7.)

Osztálydeklaráció: Az osztályváltozó az osztály saját változója, az egyes példányokban nem szerepel. Az **osztálymetódus** az osztály saját metódusa, amely csak osztályváltozókon dolgozik. Egy osztályt csak osztálymetódussal lehet megszólítani; egy objektumot meg lehet szólítani példánymetódussal és osztálymetódussal egyaránt. (Lásd 5.8.)

Objektumok, osztályok sztereotípusai: Egy objektum (osztály) sztereotípusának nevezzük annak fajtáját, jellegét. A sztereotípusok között nem minden húzható éles határ, és előfordulhat, hogy egy objektumra két sztereotípus is jellemző. Az objektumok, osztályok alapvető sztereotípusai: információhordozó, határ, kontroll, konténer. (Lásd 5.11.)

Objektumdiagram (példánydiagram): Elsősorban objektumokat és azok kapcsolatait ábrázoló diagram. Az objektumdiagramon osztály is feltüntethető (az osztálydeklarációk miatt). (Lásd 6.1.)

Két osztály közötti egy-egy kapcsolat: az egyik osztály egy példánya a másik osztály legfeljebb egy példányával állhat kapcsolatban. A másik osztályra ugyanez vonatkozik. (Lásd 6.2.)

Osztálydiagram: Az osztálydiagram a feladatban szereplő objektumok osztályait és azok (társítási és öröklési) kapcsolatait ábrázolja. A rendszert egyetlen osztálydiagram írja le. Az osztálydiagramon szerepel a rendszer összes olyan objektumának osztálya, melyet a forráskódban meg kell valósítani (le kell kódolni). Az osztálydiagramon az API osztályait csak akkor szokás feltüntetni, ha az szükséges a rendszer megértéséhez. (Lásd 6.2.)

Együttműködési diagram: Olyan objektumdiagram, amelyen feltüntetjük az egyes objektumoknak küldött üzeneteket. Az együttműködési diagramon osztályok is szerepelhetnek az osztályváltozók, illetve osztálymetódusok használata miatt. A diagram üzeneteit a végrehajtás sorrendjében be lehet sorszámozni. Az együttműködési diagram segítségével szemléletessé tehetjük az egyes használati eseteket, illetve operációkat működés közben. Egy osztálydiagramhoz több együttműködési diagram is tartozhat. Az együttműködési diagram az osztálydiagram egy példánya, a működő program egy pillanatfelvétele. (Lásd 6.2.)

Két osztály közötti egy-egy kapcsolat megvalósítása: A kliens osztályban létrehozzuk a szerver objektumra mutató referenciát. (Lásd 6.3.)

Láthatóság: Az osztály deklarációi a következő láthatósággal (hozzáférési móddal) rendelkezhetnek: **Nyilvános** (public, +): minden kapcsolatban álló kliens eléri és használhatja;

Védett (protected, #): hozzáférés csak öröklésen keresztül lehetséges; **Privát** (private, -): az osztály privát deklarációja, csak az osztály saját metódusai férhetnek hozzá. (Lásd 7.5.)

☞ Nézze át a II. rész, OO paradigma megfelelő részeit!

17.2. Első mintaprogram – Raktárprogram

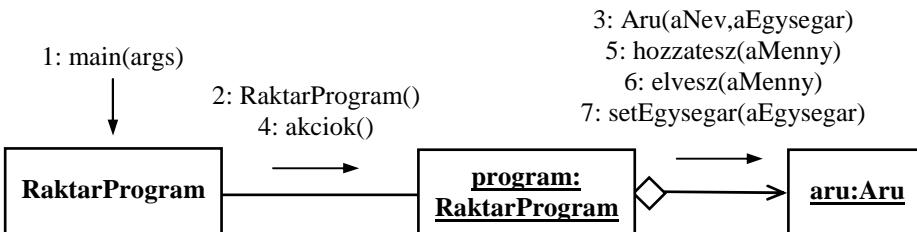
Feladat

Adott egy zöldségraktár, melyben pillanatnyilag egyetlen árut, paradicsomot raktározzunk. A raktárba gyakran teszünk be, illetve veszünk ki onnan paradicsomot. A paradicsom pillanatnyi egységára 300 Ft, de ez változhat. Készítsünk olyan programot, mely segítségével rögzíteni tudjuk a megfelelő adatokat, és bármikor jelentést tudunk adni a paradicsom aktuális mennyiségről, egységáráról és értékéről!

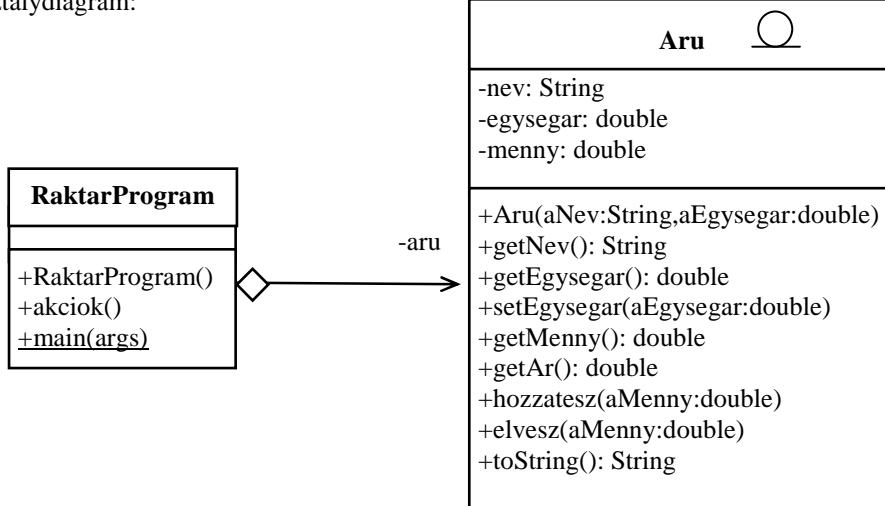
Végezzük el a következő akciókat: Tegyük a raktárba 125 kg paradicsomot, aztán vegyük ki 25 kg-ot, majd szállítsuk le a paradicsom egységárát 210 Ft-ra! Mindhárom akció után írjuk ki, mennyi paradicsom van raktáron és milyen értékben!

A program terve

Együttműködési diagram:



Osztálydiagram:



17.1. ábra. A raktárprogram együttműködési- és osztálydiagramja

A feladat szövegéből első látásra két objektum világlik ki: egy zöldségraktár és abban egy áru. A raktárba beteszünk paradicsomot, a paradicsom árát megváltoztatjuk, stb. De ha jobban belegondolunk, a paradicsom mennyisége nem a raktárra jellemző, hanem magára az árura. Amikor egy raktárba beteszünk 2 kg paradicsomot, akkor ezt a mennyiséget a paradicsomhoz hozzá tesszük. Elvileg lehetne több áru is a raktárban, de most nincs több, és a raktárra pillanatnyilag nem is jellemző más, mint hogy paradicsom van benne. Az akciókat sem a raktár végzi, hanem egy vezérlő, aki a raktárba betesz, illetve kivesz. Mivel a raktárnak pillanatnyilag csak továbbító szerepe lenne, őt most kiiktatjuk a megoldásból – marad az áru és egy vezérlő, amely az áruhoz hozzátesz, illetve abból elvesz egy adott mennyiséget. A program vezérlőjének kitalálunk egy jellemző nevet, legyen ez raktárprogram. A program együttműködési- és osztálydiagramja a 17.1. ábrán látható.

Osztályleírások

- ◆ **Aru:** Információhordozó osztály: van neve, egységára és aktuális mennyisége. Az áru létrehozásakor megadjuk annak nevét és egységárát: new Aru("Paradicsom", 300), a raktáron levő induló mennyiség 0. Az árunak le lehet kérdezni minden tárolt adatát (getNev, getEgysegar, getMenny), le lehet kérdezni az árát (getAr), valamint meg lehet változtatni az egységárát (setEgysegar) és a raktáron levő mennyiséget (hozzatesz, elvesz). A toString() által visszaadott szöveg az áru objektumot reprezentálja – a System.out.println(aru) ezt a szöveget jeleníti meg a konzolon.
- ◆ **RaktárProgram:** Kontroll osztály, mely nem csinál mást, mint végrehajtja a kért akciókat (hozzatesz, elvesz, setEgysegar). A raktárnak nincsenek tulajdonságai.

A RaktárProgram és az Aru osztályok egy-egy, tartalmazási kapcsolatban vannak egymással. Ha a program megszűnik, akkor az áru is vele együtt elpusztul. A program:RaktárProgram objektum tehát ismeri az aru:Aru objektumot, annak üzeneteket tud küldeni.

Az **együttműködési diagramon** jól követhető a program futása. Az egyes tevékenységek a végrehajtás sorrendjében sorszámozva vannak:

- ◆ 1. main(): ez a program belépési pontja. A publikus RaktárProgram osztály statikus main metódusával kezdődik a program futása.
- ◆ 2. RaktárProgram(): a main létrehozza a program:RaktárProgram objektumot saját osztályából.
- ◆ 3. Aru(): A raktárProgram rögtön a konstruktőrben létrehozza az aru objektumot.
- ◆ 4. akciok(): meghívja az aru.hozzatesz(), aru.elvesz() stb. metódusokat.

Az akciok() a következő tevékenységeket foglalja magában:

- ◆ 5. hozzatesz(aMenny): a program hozzátesz egy adott mennyiséget az áruhoz.
- ◆ 6. elvesz(aMenny): a program objektum elvesz egy adott mennyiséget az áruból.
- ◆ 7. setEgysegar(): a program beállítja az áru egységárát.

Az egyes akciók között képernyőre írjuk az aru objektumot – ekkor közvetve meghívjuk annak `toString()` metódusát.

Ha az együttműködési diagramon egy objektum üzenetet fogad, **akkor az üzenetnek megfelelő metódusnak szerepelnie kell az objektum osztályában.**

Az ábrán például a 3. konstruktur, valamint az 5., 6., és 7. üzenetek rendre megtalálhatók az aru objektum osztályában, az Aru-ban.

Forráskód

```
// RaktarProgram.java
class Aru { //1
    private String nev; //2
    private double egysegar; //3
    private double menny; //4

    public Aru(String aNev, double aEgysegar) { //5
        nev = aNev;
        egysegar = aEgysegar;
        menny = 0;
    }

    public String getNev() { //6
        return nev;
    }

    public double getEgysegar() { //7
        return egysegar;
    }

    public void setEgysegar(double aEgysegar) { //8
        if (aEgysegar >= 0)
            egysegar = aEgysegar;
    }

    public double getMenny() { //9
        return menny;
    }

    public double getAr() { //10
        return menny*egysegar;
    }

    public void hozzatesz(double aMenny) { //11
        if (aMenny>0)
            menny += aMenny;
    }

    public void elvesz(double aMenny) { //12
        if (aMenny>0 && aMenny<=menny)
            menny -= aMenny;
    }
}
```

```

public String toString() {                                //13
    return nev+"\tEgysegár: "+egysegar+
           "\tMenny: "+menny+"\tÁr: "+getAr();
}
}

public class RaktarProgram {                           //14
    private Aru aru;                                  //15

    public RaktarProgram() {                           //16
        aru = new Aru("Paradicsom", 300);          //17
    }

    public void akciok() {                           //18
        aru.hozzatesz(125);                         //19
        System.out.println(aru);                      //20
        aru.elvesz(25);                            //21
        System.out.println(aru);                      //22
        aru.setEgysegar(210);                        //23
        System.out.println(aru);                      //24
    }

    public static void main(String[] args) {           //25
        RaktarProgram program = new RaktarProgram();   //26
        program.akciok();                            //27
    }
}

```

A program futása

| | | | |
|------------|-----------------|------------|-------------|
| Paradicsom | Egysegár: 300.0 | Menny: 125 | Ár: 37500.0 |
| Paradicsom | Egysegár: 300.0 | Menny: 100 | Ár: 30000.0 |
| Paradicsom | Egysegár: 210.0 | Menny: 100 | Ár: 21000.0 |

A forráskód elemzése

A forráskódban két osztályt kódoltunk: a `RaktarProgram` a `main`-t tartalmazó publikus osztály, a másik osztály (`Aru`) nem is lehet publikus.

Egy osztály deklarációja egy fejből és egy blokkból áll. Az osztály fejét a `class` kulcsszó jelzi (//1); ha az osztály publikus, akkor a `class` előtt a `public` kulcsszó áll (//14). Az osztály blokkját nyitó és csukó kapcsos zárójelek közé tesszük. A blokk adat és metódusdeklarációkat tartalmaz. Feladatunkban csak példánydeklarációk szerepelnek. Nézzük végig a deklarációkat!

Áru osztály deklarációi:

- ◆ //2-4: Adatok. A `nev`, `egysegar` és `menny` példányváltozók, azok szerepelni fognak a létrehozott `aru` objektumban. Láthatóságuk privát – //20-ban például szintaktikailag hibás lenne a következő utasítás:

`System.out.println(aru.nev);`

- ◆ //5: Konstruktor. Feladata, hogy a paraméterben megadott aNev és aEgysegar értéket „betöltsé” a nev és egysegar példányváltozókba, és hogy a menny változónak 0 értéket adjon. **A konstruktor neve minden esetben megegyezik az osztály nevével, és nincs visszatérési típusa** (nem is void).
- ◆ //6-13: Példánymetódusok (egyik előtt sem szerepel a static kulcsszó). A példánymetódusok (eljárások és függvények) az objektum adatain, a példányváltozókon dolgoznak. Az egyes metódusok blokkjaiban hivatkozhatunk bármely másik deklarációra: //6-ban például a getNev metódus a példány nev változójának értékét adja vissza, //11-ben a hozzatesz metódus a paraméterben megadott aMenny értéket hozzáadja a menny változóhoz.

RaktárProgram osztály deklarációi:

- ◆ //15: Az aru referencia a RaktarProgram osztály egyetlen adata – ez valósítja meg az egy-egy kapcsolatot a két osztály között. A forráskóban az osztály „igazi” adatait és kapcsolatait egyaránt adatként kódoljuk. Az aru példányadat, vagyis csak a program objektum létrejötte után, és csak példánymetódusból lehet rá hivatkozni (a main-ből tehát nem).
- ◆ //16: A RaktarProgram osztály konstruktora. //17-ben létrehozza az árut.
- ◆ //18: Az akciok() metódus az aru objektumnak üzeneteket küld: hozzatesz, elvesz stb. A System.out.println(aru) konzolra írja az aru.toString() függvény értékét.
- ◆ //25: A main osztálymetódus létrehozza a programot //26), és működésre bírja azt //27). Itt indul be tehát a gépezet: létrejön a program objektum, amely létrehozza az aru objektumot. Amíg nem hozzuk létre az első objektumot, addig a többi objektum sem születhet meg – addig pusztán csak leírások léteznek...

Objektum létrehozása saját osztályából

A main osztálymetódust a futtató rendszer automatikusan meghívja, hogy a program elindulhasson. A RaktarProgram osztály main metódusa saját osztályából hoz létre egy példányt, majd meghívja a létrehozott objektum akciók() példánymetódusát. Ezt az esetet a 17.2. ábra 1. esete mutatja. Egy osztály elvileg akárhány példányt létre tud hozni önmagából. Megtehetük volna, hogy készítünk egy StartRaktarProgram osztályt, és a main metódust abba teszszük; ez a 17.2. ábra 2. esete. Ebben az esetben két osztály helyett hármat kell lekódolnunk.

A 2. eset forráskódja a következő:

```
class Aru { //1
    // Az Aru osztály kódja nem változik
}
```

```

class RaktarProgram {
    private Aru aru; //14
    //15

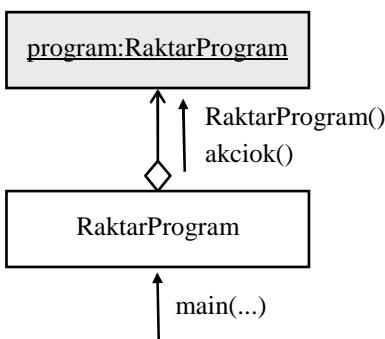
    public RaktarProgram() { //16
        aru = new Aru("Paradicsom", 300); //17
    }

    public void akciok() { //18
        aru.hozzatesz(125); //19
        System.out.println(aru); //20
        aru.elvesz(25); //21
        System.out.println(aru); //22
        aru.setEgysegar(210); //23
        System.out.println(aru); //24
    }
}

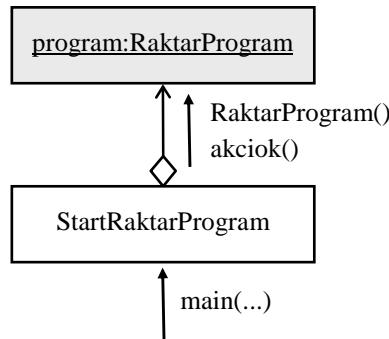
public class StartRaktarProgram { //25
    public static void main(String[] args) { //26
        RaktarProgram program = new RaktarProgram(); //27
        program.akciok(); //28
    }
}

```

1.



2.



17.2. ábra. Objektum létrehozása saját osztályából, illetve kívülről

A `main` metódusnak minden esetben ugyanaz a feladata, és a `program:RaktarProgram` objektum feladatai sem változnak. A két eset között a különbség csak a `main` metódus „tartózkodási helyében” és az osztályok láthatóságában van:

- ◆ 1. eset: a main a RaktarProgram osztály metódusa. A RaktarProgram láthatósága publikus.
- ◆ 2. eset: a main a StartRaktarProgram publikus osztály egyetlen metódusa. A StartRaktarProgram publikus, a RaktarProgram nem publikus.

Bár az 1. eset első látásra bonyolultabbnak tűnhet, ez az eset a szokványosabb, könnyebb programozni, és logikailag ugyanolyan jó, mint a 2. megoldás.

A létrehozott objektum osztálya lehet az az osztály is, amelyben a létrehozást végezzük. Ily módon **az objektumnak több példánya is létrehozható saját magából**. A létrehozás osztály- vagy példánymetódusból egyaránt történhet.

Üzenet „röptében”

A main metódusban nem lenne szükséges deklarálni a program objektumot. Mivel a program során annak minden össze egyetlen üzenetet küldünk, megtehetjük ezt a létrehozás után közvetlenül is, „röptében”:

```
public static void main(String[] args) {  
    new RaktarProgram().akciok();  
}
```

17.3. Második mintaprogram – Bank és a „Jószerencse”

Feladat – Bank

Az OTP bank „Jószerencse” fantázianevű befektetési jegyeket forgalmaz. Egy ügyfél számlát nyithat ehhez a befektetési formához, majd vásárolhat vagy eladhat befektetési jegyeket. A számla nyitásakor meg kell adni a számlatulajdonos nevét, és megadható a kedvezményezett neve is (azé, aki örökli a számlát). Az induló összeg a számla nyitásakor nulla. A befektetési jegynek árfolyama van; a vásárlás és az eladás mindenkor az aktuális árfolyamon történik. Vásárláskor 200 Ft, eladáskor 400 Ft kezelési költséget kell fizetni.

Készítsünk olyan programot, amely a bank „Jószerencse” befektetési jegyeinek forgalmát szimulálja! A program elején nyissunk két ügyfélnek számlát, majd menüből választhatóan tegyük lehetővé, hogy az ügyfelek vásároljanak vagy eladják befektetési jegyeket! Lehessen az árfolyamot is változtatni!

Megjegyzés: A feladat kissé erőltetett, hiszen egy bankban a számlák száma elvileg korlátlan. Tetszőleges számlát akkor tudunk majd nyitni, ha a konténer programozásával is megismerkedünk. Konténerekkel a könyv VI. része foglalkozik.

A program terve

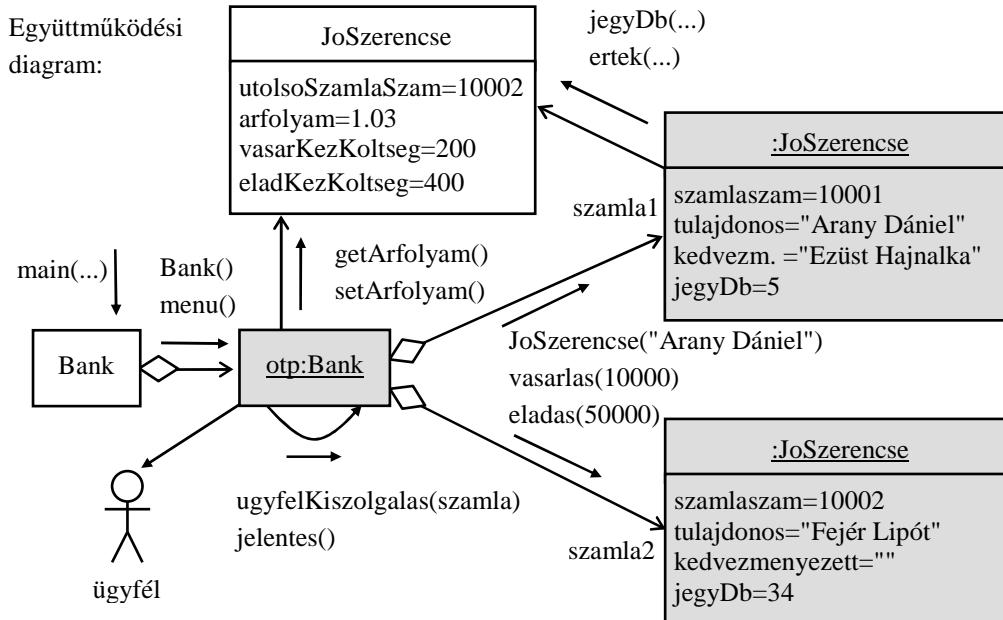
Az OTP, valamint az ügyfelek számlái nyilvánvalóan objektumok lesznek. Az OTP feladata lesz a befektetési számlák kezelése. Az ügyfelek aktorok, akiket az OTP megkérdez, mit szeretnének: vásárolni, eladni stb.

A program terve (együttműködési és osztálydiagramja) a 17.3. ábrán látható. Az otp osztálya Bank. Ebben az osztályban helyezzük el az induló `main()` metódust, és itt fogjuk létrehozni az otp objektumot – a Bank osztály tehát létrehoz egy példányt saját magából. Az otp két JoSzerencse osztályú objektummal (befektetési számlával) áll kapcsolatban. A számla objektum feladata, hogy megjegyezze a számlán lévő befektetési jegyek darabszámát, és lebonolítsa az eladásokat, illetve vásárlásokat. Az otp indulásként létrehozza a két számlát, ezután felkérjük a `menu()` végrehajtására. A menüből az otp saját magát kéri meg az `ugyfelKiszolgulas()` elvégzésére. Az együttműködési diagramon két osztályt is elhelyeztünk: a Bank osztály azért szerepel a diagramon, mert a program futása a benne levő `main()` osztálymetódussal kezdődik; a JoSzerencse osztályra azért van szükség, mert az ábrán látható `utolsoSzamlaSzam`, `arfolyam`, `vasarKezKoltseg` és `eladKezKoltseg` adatokat ő jegyzi, és a `setArfolyam()`, `getArfolyam()` tevékenységek elvégzésére őt fogjuk felkérni. Az objektumokat az átláthatóság kedvéért besötétítettük. Az objektumoknak és az osztályoknak az adatrészét is megjelenítettük: világosan látható, hogy a JoSzerencse osztályadatai (`utolsoSzamlaSzam`, `arfolyam`...) az osztályban szerepelnek, míg a példányadatok (`szamlaszam`, `tulajdonos`...) az egyes példányokban vannak jelen. A feladatok szétosztása az osztályok között a következő:

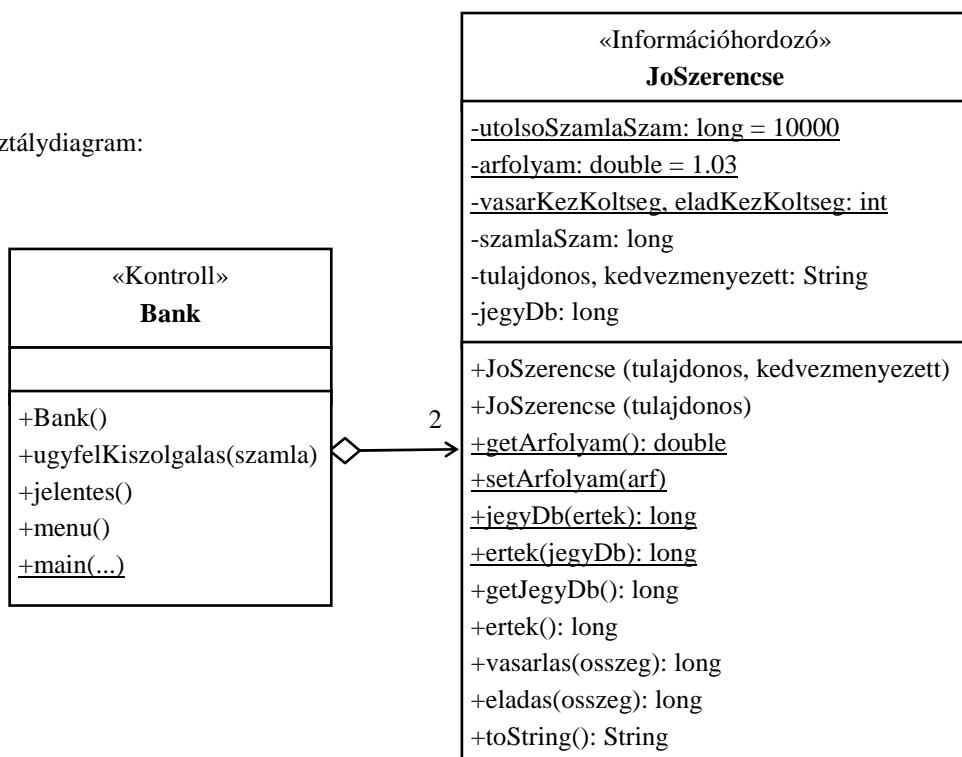
JoSzerencse osztály – osztályleírás

Adatok

- ◆ `utolsoSzamlaSzam:long` Induló sorszám: 10000. Osztályadat, mely az új számlák számlaszámának kiosztásához szükséges. Csak egyszer szerepel az osztályban. Az új számla minden az előzőhez képest egyel nagyobb sorszámot kap.
- ◆ `arfolyam:double` Osztályadat, a befektetési jegy aktuális árfolyama. Ha a jegyek darabszámát megsorozzuk az árfolyammal, megkapjuk a számlán levő befektetési jegyek pillanatnyi értékét forintban.
- ◆ `vasarKezKoltseg=200, eladKezKoltseg=400` Osztályadatok: kezelési költség vásárláskor, illetve eladáskor.
- ◆ `szamlaSzam:long` A számla száma. Az objektum élete során nem változtatható.
- ◆ `tulajdonos:String` A számla tulajdonosának neve.
- ◆ `kedvezmenyezett:String` A számla kedvezményezettjének a neve. Ő kapja meg az összeget, ha a tulajdonossal történik valami.
- ◆ `jegyDb:long` A számlán lévő befektetési jegyek száma. Vásárláskor és eladáskor változik.



Osztálydiagram:



17.3. ábra. A Bank program terve

Konstruktörök

- ◆ `JoSzerencse(tulajdonos:String, kedvezmenyezett:String)` Nyit egy „Jószerencse” befektetési számlát a megadott tulajdonos és kedvezményezett nevével. A számla sorszámát a program adja, automatikusan az előzőleg megnyitott számla számánál egyelőre nagyobbat. A jegyDb induláskor 0.
- ◆ `JoSzerencse(tulajdonos:String)` Mint az előző, csak nincs kedvezményezett.

Metódusok

- ◆ `getArfolyam():double` Osztálymetódus. Visszaadja az aktuális árfolyamot.
- ◆ `setArfolyam(arf:double)` Osztálymetódus. Beállítja az aktuális árfolyamot.
- ◆ `jegyDb(ertek:long):long` Osztálymetódus. Megadja, hogy egy adott értékű összeg hány darab jegyet ér az aktuális árfolyamon.
- ◆ `ertek(jegyDb:long):long` Osztálymetódus. Megadja, hogy az adott darabszámú jegynek mennyi az értéke az aktuális árfolyamon.
- ◆ `getJegyDb():long` Visszaadja a számlán levő jegyek darabszámát.
- ◆ `ertek():long` Visszaadja a számla aktuális értékét.
- ◆ `vasarlas(osszeg:long):long` Az ügyfél befektetési jegyet vásárol ezért az összegért. A metódus visszatéríti értéke a ténylegesen befizetendő összeg (a kerekítés miatt).
- ◆ `eladas(osszeg:long):long` Az ügyfél befektetési jegyet ad el ezért az összegért. A metódus visszatéríti értéke a ténylegesen kifizetett összeg (a kerekítés miatt).
- ◆ `toString():String` Megadja az objektum szöveges reprezentációját.

Bank osztály – osztályleírás

Adatok: Nincsen adat. A bank kapcsolatban áll két befektetési számlával – a kapcsolatokat a forráskódban referenciaikkal valósítjuk meg, ezeket a terven nem kell adatként feltüntetni.

Metódusok:

- ◆ `Bank()` Konstruktur. Létrehozza a két „Jószerencse” objektumot, vagyis számlát nyit két ügyfelének.
- ◆ `ugyfelKiszolgulas(szamla:JoSzerencse)`: Kiszolgál egy ügyfelet a paraméterben megadott számla alapján. Megkérdezi, hogy vásárlás lesz-e vagy eladás, bekéri a vásárolni vagy eladni kívánt összeget. Módosítja a számlát, és kiírja a ténylegesen befizetendő vagy kifizetendő összeget.
- ◆ `jelentes()` Kiírja az aktuális árfolyamot és a számlák szöveges reprezentációit (a `toString`-ben megadott, az objektumra jellemző adatokat).
- ◆ `menu()` Ki lehet szolgálni az 1. vagy a 2. ügyfelet, jelentést lehet kérni, vagy árfolyamat lehet módosítani. minden választás előtt az `otp` jelentést ad a számlákról.
- ◆ `main(...)` Saját magából létrehoz egy példányt, az `otp:Bank` objektumot, majd meghívja annak `menu()` metódusát.

Forráskód

```
import extra.*;
class JoSzerencse {
    private static long utolsoSzamlaSzam = 10000;
    private static double arfolyam = 1.03;
    private static int vasarKezKoltseg = 200;
    private static int eladKezKoltseg = 400;
    private long szamlaSzam;
    private String tulajdonos, kedvezmenyezett;
    private long jegyDb;

    public JoSzerencse(String tulajdonos, String kedvezmenyezett) {
        szamlaSzam = ++utolsoSzamlaSzam;
        this.tulajdonos = tulajdonos;
        this.kedvezmenyezett = kedvezmenyezett;
        jegyDb = 0;
    }

    public JoSzerencse(String tulajdonos) {
        this(tulajdonos, "");
    }

    public static double getArfolyam() {
        return arfolyam;
    }

    public static void setArfolyam(double arf) {
        if (arf >= 0)
            arfolyam = arf;
    }

    public static long jegyDb(long ertek) {
        return (long)(ertek / arfolyam);
    }

    public static long ertek(long jegyDb) {
        return (long)(jegyDb * arfolyam);
    }

    public long getJegyDb() {
        return jegyDb;
    }

    public long ertek() {
        return ertek(jegyDb);
    }

    public long vasarlas(long osszeg) {
        long db = jegyDb(osszeg);
        jegyDb += db;
        return ertek(db) + vasarKezKoltseg; // a befizetendő összeg
    }

    public long eladas(long osszeg) {
        long db = jegyDb(osszeg);
        jegyDb -= db;
        return ertek(db) - eladKezKoltseg; // a kifizetendő összeg
    }
}
```

```

public String toString() {
    return "Számlaszám: "+szamlaSzam+
        " Tulajd.: "+Format.left(tulajdonos,15)+"
        " Kedv.: "+Format.left(kedvezmenyezett,15)+"
        " Jegyszám: "+Format.right(jegyDb,8)+"
        " Érték: "+Format.right(ertek(),8);
}
}

public class Bank {
    private JoSzerencse szamlal, szamla2;

    public Bank() {
        szamlal = new JoSzerencse("Arany Dániel", "EZÜST Hajnalka");
        szamla2 = new JoSzerencse("Fejér Lipót");
    }

    public void ugyfelKiszolgulas(JoSzerencse szamla) {
        long osszeg;
        System.out.println("\n"+szamla);
        char valasz;
        do {
            valasz = Character.toUpperCase(
                Console.readChar("V(ásárlás)/E(ladás) ?"));
        } while (valasz!='V' && valasz != 'E');
        if (valasz == 'V') {
            osszeg=szamla.vasarlas(Console.readInt("Mennyírt vásárol?"));
            System.out.println("Befizetendő: "+osszeg+" Ft");
        }
        else {
            osszeg=szamla.eladas(Console.readInt("Mennyírt ad el? "));
            System.out.println("Kifizetendő: "+osszeg+" Ft");
        }
    }

    public void jelentes() {
        System.out.println("\n*-*-*-* JELENTÉS -*-*-*-*");
        System.out.println("Árfolyam : "+JoSzerencse.getArfolyam());
        System.out.println(szamlal);
        System.out.println(szamla2);
    }

    public void menu() {
        char valasz;
        do {
            jelentes();
            System.out.println("\n1: 1. ügyfél kiszolgálása");
            System.out.println("2: 2. ügyfél kiszolgálása");
            System.out.println("A: Árfolyam módosítás");
            System.out.println("V: Vége");
            valasz = Character.toUpperCase(Console.readChar(" ?"));
            switch (valasz) {
                case '1': {
                    ugyfelKiszolgulas(szamlal);
                    break;
                }
            }
        }
    }
}

```

```

        case '2': {
            ugyfelKiszolgolas(szamla2);
            break;
        }
        case 'A': {
            JoSzerencse.setArfolyam(
                Console.readDouble("Új árfolyam: "));
            break;
        }
    } while (valasz != 'V');
}

public static void main(String[] args) {
    Bank otp = new Bank();
    otp.menu();
}
}

```

A program futását terjedelmi okokból nem mellékeljük. Futtassa a programot!

17.4. Az osztály felépítése, az osztály deklarációi

Az osztály egy fejből és egy blokkból áll:



Az osztály feje

```

[<módosítók>] class <OsztályAzon> [extends <OsztályAzon>]
[implements <InterfészAzon>, <InterfészAzon> ... ]
{
    <osztály blokk (az osztály deklarációi)>
}

```

Nézzük meg részletesen először a fej alkotóelemeit, majd az osztály deklarációt!

Osztály feje

Az osztály feje tartalmazza az esetleges módosítókat, a `class` kulcsszó után az osztály azonosítóját, majd opcionálisan az `extends` kulcsszó után az ős osztály azonosítóját és az `implements` kulcsszó után az implementálandó interfések azonosítóiit.

Osztályfejek például:

```

class JoSzerencse
public class Bank
public final class System
public final class String implements java.io.Serializable, Comparable

```

Módosítók

Egy osztálynak a következő módosítói lehetnek:

- Hozzáférési módosítók: csak a `public` használható. Ha nem adunk meg semmit, akkor alapértelmezés szerint az osztály csak a saját csomagjában látható. Egy fordítási egységen (Java forrásállományban) legfeljebb egy publikus osztály lehet.
- Egyéb módosítók: Használható az `abstract` és a `final`. Egy absztrakt osztályt nem lehet példányosítani (arra való, hogy örökösséssel továbbfejlesszék). Ha egy osztály `final` (végeleges), akkor azt nem lehet örökölni. Az `abstract` és a `final` kizárták egymást. Örökléssel a könyv 2. kötete foglalkozik majd.

extends <OsztályAzon>

A deklarált osztály az itt megadott `OsztályAzon`-t terjeszti ki (belőle származik). Az `extends` után legfeljebb egy osztály adható meg (egy osztálynak csak egy őse lehet). A Java összes osztálya implicit módon az `Object` osztályból származik, azt sosem kell megadni.

implements <InterfészAzon>[, <InterfészAzon> ...]

Az osztály implementálja a felsorolásban megadott interfészeket. Egy osztály több interfészt is implementálhat. Interfészekkel a 20. fejezet foglalkozik.

Osztály blokkja, az osztály deklarációi

Osztály szintű deklaráció: közvetlenül az osztály blokkjában lévő deklaráció.

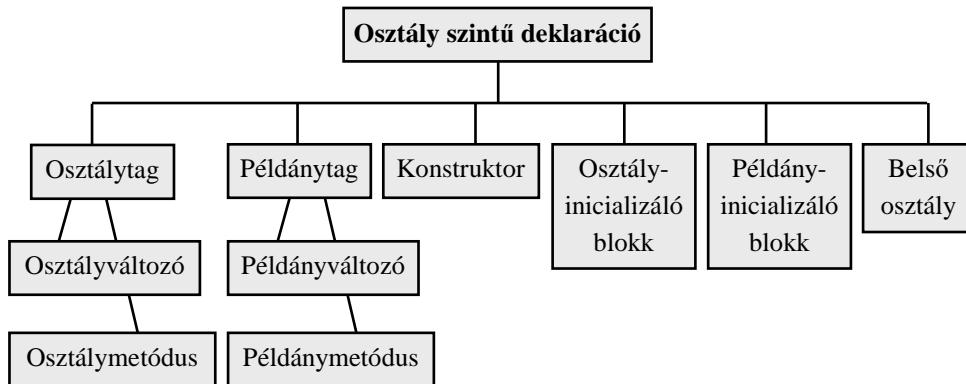
Az osztály blokkja a következő deklarációkat tartalmazhatja:

- osztálytagok (osztályváltozók és osztálymetódusok);
- példánytagok (példányváltozók és példánymetódusok);
- konstruktörök;
- osztályinicializáló blokkok;
- példányinicializáló blokkok;
- belső osztályok;

A deklarációk sorrendje a blokkon belül tetszőleges. Érdemes azonban betartani a sorrendre vonatkozó kódolási konvenciókat, hogy a kód olvashatóbb legyen. Egy **ajánlott deklarálási sorrend** a következő (többféle ajánlás is létezik):

- változók – előbb az osztály kapcsolatait szokás megadni;
- konstruktörök;
- metódusok – először az adatkiolasó (`get`) és adatbeállító (`set`) metódusokat szokás megadni;
- `main` metódus, ha van.

A deklarációk fajtait a 17.4. ábra mutatja. Az egyes deklarációkat külön pontok tárgyalják. A belső osztályokról a 2. kötetben lesz szó.



17.4. ábra. Az osztály szintű deklarációk fajtái

17.5. Osztálytag, példánytag

Osztálytag vagy más néven **statikus tag** (class member, static member): Az osztályhoz tartozik, és az osztály minden objektumára egyformán érvényes. Az osztálytag módosítója **static**. Az osztálytag lehet

- **osztályváltozó** vagy más néven **statikus változó** (class variable, static variable): az osztályhoz tartozó változó, az egyes példányokban nem jelenik meg.
- **osztálymetódus** vagy más néven **statikus metódus** (class method, static method): az osztály saját metódusa, mely csak az osztálytagokat látja.

Példánytag (instance member): A példányhoz tartozik, minden egyes példányra jellemző. A példánytagnak **nincs static** módosítója (innen ismerhető fel). A példánytag lehet:

- **példányváltozó** (instance variable): minden példányban külön szerepel, értéke a példány állapotára jellemző.
- **példánymetódus** (instance method): a megszólított példányon dolgozik, mely az osztály- és példánytagokat egyaránt látja.

Tag (member): az osztálytagok és példánytagok közös neve. Osztályon belül egy osztálytag csak az osztálytagokat látja, a példánytag minden tagot lát.

A tagok módosítói

A tagok előtt megadhatók hozzáférési és egyéb módosítók:

- ◆ Hozzáférési módosítók (accessibility modifiers):

`public`: minden honnan el lehet érni.

`protected`: kizárolag ebből az osztályból és az osztály utódaiból lehet elérni.

`private`: kizárolag csak ebből az osztályból lehet elérni.

Ha egyiket sem adjuk meg, akkor alapértelmezés szerint a saját csomagjában bárki elérheti, de más csomagból nem látható (csomagszintű hozzáférés, package accessibility).

- ◆ Egyéb módosítók:

`static`: osztálytag.

`final`: végeges, nem változtatható.

`abstract` (csak metódusra alkalmazható): üres metódus, ezt majd egy utódosztályban kell kifejteni.

Megjegyzés:

Van még néhány módosító, ezekkel egyelőre nem foglalkozunk:

- `transient` és `volatile` (csak változókra alkalmazhatók);
- `native` és `synchronized` (csak metódusokra alkalmazhatók).

A JoSzerencse osztálytagjai

A JoSzerencse osztályban többek között a következő osztálytagok találhatók:

```
private static long utolsoSzamlaSzam = 10000;
private static double arfolyam = 1.03;
public static boolean setArfolyam(double arf) {
    if (arf>=0)
        arfolyam = arf;
}
```

Az `utolsoSzamlaSzam` és `arfolyam` adatok tehát csak a JoSzerencse osztályban szerepelnek, az osztály példányai nem. Ezek a példányok közös adatai: az `utolsoSzamlaSzam` tartalmazza az utolsóként kiadott számlaszámot, ebből számítja ki a születő példány saját számlaszámát; az `arfolyam` a mindenkorai, minden számlára vonatkozó árfolyam – minden példány ezt használja a számla, illetve a vásárolt/eladott jegyek értékének kiszámításakor. Az árfolyamot a `setArfolyam(double arf)` osztálymetódus segítségével lehet megváltoztatni.

A JoSzerencse példánytagjai

A JoSzerencse osztályban többek között a következő példánytagok találhatók:

```
private long szamlaSzam;
private String tulajdonos, kedvezmenyezett;
private long jegyDb;
```

```

public long getJegyDb() {
    return jegyDb;
}

public long vasarlas(long osszeg) {
    long db = jegyDb(osszeg);
    jegyDb += db;
    return ertek(db)+vasarKezKoltseg; // a befizetendő összeg
}

```

A jegyDb példányadat: minden egyes számlában külön értékkal szerepel, vagyis minden ügyfélnek más darabszámú jegy van a számláján. A `getJegyDb()` példánymetódus, amely visszaadja a jegyDb értékét. Egy osztálymetódus, mint például a `setArfolyam()` nem látja a jegyDb-t (nem is tudná, melyik számlába kukkantson bele a sok közül). A `vasarlas()` metódus minden tagot lát: a jegyDb példányváltozót, a `vasarKezKoltseg` osztályváltozót és az `ertek(long jegyDb)` osztálymetódust. Megjegyzendő, hogy az osztály két `érték()` metódust is deklarál: egy osztálymetódust, amely a paraméterként megadott darabszámú befektetési jegyek értékét adja meg; és egy példánymetódust, amelynek nincsen paramétere, és az objektum saját befektetési jegyeinek az értékét számolja ki.

17.6. Azonosító, hivatkozási kör, takarás

Az egyes deklarációkra a következő szabályok szerint lehet **hivatkozni**:

- A metódus blokkjából hivatkozni lehet az osztály bármely tagjára (osztálymetódusból csak osztálytagra).
- Egy adattag csak a fizikailag előtte deklarált tagokra hivatkozhat (osztályadat csak osztályadatra).
- Egy metódus lokális változójára csak az őt deklaráló metódus hivatkozhat.

Például a `vasarlas()` példánymetódus hivatkozik a `jegyDb` példányadatra, a `jegyDb()` és `ertek()` statikus metódusokra. Az `osszeg` paraméter, a `db` lokális változó:

```

public long vasarlas(long osszeg) {
    long db = jegyDb(osszeg);
    jegyDb += db;
    return ertek(db)+vasarKezKoltseg; // a befizetendő összeg
}

```

Az osztályban ugyanazon a néven deklarálható metódus és változó, sőt a metódusok túlterhelhetők.

Helyes lenne például a következő kódrészlet:

```

class JoSzerencse {
    int joSzerencse; //1

    int joSzerencse() { //2
        int joSzerencse = this.joSzerencse; //3
        return joSzerencse; // a lokális deklarációt adja vissza
    }
    ...
}

```

//1-ben a joSzerencse osztály szintű változó; //2-ben a joSzerencse() osztály szintű metódus; //3-ban a joSzerencse lokális változó, mely takarja az azonos nevű példányváltozót.

A lokális változók **eltakarják** az ugyanolyan nevű osztály-, illetve példányváltozókat. Ha az osztály deklarációjára szeretnénk hivatkozni, akkor osztályváltozó esetén az osztály nevével, példányváltozó esetén pedig a this referenciaival kell azt minősítenünk.

Például:

```

class Proba {
    int szam=1; //1
    static char kar='A'; //2

    void proba(){ //3
        int szam=2; //4
        char kar='B'; //5

        System.out.println(szam); //6 -> 2
        System.out.println(this.szam); //7 -> 1
        System.out.println(kar); //8 -> B
        System.out.println(Proba.kar); //9 -> A
    }
}

```

//6 a metódus saját szam lokális változóját írja ki a konzolra. Ha az osztály szintű szam példányváltozót szeretnénk kiírni, akkor azt a this objektumreferenciával kell minősítenünk (//7).

//8 a metódus saját kar lokális változóját írja ki a konzolra. Ha az osztály szintű kar osztályváltozót szeretnénk kiírni, akkor azt a Proba osztálynévvel kell minősítenünk (//9).

⚠ A változó takarása kerülendő, hiszen a program olvashatósága ettől erősen romlik!

17.7. Változók alapértelmezés szerinti kezdeti értékei

Ha a változó osztályszenzinten van deklarálva, vagyis osztálytag vagy példánytag, akkor a rendszer a változóhoz rendel egy alapértelmezés szerinti kezdeti értéket. Ez az érték minden esetben nullaszerű, vagyis a memóriaterület csupa 0 bit: boolean esetén `false`, char esetén `\u0000`, szám esetén 0, referencia esetén `null`. Ha a változó lokális, vagyis metódusban van deklarálva, akkor születésekor értéke definiálatlan lesz attól függetlenül, hogy típusa primitív vagy referencia. A változók alapértelmezés szerinti kezdeti értékeit a 17.5. ábra foglalja össze.

| | Osztályváltozó, példányváltozó (nullaszerű kezd. érték) | Lokális változó |
|---------------------------|--|-----------------|
| Primitív típusú változó | <code>boolean: false;</code> <code>char: '\u0000'; szám: 0</code> | Definiálatlan |
| Referencia típusú változó | <code>null</code> | Definiálatlan |

17.5. ábra. A változók alapértelmezés szerinti kezdeti értékei

17.8. A this objektumreferencia

Amikor egy példánymetódust megírunk, objektumnak még híre sincs. Az osztályt tartalmazó egységet már rég lefordítottuk – de még mindig nincs objektum. Milyen adatra hivatkozik hát a metódus? Azt mondta egészen eddig, hogy a példánymetódus minden esetben a hívott objektum adatain dolgozik. És ez így is van. Igen ám, de vajon honnan tudjuk a metódusban, hogy most éppen melyik objektum van „teréken”?! Az osztály egységének fordításakor még azt sem tudjuk biztosan, hány objektum lesz a programban!

A technikai megvalósítás borzasztó egyszerű: minden példánymetódusnak van egy utolsó, rejtett paramétere, a `this` (ez), amely a hívott objektum referenciája, típusa pedig maga az osztály. Egy példánymetódus minden esetben a `this` objektumon dolgozik. Ha a megszólított objektum a `szamla1`, akkor a `szamla1` címe kerül a `this` változóba, ha a megszólított objektum a `szamla2`, akkor a `this` értéke `szamla2` lesz. A metódus blokkjában pedig a fordítóprogram minden olyan példánytag előtt, amely nem objektumreferenciával kezdődik, egyszerűen odateszi a `this` referenciát, biztosítva ezzel azt, hogy minden, ami a metódus belsejében történik, az aktuális objektumra vonatkozzék. A `JoSzerencse` osztály `vasarlas()` metódusának **rejtett paramétere** a `this:JoSzerencse`. A példányadatok és példánymetódusok a `this` objektumra vonatkoznak. A vastagon szedett részek csak magyarázat céljából vannak a forráskódban:

```

public long vasarlas(long osszeg, Joszerencse this) {
    long db = jegyDb(osszeg);
    this.jegyDb += db;
    return ertek(db)+vasarKezKoltseg;
}

```

implicit paraméter
(a fordító így látja)

A **this** implicit paraméter nem más, mint memóriacím, amely a megszólított objektum referenciajára. Egy példánymetódus innen tudja, hogy éppen melyik példányon dolgozik (a **this** által mutatott objektumon). Egymásból hívott metódusok esetén a **this** automatikusan továbbadódik.

A **toString()** példánymetódus például továbbadja a **this** értékét, amikor meghívja az **ertek()** példánymetódust:

```

public long ertek(Joszerence this) {
    return ertek(this.jegyDb);
}

public String toString(Joszerence this) {
    return "Szamlaszám: "+this.szamlaSzam+
        " Tulajd.: "+Format.left(this.tulajdonos,15)+"
        " Kedv.: "+Format.left(this.kedvezmenyezett,15)+"
        " Jegyszám: "+Format.right(" "+this.jegyDb,8)+"
        " Érték: "+Format.right(" "+this.ertek(this),8);
}

```

17.9. Konstruktorkák

Amikor egy objektumot a **new** operátorral létrehozunk, akkor azt inicializálni kell. A konstruktur elsőleges feladata az objektum kezdeti állapotának (adatainak és kapcsolatainak) beállítása, felépítése.

A konstruktur általános szintakszisa:

a konstruktur feje

[<módosítók>] <OsztályAzonosító>(<formális paraméterlista>)
 {
 <konstruktur blokkja>
 }

A konstruktur hasonlít a metódushoz – a következő szabályok érvényesek rá:

- A konstruktur neve kötelezően megegyezik az osztály nevével.
- Csak a **new** operátorral hívható. Konstruktornal nem lehet újra inicializálni egy objektumot.
- A módosítók közül csak a hozzáférési (láthatósági) módosítók használhatók.
- A konstruktur túlterhelhető.
- A konstruktornak nincs visszatérési értéke, és nem is **void**.
- A konstruktur nem öröklödik.

Alapértelmezés szerinti konstruktur

Ha egy osztályban nem adunk meg explicit módon konstruktort, akkor az osztálynak lesz egy **alapértelmezés szerinti** (default), paraméter nélküli **konstruktora**. Ha tehát az osztályban nem adtak meg konstruktort, akkor a példány létrehozásakor a rendszer ezt az alapértelmezés szerinti konstruktort hívja meg. Az objektum adatai ekkor az alapértelmezések szerint lesznek beállítva.

Ha az osztályban létezik egy akármilyen explicit konstruktur (akár paraméteres, akár paraméter nélküli), akkor az osztálynak nem lesz implicit, alapértelmezés szerinti konstruktora.

Tekintsük a következő deklarációt:

```
class Kor { // Kör
    private double atmero;
    public double getAtmero() { return atmero; }
    public void setAtmero(double a) { atmero = a; }
    public double terulet() {
        return Math.pow(atmero/2,2)*Math.PI;
    }
}
```

Bár a `Kor` osztály nem definiál konstruktort, mégis létrehozható `Kor` objektum a `new` operátorral. A kör átmérője ebben az esetben induláskor 0 lesz; értékét a `setAtmero()` metódussal változtathatjuk meg:

```
Kor kor = new Kor();
kor.setAtmero(5);
```

A következő példában az alapértelmezés szerinti konstruktur használata fordítási hibát eredményez, mivel deklaráltunk egy másik, paraméteres konstruktort:

```
class Kor {
    private double atmero;

    public Kor(double atmero) { this.atmero = atmero; }
    public double getAtmero() { return atmero; }
    public void setAtmero(double a) { atmero = a; }
    public double terulet() {
        return Math.pow(atmero/2,2)*Math.PI;
    }
}

Kor kor = new Kor(5);
// Kor kor = new Kor(); Fordítási hiba!!!
```

Ha deklarálunk paraméteres konstruktort, és ezenkívül szükségünk van paraméter nélküli konstruktorra is, akkor azt meg kell írnunk.

A konstruktur túlterhelése

A konstruktorok ugyanúgy **túlterhelhetők**, mint más metódusok. Egy objektum így többféleképpen is inicializálható.

Ha egy osztály több konstruktort definiál, akkor az egyik konstruktorból – annak első utasításaként – meghívható egy másik konstruktur a `this` referencia segítségével. A `this` ekkor eljárásként hajtódik végre:

```
    this(paraméterek)
```

Például:

```
public JoSzerencse(String tulajdonos, String kedvezmenyezett) {  
    szamlaSzam = ++utolsoSzamlaSzam;  
    this.tulajdonos = tulajdonos;  
    this.kedvezmenyezett = kedvezmenyezett;  
    jegyDb = 0;  
}  
  
public JoSzerencse (String tulajdonos) {  
    this(tulajdonos,"");  
}  
  
...  
szamlal = new JoSzerencse("Arany Dániel", "Ezüst Hajnalka");  
szamla2 = new JoSzerencse("Fejér Lipót"); // nincs kedvezm.
```

Konstruktorok írásakor is kerüljük a redundanciát! Tipp: Az adatokat a több paraméteres konstruktur állítsa be, a többi konstruktur pedig ezt a konstruktort hívja meg!

Vannak esetek, amikor nincs értelme egy osztályból példányt létrehozni (például csak osztálytagjai vannak). Ilyenkor az osztály írója letilthatja a példányosítást. A letiltást úgy végezhetjük el, hogy az osztályban írunk egy paraméter nélküli konstruktort, és annak privát láthatóságot adunk, mert így nem érvényes többé az alapértelmezett publikus konstruktur. Ilyen osztály például a `java.lang.Math`.

17.10. Inicializálók

Inicializálók segítségével beállítható az osztályok és objektumok kezdeti állapota, és elvégezhetők egyéb, tetszőleges kezdeti tevékenységek. Objektumok esetén e funkciókat a konstruktur `is` el tudja látni; lehetőség szerint azt használjuk! Osztályok esetén a kezdeti tevékenységek automatikus elvégzésére csak az inicializáló alkalmas. Az inicializálók lehetnek kifejezések vagy blokkok.

Inicializáló kifejezés

Inicializáló kifejezéssel egy változónak, illetve konstansnak adhatunk kezdeti értéket. A kifejezés kiértékelése

- osztálytag esetén az osztály betöltésekor;
- példánytag esetén az objektum születésekor (a new operátor végrehajtásakor) történik meg.

Az inicializáló kifejezés a változó, illetve a konstans deklarációjának a része.

Egyéb szabályok:

- Az inicializáló kifejezések a deklarálás sorrendjében kerülnek kiértékelésre; előbb az osztály, aztán a példánytagok inicializáló kifejezései.
- A kifejezésben csak a kifejezés előtt szereplő deklarációkra hivatkozhatunk.
- Osztálytagokból nem hivatkozhatunk példánytagokra.

Például:

```
class Tanulo1 {  
    static int alapTandij = 2000;  
    double atlag = Console.readDouble("Átlag: ");  
    int tandij =  
        alapTandij + (int)Math.round(3000*(5-atlag));  
    // ...  
    public void print() {  
        System.out.println("Tandíj: "+tandij);  
    }  
}
```

Példánkban az alapTandij osztályváltozó; ez egyetlenegyszer kap értéket, mégpedig a program indulásakor, amikor az osztály betöltődik. Az atlag és tandij változók példányonként szerepelnek, azok az egyes objektumok létrehozásakor kapnak értéket.

Inicializáló blokk

Egy osztályt az osztályinicializáló blokk, egy példányt pedig a példányinicializáló blokk segítségével inicializálhatunk. Az inicializáló blokk olyan programkód, amely az osztály, illetve a példány életében egyetlenegyszer, első tevékenységeként fut le. Egy osztályban akárhány osztály-, illetve példányinicializáló blokk deklarálható, azok elvileg bárhol szerepelhetnek, és tetszőleges kódot tartalmazhatnak.

- **Osztályinicializáló blokk (statikus inicializáló blokk):** Egyetlenegyszer, az osztály betöltésekor hajtódik végre. Az osztályinicializáló blokkot osztály szintű változók inicializálására szokták használni. Az osztályinicializáló blokk egyszerű blokk (csak egy {} zárójelpár; nincs feje), melyet a static kulcsszó vezet be.

- **PéldányinicIALIZÓ blokk:** Példányonként egyszer, a példány születésekor hajtódiik végre. A példányinicIALIZÓ blokkot elsősorban a példány szintű változók inicializálására szokták használni. A példányinicIALIZÓ blokk egyszerű blokk, mely előtt nem szerepel a static kulcsszó.

Előbb az osztálytagok aztán a példánytagok inicializáló blokkjai kerülnek végrehajtásra, a deklarálás sorrendjében. Az inicializáló blokkok csak olyan deklarációkra hivatkozhatnak, amelyek fizikailag a blokk előtt szerepelnek.

Példaként az előző, Tanuló1 osztályt átírjuk úgy, hogy az osztály- és példányváltozók inicializálását most a kifejezések helyett statikus, illetve példányinicIALIZÓ blokkok végezzék el:

```
class Tanulo2 {
    static int alapTandij;
    double atlag;
    int tandij;

    // OsztályinicIALIZÓ blokk:
    static {
        alapTandij = 2000;
    }
    // PéldányinicIALIZÓ blokk:
    {
        atlag = Console.readDouble("Átlag: ");
        tandij = alapTandij + (int)Math.round(3000*(5-atlag));
    }
    // ...
    public void print() {
        System.out.println("Tandíj: "+tandij);
    }
}
```

Az inicializálás sorrendje

Mindenekelőtt az osztályadatok kapnak kezdeti értéket:

- először felveszik az alapértelmezés szerinti értékeket, majd
- kiértékelődnek az osztályinicIALIZÓ kifejezések, majd lefutnak az osztályinicIALIZÓ blokkok a deklarálás sorrendjében.

Egy objektum születésekor az objektum adatai a következő szabályok szerint kapnak kezdeti értéket:

- először felveszik az alapértelmezés szerinti értékeket, aztán
- kiértékelődnek az inicializáló kifejezések, majd lefutnak a példányinicIALIZÓ blokkok a deklarálás sorrendjében;
- végül sorban lefutnak az esetlegesen egymást hívó konstruktörök.

Tesztkérdések

17.1. Jelölje meg az összes szintaktikailag helyes osztályfejet!

- a) class public Kalitka extends Lakas
- b) final public class
- c) class LakoTelep
- d) public class Bohoc implements Comparable

17.2. A következő fogalmak közül jelölje meg az osztály szintű deklarációkat!

- a) osztályváltozó
- b) lokális változó
- c) formális paraméter
- d) példányinicializáló blokk

17.3. Tekintse a következő tagdeklarációkat:

```
double d;                                //1
static double d;                            //2
double m() { return d; }                   //3
static double m() { return d; }             //4
```

Jelölje meg az összes helyes állítást a következők közül!

- a) A //1 és a //3 deklarációk nem lehetnek egy osztályban.
- b) A //2 és a //4 deklarációk nem lehetnek egy osztályban.
- c) A //1 és a //4 deklarációk nem lehetnek egy osztályban.
- d) A //2 és a //3 deklarációk nem lehetnek egy osztályban.

17.4. Jelölje meg az összes helyes állítást a következők közül!

- a) A this az osztálymetódus implicit paramétere.
- b) A this a megszólított objektum referenciája.
- c) A this osztálya minden esetben egy Object.
- d) Egymásból hívott példánymetódusok esetén a this automatikusan továbbadódik.

17.5. Jelölje meg az összes helyes állítást a következők közül!

- a) A konstruktur feladata, hogy beállítsa az objektum kezdeti állapotát.
- b) A konstruktur fejében a módosítók közül csak hozzáférési módosító adható meg.
- c) A konstruktur visszatérési értéke boolean.
- d) A konstruktur neveként ajánlatos az osztály nevét adni, de ez nem kötelező.

17.6. Jelölje meg az összes helyes állítást a következők közül!

- a) Egy statikus metódus meghívhatja ugyanazon osztály egy nem statikus metódusát a this kulcsszó segítségével.
- b) Az osztály összes példány- és osztálymetódusa híváskor átad egy implicit this paramétert.
- c) minden objektumban helyet kap az a változó, amelynek van static módosítója.
- d) minden objektumban helyet kap az a válto, amelynek nincs static módosítója.

- 17.7. Jelölje meg az összes helyes állítást a következők közül!
- A konstruktor túlterhelhető.
 - Az osztály konstruktora meghívható az osztály egy másik, túlterhelt konstruktora, annak nevére való hivatkozással.
 - Egy osztálynak minden esetben van egy paraméter nélküli konstruktora.
 - A konstruktor visszatérési értéke void.
- 17.8. Jelölje meg az összes helyes állítást a következők közül!
- Ha az osztálynak nincs explicit konstruktora, akkor a rendszer megad egy alapértelmezés szerinti, paraméter nélküli konstruktort.
 - A konstruktor lehet final.
 - Az osztály konstruktora meghívható az osztály egy másik, túlterhelt konstruktora a this referencia segítségével.
 - A konstruktor blokkja lehet üres.
- 17.9. Jelölje meg az összes helyes állítást a következők közül!
- Az osztályinicializáló blokk beállítja az objektum kezdeti értékeit.
 - Az inicializálók közül először futnak le az osztályinicializálók, és csak azután kerülnek végrehajtásra a példányinicializálók.
 - Egy objektum létrehozható saját osztályából, de csak osztálymetódusból.
 - Az osztály bármely példánymetódusából meghívható a main metódus.

Feladatok

A következő feladatspecifikációk alapján **tervezze meg**, és **kódolja le** a programokat! A tervhez készítse el az osztálydiagramot és legalább egy együttműködési diagramot!

- 17.1. (A) A bor világnapján egy kis falu vezetősége úgy dönt, hogy egy teljes 100 literes hordó bort kioszt az arra járó borkedvelőknek, mindenkinél, akinél éppen van a bor tárolására alkalmas edény. A borkedvelő megmondja, hány litert akar (lehet tört érték is), és a csapláros már is kitölti a kért mennyiséget. Ha nincs annyi bor már a hordóban, akkor sajnálkozunk, és egyáltalán nem adunk. minden osztás előtt írjuk ki információként a hordó kapacitását és az aktuális mennyiséget! Az osztogatás addig folyik, amíg még legalább egy deci bor marad a hordóban. (*BorFesztival.java*)
- 17.2. (A) A fejezet 2. pontjában megadott raktárprogramot egészítse ki úgy, hogy a programban két áru szerepel: paradicsom és paprika. Hol paradicsomból, hol paprikából veszünk ki, illetve teszünk be egy adott mennyiséget. Közben bármelyik árunak megváltoztathatjuk az egységárát. (*RaktarProgram2.java*)
- 17.3. Készítse el az 5. fejezet 7. pontjában tárgyalt Ember osztályt! Az embernek van pozíciója és irányszöge, és a következő üzenetek küldhetők neki: `megy(táv)`, `elmegy(x,y)` és `fordul(szög)`. A létrehozáskor minden adat legyen megadható, de egyik adat megadása se legyen kötelező! Ezután

- a) (A) programozza be Jánost az ott megadott módon! Ellenőrizze állapotait!
- b) (B) hozzon létre az Ember osztályból egy Gergő azonosítójú embert! Kezdeti pozíciója legyen (30,5), irányisége pedig 120! Kövesse végig Gergő „állapotát”, ha a következő üzeneteket kapja: megy(10), fordul(120), megy(13), elmegy(0,0)!
- c) (B) hozza létre Gergőt és Annát tetszőleges állapotokkal, és utasítsa őket a következőkre: Forduljanak meg! Menjenek a 0° irányában 3 egységet! Forduljanak egymás felé! (*Emberek.java*)

17.4. (B) Készítsen egy osztályt az óra ábrázolására (óra, perc másodperc)! Hozzon létre két órát és használja őket: kérdezze le az órát, állítsa át be egy adott időre, állítsa előrebb/hátrébb egy adott idővel! (*Ora.java*)

17.5. (B) Készítsen egy osztályt a komplex számok ábrázolására! Számoljon a komplex objektumokkal! (*Komplex.java*)

17.6. (C) Egy asztalosműhelyben pálcikákat esztergálnak. Kétféle típusú géppel dolgoznak, ezek a megadott méretre esztergálják a pálcikákat. A méretek (átmérő, magasság) a következők:

- kicsi: 0.6×5 cm;
- nagy: 1.2×13 cm.

A pálcikák minden egyfélét, az éppen raktáron levő faanyagból készülnek. Készítsünk olyan programot, amely a különböző típusú pálcikákhoz címkeket gyárt! A címkeken a következő adatoknak kell szerepelniük: a pálcika típusa (kicsi vagy nagy), átmérője, magassága, térfogata, a fa neve és fajsúlya, valamint a pálcika súlya. A címke nyomtatása előtt kérdezzük meg a nyomtatandó darabszámot! A címkeket ragadós papír helyett most konzolra nyomtatjuk.

A hozott fa alapértelmezésben fenyő, melynek fajsúlya 0.7 gramm/cm³. Ezek az értékek azonban a különböző szállítmányok esetén megváltozhatnak. A fa adatait ezért programból állíthatóvá kell tenni, és az átállított érték addig érvényes, amíg azt újra megadják.

Menüből kérhetjük a következőket:

- Címkek nyomtatása kicsi pálcikatípusból
- Címkek nyomtatása nagy pálcikatípusból
- A fa adatainak megadása
- Statisztika (melyik pálcikatípushoz összesen mennyi címkét nyomtattunk)
- Vége

A program legyen könnyen bővíthető, mivel elképzelhető, hogy egyéb típusú címkekre is szükség lesz majd! (*PalcikaCimkezo.java*)

- 17.7. (C) A fejezet 2. pontjában megadott raktárprogramot egészítse ki! Legyen most két raktár, és minden raktárban két áru: paradicsom és paprika. Menüből választhatóan tegye lehetővé, hogy egyik raktárból a másikba átvehessünk akár paradicsomot, akár paprikát! Az egyes áruk árait is meg tudjuk változtatni. (*KetRaktarProgram.java*)

I. BEVEZETÉS A PROGRAMOZÁSBA

1. A számítógép és a szoftver
2. Adat, algoritmus
3. A szoftver fejlesztése

II. OBJEKTUMORIENTÁLT PARADIGMA

4. Mitől objektumorientált egy program?
5. Objektum, osztály
6. Társítási kapcsolatok
7. Öröklődés
8. Egyszerű OO terv – Esettanulmány

III. JAVA KÖRNYEZET

9. Fejlesztési környezet – Első programunk
10. A Java nyelvről

IV. JAVA PROGRAMOZÁSI ALAPOK

11. Alapfogalmak
12. Kifejezések, értékadás
13. Szelekciók
14. Iterációk
15. Metódusok írása

V. OSZTÁLYOK HASZNÁLATA, KÉSZÍTÉSE

16. Objektumok, karakterláncok, csomagolók
17. Osztály készítése



VI. KONTÉNEREK

18. Tömbök
19. Rendezés, keresés, karbantartás
20. A Vector és a Collections osztály

VI.

FÜGGE LÉK

- A tesztkérdések megoldásai
Irodalomjegyzék
Tárgymutató

18. Tömbök

A fejezet pontjai:

1. A tömb általános fogalma
 2. Egydimenziós tömb
 3. A tömb szekvenciális feldolgozása
 4. Gyűjtés
 5. Kétdimenziós tömb
 6. Többdimenziós tömb
 7. A tömb átadása paraméterként
 8. A program paraméterei
 9. Feladat – Szavazatkiértékelés
-

A tömb fix elemszámlalú, azonos típusú elemekből álló sorozat (összetett adatszerkezet), amelyben az elemek sorszámozhatók (indexelhetők), egyenként manipulálhatók. A Javában a tömb elemei lehetnek akár primitív, akár referencia típusúak, és az index minden esetben egy 0-ról induló egész szám. A tömbök egymásba ágyazhatók (a tömb eleme tömb is lehet) – az egymásba ágyazások száma a tömb dimenziója. A tömb rögzített elemszámlalú sorozatok kezelésére és az egy–sok kapcsolat megvalósítására egyaránt jól használható. A tömb egy alacsonyszintű konténer, melynek csak adatai vannak, nincsenek funkciói. A következő fejezet az egy-, két- és többdimenziós tömbök használatát mutatja be Javában.

18.1. A tömb általános fogalma

A tömb azonos típusú elemek sorozata, ahol az elemek száma előre rögzített. A tömb elemei **indexelhetők**, az elemekre az index segítségével hivatkozhatunk.

Egydimenziós tömb

Az egydimenziós tömb minden egyes eleméhez egy egyértelmű indexérték tartozik, amely meghatározza az elem tömbön belüli helyét (18.1. ábra). Az index típusának sorszámozhatónak kell lennie. Az index tartomány két határérték közötti, és az indexek egyesével növekszenek.

Indexelések például:

- ◆ A matematikában az index általában egész típusú, és az alsó határ 1.
- ◆ A Javában az index egész típusú, és az alsó határ minden esetben 0.

Megjegyzések:

- Vannak olyan programozási nyelvek (mint például a Pascal), amelyekben az index bármilyen sorszámozott típus lehet, és megadható az index tartomány alsó és felső határa. Például: `1..20, -5..60000, A..Z`.
- Matematikában az egydimenziós tömböt vektornak nevezzük.

Az egydimenziós tömb megadása pszeudokódval:

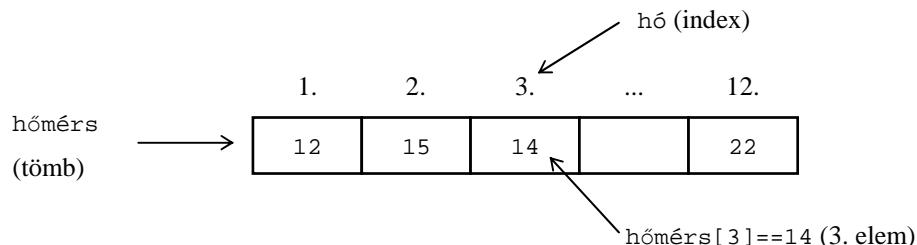
```
<tömbAzonosító>: <elemtípus>[<index>:tól..ig]
```

A következő egydimenziós tömb például az év minden egyes hónapjához tárolja a havi átlaghőmérsékletet (18.1. ábra):

```
hőmérs: number[hő:1..12]
```

Ekkor:

- ◆ A tömb azonosítója: `hőmérs`
- ◆ A tömb elemeinek száma: 12
- ◆ Az index tartomány alsó határa: 1, felső határa: 12
- ◆ A tömb elemeinek típusa: `number`
- ◆ Elemek: `hőmérs[1]==12, hőmérs[2]==15, ... hőmérs[12]==22`



18.1. ábra. Egydimenziós tömb

Az egyes elemek típusuktól függően manipulálhatók. A tömb használatában az a nagyszerű, hogy nem kell az elemeknek egyedi neveket kitalálni, hanem egy gyűjtőnevet adunk az egész csoportnak (`hőmérs`), és az egyes elemekre indexekkel hivatkozunk (`hőmérs[2]` a február havi átlag hőmérséklet).

Két-, illetve többdimenziós tömb

A tömb elemei elvileg bármilyen típusúak, így tömbök is lehetnek. Ha egymelenziós tömböket mint elemeket egy tömbbe foglalunk, rögtön kétdimenziós tömböt kapunk. A háromdimenziós tömb úgy keletkezik, hogy tömbbe foglaljuk a kétdimenziós tömböket. Az egymásba ágyazásnak elvileg nincs határa – a tömb dimenzióinak száma elvileg korlátlan. A kétdimenziós tömb megadása pszeudokód segítségével:

```
<tömbAzonosító>:  
<elemtípus>[<sorindex>:tól..ig][<oszlopindex>:tól..ig]
```

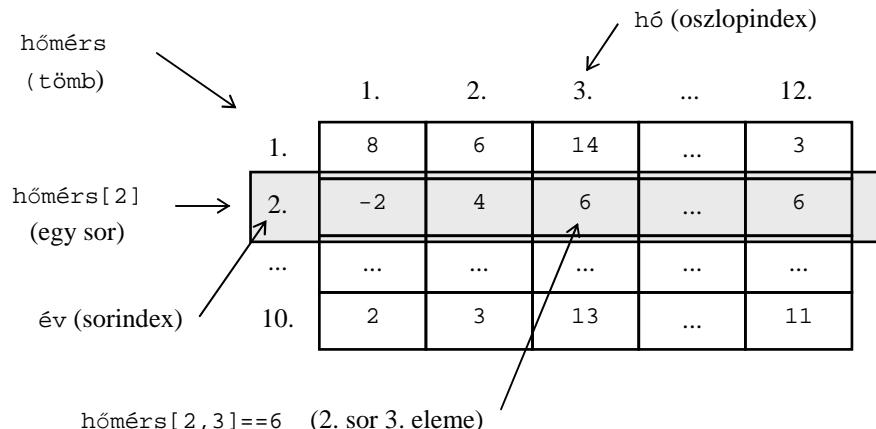
Többdimenziós tömbökkel egymásba ágyazott, többszintű indexelés valósítható meg. A többdimenziós tömb egy elemét egy indexhalmaz határozza meg. minden egyes elemnek annyi indexe van, ahány dimenziós a tömb.

A következő kétdimenziós tömb például 10 éven keresztül az év minden hónapjára megadja a havi átlaghőmérsékletet (18.2. ábra):

```
hőmérs: number[év:1..10][hó:1..12]
```

Ekkor:

- ◆ A tömb azonosítója: hőmérs
- ◆ A tömb dimenziója: 2
- ◆ Elemek: `hőmérs[1][1]==8`, `hőmérs[1][12]==3`, `hőmérs[10][1]==2`, ...
`hőmérs[10][12]==11` (A 10. év decemberében az átlag hőmérséklet 11 fok volt.)



18.2. ábra. Kétdimenziós tömb

A 18.2. ábrán látható hőmérs kétdimenziós tömb előállítása a következőképpen képzelhető el: veszünk egy sort, melynek `number` típusú elemeit 1 és 12 között indexeljük. Ezután veszünk ilyen sorból 10 darabot, s azokat 1-től 10-ig indexeljük. A hőmérs[2][3] a második sor harmadik eleme. A kétdimenziós tömb egy teljes sorára is hivatkozhatunk: hőmérs[2] a második sort jelenti.

Példák a kétdimenziós tömb alkalmazására:

- ◆ 1990 és 2010 között évente és azon belül havonta szeretnénk gyűjteni a beérkezett e-mailek számát. Ekkor az egyik index lehet az év, a másik a hónap:

`emailSzám: number[év:1990..2010][hó:1..12]`

Az 1998. 11. havi darabszámról így hivatkozhatunk: `emailSzám[1998][11]`.
- ◆ Statisztikát szeretnénk készíteni arról, hogy egy angol dokumentum egyes lapjain milyen betűből hány szerepel (kis- és nagybetűk között nem teszünk különbséget). Ekkor az egyik index a lap (1..lapokSzáma), a másik index a betű (A..Z). A tömb elemei egész típusúak:

`betűkSzáma: number[lap:1..lapokSzáma][betű:'A'..'Z']`

A 26. lapon található A betűk száma: `betűkSzáma[26]['A']`

Megjegyzések:

- Matematikában a kétdimenziós tömböt mátrixnak nevezzük.
- A Javában a soroknak nem kell egyforma hosszúnak lenniük.
- A Javában nem kényelmes, hogy az index csak nulláról indulhat. Az előbbi példák kidolgozásához az indexeket transzformálni kell!

18.2. Egydimenziós tömb

A Javában a tömb elemei bármilyen típusúak lehetnek, vagyis:

- primitív típusú, vagy
- referencia típusú (osztály típusú vagy tömb típusú).

Az elemek deklarált típusa azonos, de a referencia típusú változók utód típusú objektumokat is azonosíthatnak.

Tömb változó deklarálása

A tömb referencia típusú változó, melyet deklarálnunk kell. Először megadjuk az elemek típusát, ezt egy szögletes zárójelpár, a tömbképző operátor követi, végül megadjuk a tömb azonosítóját. Az elemtípus lehet akár primitív, akár referencia. Az így deklarált változó képes egy megadott elemtípusú tömbre mutatni:

```
<elemtípus>[ ] <tömbAzonosító>; //1
```

A tömbképző operátort a tömbazonosító után is tehetjük:

```
<elemtípus> <tömbAzonosító>[ ]; //2
```

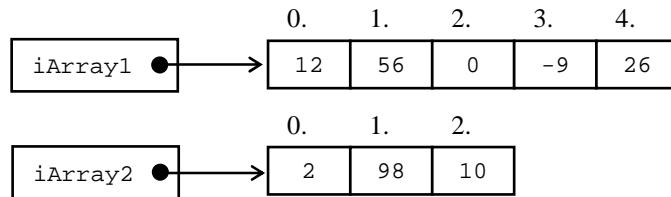
Például:

```
int[] iArray1;           // iArray1 egy tömb
int iArray2[];           // iArray2 is egy tömb!
```

• Egy deklarációban a vesszővel elválasztott azonosítók mindegyikére a deklaráció elején szereplő típus vonatkozik. A következő példában iArray1, iArray2 és iArray3 tömbök, de iArray4 csak egy egyszerű int típusú változó!

```
int[] iArray1, iArray2;      // iArray2 is tömb!
int iArray3[], iArray4;      // iArray4 nem tömb!
```

Az iArray1 és az iArray2 változók egy-egy tetszőleges méretű, int elemtípusú tömbre képesek mutatni (18.3. ábra).



18.3. ábra. Tömbreferencia

A deklarálással egyelőre csak a referenciának (memóriacímnek) foglaltunk helyet, ezen a pontron még nincs tömb. A tömböt külön létre kell hoznunk.

A tömb létrehozása

A Javában a tömböt a program futása közben, a new operátorral hozhatjuk létre. A new után meg kell adnunk az elemek típusát és a tömb méretét (az elemek számát):

```
new <elemtípus> [<méret>]
```

A méret egy nem negatív, int típusú kifejezés. Maximum Integer.MAX_VALUE eleme lehet a tömbnek.

Például:

```
new int[5]           // méret:5           elemtípus:int
int k=20;
new double[k/3+1]   // méret:k/3+1==7   elemtípus:double
new String[3]        // méret:3           elemtípus:String
```

A new operátor létrehozza a tömböt a memóriában, és visszaadja a tömb referenciáját, vagyis memóriabeli mutatóját. Ezt a referenciait aztán értékel adhatjuk egy olyan referencia típusú változónak, amely egy ilyen tömbre képes rámutatni:

```
int[] iArray;
double[] dArray;
String[] sArray;
iArray = new int[5];
dArray = new double[k/3+1];
sArray = new String[3];
```

A deklarációt és a létrehozást egy lépésben is elvégezhetjük:

```
int[] iArray = new int[5];
double[] dArray = new double[k/3+1];
String[] sArrray = new String[3];
```

A tömb length konstansa

Minden tömb objektumnak van egy length konstansa (módosítója final), amely megadja a tömb hosszát: tömbAzonosító.length. A length értéke a tömb létrehozásától kezdve állandó, azt nem lehet megváltoztatni. A tömb indexei 0 és length-1 közöttiek.

Megjegyzés: Változtatható méretű konténerként az API Vector osztályát fogjuk alkalmazni.

Indexelés

A tömb egyes elemeire így hivatkozhatunk:

<tömbAzonosító>[index]

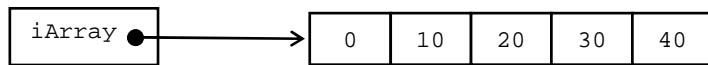
Az érvényes indextartomány 0 és length-1 között van. Az ezen kívül eső indexelés ArrayIndexOutOfBoundsException kivételt vált ki.

Például:

- ◆ Az iArray tömb elemeinek feltöltése:

```
int[] iArray = new int[5];
for (int i=0; i<iArray.length; i++)
    iArray[i] = i*10;
```

0. 1. 2. 3. 4.

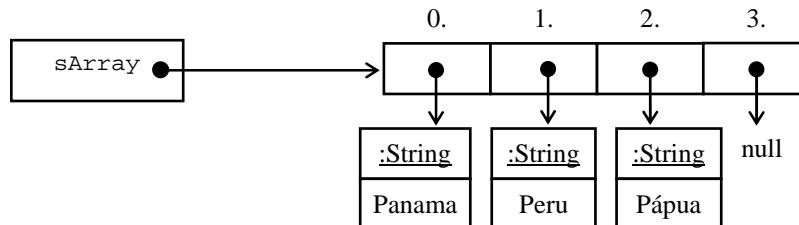


- ◆ A dArray tömb elemeinek beolvasása a konzolról:

```
double[] dArray = new double[20];
for (int i=0; i<dArray.length; i++)
    dArray[i] = Console.readDouble();
```

- ◆ Az sArray tömb elemeinek értékkadása (az elemek referenciaiák):

```
String[] sArray = new String[4];
sArray[0] = "Panama";
sArray[1] = "Peru";
sArray[2] = "Pápuá";
```



- ◆ A szep két elemű, boolean elemtípusú tömb:

```
boolean[] szep;
szep = new boolean[2];
szep[0]=true; szep[1]=false;
```

- ◆ A maganhangzok öt elemű, char elemtípusú tömb:

```
char[] maganhangzok = new char[5];
maganhangzok[0]='a'; ...; maganhangzok[4]='u';
```

Alapértelmezés szerinti kezdeti értékek

A tömb referenciajának alapértelmezés szerinti értéke osztályszintű deklaráció esetén null, lokális deklaráció esetén definiálatlan (mint minden más referencia esetén).

Az újonnan létrehozott tömb elemei alapértelmezés szerinti értékeket vesznek fel függetlenül attól, hogy a deklaráció osztályszintű, vagy lokális. Referenciák esetén ez az érték null, primitív típusú változók esetén 0, \u0000, illetve false.

Inicializáló blokk

Deklaráláskor inicializáló blokkal a tömb elemeinek kezdeti értékek adhatók:

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

A kapcsos zárójelek által határolt blokkot inicializáló blokknak nevezzük. Ebben az esetben nem kell és nem is szabad a tömböt a new operátorral létrehozni: a {} blokk-képző operátor mindezt megteszi helyettünk! A tömb mérete pontosan akkora lesz, amennyi a felsorolt értékek száma.

Példák:

```
int[] iArray = {0,10,20,30,40}; // 5 elemű
char[] maganhangzok = {'a','e','i','o','u'}; // 5 elemű
boolean[] szep = {true,false}; // 2 elemű
double[] arak = {12.4,5000,3.7}; // 3 elemű
```

Értékadás

A t2 tömb értékadás szerint **kompatibilis** a t1 tömbbel (azaz t1=t2 megengedett), ha

- Primitív elemtípus esetén t1 és t2 elemtípusa azonos;
- Referencia elemtípus esetén t2 elemtípusa t1 elemtípusával azonos, vagy annak leszármazottja.

Tömb értékadásakor a tömb referenciaja kap értéket, nem a tömb elemei. Így különböző hosszúságú tömbök is értékül adhatók egymásnak.

Például adottak a következő tömb deklarációk:

```
int[] iArray1 = {1,3,5,7}, iArray2 = {10,20};
double[] dArray1 = {1,2,3,4,5,6,7.0}, dArray2 = {1.5,6,0};
Object[] oArray = null;
String[] sArray = {"Get Back", "Let It Be"};
```

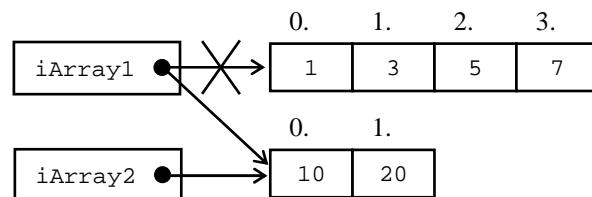
◆ Szintaktikailag helyes értékadások:

```
iArray1 = iArray2; // iArray1 == {10,20}, iArray2=={10,20}
dArray1 = dArray2; // dArray1 == {1.5,6,0}
oArray = sArray; // oArray == {"Get Back", "Let It Be"}
```

◆ Szintaktikailag helytelen értékadások:

```
iArray = dArray; // nem egyforma primitív típusú elemek
sArray = dArray; // egyik elemei objektumok, másiké primitívek
sArray = oArray; // az Object-nek nem őse a String!
```

Értékadáskor tehát a tömb nem másolódik, hanem átállítódik a mutatója. Az iArray1=iArray2 értékadás után iArray1 már nem arra a tömbre mutat, amelyikre eddig, hanem arra, amelyikre az iArray2 is mutat. Az értékadás után tehát minden mutató ugyanazt a tömböt azonosítja, az iArray1 által mutatott tömb pedig, hacsak más nem mutat rá, előbb vagy utóbb a szemérgyűjtő áldozata lesz. A tömb értékadását a 18.4. ábra szemlélteti.



18.4. ábra. Tömb értékadás (iArray1=iArray2)

18.3. A tömb szekvenciális feldolgozása

A tömb szekvenciális feldolgozásakor a tömb elemeit sorban, egyesével dolgozzuk fel. A tömb szekvenciális feldolgozását néhány feladat segítségével mutatjuk be.

Feladat – Számok

Kérjünk be 10 számot a konzolról – minden bekérésnél írjuk ki az elem sorszámát! Ezután írjuk ki a számokat először a bekérés sorrendjében, azután pedig visszafelé!

Forráskód

```
import extra.*;
public class Szamok {
    int[] szamok = new int[10];      //1

    // A 10 db szám bekérése:
    void beker() {
        for (int i=0; i<szamok.length; i++)
            szamok[i] = Console.readInt(i+1+". szám: ");
    }

    // A számok kiírása a bekérés sorrendjében:
    void kiirElore() {
        for (int i=0; i<szamok.length; i++)
            System.out.print(szamok[i]+" ");
        System.out.println();
    }

    // A számok kiírása fordított sorrendben:
    void kiirVissza() {
        for (int i=szamok.length-1; i>=0; i--)
            System.out.print(szamok[i]+" ");
        System.out.println();
    }

    public static void main(String[] args) {
        Szamok szamok = new Szamok();
        szamok.beker();
        szamok.kiirElore();
        szamok.kiirVissza();
    }
}
```

A program elemzése

A számokat egy tömbben tároljuk. A tömböt és az azon dolgozó algoritmusokat egy szamok nevű tömbkezelő objektumba tesszük. A Szamok osztály felelőssége a 10 darab szám beolvasása, tárolása és kiírása előre és visszafelé. A szamok tömböt (amely történetesen egy ugyanilyen nevű objektumba van bezárva) az objektum születésekor, egy inicializáló kifejezéssel hoztuk létre (//1), minden elem kezdeti értéke nulla. A beker(), a kiirElore() és a

`kiirVissza()` metódusok minden számokat tartalmazó tömbön dolgoznak. A `beker()`, és a `kiirElőre()` metódusokban a ciklus 0-ról indul és `length-1`-ig megy; a `kiirVissza()` metódusban a tömb elemeit fordított sorrendben dolgozzuk fel, ott a ciklus `length-1-től 0-ig` megy. Bekéréskor a felhasználónak minden egyelőre nagyobb indexet írunk ki, mint a programbeli valóság: számára az 1-től induló sorszámozás a természetes.

Feladat – Szövegek

Kérjünk be szövegeket a konzolról az üres szöveg végjelig. Ha nem tudunk már több szöveget tárolni, informáljuk erről a felhasználót! Végül írjuk ki a szövegeket először a bekérés sorrendjében, azután pedig visszafelé!

Ez a feladat annyiban tér el az előzőtől, hogy most nem számokat, hanem szövegeket kérünk be, és a bekért szövegek számát nem tudjuk előre. Egy akkora tömböt kell deklarálnunk, amelybe nagy valószínűséggel beleférnek a szövegek: a tömb mérete legyen 100! A szövegek valódi száma (`nSzoveg`) a program futásától függ majd, és természetesen az is előfordulhat, hogy a szövegek nem férnek majd bele a tömbbe. A számozás – alkalmazkodva a Javához – nulláról indul:

| 0. | 1. | nSzoveg-1 | | length-2 | length-1 |
|----|----|-----------|---------|----------|----------|
| | | | ... | ... | |

Forráskód

```
import extra.*;

public class Szovegek {
    String[] szovegek = new String[100]; // 100 szöveg fér be
    int nSzoveg = 0; // szövegek száma

    // A szövegek bekérése:
    void beker() {
        String szoveg;
        while (true) {
            if (nSzoveg == szovegek.length) {
                System.out.println("Betelt");
                break;
            }
            szoveg = Console.readLine(nSzoveg+1 + ". szöveg: ");
            if (szoveg.equals(""))
                break;
            szovegek[nSzoveg] = szoveg;
            nSzoveg++;
        }
    }
}
```

```
// Szövegek kiírása a bekérés sorrendjében:  
void kiirElore() {  
    for (int i=0; i<nSzoveg; i++)  
        System.out.print(szovegek[i] + " ");  
    System.out.println();  
}  
  
// Szövegek kiírása fordított sorrendben:  
void kiirVissza() {  
    for (int i=nSzoveg-1; i>=0; i--)  
        System.out.print(szovegek[i] + " ");  
    System.out.println();  
}  
  
public static void main(String[] args) {  
    Szovegek szovegek = new Szovegek();  
    szovegek.beker();  
    szovegek.kiirElore();  
    szovegek.kiirVissza();  
}
```

A program elemzése

A szövegek objektumban most a bevitt szövegek számát (`nSzoveg`) is tárolnunk kell, hiszen az futáskor változik. A `beker()` eljárás ciklusában a kilépés feltételét felesleges megfogalmaznunk – ha a tömb betelt, vagy a felhasználó végjelet üt, egyszerűen kiugrunk a ciklusból. Az előbbi esetben figyelmeztetjük a felhasználót, nehogy feleslegesen fáradjon. Ha a tömbben van hely: `nSzoveg < szovegek.length`, és a szöveg nem a végjel: `!szoveg.equals("")`, a szöveget betesszük a tömb soron következő elemébe: `szovegek[nSzoveg]=szoveg`, és növeljük a tömbben lévő szövegek számát: `nSzoveg++`.

18.4. Gyűjtés

Gyűjtéskor valamilyen szempontból összetartozó elemek darabszámát, illetve értékeit összegezzük, gyűjtjük. Gyűjtéskor a bejövő elemeket szétválogatjuk: minden elemet sorban megvizsgálunk, hogy az melyik gyűjtőbe tartozik. Ha a gyűjtő indexelhető tömb, akkor azt kell eldöntenünk, hogy melyik indexű gyűjtőt és hogyan kell módosítanunk.

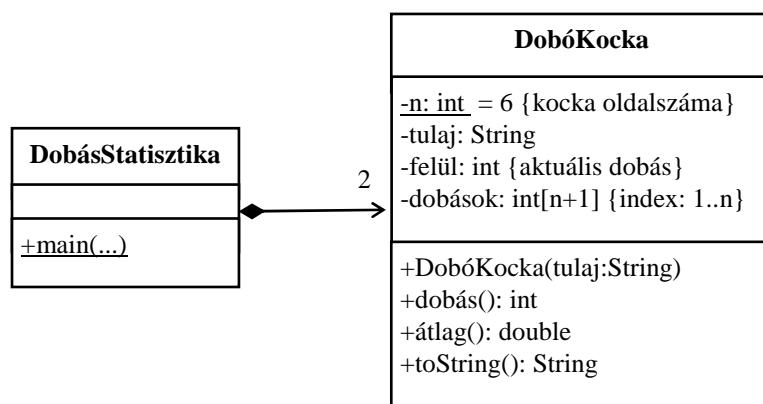
Feladat – Dobás-statisztika

Két ember, Bush és Gates dobókockával versenyez: saját kockájukkal mindenketten 10-szer dobnak, és amelyiküknek az átlaga jobb, az nyer. A dobásokat véletlenszám-generátorral szimuláljuk! A verseny végén írjuk ki mindkét versenyző nevét, és azt, hogy hány darab 1-es, 2-es... 6-os dobásuk volt, valamint dobásainak átlagát!

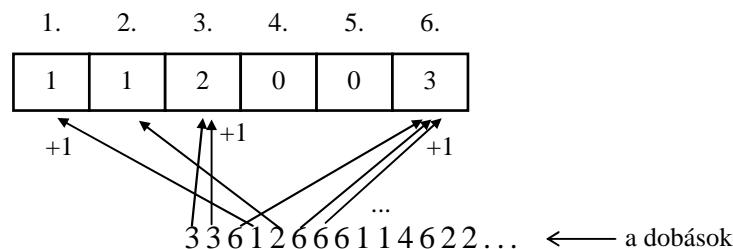
A program terve

A program terve a 18.5. ábrán látható.

- ◆ Készítünk egy DobóKocka osztályt. Egy osztályváltozóban tároljuk a kocka oldalainak a számát (hogy alkalmasint hétdalalú kockával is játszhassunk). Egy kocka születésekor meg kell adni tulajdonosának nevét. A kockával lehet dobni, ilyenkor a kocka generál egy véletlen értéket 1 és a kocka oldalszáma (n) között. minden kocka maga gyűjti a dobások tömbben, hogy élete során hányasokat dobtak vele (melyik szám hányszor szerepelt felül). A kockától meg lehet kérdezni eddigi dobásainak átlagát.
- ◆ A DobásStatisztika osztály létre fogja hozni a két játékos dobókockáját, és tízszer felkéri az egyes kockákat dobásra. Végül kiírja az eredményt. A megfelelő információt minden kocka maga állítja elő `toString()` metódusával.



18.5. ábra. A DobásStatisztika program osztálydiagramja



18.6. ábra. Gyűjtés

Elemezzük a DoboKocka osztályban szereplő dobások tömböt! A gyűjtés mechanizmusát a 18.6. ábra szemlélteti. A tömb 1. rekeszében gyűjtjük az 1-es dobásokat, a 2-esben a 2-eseket stb. Ha a dobás n, akkor a dobasok[n] értékét megnöveljük eggel. Kezdetben a gyűjtő rekeszei nulla értékűek. A könnyebb számolás érdekében egy trükköt alkalmazunk: a tömb nulladik elemét egyszerűen nem szerepeltetjük a megoldásban. Így egy rekesz ugyan kihasználatlan marad, de a programot sokkal egyszerűbb követni!

Forráskód

```
import extra.*;
class DoboKocka {
    private static int n = 6;
    private String tulaj;
    private int felul;
    private int[] dobasok = new int[n+1]; // 0. elem felesleges

    public DoboKocka(String tulaj) {
        this.tulaj = tulaj;
        felul = 1;
    }

    public int dobas() {
        felul = (int)(Math.random()*n+1);      // 1 és n között
        dobasok[felul]++;
        return felul;                      // a dobott értéket visszaadja
    }

    public double atlag() {
        int osszeg=0, dobasSzam=0;
        for (int i=1; i<=n; i++) {
            dobasSzam += dobasok[i];
            osszeg += dobasok[i]*i;
        }
        return osszeg*1.0/dobasSzam;
    }

    public String toString() {
        String str=tulaj;
        for (int i=1; i<=n; i++) {
            str = str+" "+dobasok[i];
        }
        str = str+" Átlag: "+atlag();
        return str;
    }
}

public class DobasStatisztika {
    public static void main(String[] args) {
        final int DOBASSZAM = 10;
        DoboKocka d1 = new DoboKocka("Bush ");
        DoboKocka d2 = new DoboKocka("Gates ");
```

```

// Mindketten dobnak tizet:
for (int i=1; i<=DOBASSZAM; i++) {
    d1.dobas();
    d2.dobas();
}

// Eredmény kiírása:
System.out.println(d1);
System.out.println(d2);
}
}

```

A program egy lehetséges futása

| | | | | | | | | |
|-------|---|---|---|---|---|---|--------|-----|
| Bush | 1 | 4 | 1 | 2 | 0 | 2 | Átlag: | 3.2 |
| Gates | 0 | 2 | 2 | 1 | 2 | 3 | Átlag: | 4.2 |

Sokszor előfordul, hogy több különböző értékű elemnek ugyanazt a gyűjtőt kell növelnie – a következő feladat erre mutat példát.

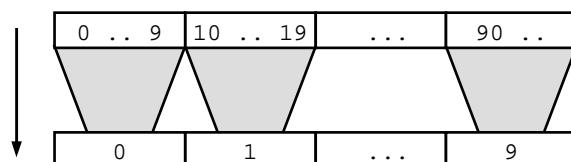
Feladat – Életkor

Kérjük be 5 ember nevét és életkorát. Ezután készítsünk olyan statisztikát, amely megmutatja, hogy melyik korcsoportba hány ember tartozik (a 100 éveseket vagy annál idősebbeket soroljuk az utolsó korcsoportba)! A képernyőterv a következő:

| Korcsop | Darab |
|---------|-------|
| 0 - 9: | 999 |
| 10-19: | 999 |
| 20-29: | 999 |
| ... | |
| 80-89: | 999 |
| 90-99: | 999 |

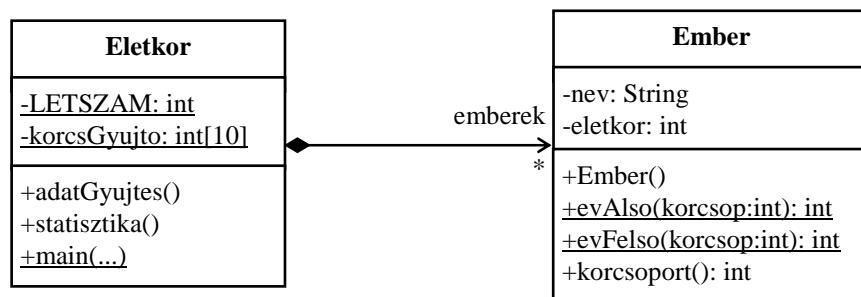
A program terve

A gyűjtő indextípusa és az index képzése most nem annyira egyértelmű, mint az előző feladatban. Mivel a gyűjtő indexeinek egy sorszámozott típus egymást követő értékeinek kell lenniük, ki kell találnunk egy olyan indextípust, valamint egy olyan indexképző algoritmust (transzformációt), amelyikkel a csoporthoz tartozó index egyértelműen meghatározható. E feladatban kézenfekvő a 0..9 indextípus, az indexképző algoritmus pedig legyen a 10-zel való egész osztás:



A tömb 0. rekeszét növelik a 0..9 értékek, az 1. rekeszét a 10..19 értékek stb. A 9. rekeszt növelik a 90 vagy annál nagyobb életkorok.

A program terve a 18.7. ábrán látható. Készítünk egy Ember osztályt, amelynek konstruktora bekéri az ember adatait. Az ember meg tudja mondani saját korcsoportját (0..9), és osztálymetódusokkal lekérdezhetők az egyes korcsoportok alsó és felső évei. Az Eletkor osztályban felveszünk egy gyűjtő tömböt a korcsoportoknak. A main metódus létrehozza az Eletkor vezérlő objektumot, majd felkéri őt adatgyűjtésre és a statisztika elkészítésére. Az adatGyujtes() metódus létrehoz 10 embert; a statisztika() metódus sorban megvizsgálja az embereket, hogy melyik korcsoportba tartoznak, az ennek megfelelő gyűjtőrekeszt eggyel növeli, végül kiírja az eredményt.



18.7. ábra. Az Életkor program osztálydiagramja

Forráskód

```

import extra.*;
class Ember {
    private String nev;
    private int eletkor;

    public Ember() {
        System.out.println("\nAz ember adatai");
        nev = Console.readLine("Név: ");
        eletkor = Console.readInt("Életkor: ");
    }

    public static int evAlso(int korcsop) {
        return korcsop*10;
    }

    public static int evFelso(int korcsop) {
        return korcsop*10+9;
    }
}
  
```

```

public int korcsoport() {
    int korcsop = eletkor/10;
    if (korcsop > 9) korcsop = 9;
    return korcsop;
}

public class Eletkor {
    private final int LETSZAM=5;
    private int[] korcsGyujto = new int[10];
    private Ember[] emberek = new Ember[LETSZAM];

    public void adatGyujtes() {
        for (int i=0; i<LETSZAM; i++)
            emberek[i] = new Ember();
    }

    public void statisztika() {
        for (int i=0; i<LETSZAM; i++)
            korcsGyujto[emberek[i].korcsoport()]++;

        System.out.println("\nKorcsoport Darab");
        for (int i=0; i<korcsGyujto.length; i++) {
            System.out.print(Format.right(Ember.evAlso(i),2)+" - "
                Format.right(Ember.evFelső(i),2));
            System.out.println(Format.right(korcsGyujto[i],3));
        }
    }

    public static void main(String[] args) {
        Eletkor ek = new Eletkor();
        ek.adatGyujtes();
        ek.statisztika();
    }
}

```

Feladat – Betűgyűjtés

Kérjünk be egy szöveget, majd írjuk ki, hogy A-tól Z-ig melyik betűből hány van a szövegben! A kis- és nagybetűket ne különböztessük meg, és a nem ebbe a tartományba eső karaktereket ne számoljuk!

Forráskód

```

import extra.*;
public class BetuGyujtes {
    public static void main (String args[]) {
        String szoveg = Console.readLine("Szöveg: ");
        int[] karakterek = new int['Z'-'A'+1]; //1
        char karakter;
        for (int i=0; i<szoveg.length(); i++) { //2
            karakter = Character.toUpperCase(szoveg.charAt(i));
            if (karakter>='A' && karakter<='Z')
                karakterek[karakter-'A']++;
        }
    }
}

```

```

for (int i=0; i<karakterek.length; i++) {                                //3
    if (karakterek[i] != 0)
        System.out.println((char)('A'+i)+": "+karakterek[i]);
}
}
}

```

A program egy lehetséges futása

| |
|------------------|
| Szöveg: Merem-e? |
| E: 3 |
| M: 2 |
| R: 1 |

A program elemzése

A Javában sajnos nem tudunk karakterekkel indexelni, ezért az 'A'..'Z' karaktertartományt kell képeznünk a 0..25 számtartományra. A karakterek tömbben fogjuk gyűjteni a megfelelő karakterek darabszámát: a 0. elemben az 'A' betükét, az 1. elemben a 'B' betükét stb., a 25. elemben a 'Z' betükét (//1). A tartomány méretét a szélső karakterek unikódjaiból számoltuk ki ('Z' - 'A' +1). A //2-nél kezdődő for ciklusban végigmegyünk a szóban forgó szövegen, minden egyes karakterét nagybetűssé alakítjuk, és megvizsgáljuk, beleesik-e az 'A'..'Z' karaktertartományba. Ha igen, akkor a megfelelő tömbelemet növeljük eggyel (az indexelésnél implicit típuskonverzió történik). //3-ban végigmegyünk a karakterek tömbön, és kiírjuk az eredményt.

Megjegyzés: A karakterek tömb nullázása felesleges lenne, mert azt a rendszer alapértelmezés szerint elvégzi.

Oldja meg a fejezet végén található egydimenziós tömbökkel kapcsolatos feladatokat!

18.5. Kétdimenziós tömb

Kétdimenziós tömb **deklarálásakor** két tömbképző operátort teszünk az elemtípus után:

```
<elemtípus>[][] <tömbAzonosító>
```

A teljes kétdimenziós tömb **létrehozása**:

```
new <elemtípus> [méret0][méret1]
```

Így a tömb elemeinek a száma $\text{méret0} * \text{méret1}$ lesz. A kétdimenziós tömb lépéseként is létrehozható – ekkor a beágyazott tömbök különböző méretűek is lehetnek. A kétdimenziós tömb sor-referenciáinak létrehozása:

```
new <elemtípus> [méret0][]
```

A kétdimenziós tömb alapelemeinek **indexelése**:

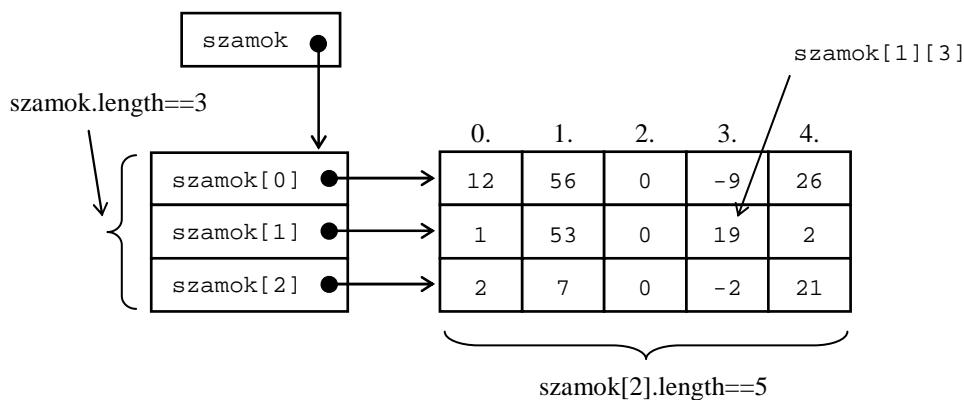
```
<tömbazonosító> [index0][index1]
```

Minden indexI egy 0 és mereti-1 közötti érték.

A többdimenziós tömb elemeinek **alapértelmezés szerinti kezdeti értékei** ugyanolyanok, mint az egydimenziós tömb esetén: referenciaiak esetén az érték null, primitív típusú változók esetén 0, \u0000, illetve false.

Az **inicializáló blokkok** egymásba ágyazhatók:

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



18.8. ábra. Primitív elemtípusú kétdimenziós tömb

Példaként vegyük a `szamok` tömböt, melynek 3 sora és 5 oszlopa van, elemtípusa int:

```
int[][] szamok = new int[3][5];
```

A primitív elemtípusú kétdimenziós tömböt a 18.8. ábra mutatja. A sorok indexei 0 és 2, az oszlopok indexei 0 és 4 között lehetnek. A `szamok` 1. sorának 3. eleme: `szamok[1][3]`. Az 5 elemű sorokra külön lehet hivatkozni: `szamok[0]`, `szamok[1]`, `szamok[2]`. Az oszlopokra külön nem hivatkozhatunk.

Másik példaként vegyük egy olyan kétdimenziós tömböt, amelyben korosztályonként tároljuk az első három helyezett versenyzőt. A korosztályok száma 4. A referencia elemtípusú két-dimenziós tömböt a 18.9. ábra szemlélteti.

- ◆ 1. eset: Egyetlen utasítással létrehozzuk az összes referenciát:

```
Ember[][] versenyzok = new Ember[4][3];
```

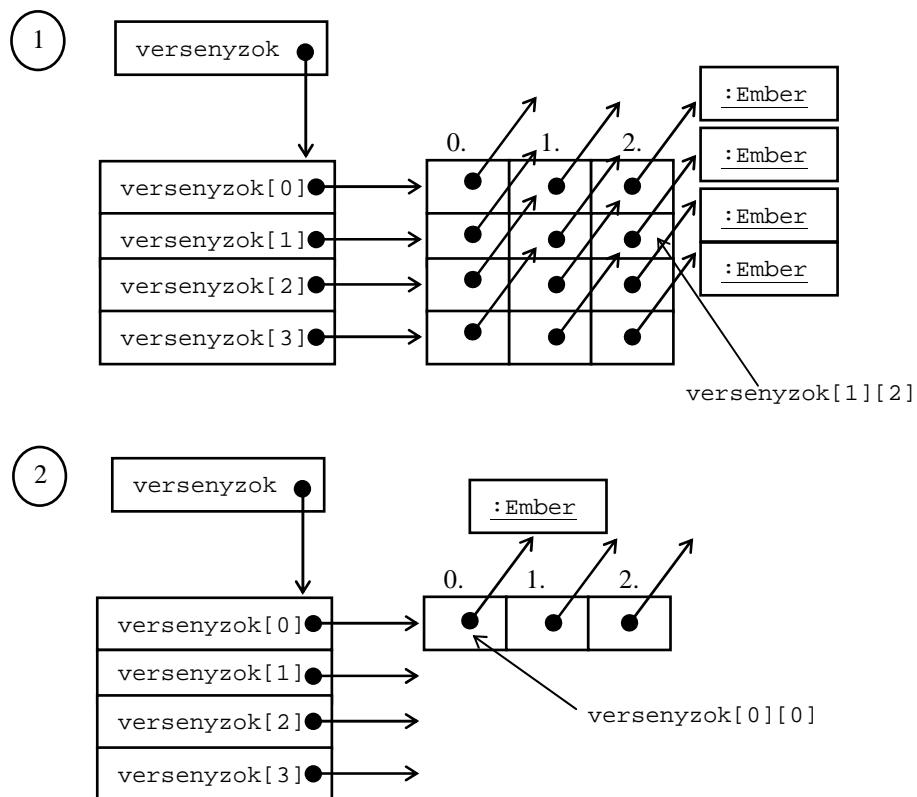
A létrehozás után a versenyzők tömb mind a 12 referenciajának értéke null – emberek még nincsenek a tömbhöz rendelve. Az embereket külön kell létrehozni:

```
versenyzok[i][j] = new Ember(...);
```

- ◆ 2. eset: Megtehetjük, hogy az egyes korosztályokhoz tartozó tömböket külön hozzuk létre, sőt azok különböző méretűek is lehetnek:

```
Ember[][] versenyzok = new Ember[4]; //1
versenyzok[0] = new Ember[3]; //2
versenyzok[0][1] = new Ember(...); //3
```

Az egyes korosztályok tömbjeinek létrehozzuk a referenciáit (//1), de most csak a legelső korosztály versenyzőinek foglalunk helyet (//2). A létrehozott tömbben az első referenciahoz egy embert rendelünk, a többi még meghatározatlan (értéke null).



18.9. ábra. Referencia elemtípusú kétdimenziós tömbök

Tömbök másolása

A `java.lang` csomag `System` egységében van egy statikus `arraycopy` metódus, melynek segítségével azonos típusú tömbrészleteket lehet átmásolni egyik tömbből a másikba:

- `static void arraycopy(Object src, int src_position, Object dst, int dst_position, int length);`
src (source) a forrástömb, melynek src_position elemétől kezdve a metódus átmásolja az elemeket a dst (destination) nevű cél tömbbe a dst_position elemtől kezdve, length hosszúságban.

Tesztprogram – Mátrix

A következő tesztprogram a `matrix` nevű kétdimenziós tömböt manipulálja. minden egyes mátrixművelet után kiírjuk konzolra a mátrix tartalmát a `printMatrix()` metódussal. Kövesse végig a programot, és minden egyes művelet után ellenőrizze a program után szereplő futási eredményt.

Forráskód

```

// A 2. sor 1. eleme 999 lesz:
matrix[2][1] = 999;
printMatrix();                                // mátrix kiírása      //3

// 2. oszlop feltöltése 0 értékekkel:
for (int i=0; i<5; i++)
    matrix[i][2] = 0;
printMatrix();                                // mátrix kiírása      //4

// Ezen értékkadás után a 0. sor ugyanaz mint a 4.!
// A két referencia ugyanarra az tömbre mutat.
matrix[0] = matrix[4];                         // A 0. sor elvész!
printMatrix();                                // mátrix kiírása      //5

// Értékkadás a 4. sor 1. elemnek. 0. sor. 1. elemnek is!
matrix[4][1] = -1;
printMatrix();                                // mátrix kiírása      //6
}
}

```

A program futása

| | | |
|----------------|-----|----|
| Sorok száma | =5 | |
| Oszlopok száma | =3 | |
| ---- 1 ----- | | |
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |
| 10 | 11 | 12 |
| 13 | 14 | 15 |
| ---- 2 ----- | | |
| 1 | 2 | 3 |
| 7 | 8 | 9 |
| 7 | 8 | 9 |
| 10 | 11 | 12 |
| 13 | 14 | 15 |
| ---- 3 ----- | | |
| 1 | 2 | 3 |
| 7 | 8 | 9 |
| 7 | 999 | 9 |
| 10 | 11 | 12 |
| 13 | 14 | 15 |
| ---- 4 ----- | | |
| 1 | 2 | 0 |
| 7 | 8 | 0 |
| 7 | 999 | 0 |
| 10 | 11 | 0 |
| 13 | 14 | 0 |
| ---- 5 ----- | | |
| 13 | 14 | 0 |
| 7 | 8 | 0 |
| 7 | 999 | 0 |
| 10 | 11 | 0 |
| 13 | 14 | 0 |

| ----- | | |
|-------|-----|---|
| 13 | -1 | 0 |
| 7 | 8 | 0 |
| 7 | 999 | 0 |
| 10 | 11 | 0 |
| 13 | -1 | 0 |

A program elemzése

A program remélhetőleg „magáért beszél”, csupán néhány dolog érdemel külön említést.

A program azzal kezdődik, hogy kiírjuk a mátrix sorainak számát (`matrix.length`), vagyis a tömb külső méretét, majd a 0. sor oszlopainak számát (`matrix[0].length`). Itt most minden sor egyforma hosszú. Ezután az egyes lépések a következők:

- //1: Feltöltyük a mátrix elemeit.
- //2: A tömb 1. sorát átmásoljuk a tömb 2. sorába az `arraycopy` metódus segítségével.
- //3: A tömb 2. sora 1. elemének új értéket adtunk, a 999-et.
- //4: Feltöltyük a 2. oszlopot nullákkal.
- //5: A `matrix[0]=matrix[4];` utasítás hatására a tömb 0. és 4. sora eggyaránt az eddig negyedik sorra fog mutatni. Ezért ezzel az utasítással nemcsak azt érjük el, hogy a két sor most ugyanolyan értékű elemeket fog tartalmazni, hanem bármilyen változtatást hozunk létre akár az egyik, akár a másik soron, az minden sor megváltoztatását jelenti, hiszen a két sor egy és ugyanaz.
- //6: A 4. sor 1. elemét megváltoztatjuk. Figyelje meg, hogy //5 következményeként a 0. sor 1. eleme is megváltozik!

18.6. Többdimenziós tömb

Az N dimenziós tömb deklarálása (N darab tömbképző operátor):

```
<elemtípus>[ ][ ]...[ ] <tömbAzonosító>
```

A teljes N dimenziós tömb létrehozása:

```
new <elemtípus> [méret0][méret1]...[méretN-1]
```

Így a tömb elemeinek a száma $\text{méret0} * \text{méret1} * \dots * \text{méretN-1}$ lesz. A létrehozás lépésenként is elvégezhető – ekkor a beágyazott tömbök különböző méretűek is lehetnek.

A többdimenziós tömb alapelemeinek indexelése:

```
<tömbAzonosító> [index0][index1]...[indexN-1]
```

Minden `indexI` egy 0 és `méretN-1` közötti érték.

A `<tömbAzonosító>.length` a `méret0-t` adja vissza. A `<tömbazonosító>[index0]` résztömb méretét a `<tömbAzonosító>[index0].length` adja meg stb.

Az egész N dimenziós tömbre a tömb azonosítójával hivatkozunk. A MéretN darab résztömb mindegyikére már eggyel kevesebb indexre van szükség. Ahogy az indexek fogynak, úgy fogy

a résztömb dimenziója. A tömb egy alapelemére való hivatkozáshoz már N indexre van szükség.

Például a 4 dimenziós t tömb hivatkozásai a következők:

| | |
|----------------------------------|-----------------------------------|
| az egész 4 dimenziós tömb: | t |
| egy 3 dimenziós résztömb: | t[index0] |
| egy 2 dimenziós résztömb: | t[index0][index1] |
| egy 1 dimenziós résztömb (sor): | t[index0][index1][index2] |
| egy 0 dimenziós résztömb (elem): | t[index0][index1][index2][index3] |

Példaként nézzük a dArray háromdimenziós tömböt, amely 3 magas, 2 soros, 10 oszlopos, és elemtípusa double (18.10. ábra). Deklarációja a következő:

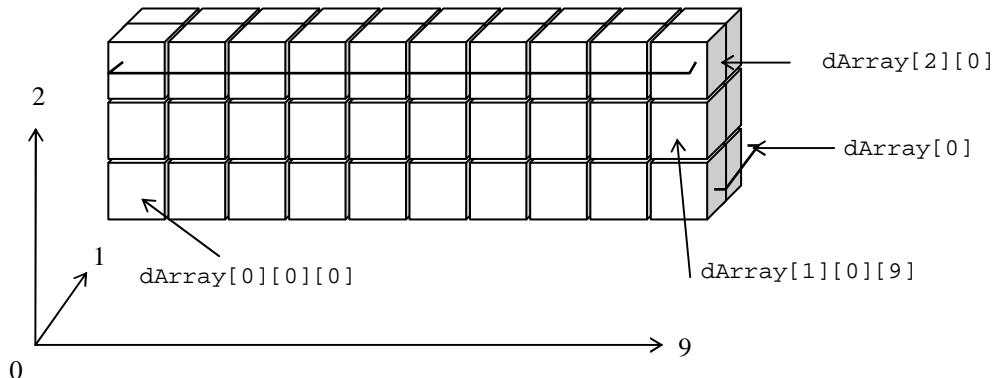
```
double[][][] dArray = new double[3][2][10];
```

Igazak a következő egyenlőségek:

```
dArray.length == 3           // a dArray magassága 3
dArray[0].length == 2         // a dArray[0] lap 2 soros
dArray[2][0].length == 10    // a dArray[2][0] sor 10 elemű
```

Az ábrán jelölt felső sor (2-es szint 0. sora) feltöltése 9-es értékekkel:

```
for (int k=0; k<dArray[2][0].length; k++)
    dArray[2][0][k] = 9;
```



18.10. ábra. Háromdimenziós tömb

18.7. A tömb átadása paraméterként

Egy tömb átadható paraméterként egy metódusnak. Formális paraméterként egy tömbreferenciát kell deklárnunk, amely fogadja az aktuális tömb referenciáját. Az aktuális tömbnek értékkadás szerint kompatibilisnek kell lennie a formális tömbbel, vagyis:

- Primitív elemtípus esetén a formális tömb elemtípusa ugyanolyan kell, hogy legyen, mint az aktuális tömbé;
- Referencia elemtípus esetén egy formális tömb elemtípusának az aktuális tömb elemtípusával azonosnak, vagy annak egy ősének kell lennie.

A formális paraméter tetszőleges hosszúságú tömböt fogadhat.

Feladat – Paraméter teszt

Írunk eljárást, amely egy

- int elemtípusú tömb elemeit kilistázza!
- Object elemtípusú tömb elemeit kilistázza!

Forráskód

```
public class ParamTeszt {

    // A formális paraméter primitív elemtípusú tömb:
    static void kiir(int[] tomb) {
        for (int i=0; i<tomb.length; i++)
            System.out.print(tomb[i]+" ");
        System.out.println();
    }

    // A formális paraméter Object elemtípusú tömb:
    static void kiir(Object[] tomb) {
        for (int i=0; i<tomb.length; i++)
            System.out.print(tomb[i]+" ");
        System.out.println();
    }

    public static void main(String[] args) {
        int[] szamok1 = new int[2], szamok2 = new int[5];
        szamok1[1] = 5;
        szamok2[0] = 99; szamok2[4] = -5;
        kiir(szamok1); kiir(szamok2);

        String[] szerzok1 = {"Mozart", "Wagner", "Beethoven"};
        StringBuffer[] szerzok2 = new StringBuffer[2];
        szerzok2[0] = new StringBuffer("Monteverdi");
        szerzok2[1] = new StringBuffer("Corelli");
        kiir(szerzok1); kiir(szerzok2);
    }
}
```

A program futása

```

0 5
99 0 0 0 -5
Mozart Wagner Beethoven
Monteverdi Corelli

```

A program elemzése

A `kiir(int[] tomb)` eljárásnak csak `int` elemtípusú tömb adható át, de az természetesen bármilyen hosszú lehet. Az aktuális tömbök 2 és 5 hosszúak. A `kiir(Object[] tomb)` eljárásnak csak referencia elemtípusú tömb adható át paraméterként. Az elemek osztályainak az `Object` osztályból kell származniuk. Az egyik aktuális tömb elemei `String` típusúak, a másiké pedig `StringBuffer`. A `tomb` akármilyen méretű elemekből álló tömböt képes fogadni. A fenti `kiir` metódus kiírja az aktuális tömb elemeit.

18.8. A program paraméterei

A program elindításakor a programnak tetszőleges számú paramétert adhatunk át:

```
java <OsztályAzonosító> <param0> <param1> <param2> ...
```

A paraméterek (argumentumok) szövegek, melyeket fehér szóközök választanak el egymástól. Az aktuális paramétereket a `main` metódus fogadja a `String` elemtípusú formális paramétertömbben (szokásos elnevezése `args`). Az aktuális paraméterek számát az `args.length` adja meg.

A program paramétereit parancssor-paramétereknek vagy a program argumentumainak is nevezik.

Feladat – Program paraméter teszt

Készítsen egy programot, mely a program paramétereit kiírja a konzolra először egyenként, külön sorban, majd egy sorban, vesszővel elválasztva! Ha nem adnak meg paramétert, akkor írjuk ki, hogy "A programnak nincs paramétere."!

Próbaként futtassa le a programot a következő három paraméterrel:

```
java ProgParamTeszt Yellow Submarine Yesterday
```

Forráskód

```

public class ProgParamTeszt {

    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("A programnak nincs paramétere.");
            System.exit(0);
        }
    }
}

```

```
System.out.println("\nParaméterek egyenként, külön sorban:");
for (int i = 0; i < args.length; i++)
    System.out.println(args[i]);

System.out.println("\nParaméterek vesszővel elválasztva:");

StringBuffer sor = new StringBuffer(args[0]);
for (int i = 1; i < args.length; i++) {
    sor.append(", " + args[i]);
}
System.out.println(sor);
}
```

A program futása a fenti paraméterek mellett

```
| Paraméterek egyenként, külön sorban:
| Yellow
| Submarine
| Yesterday

| Paraméterek vesszővel elválasztva:
| Yellow, Submarine, Yesterday
```

JBuilder – futási konfiguráció

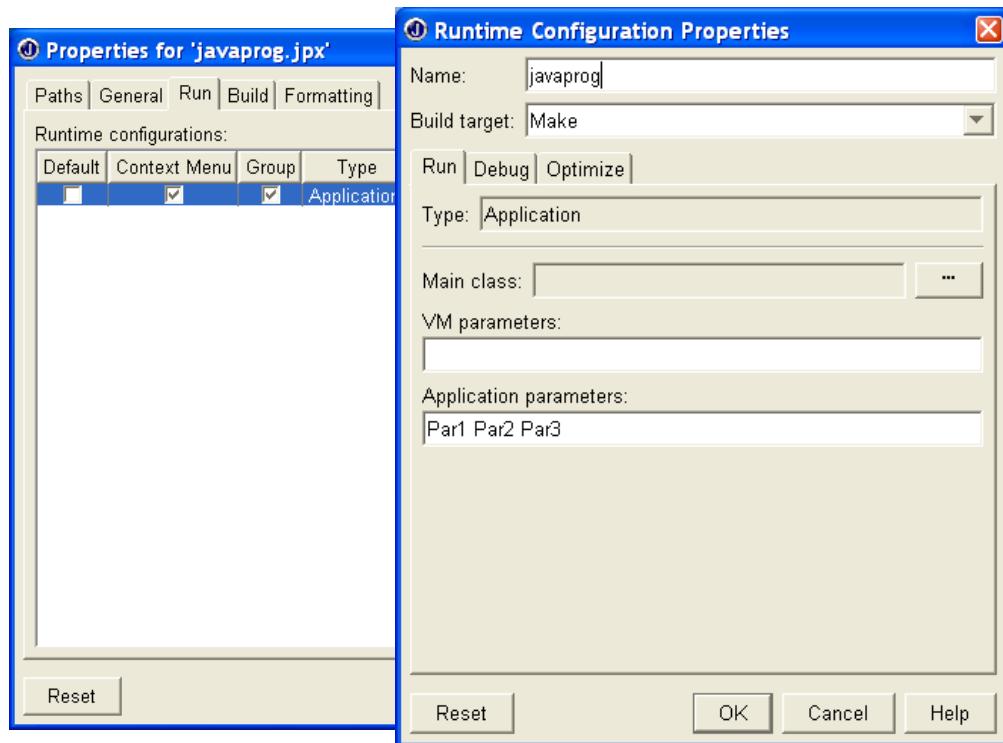
A program paramétereit JBuilderben a futási konfiguráció ablakában állíthatjuk be. Válasszuk ki a következő menüpontok egyikét:

- ◆ *Project/Project Properties...*
- ◆ *Run/Configurations...*

Mindkét esetben a Projekt tulajdonságai (*Project Properties*) ablak jelenik meg, második esetben rögtön a *Run* fül lesz az aktuális. A *Run* fülön ki kell választanunk az aktuális futási konfigurációt (*Runtime Configuration*), melyet szerkesztenünk kell: *Edit*. Ha még nincs futási konfiguráció, hozzunk létre egyet!

Megjelenik a Futási konfiguráció tulajdonságai ablak (*Runtime Configuration Properties*, 18.11. ábra). A program paramétereit az *Application parameters* mezőbe kell beírni, szóközzel elválasztva.

Megjegyzés: Ebben az ablakban állíthatjuk be többek között a projekt fő osztályát (*Main class*) is, amely a *main* metódust tartalmazza. E beállításnak csak valódi projekt esetén van értelme, ahol futtatjuk is a projektet.



18.11. ábra. A program paraméterei

18.9. Feladat – Szavazatkiértékelés

Feladat

Egy sportversenyen közönségszavazást tartanak. minden néző egyetlen versenyőre szavazhat. A versenyzők különböző kategóriákban indulnak, a kategóriák számozása 1-től indul. minden kategóriában a versenyzők sorszámot kapnak 0-tól kezdődően. Hogy hány kategória van összesen, és kategóriánként hány versenyző indul, azt a program paramétereiként adjuk meg. 5 kategória esetén például: 6 10 5 3 7.

A közönségben mindenki kap egy szavazócédulát, melyen kitöltheti a kategóriát és a versenyző sorszámát. A cédrulát egy ládikába kell bedobni. A ládikában levő cédrulák száma tetszőleges.

Készítsünk egy olyan programot, amely feldolgozza és kiértékeli a szavazócédulákat!

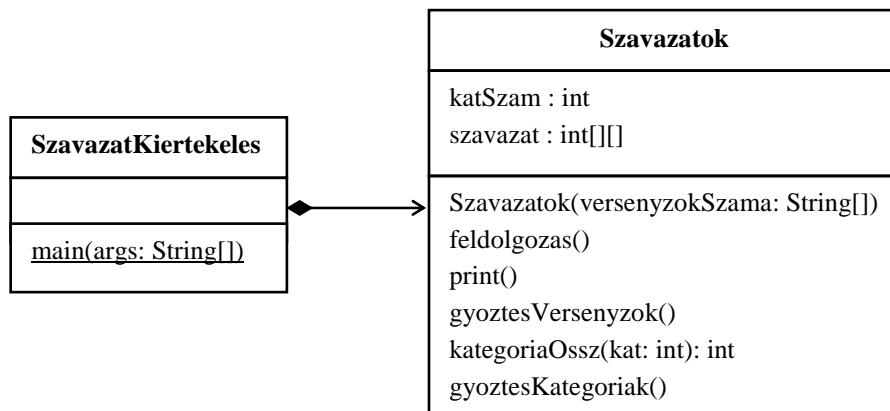
- Először vigyük be a szavazócédulák adatait (rossz adatokat tartalmazó cédrulák nem vesznek részt a szavazásban)!

Végül készítsük el a következő kimutatást:

- Kategóriánként és versenyzőnként írjuk ki a szavazatok számát!
- Írjuk ki azon versenyzők kategóriáját és sorszámát, akik a legtöbb szavazatot kapták a teljes mezőnyben! (Egyforma eredmény esetén mindegyiket írjuk ki!)
- Mely kategória kapta összesen a legtöbb szavazatot? (Egyforma eredmény esetén mindegyiket írjuk ki!)

A program terve

A Szavazatok osztály lesz a felelős a szavazatok tárolásáért, feldolgozásáért és a különböző kiértékelésekért. Tervezzük meg az osztály adatait és algoritmusait! A program terve a 18.12. ábrán látható.



18.12. ábra. A SzavazatKiértékelés program terve

Szavazatok – osztályleírás

A versenyző sorszáma

| Adatok: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|-----------------|---|---|---|---|---|---|---|---|---|---|
| szavazat: | 1 | 3 | 0 | 7 | 2 | 0 | 0 | | | |
| | 2 | 0 | 0 | 0 | 1 | 2 | 3 | 5 | 1 | 0 |
| Kategória száma | 3 | 1 | 3 | 0 | 0 | 1 | | | | |
| | 4 | 3 | 1 | 5 | | | | | | |
| | 5 | 1 | 0 | 0 | 5 | 4 | 9 | 4 | | |

A szavazat tömbben paramétertől függően állítjuk be a sorok (kategóriák) és azon belül az oszlopok (versenyzők) számát. Az oszlopok száma soronként különböző lehet. A kategóriát a tömbben 1-től számozzuk, hogy ne kelljen transzformálni a szavazócédulán szereplő értékeket. **katSzam** a kategóriák számát tárolja.

Metódusok

- ▶ **Szavazatok(String[] versenyzokSzama)**
Konstruktur. A program paramétere alapján létrehozza a szavazat tömböt.
- ▶ **void feldolgozas()**
Ha egy cédulán a kategória **kat**, és a versenyző sorszáma **vsz**, akkor a tömb **kat**. **sorának** **vsz**. elemét meg kell növelnünk eggyel. A szavazócédulák végét a kategóriánál beütött 0 jelzi. Tegyük fel, hogy az eredmény az lesz, amit a kitöltött ábra mutat.
- ▶ **void print()**
A külső ciklus kategóriánként fut. minden kategóriára kiírjuk a versenyzők sorszámát, alá pedig a megfelelő versenyző szavazatainak számát.
- ▶ **void gyozesVersenyzok()**
Megállapítja, kik kapták a legtöbb szavazatot: végigmegyünk sorfolytonosan a tömbön, és maximumszámítási algoritmussal kiválasztjuk a legnagyobb értéket. Ezután még egyszer végigmegyünk, és kiírjuk, ki kapott ennyi szavazatszámot.
- ▶ **void gyozesKategoriak()**
A legtöbb szavazatban részesített kategóriák meghatározása: készítünk egy függvényt (**kategoriaOssz**), mely egy adott kategória összes szavazatát kiszámítja. E függvény segítségével végigmegyünk a kategóriákon, és szintén maximumot számítunk.
- ▶ **int kategoriaOssz(int kat)**
Kiszámítja az adott kategóriában leadott összes szavaztok számát: a kategória sorának összegét.

Forráskód

```
import extra.*;  
  
class Szavazatok {  
    private int katSzam;  
    private int[][] szavazat;  
  
    public Szavazatok(String[] versenyzokSzama) {  
        System.out.println(versenyzokSzama.length);  
        katSzam = versenyzokSzama.length;
```

```
// Kategóriák létrehozása:  
szavazat = new int[katSzam+1][]; // 0. kategória nincs  
for (int k=1; k<=katSzam; k++)  
    // Kategóriánként résztömb létrehozása:  
    szavazat[k] = new int[Integer.parseInt(  
        versenyzokSzama[k-1])];  
  
}  
  
public void feldolgozas() {  
    int kat, vsz;  
    do {  
        kat = Console.readInt("Kategória : ");  
        if (kat == 0)  
            continue;  
        if (kat<1 || kat>katSzam) {  
            System.out.println("Rossz kategória");  
            continue;  
        }  
        vsz = Console.readInt("Versenyző: ");  
        if (vsz<0 || vsz>=szavazat[kat].length) {  
            System.out.println("Rossz versenyző sorszám");  
            continue;  
        }  
        szavazat[kat][vsz]++; // a megfelelő érték növelése  
    } while (kat != 0);  
}  
  
public void print() {  
    for (int i=1; i<=katSzam; i++) {  
        System.out.println("\nSzavazatok az "+i+". kategóriában");  
        // A versenyzők sorszámainak kiírása:  
        System.out.print("Vers. sorszáma: ");  
        for (int j=0; j<szavazat[i].length; j++)  
            System.out.print(Format.right(j,3));  
        System.out.println();  
  
        // A versenyzők szavazatainak kiírása:  
        System.out.print("Szavazatok : ");  
        for (int j=0; j<szavazat[i].length; j++)  
            System.out.print(Format.right(szavazat[i][j],3));  
        System.out.println();  
        System.out.println("Összesen: "+kategoriaOssz(i)+" szav.");  
    }  
}  
  
public void gyoztesVersenyzok() {  
  
    // Maximális érték meghatározása  
    int max = 0;  
    for (int i=1; i<szavazat.length; i++)  
        for (int j=0; j<szavazat[i].length; j++)  
            if (szavazat[i][j] > max)  
                max = szavazat[i][j];
```

```

// A győztesek kiírása
System.out.println("Győztes versenyzők:");
for (int i=1; i<szavazat.length; i++) {
    for (int j=0; j<szavazat[i].length; j++)
        if (szavazat[i][j] == max)
            System.out.println(i+". kat.ban a "+j+". vers.");
    System.out.println();
}
}

public int kategoriaOssz(int kat) {
    int sum = 0;
    for (int j=0; j<szavazat[kat].length; j++)
        sum += szavazat[kat][j];
    return sum;
}

public void gyoztesKategoriak() {
    int max = 0;
    for (int i=1; i<szavazat.length; i++)
        if (kategoriaOssz(i) > max)
            max = kategoriaOssz(i);

    System.out.println("Győztes kategoriák:");
    for (int i=1; i<szavazat.length; i++)
        if (kategoriaOssz(i) == max)
            System.out.print(i+" ");
    System.out.println();
}
}

public class SzavazatKiertekeles {
    public static void main(String[] args) {
        Szavazatok szavazatok = new Szavazatok(args);

        szavazatok.feldolgozas();
        szavazatok.print();
        szavazatok.gyoztesVersenyzok();
        szavazatok.gyoztesKategoriak();
    }
}

```

Tesztkérdések

- 18.1. Mely deklarációk határoznak meg egy olyan tömböt, amely 8 darab karakterelemet tartalmaz? Jelölje meg az összes jó választ!
- char[] betuk = new char[7];
 - String[] szo = new szo[8];
 - char[] jel = new char[8];
 - char jel[] = new char[8];

- 18.2. Jelölje meg az összes szintaktikailag helyes deklarációt!
- a) `int[] t1;`
 - b) `int t2[] = new t2;`
 - c) `int[][] t3;`
 - d) `int[] t4 = new int[5];`
- 18.3. Jelölje meg az összes helyes állítást a következők közül!
- a) A kétdimenziós tömb sorai mindenkor egyforma hosszúak.
 - b) A kétdimenziós tömb méretét deklaráláskor meg kell adni.
 - c) A kétdimenziós tömb egyes sorai külön létrehozhatók.
 - d) A kétdimenziós tömb sorai futás közben bővíthetők (az eredeti sor megszüntetése nélkül).
- 18.4. Jelölje meg az összes helyes állítást a következők közül!
- a) Ha a metódus formális paramétere egy `int` elemtípusú tömb, akkor az aktuális paraméter is csak `int` elemtípusú tömb lehet.
 - b) Ha a metódus formális paramétere `Hallgato[]` típusú, akkor az aktuális paraméter is csak `Hallgato[]` típusú lehet.
 - c) A programnak a `main` metódus paramétereinek átadhatók adatok.
 - d) A `main` metódus formális paramétere kötelezően `String` elemtípusú tömb.
- 18.5. Adva van egy kétdimenziós tömb:
- ```
int[][] tomb = new int[3][6];
```
- Hogy hivatkozunk a tömb utolsó sorára? Jelölje meg az összes jó választ!
- a) `tomb[3]`
  - b) `tomb[3][]`
  - c) `tomb[2]`
  - d) `tomb[2][]`
- 18.6. Adva van egy háromdimenziós tömb:
- ```
int[][][] tomb = new int[2][5][3];
```
- Melyek azok a hivatkozások, amelyek szintaktikailag is helyesek, és nem mutatnak a tömbön kívülre? Jelölje meg az összes jó választ!
- a) `tomb[2][4]`
 - b) `tomb[1]`
 - c) `tomb[0,0,2]`
 - d) `tomb[][4][1]`
- 18.7. Mely kifejezések adják meg a kétdimenziós tömb utolsó sorának a hosszát? Jelölje meg az összes jó választ!
- ```
int[][] tomb;
...
```
- a) `tomb.length(0)`
  - b) `tomb[length-1].length()`
  - c) `tomb[tomb.length-1].length`
  - d) `tomb[length-1].length`

## Feladatok

### Egydimenziós tömbök

- 18.1. **(A)** Kérjen be 10 darab egész számot! Ezután írja ki a 10-nél nagyobb számokat, majd a 10-nél kisebbeket! (*Szamok.java*)
- 18.2. **(A)** Mérjük meg 10 darab alma súlyát, és a mért értékeket sorban vigyük be a számítógéphez. Kérdés, hogy melyik alma súlya tér el legjobban az átlagtól? Írjuk ki a kérdéses alma sorszámát, és eltérését az átlagtól (ha több van, akkor csak az elsőt)! Hogy ne tévesszük össze az almákat, számozzuk be azokat 1-től kezdve. (*Sulyok.java*)
- 18.3. **(A)** Egy börtönben 1000 cella van, minden egyikben egy rab ül. Kezdetben minden cella zárva van. A börtönök játszani támad kedve: végigmegy az összes cella előtt, és minden egyik ajtó zárján fordít egyet. Fordításkor a nyitott cellát bezárja, illetve a zártat kinyitja. Ha végigment, elkezdi előlről, és minden második cella zárján fordít egyet. Aztán minden harmadikon fordít, és így tovább. Legvégül fordít egyet az ezrediken, és kész. Ezután amelyik cella ajtaja nincs bezárva, abból a rab elmehet. Kik a szerencsés rabok? (*Borton.java*)
- 18.4. **(A)** Generáljon 100 darab 100 és 200 közötti véletlen egész számot (a határokat is beleértve)! Vizsgálja meg és írja ki, hogy melyik számból összesen mennyi keletkezett! (*VeletlenEloszlas.java*)
- 18.5. **(B)** Kérjen be egy szöveget, és számolja meg a benne levő
  - a) A, B, ... , Z betűk számait (betűnként)! Ne különböztesse meg a nagy és kisbetűket!
  - b) betűk számait! A nagy- és kisbetűket is különböztesse meg!
  - c) 0, 1, 2,...9 számjegyek számait!
  - d) az ASCII karakterek számait! (*KarStat.java*)
- 18.6. **(B)** Kérjen be egy számot, majd írja ki eddig a számig az összes prímszámot! minden prím elő írja ki, hogy hányadik prím!  
Ötlet: Oldjuk meg a feladatot az Eratoszthenészi szita segítségével! Vegyük fel egy tömböt, ahol minden számhoz (indexhez) egy logikai érték tartozik, jelezvén, hogy a szám prím-e (kezdetben minden elemet true-ra kell állítani). Menjünk végig a tömbön, és állítsuk be a 2 többszöröseihez tartozó értékeket (4, 6, 8...) false-ra (ezek biztosan nem prímek)! Ugyanezt végezzük el a 3 többszöröseire, és így tovább: csak a „még” prím értékek többszöröseit nézzük! Végül a true értékű rekeszek indexei lesznek a prímszámok. (*Primek.java*)

18.7. (B) Egy cégnél felmérést végeznek a fizetésekkel. Először kérje be a dolgozók nevét és fizetését! Ezután

- a) állapítsa meg a fizetések átlagát!
- b) írja ki, kiknek tér el a fizetése legjobban az átlagtól!
- c) emelje meg 10%-kal mindenki fizetését!

A megoldáshoz készítsen egy Dolgozó osztályt! (*CegFizetesek.java*)

18.8. (C) Készítse el a számlázó programok elengedhetetlen „azaz” függvényét, amely egy tetszőleges egész számnak meghatározza a szöveges, kimondott alakját. Például:

2002, azaz Kettőezerkettő; 30996, azaz Harmincezerkilencszázkilencvenhat. (*Azaz.java*)

18.9. (C) Egy téren sok ember tartózkodik. Mindegyik emberre jellemző a tartózkodási helye (a tér közepéhez viszonyított pozíciója) és az iránya, képesek menni egy adott távolságot, és tudnak fordulni. Az ünnepi szónok ilyen utasításokat ad ki:

- a) minden ember menjen egy métert!
- b) Akinek az iránya 0 és 90 fok közé esik, menjen két métert!
- c) mindenki forduljon el 30 fokkal!
- d) Aki a tér közepétől legalább 20 méter távolságra van, forduljon meg!

Szimulálja az ünnepi műsort! Az ütközésektől most tekintsünk el! (*MusoraTeren.java*)

### Két- és többdimenziós tömbök

18.10. (A) Adott egy 8\*8-as sakktábla. Úgy játszunk, hogy 100-szor véletlenszerűen rábökünk a sakktábla valamely mezőjére. A játék végén írjuk ki, hogy melyik mezőre hányszor bökünk rá. Egy mezőt egy betűvel (A..H) és egy számmal (1..8) azonosítunk. A rábökést véletlenszám-generátorral végezzük. (*Bokodes.java*)

18.11. (B) Szimuláljuk egy mozi „kissé primitív” jegyeladását. A nézőtéren 10 sor található, abból az első 5 sorban 8 hely van, a többiben 12. Menüből lehet választani:

**J:** Eladnak egy jegyet. Kérjük be, hova kéri a néző a jegyet (sor és székszám)! Csak jó értéket fogadjunk el! Ha a hely már foglalt, akkor adjunk egy figyelmeztető üzenetet! Ha még nem foglalt, foglaljuk le a helyet!

**M:** A néző visszahoz egy jegyet. Kérjük be a jegyén levő sor és székszámot! Csak jó értéket fogadjunk el! Ha a hely nem foglalt, akkor adjunk egy figyelmeztető üzenetet! Ha foglalt, akkor szabadítsuk fel a helyet!

**V:** Vége a programnak.

Minden funkció után „rajzoljuk ki” a nézőteret! Különböztessük meg a foglalt és a szabad helyeket! A nézőter oldalán jelezzük a sor és oszlopszámokat!

Tipp: A nézőternek vegyen fel egy kétdimenziós logikai tömböt, melynek sorai változó hosszúságúak! (*Mozijegyek.java*)

- 18.12. **(C)** Kérjen be számokat konzolról! Készítsen kímutatást, hogy a bekért számok között
- a) összesen hány egyjegyű, kétjegyű stb. szám van; azon belül hány végződik 0-ra, 1-re, 2-re stb.;
  - b) összesen hány szám végződik 0-ra, 1-re, 2-re stb.; azon belül hány egyjegyű, kétjegyű stb. szám van.
  - c) hány olyan szám van, melyben a jegyek száma és a végződés a megadott;
  - d) hány olyan szám van, melyben a jegyek száma a megadott;
  - e) hány olyan szám van, melyben a végződés a megadott;

Csak a feltétlenül szükséges adatokat tárolja! (*JegyVegzodes.java*)

- 18.13. **(B)** Egész évben gyűjtött számláink rendezetlenül hevernek a fiókunkban. Kiadásainkról szeretnénk egy összesítést készíteni naponta, havonta és egész évre vonatkozóan. Készítse el ehhez a programot! Bevitelnél ellenőrizze a dátumot, hogy az adott évben érvényes-e a hónap és a nap! Csak a nem nulla napi eredményeket írja ki! (A számlákat egyenként nem kell megjegyezni, és korrigálásra sincs szükség. A megoldáshoz vegyük fel egy kétdimenziós tömböt, amely alapján havonta, azon belül naponta elvégzhetjük az összesítést!) (*Szamlak.java*)

### Paraméterátadás

- 18.14. Készítsen metódust, amely

- a) **(A)** a paraméterben megadott `double` elemtípusú tömb elemeit a paraméterben megadott határok közötti véletlen értékekkel feltölti!
- b) **(A)** a paraméterben megadott `String` elemtípusú tömb elemeit bekéri a konzolról!
- c) **(A)** konzolra írja egy `double` elemtípusú tömb elemeit, sorszámokkal ellátva!
- d) **(A)** konzolra írja egy `String` elemtípusú tömb elemeit, sorszámokkal ellátva!
- e) **(A)** kiszámítja bármely `double` elemtípusú tömb elemeinek átlagát!
- f) **(B)** bármely `String`eket tartalmazó tömb elemeit fordított sorrendbe teszi!

Tesztelje a metódusokat! (*TombParam.java*)

- 18.15. **(B)** Készítsen egy programot, amely a program paramétereiben megadott számokat összegezi. Tegyük fel, hogy a program paraméterei számok, és azokat fehér szóközök választják el egymástól. (*Osszead.java*)

A program hívása például: `java Osszead 23.6 4 1`

Eredmény: `28.6`



## 19. Rendezés, keresés, karbantartás

---

A fejezet pontjai:

1. Rendezés
  2. Keresés
  3. Karbantartás
  4. Primitív elemek rendezése, keresése
  5. String objektumok rendezése, keresése
  6. Saját osztályú objektumok rendezése, keresése
  7. Szövegek rendezett karbantartása
- 

Szinte nem létezik olyan szoftver, amelyben ne kellene adatokat, objektumokat nyilvántartani. Rögzíteni kell személyi adatokat, megírt leveleket, mérési eredményeket. Gyakori feladat a rögzített elemek (objektumok) valamilyen szempont szerinti rendezése, egy adott tulajdonsággal rendelkező elem megkeresése, illetve az elemek karbantartása (új elem felvitele, régiinek a törlése vagy módosítása). Fontos szempontok a következők:

- ◆ **Az elemek rendezéséhez a sorozat (konténer) bármely két elemét össze kell tudnunk hasonlítani**, azaz meg kell tudnunk állapítani, hogy a rendezett sorozatban melyik szerepel majd előbb. Számos rendezési algoritmus létezik; ezeket a programozó általában készen kapja. A rendezési módszer kiválasztásához és helyes alkalmazásához azonban fontos, hogy nagy vonalakban ismerjük a rendezés elméletét. A fejezet az egyik legegyszerűbb, a minimum-kiválasztásos rendezési eljárást tárgyalja. Tudni kell azonban, hogy ennél vannak sokkal hatékonyabb (és egyben bonyolultabb) rendezések is.
- ◆ **Az elemek karbantartásához** (felvitel, törlés, módosítás) **és az elemek kereséséhez az elemeket azonosítanunk kell**, meg kell tudnunk állapítani egy adott elemről, hogy ez-e az, amit keresünk.

A fejezetben először a rendezés, a keresés és a karbantartás általános elvéről lesz szó. Ezután rendezni fogunk primitív és referencia elemtípusú tömböket, keresni fogunk a rendezetlen és a rendezett tömbökben, valamint a tömbökön karbantartási műveleteket fogunk végezni.

## 19.1. Rendezés

Egy sorozat rendezésének alapfeltétele, hogy a sorozat bármely két eleme összehasonlítható legyen, vagyis eldönthető legyen sorrendiségek. A rendezett sorozatot előállító algoritmust **rendezési algoritmusnak** (eljárásnak, módszernek) nevezük.

A Javában a primitív adatok összehasonlítása a `<`, `>`, `<=`, `>=`, `==`, `!=` összehasonlító operátorokkal történik; **objektumokat** azonban **kizárolag metódusokkal lehet összehasonlítani**. Hogy két objektum közül melyik a nagyobb, valamint az, hogy két objektum egyenlő-e, az meghatározás kérdése.

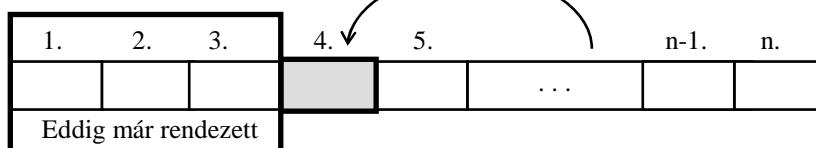
### Minimumkválasztásos rendezés

**A minimumkválasztásos rendezési algoritmus** lényege a következő (19.1. ábra):

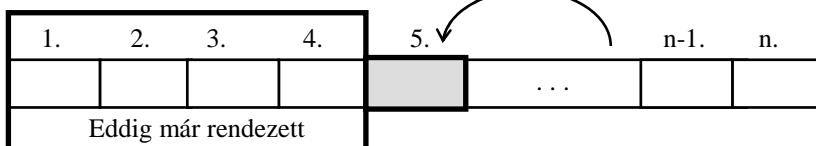
- 1. lépés: Megkeressük a teljes n elemű tömb legkisebb elemét; azt kicseréljük az első elemmel; következésképpen most az első helyen lesz a legkisebb elem, ahogy ezt szereznénk is.
- 2. lépés: most a „maradék” részt, a 2.-től az n. elemig levő tömbrészt kell rendeznünk. Keressük meg itt is a legkisebb elemet! Ezt az elemet kicseréljük a második elemmel, és így most már két elemünk van jó helyen.
- További lépések: folyassuk ezt az eljárást: tegyük a helyére a 3. elemet, a 4. elemet, az 5. elemet stb. Utolsó lépésként az utolsó előtti helyre kell kiválasztanunk a legkisebb elemet, ezzel az utolsó elem is a helyére kerül.

Melyik a legkisebb? Az jöjjön ide!

**4. lépés:**



**5. lépés:**



19.1. ábra. Minimumkválasztásos rendezés

**Megjegyzés:**

Sok féle rendezési algoritmus létezik, például:

- **Beszúrásos vagy más néven beillesztéses rendezés:** Hasonlít arra módszerre, ahogy az ember elrendezi kezében a kártyákat leosztás után. Feltételezzük, hogy van egy szigorú rend, amely szerint a kártyákat sorba kell tenni. Felvesszük az első kártyát. Ezután felvesszük a másodikat, és a helyére tesszük. Felvesszük a harmadikat, helyére tesszük, és így tovább. minden esetben megkeressük a kérdéses kártyának a helyét a már rendezett sorban, és oda beszúrjuk. A tömbbe való beszúrás az elemek feljebb tolásával történik.
- **Buborékos vagy más néven szomszédos elemek cseréjével történő rendezés:** Első lépésben a legnagyobb elem „fel fog szállni” a tömb utolsó helyére, mégpedig a következőképpen: Összehasonlítjuk a tömb első két elemét, és ha az 1. nagyobb, mint a 2., akkor felcseréljük őket. Ezután a 2. és 3. elemmel tesszük ugyanezt stb., a szomszédos elemeket sorban hasonlígtatjuk, és ha kell, cseréljük. A legnagyobb elem, mint egy buborék, felszáll a tömb tetejére. Második lépésként a tömb utolsó előtti helyére szállítjuk fel a bubaréket, majd egyre kisebb tömbréssel buborékoltatunk. minden egyes menetben figyeljük, hogy a tömb nem rendezett-e már. Ha már rendezett, akkor többször nem megyünk végig a tömbön. A buborékos rendezés csupán akkor hatékony, ha rendezettségében „csak egy kicsit elromlott” sorozatról van szó.
- Az egyik leghatékonyabb rendezés a **gyorsrendezés** (quicksort), melynek algoritmusa rekurzív.

A rendezés gyorsaságát elsősorban az összehasonlítások és az elemcserék száma befolyásolja. A rendezési algoritmust a feladat és a rendezendő sorozat jellemzőinek függvényében kell megválasztani.

**Direkt rendezésről** akkor beszélünk, ha a rendezési eljárással az elemek sorrendjét végér-vényesen megváltoztatjuk. **Indexes rendezésről** beszélünk ellenben, ha a sorozat elemeinek eredeti sorrendjét meghagyva, egy olyan indexsorozatot rendezünk, amely azonosítja a sorozat elemeit.

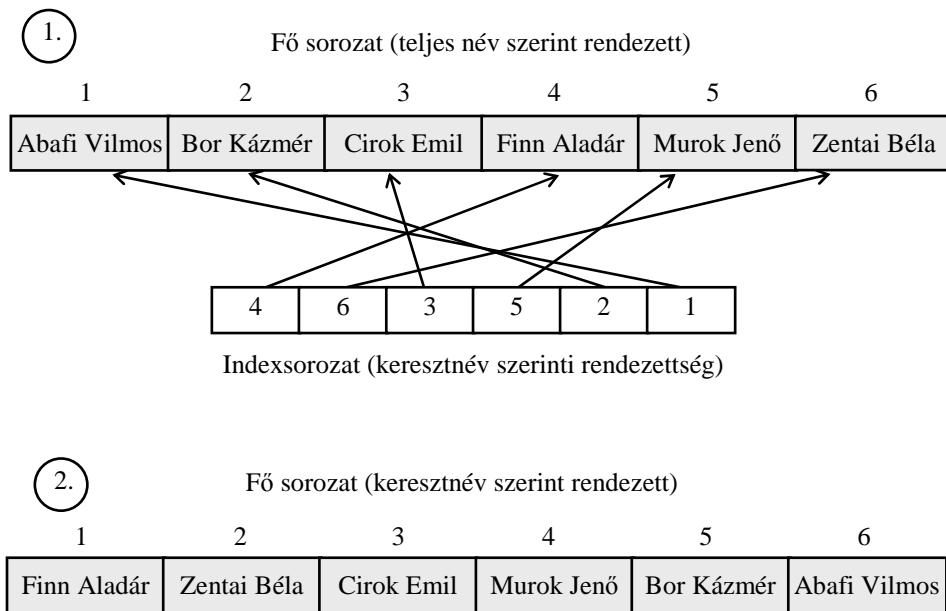
### Indexes rendezés

Az **indexes rendezés** lényege, hogy az eredeti sorozatot változatlanul hagyjuk, és a sorozathoz egy indexsorozatot rendelünk. Az indexsorozatban szereplő indexek az eredeti sorozat elemeit azonosítják, és az adott rendezési szempont szerint követik egymást. Ilyen közvetett módon egyszerre több szempont szerinti rendezettséget is elő tudunk állítani több indexsorozat felépítésével.

A 19.2. ábra 1. esetében a fő sorozat teljes név (vezetéknév+keresztnév) szerint van rendezve. A sorozathoz hozzárendeltünk egy indexsorozatot (4,6,3,5,2,1), amelyben az indexek egy keresztnév szerinti rendezettséget állítanak elő: az indexsorozat legelső eleme a 4-es, ez a fő

sorozatban Aladár indexe (keresztnév szerinti rendezettségen Aladár az első). Az indexsorozat utolsó eleme az 1-es, ez a fő sorozatban Vilmos indexe. Vilmos a keresztnév szerinti rendezettségen az utolsó, bár a fő sorozatban, amely teljes névre van rendezve, első helyen áll. Az indexsorozatban lévő indexekkel az elemek eredeti helyére hivatkozunk. A „rendezett” indexsorozatot úgy tudjuk előállítani, hogy azt feltöljük a rendezendő elemek indexeivel (egyéb információ is kapcsolódhat hozzá), majd az indexsorozat elemeit az indexelt elemek rendezettségének függvényében sorba rakjuk. A hasonlítást tehát minden az indexelt elemekre végezzük el, a csere pedig az indexeket érinti. Így az eredeti sorozatunk változatlan maradhat.

A 2. esetben az eredeti tömböt keresztnév szerint direkt módon rendeztük. Ezzel a teljes név szerinti rendezettség elromlott, így az 1. eset indexei már hamis rendezettséget mutatnak (az indexsorozatot újra fel kell építeni).



19.2. ábra. Indexes és direkt rendezés

## 19.2. Keresés

Kereshetünk akár primitív-, akár referencia típusú elemeket tároló konténerben; ilyenkor minden adott feltételt kielégítő elemet keresünk. Az elemeket sorban megvizsgáljuk, hogy azok a **keresési feltételnek** eleget tesznek-e (valamelyen szempontból egyenlők-e a keresettel): ha igen, akkor az elemet megtaláltuk.

Elképzelhető, hogy több elem is eleget tesz a keresési feltételnek. Primitív adatok esetén egy indexelhető konténerben a keresési feltétel lehet például a következő:

```
if (kontener[i]==10)
 System.out.println("Az i. elem 10");
```

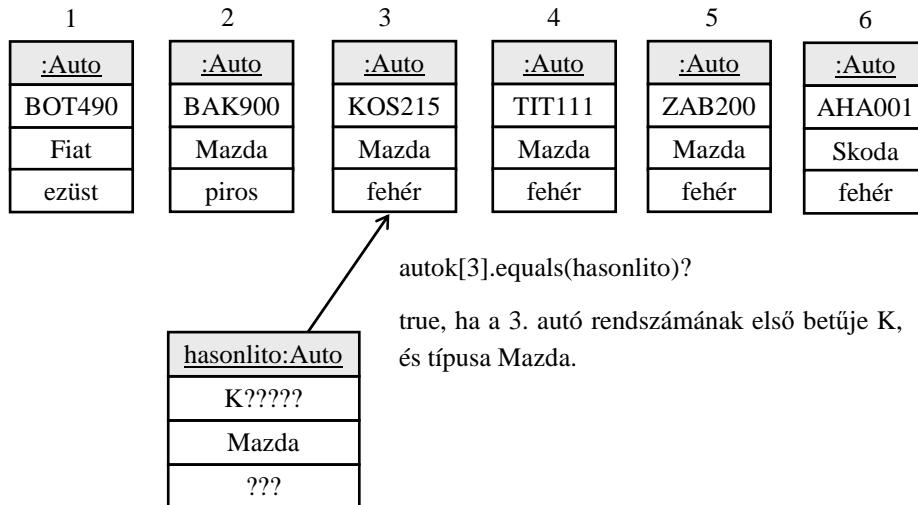
Objektumok keresésekor megfogalmazhatjuk a feltételt az objektumok tulajdonságaival:

```
if (kontener[i].getDb()==10)
 System.out.println("Az i. objektum darabszáma 10");
```

Elegáns megoldás, ha az objektumokat függvényel hasonlítjuk össze, különösen akkor, ha a hasonlításra már vannak kész metódusaink. A hasonlítást elvégző metódus az objektum egy példánymetódusa, amelynek paramétere egy másik objektum. Ilyenkor az objektum összehasonlíta magát egy másik, ún. **hasonlító objektummal**, például:

```
if (kontener[i].equals(hasonlito))
 System.out.println("az i. elem és a hasonlító egyenlők");
```

Az objektumok összehasonlításához a hasonlító objektumot létre kell hoznunk (hacsak nincs kész hasonlítandó objektumunk), amelyben kitöljtük a feltételben részt vevő tulajdonságokat; majd a keresés során ezt az objektumot hasonlítjuk a konténer elemeihez.



19.3. ábra. Objektum keresése

A keresést a 19.3. ábra szemlélteti. Példánk Autó osztályában a következő adatokat deklaráltuk: rendszám (ez egyedi adat), típus és szín. A keresési feltétel most az, hogy a típus, valamint a rendszám kezdőbetűje megegyezzen a megadott értékkal: „Keresendő egy Mazda, amelyiknek a rendszáma K betűvel kezdődik”. A hasonlítás céljára létrehozunk egy Autó objektumot, amelyben kitöljtük a keresendő tulajdonságok értékeit. Ezután sorban megvizsgáljuk a sorozatot, hogy van-e közöttük olyan autó, amelyik „egyenlő” (equals) a hasonlító objektummal, azaz keresünk egy olyan esetet, amikor a megfogalmazott feltételt a két objektumra alkalmazva igaz értéket kapunk. A keresés gyorsabb, ha az autók rendezve vannak típus és azon belül rendszám szerint.

**Szekvenciális keresésről** beszélünk, ha a sorozat elemein sorban, egyesével végighaladva a keresendő elemet minden egyes elemmel összehasonlítjuk addig, amíg azt meg nem találtuk, és amíg még van értelme a keresésnek.

Kereséskor tudnunk kell, hogy a szóban forgó sorozat elemei a keresési szempont szerint vannak-e rendezve, vagy sem, mert e két esetben másképp kell keresnünk:

- Rendezetlen sorozat esetén az elemet addig keressük, amíg meg nem találjuk, vagy a sorozat végére nem érünk.
- Rendezett sorozat esetén a keresést nemcsak akkor hagyjuk abba, ha a keresett elemet megtaláljuk, hanem akkor is, ha rendezettségen túlhaladunk rajta, hiszen a sorozat további részében a kérdéses elem már nem fordulhat elő.

Belátható, hogy rendezett sorozatban való szekvenciális kereséskor átlagban az elemek felét kell megnéznünk.

**Megjegyzés:** Rendezett sorozatban az egyik leghatékonyabb keresési módszer a **bináris keresés**, amelynek lényege a következő:

- Meghatározzuk a tömb középső elemét. Ha ez a keresett elem, akkor már készen is vagyunk.
- Ha az elemet még nem találtuk meg, akkor megvizsgáljuk, hogy a keresett elem a tömb alsó vagy felső felébe esik-e. Mivel a tömb rendezett, ez egyértelműen eldönthető a középső elemmel való összehasonlítással.
- Ha a keresett elem a tömb alsó felébe esik, akkor az új tömbrész a tömb alsó fele, egyébként a tömb felső fele lesz.
- Az új tömbrészben a keresést ugyanúgy folytatjuk, mint ahogy azt eddig leírtuk: meghatározzuk a tömb középső elemét...
- Ezt az eljárást addig folytatjuk, amíg nincs meg a kérdéses elem, és van mit felezni.

Ha a rendezett sorozat elemszáma  $n$ , akkor szekvenciális kereséssel egy elemet átlagban  $n/2$  lépésekben találunk meg; bináris kereséssel a lépések száma legfeljebb  $\log_2(n)$ . 1 millió elem esetén tehát az eredmény 500000:20, a bináris keresés javára.

### 19.3. Karbantartás

**Karbantartás:** az adat-, illetve objektumsorozat folyamatos módosítására irányuló műveletek (tevékenységek) összessége. **Karbantartási műveletek:**

- **beszúrás**, vagy más néven **felvitel** (insert, add): új elem hozzáadása a már meglévő sorozathoz.
- **törlés** (delete, remove): elem eltávolítása a sorozatból.
- **módosítás** (update, modify): a sorozatban lévő elem tulajdonságainak megváltoztatása.

Karbantartani lehet akár primitív, akár referencia típusú elemeket.

Fontos, hogy a felhasználónak minél kevesebbet kelljen várnia az egyes műveletek elvégzésére, és hogy ne foglaljunk le feleslegesen sok memóriát. **Egy programot mind a futási időre, mind a helyfoglalásra optimalizálni kell!**

#### Azonosító, kulcs

Az objektumok karbantartásához az objektumokat azonosítanunk kell! Szerencsére minden objektumnak van programbeli azonosítója. Tudjuk azonban, hogy két nem azonos objektumnak lehet ugyanaz az állapota. A felhasználó elé általában csak az objektum állapotát, vagyis a tulajdonságok értékeit „tállaljuk”. Ha az objektumokat állapotai alapján is meg akarjuk különböztetni, akkor meg kell határoznunk az objektumnak egy olyan elemi vagy összetett tulajdon-ságát, amely egyedi, tehát objektumonként más.

Az objektum egyedi tulajdonságát (tulajdonságait) **kulcsnak** nevezzük. **Karbantartáskor szem előtt kell tartani a kules egyediségét:**

- Új elem felvitelekor már létező kulcsot nem adhatunk meg!
- Az objektum kulcsát megváltoztatni nem szabad!

**Megjegyzés:** Az objektum kulcsát azért nem szabad megváltoztatni, mert annak programozása bonyolult, hibalehetőségeket rejt magában. Ajánlatos az objektumnak születésekor egy belső, mesterséges kulcsot adni, amelyet a felhasználó nem is lát, így nem is akarja azt meg-változtatni.

#### A karbantartandó elemek tárolásának módja

Meg kell gondolnunk, hogy a karbantartandó elemeket memóriában vagy lemezen tároljuk-e, és hogy pontosan milyen konténerben. A programozó a konténerek széles skálájából választhat: a feladat nagyságától és bonyolultságától függően alkalmazhat akár egy egyszerű tömböt, akár egy bonyolult adatbázis-kezelő rendszert. Mi most az egyszerű tömb segítségével mutat-juk be a karbantartás alapvető elveit.

El kell dönteni, hogy a sorozatot rendezetlenül tároljuk, vagy rendezetten. Ez utóbbi esetben el kell dönteni, hogy a rendezettség kulcs szerint vagy az objektum valamely más tulajdonsága szerint történék (indexsorozatok segítségével több szempont szerinti rendezettség is előállítható). Az elemek rendezett tárolása esetén a sorozatnak minden rendezettnek kell lennie, a karbantartó műveleteket eszerint kell elvégezni. Összefoglalva:

- ◆ **Rendezetlen sorozat** esetén sokkal egyszerűbb az objektumokat beszúrni és törölni, a keresés viszont lassú.
- ◆ **Rendezett sorozat** esetén az objektumokat könnyű rendezetten listázni, és a keresés is sokkal gyorsabb. Figyelembe kell azonban venni, hogy egy új elem felvitekor, illetve egy már meglévő elem törlésekor vagy módosításakor a rendezettség megtartásának súlyos ára is lehet.

Karbantartó műveletek esetén a kulcsra mindenkorában rá kell keresnünk! A karbantartás megtervezése nagy elemszám esetén nehéz feladat.

A karbantartás technikájának kidolgozása előtt két dolgot el kell döntenünk:

- Van-e az objektumoknak kulcsa;
- Rendezett-e a konténer a kulcs szerint.

Ezektől függően a beszúrás, módosítás és törlés funkciók alapvetően mások lesznek. Négy esetet kell végignéznünk:

|             | Van kulcs                     | Nincs kulcs                     |
|-------------|-------------------------------|---------------------------------|
| Rendezetlen | <b>Rendezetlen, van kulcs</b> | <b>Rendezetlen, nincs kulcs</b> |
| Rendezett   | <b>Rendezett, van kulcs</b>   | <b>Rendezett, nincs kulcs</b>   |

Ha a konténer rendezett ugyan, de nem a kulcs szerint, az megkönnyít bizonyos kereséseket és listázásokat, de nem könnyíti meg a karbantartást. Karbantartáskor ugyanis a kulcsra mindenkorában rá kell keresnünk, hogy a kulcs duplikálását megakadályozzuk.

Általános programozási szempontok:

- ◆ A kulcsot ne módosítsuk!
- ◆ Új elem felvitele előtt – ha szükséges –, megvizsgáljuk, van-e az elem számára hely. Ha már nincs, akkor a beszúrást nem végezzük el, és erről üzenetet adunk a felhasználónak.
- ◆ Törlés vagy jelentősebb változást előidéző módosítás előtt – igény szerint – kérjünk megerősítést a felhasználótól: „Biztosan törölni/módosítani akarja?”. A tényleges törlést csak az „Igen” válasz esetén végezzük el.

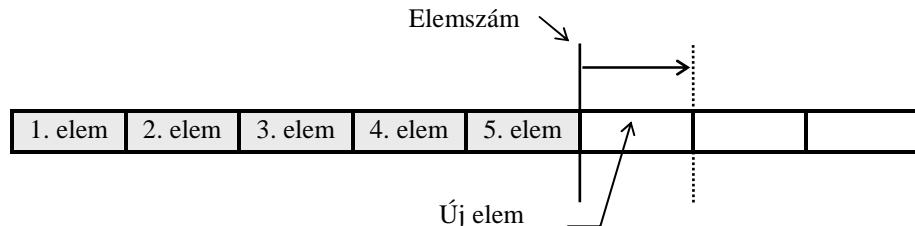
A karbantartási műveleteket most az indexelhető tömb konténerben fogjuk szemléltetni. Az elmélet kiterjeszthető egyéb konténerekre is.

## Rendezetlen konténer, van kulcs

**Például:** Autó (rendszer, típus, szín) objektumok karbantartása. A rendszám egyedi adat. A keresési feltétel egy adott rendszám egyezősége. Az autók nincsenek rendezve.

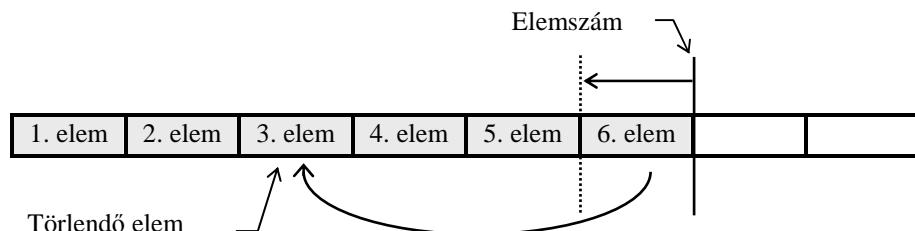
### Beszúrás

Keresünk a rendezetlen tömbben. Ha létezik már az adott elem, akkor az elemet nem visszük fel (hiszen az elemek egyediek), és erről üzenetet adunk a felhasználónak. Ha nincs, akkor az új elemet az utolsó elem utáni helyre tesszük (nem kell, hogy az elemek rendezettek legyenek, így a legkényelmesebb az elemet erre a helyre tenni). A nyilvántartásban szereplő elemek száma minden eggyel növeljük:



### Törlés

Keresünk a rendezetlen tömbben. Ha nem található az adott elem, akkor természetesen nem tudunk törölni, és a felhasználót informáljuk erről. Ha megvan az elem, akkor azt úgy töröljük, hogy az utolsó elemet rámásoljuk. Az elemek számát egygyel csökkentjük:



### Módosítás

Ha nem található az adott elem, akkor erről üzenetet adunk a felhasználónak. Ha van ilyen elem, akkor annak a kulcson kívül bármely adatát módosíthatjuk.

## Rendezetlen konténer, nincs kulcs

**Például:** Hordó (átmérő, magasság, faanyag) objektumok karbantartása. Nincs egyedi adat. A keresési feltétel az átmérő és a magasság egyezősége. A hordók nincsenek rendezve semmilyen szempont szerint. Új hordót egyszerűen a hordósor végére teszünk; törlés esetén – ha az adott

átmérőjű és magasságú hordót megtaláljuk – kitöröljük, mindegy, hogy melyiket (a tölgyet vagy a fenyőt).

A nyilvántartott elemek számát állandóan jegyezzük. Az új elemet mindenig a tömb végére teszik. Ellenőrzést nem kell végeznünk, hiszen lehetnek ismétlődő, egyforma elemek is. Törlés és módosítás esetén kezelník kell az egyforma elemeket is!

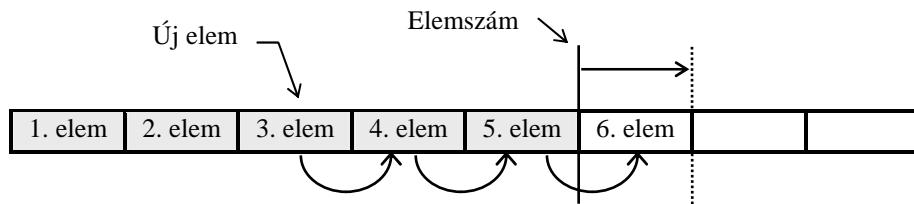
### Rendezett konténer, van kulcs

**Például:** Autó (rendszer, típus, szín) objektumok karbantartása. A rendszám egyedi adat. A keresési feltétel egy adott rendszám egyezősége. Az autók rendszám szerint rendezve vannak.

A tömböt állandóan „rendben tartjuk”, vagyis minden egyes karbantartási művelet elvégzésekor ügyelünk arra, hogy a rendezettség megmaradjon.

### Beszúrás

Ha létezik az adott elem a tömbben, akkor a beszúrást nem végezzük el, és erről értesítjük a felhasználót. Ha még nincs ilyen elem, akkor megkeressük azt a helyet a tömbben, ahová az új elemet a rendezettség szerint be kellene szúrnunk. Az elemeket onnan kezdve eggyel feljebb toljuk, és az így keletkezett üres helyre beírjuk (beszúrjuk) az új elemet. A nyilvántartásban szereplő elemek számát eggyel növeljük:



### Törlés

Ha nem található az adott elem a tömbben, akkor erről értesítjük a felhasználót. Ha igen, akkor azt úgy töröljük, hogy ettől az elemtől kezdve az elemeket a tömbben eggyel lejjebb másoljuk – a rendezettség tehát megmarad. A nyilvántartásban szereplő elemek számát eggyel csökkentjük.

### Módosítás

Kulcsot nem szabad módosítani, a többi adat módosítható.

## Rendezett konténer, nincs kulcs

**Például:** Hordó (átmérő, magasság, faanyag) objektumok karbantartása. Nincs egyedi adat. A keresési feltétel a faanyag egyezősége. A hordók rendezve vannak faanyaguk szerint.

A beszúrás hasonló az előző esethez (a rendezettségen kívül következő elemeket feltoljuk) azzal a különbséggel, hogy a beszúrást mindenkorban elvégezzük, ha van a tömbben hely.

Törlés és módosítás esetén vigyázni kell, mert több egyforma elem lehetséges: el kell tehát törleni, melyik elemre (esetleg az összesre) akarjuk a műveletet elvégezni.

## 19.4. Primitív elemek rendezése, keresése

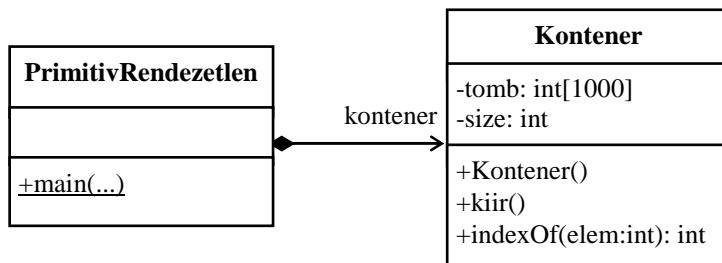
### Keresés rendezetlen tömbben

#### Feladat – Primitív rendezetlen

Kérjünk be a felhasználótól számokat 0 végig, majd írjuk ki a számokat a bevitel sorrendjében! Ezután tegyük lehetővé, hogy a felhasználó kereshessen számokat a bevitt számsorozatban: ha az általa megadott szám benne van a sorozatban, írjuk ki első előfordulásának indexét (nullától számozva), egyébként írjuk ki, hogy "Nincs ilyen szám"! Ha a megadott szám 0, legyen vége a programnak!

#### A program terve

Készítünk egy Kontener osztályt, amely a számokat egy tömbben tárolja. A tömb méretét 1000-re vesszük, feltételezzük, hogy ennél több számot nem fog beütni a felhasználó. A program terve a 19.4. ábrán látható.



19.4. ábra. A PrimitivRendezetlen program terve

A Konténer osztály metódusai:

- ▶ `Kontener()`  
Konstruktor. Bekéri a számokat a felhasználótól. A bekért számok száma `size`.
- ▶ `void kiir()`  
Kiírja a tömb elemeit a konzolra.
- ▶ `int indexOf(int elem)`  
Visszaadja a paraméterként megadott `elem` első előfordulásának tömbbeli indexét, illetve -1-et, ha nincs ilyen elem.

### Forráskód és a program elemzése

```
import extra.*;
class Kontener {
 private int[] tomb = new int[1000];
 private int size=0;
```

A konstruktorban bekérjük a számokat. A ciklus addig halad, amíg van hely a tömbben, és nem végjelet ütnek. A számot betesszük az eddigi utolsó elem utáni helyre, és az eltárolt számok darabszámát (`size`) eggyel megnöveljük:

```
public Kontener() {
 int elem;
 while ((size<tomb.length) &&
 (elem = Console.readInt("Szám: "))!=0) {
 tomb[size++] = elem;
 }
}
```

A metódus rendezetlenül, a bevitel sorrendjében írja ki a konzolra a számokat.

```
public void kiir() {
 for (int i=0; i<size; i++)
 System.out.print(tomb[i]+ " ");
 System.out.println();
}
```

A rendezetlen tömbben az `indexOf()` metódus keres. A ciklussal végighaladunk a tömbön; ha közben megtaláljuk a keresett elemet, akkor elhagyjuk a függvényt úgy, hogy annak visszatérési értéke a keresett index lesz. A ciklus teljes lefutása (vagyis hogy nem ugrunk ki a ciklusból) azt jelenti, hogy nincs meg az `elem`: ekkor a visszatérési érték -1 lesz. Az elemek összehasonlítását az == (egyenlő-e) operátorral végezzük el, mivel primitív típusokról van szó:

```
public int indexOf(int elem) {
 for (int i=0; i<size; i++) {
 if (tomb[i] == elem)
 return i;
 }
```

```

 return -1;
 }
} // Kontener

public class PrimitivRendezetlen {
 public static void main(String[] args) {
 Kontener kontener = new Kontener();
 kontener.kiir();
 }
}

int elem;
while ((elem = Console.readInt("Keresendő szám: "))!=0) {
 int index = kontener.indexOf(elem);
 if (index >= 0)
 System.out.println("Indexe: "+index);
 else
 System.out.println("Nincs ilyen szám");
}
} // main
} // PrimitivRendezetlen

```

### A program egy lehetséges futása

```

Szám: 4
Szám: 13
Szám: 2
Szám: 7
Szám: 0
4 13 2 7
Keresendő szám: 13
Indexe: 1
Keresendő szám: 5
Nincs ilyen szám
Keresendő szám: 0

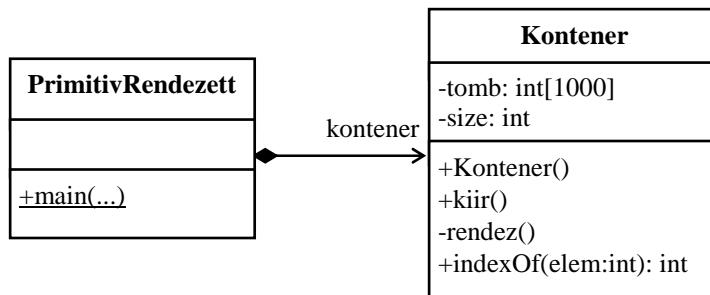
```

### Tömb rendezése, keresés a rendezett tömbben

#### Feladat – Primitív rendezett

Kérjünk be a felhasználótól számokat 0 végig, majd írjuk ki a számokat értékük szerint növekvő sorrendben! Ezután tegyük lehetővé, hogy a felhasználó kereshessen számokat a bevitt számsorozatban: ha az általa megadott szám benne van a sorozatban, írjuk ki első előfordulásának indexét (nullától számozva), egyébként írjuk ki, hogy "Nincs ilyen szám"! Ha a megadott szám 0, legyen vége a programnak!

Az előbbi Kontener osztályt most ki kell egészítenünk egy `rendez()` metódussal, amely a számok bevitelle után, a konstruktorban rendezi a tömböt. A rendező metódus privát lesz, a tömböt kívülről nem lehet rendezni. Az `indexOf()` metódus implementációja megváltozik, hiszen rendezett tömb esetén nem kell a tömb végéig keresni. A program terve a 19.5. ábrán látható.



19.5. ábra. A PrimitivRendezett program terve

*Megjegyzés:* A feladat megoldható a `rendez()` metódus nélkül is, ha a konstruktort úgy módosítjuk, hogy a bevitt számokat azonnal a helyükre szűrjuk be.

### Forráskód

```

import extra.*;

class Kontener {
 private int[] tomb = new int[1000];
 private int size=0;

 public Kontener() {
 int elem;
 while ((size<tomb.length) &&
 (elem = Console.readInt("Szám: "))!=0) {
 tomb[size++] = elem;
 }
 rendez();
 }

 public void kiir() {
 for (int i=0; i<size; i++)
 System.out.print(tomb[i]+" ");
 System.out.println();
 }
}

```

A `rendez()` metódus minimumkválasztásos algoritmussal rendezi a számokat. A külső ciklusban az `i` index értéke a tömb első (0.) elemétől az utolsó előtti (`size-2.`) eleméig megy. Ezekre a helyekre választjuk ki az adott menetben a legkisebb elemet. A belső ciklusban `minIndex` a tömb `i`-től kezdődő része (az `i, size-1` tartomány) legkisebb elemének indexe lesz. Ha ez különbözik az `i`-től (`//1`), vagyis nem az első helyen álló elem a legkisebb, akkor a két elemet ki kell cserélnünk. A rendezést a konstruktor végzi, rögtön az elemek bevitelé után:

```

private void rendez() {
 for (int i=0; i<=size-2; i++) {
 int minIndex = i;
 for (int j=i+1; j<=size-1; j++)
 if (tomb[j] < tomb[minIndex]) // elemek hasonlítása
 minIndex = j;
 if (i != minIndex) //1
 int seged = tomb[i]; // i. és minIndex. elem cseréje
 tomb[i] = tomb[minIndex];
 tomb[minIndex] = seged;
 }
}
}

```

Az `indexOf()` most a rendezett tömbben való keresést valósítja meg. Figyelje meg, hogy most nemcsak akkor hagyjuk el a metódust, ha megtaláltuk az elemet (`tomb[i]==elem`), hanem akkor is, ha a tömb aktuális eleme lehagyta a keresett elemet (`tomb[i]>elem`):

```

public int indexOf(int elem) {
 for (int i=0; i<size; i++) {
 if (tomb[i] == elem)
 return i;
 if (tomb[i] > elem)
 return -1;
 }
 return -1;
}

public class PrimitivRendezett {
 public static void main(String[] args) {
 Kontener kontener = new Kontener();
 kontener.kiir();
 }
}

```

A keresés semmiben sem különbözik a rendezetlen tömbben végzett kereséstől:

```

int elem;
while ((elem = Console.readInt("Keresendő szám: "))!=0) {
 int index = kontener.indexOf(elem);
 if (index >= 0)
 System.out.println("Indexe: "+index);
 else
 System.out.println("Nincs ilyen szám");
}
} // main
} // PrimitivRendezett

```

### A program futása

Az előző, `PrimitivRendezetlen.java` programhoz képest a program futása minden összeannyiban változik, hogy a számok rendezetten kerülnek ki a konzolra, és nagy elemszám esetén gyorsabb a keresés.

## 19.5. String objektumok rendezése, keresése

Az objektumok rendezésének feltétele, hogy azok összehasonlíthatók legyenek, vagyis meg tudjuk állapítani a sorozat két objektumáról, hogy melyik a nagyobb. Kereséskor el kell tudnunk dönteni egy objektumról, hogy azt keressük-e. A hasonlításokat el tudjuk végezni az objektum tulajdonságainak megvizsgálásával (összehasonlításával) is, de a **legtöbb esetben sokkal elegánsabb magukat az objektumokat összehasonlítani.**

Két objektum összehasonlítására **nem alkalmazhatók az összehasonlító operátorok**, azokat csak metódusokkal lehet összehasonlítani, hiszen két referencia között

- az egyenlőség operátorok (`==`, `!=`) az objektumazonosságot vizsgálják (hogy a két objektum ugyanaz-e), és nem az objektumegyenlőséget. Az objektumok közötti egyenlőséget az `equals()` vagy a `compareTo()` metódussal szokás lekérdezni.
- a hasonlító operátorok (`<`, `>`, `<=`, `>=`) nincsenek értelmezve. Objektumokat a `compareTo()` metódussal szokás összehasonlítani.

Az `equals()` metódus visszatérési értéke `boolean`, és azt mondja meg, hogy két objektum egyenlő-e, vagy sem. A `compareTo()` metódus alkalmas a kisebb/nagyobb/egyenlő összehasonlításra, amikor is a három a lehetőséget a negatív, pozitív és nulla értékek képviselik.

Egy osztályban az `equals()` és `compareTo()` metódusok egymástól függetlenül megadhatók, tűl is terhelhetők, és akár az egyenlőség, akár az összehasonlítás más metódussal is definíálható. Vannak olyan algoritmusok, amelyek megkövetelik a kötött formájú `equals()` vagy a `compareTo()` metódusok megadását, ilyen algoritmusokról majd a 20. fejezetben lesz szó. Sok osztályban az `equals()` metódus pontosan akkor ad vissza `true` értéket, amikor a `compareTo()` nullát (így van ez a `String`-ben is); ez azonban nem törvényszerű. Az egyenlőségvizsgálat és hasonlítás feltételei különbözők is lehetnek!

### A String osztály `equals()` és `compareTo()` metódusai

- ▶ `boolean equals(Object anObject)`

Összehasonlítja az objektumot a paraméterben megadott másik objektummal. A visszaadott érték `true`, ha az `anObject` osztálya `String`, és a két szöveg karakterei rendre megegyeznek. Ez a metódus az `Object` osztály metódusát írja felül.

- ▶ `boolean equalsIgnoreCase(String str)`

Összehasonlítja az objektumot a paraméterként megadott másik `String` objektummal. A visszaadott érték `true`, ha a két szöveg karakterei rendre megegyeznek úgy, hogy a nagy- és kisbetűk között nem tesz különbséget.

- ▶ `int compareTo(Object o)`
- ▶ `int compareTo(String str)`
- ▶ `int compareToIgnoreCase(String str)`

Összehasonlítják az objektumot a paraméterként megadott másik objektummal. A harmadik esetben a kis- és nagybetűk közt nem tesz különbséget. A visszaadott érték 0, ha a két szöveg lexikografikusan egyenlő; negatív, ha a szöveg kisebb, mint a paraméter szövege; és pozitív, ha a szöveg nagyobb, mint a paraméter szövege.

### Feladat – String rendezett

Kérjünk be a felhasználótól szövegeket az üres szöveg végjelig, majd írjuk ki őket ábécé szerint növekvő sorrendben! Ezután tegyük lehetővé, hogy a felhasználó szövegeket kereshessen a bevitt sorozatban: ha az általa megadott szöveg benne van a sorozatban, írjuk ki első előfordulásának indexét (nullától számozva), egyébként írjuk ki, hogy "Nincs ilyen szöveg"! Ha üres szöveget adnak meg, legyen vége a programnak!

A program az előző pontban szereplő `PrimitivRendezett.java` program hasonmása. A `StringRendezett.java` forráskódban vastagon szedtük a megváltoztatott részeket. Figyelje meg, hogy rendezéskor és kereséskor a szövegek összehasonlítására **nem a <, >, és == operátorokat alkalmaztuk, mint primitív elemek esetén, hanem a `compareTo()` metódust!**

A bevitt szöveget az `equals()` metódussal vizsgáljuk meg, hogy az üres szöveg-e. Ezt megtehettük volna a `compareTo()` metódussal is, de az `equals()` használata kényelmesebb, és ugyanaz az eredménye. Az `indexOf()` metódusban a rendezett elemek között keresünk. Mivel a rendezést a `compareTo()` metódus alapján végeztük, a keresés is eszerint történik.

### Forráskód

```
import extra.*;

class Kontener {
 private String[] tomb = new String[1000];
 private int size=0;

 public Kontener() {
 String elem;
 while ((size<tomb.length) &&
 !(elem=Console.readLine("Szöveg: ")).equals("")) {
 tomb[size++] = elem;
 }
 rendez();
 }

 public void kiir() {
 for (int i=0; i<size; i++)
 System.out.print(tomb[i]+" ");
 System.out.println();
 }

 private void rendez() {
 for (int i=0; i<=size-2; i++) {
 int minIndex = i;
```

```
 for (int j=i+1; j<=size-1; j++)
 if (tomb[j].compareTo(tomb[minIndex])<0)
 minIndex = j;

 if (i != minIndex) {
 String seged = tomb[i];
 tomb[i] = tomb[minIndex];
 tomb[minIndex] = seged;
 }
 }

// Adott elem keresése:
public int indexOf(String elem) {
 for (int i=0; i<size; i++) {
 if (tomb[i].compareTo(elem) == 0)
 return i;
 if (tomb[i].compareTo(elem) > 0)
 return -1;
 }
 return -1;
}
} //Konténer

public class StringRendezett {
 public static void main(String[] args) {
 Kontener kontener = new Kontener();
 kontener.kiir();
 // Keresés:
 String elem;
 while (!(elem=Console.readLine
("Keresendő szöveg: ")).equals("")){
 int index = kontener.indexOf(elem);
 if (index >= 0)
 System.out.println("Indexe: "+index);
 else
 System.out.println("Nincs ilyen szöveg");
 }
 } // main
} // StringRendezett
```

#### A program egy lehetséges futása

```
Szöveg: Madrid
Szöveg: Stockholm
Szöveg: London
Szöveg: Budapest
Szöveg: Róma
Szöveg:
Budapest London Madrid Róma Stockholm
Keresendő szöveg: Moszkva
Nincs ilyen szöveg
Keresendő szöveg: London
Indexe: 1
```

## 19.6. Saját osztályú objektumok rendezése, keresése

Az, hogy mikor egyenlő két objektum, valamint az, hogy mikor kisebb az egyik objektum, mint a másik, nézőpont kérdése. Adott például a következő feladat:

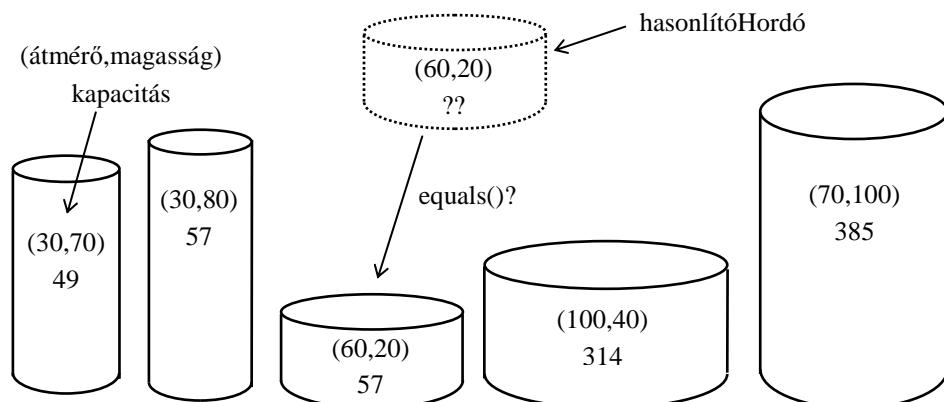
### Feladat – Hordókeresés

Egy borász különböző méretű henger alakú hordókba tölti borait. Egy hordó jellemző annak átmérője és magassága (cm-ben), ezek együttesen meghatározzák a hordó kapacitását (literben). A borász a hordók feltöltése után sorba rendezi a hordókat, hogy így könnyebb legyen majd egy adott méretű, illetve kapacitású hordót megtalálni. Írunk a borásznak egy olyan programot, amely segíti a hordók kiválasztásában!

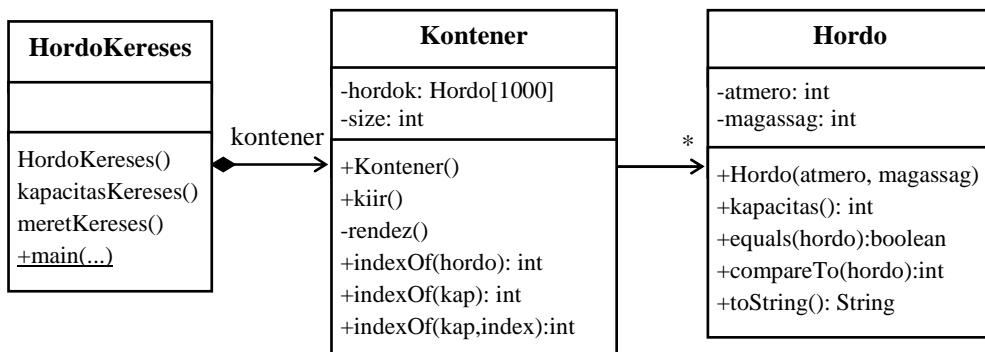
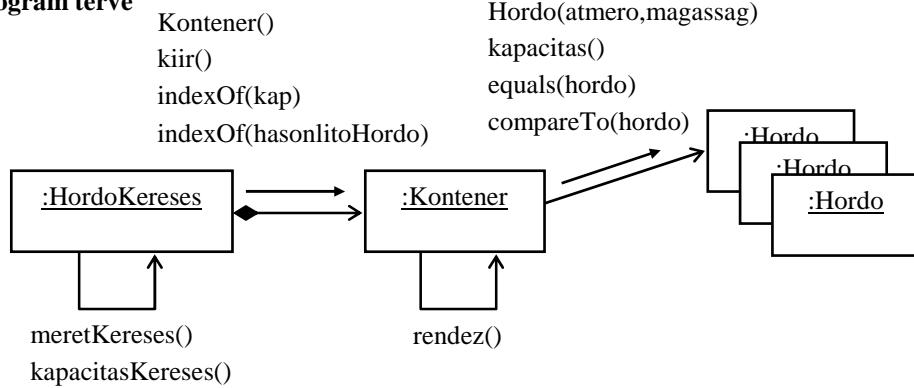
A program elején hozzuk létre a hordókat, majd keressünk a hordósorban adott méretű, illetve kapacitású hordókat!

A borász hordóit a 19.6. ábra mutatja. Hogyan rendezi sorba a borász a hordókat? Többféle szempont is lehetséges: rendezhetné őket átmérőjük, magasságuk vagy kapacitásuk szerint akár növekvő, akár csökkenő sorrendbe. Fizikailag csak egyféléképpen tudja elrendezni a hordókat; el kell tehát döntenie, mi legyen a rendezési szempont. Mivel a legtöbb vevő literben kéri a bort, és a kisebb hordóknak nagyobb a keletje, úgy dönt, hogy legyen a hordók sorrendje kapacitás szerint növekvő! Ebben az esetben egy hordó akkor kisebb, mint egy másik, ha kisebb a kapacitása. Két hordó akkor egyenlő, ha egyenlő a kapacitásuk. Ezt a feltételezett a hordó osztályának `compareTo()` metódusában fogjuk megfogalmazni, és eszerint végezzük el majd a rendezést is.

A rendezés a `compareTo()`, azaz kapacitás szerint;  
a keresés az `equals()`, azaz a pontos méret szerint történik.



19.6. ábra. A borász hordói

**A program terve**

19.7. ábra. Hordókeresés program együttműködési és osztálydiagramja

A vevő kétféleképpen fog keresni:

- ◆ **Adott méretű (átmérő és magasság) hordó keresése:** Erre a `compareTo()` metódust nem tudjuk alkalmazni, hiszen az csak a kapacitást vizsgálja; nekünk pedig minden két méretnek szigorúan meg kell egyeznie. Írjuk meg az `equals()` metódust a következőképpen: két hordó egyenlő, ha átmérője és magassága is egyenlő. A kereséshez létrehozunk egy hasonlító objektumot a kívánt adatokkal, és a sorozatban addig keresünk, amíg az `equals()` metódus `true` értéket nem ad vissza.
- ◆ **Adott kapacitású hordó keresése:** Kereséskor a `compareTo()` metódussal adódik egy nehézség: a metódus objektumokat hasonlíthat össze, ezért egy adott objektum kereséséhez össze kell állítanunk egy ún. hasonlító objektumot. A hasonlító objektumban ki kell töltenünk azokat az adatokat, amelyek részt fognak venni a hasonlításban. Egy adott kapacitású objektumot azonban nagyon nehézkes lenne meghatározni (mert az objektumnak

nincs ilyen tulajdonsága). A keresést inkább úgy végezzük el, hogy sorban minden egyes hordónak lekérdezzük a kapacitását, és összehasonlítjuk a keresett értékkel.

A fentiek figyelembevételével a program tervét a 19.7. ábra mutatja.

**Saját objektumok rendezésekor és keresésekor egyenlőségvizsgáló és hasonlító metódusokat szokás alkalmazni;** ezek **megírása a programozó feladata.** Az `equals()` metódus alkalmas az egyenlőségvizsgálatra, a `compareTo()` pedig a hasonlításra.

Az `equals()` és a `compareTo()` metódusok **objektumokat hasonlítanak össze.** E metódusok használatához össze kell állítani egy **hasonlító objektumot**, amelyben be kell állítani az egyenlőségvizsgálatban, illetve hasonlításban szereplő adatokat. A hasonlító objektumban csak egy részállapotot határozunk meg, ehhez egy külön konstruktort kell készíteni. Vigyázat! A hasonlító objektum egy „kores” objektum, abban csak a hasonlításhoz szükséges adatokat állítottuk be! **A hasonlító objektumot a keresés után ki kell dobni, vagy ha megtartjuk, akkor állapotát ki kell egészíteni!**

**Megjegyzés:** Vannak a Javában olyan kereső és rendező metódusok, amelyek megkövetelik az `equals()`, illetve a `compareTo()` metódusok megadását. A Collections osztály ilyen algoritmusairól a következő fejezetben lesz szó.

### Forráskód, a program elemzése

```
import extra.*;
class Hordo {
 private int atmero, magassag; // cm-ben

 public Hordo(int atmero, int magassag) {
 this.atmero = atmero;
 this.magassag = magassag;
 }
}
```

A hordó kapacitásának kiszámítása:  $(\text{átmérő}/2)^2 \cdot \pi \cdot \text{magasság}/1000$ . Mivel az átmérő és magasság mértékegysége cm, a térfogatot el kell osztanunk 1000-rel, hogy a kapacitást literben kapjuk meg. Az átmérőt 2.0-val osztjuk, mert az egész osztás miatt csonkulna az eredmény:

```
public int kapacitas() {
 return (int)(Math.round(Math.pow(atmero/2.0, 2) *
 Math.PI*magassag/1000));
}
```

Két hordó akkor egyenlő, ha átmérőjük és magasságuk egyaránt egyenlő:

```
public boolean equals(Hordo hordo) {
 return (hordo.atmero == atmero) &&
 (hordo.magassag == magassag);
}
```

A visszaadott érték pozitív, ha a hordó saját kapacitása nagyobb, negatív, ha a hordó saját kapacitása kisebb, mint ami a paraméterben meg van adva; és nulla, ha a két kapacitás egyenlő:

```
public int compareTo(Hordo hordo) {
 return kapacitas()-hordo.kapacitas();
}

public String toString() {
 return "\nÁtmérő:"+Format.right(atmero,6)+"
 Magasság:"+Format.right(magassag,6)+"
 Kapacitás:"+Format.right(kapacitas(),8);
}

} // Hordo

class Kontener {
 private Hordo[] tomb = new Hordo[1000];
 private int size=0;
```

A konstruktorban a borász beüti a hordók méreteit. Ha nincs több hordó, akkor az átmérőnél nullát üt. Vigyázunk, hogy ekkor már ne kérjük be a magasságát!

```
public Kontener() {
 int atmero;
 while (size<tomb.length &&
 (atmero=Console.readInt("\nÁtmérő: "))!= 0) {
 tomb[size++]= new Hordo(atmero,
 Console.readInt("Magasság: "));
 }
 rendez();
}
public void kiir() {
 for (int i=0; i<size; i++)
 System.out.print(tomb[i]+" ");
 System.out.println();
}
```

A saját készítésű osztályok objektumait ugyanúgy rendezzük, mint más objektumokat. Egyetlen kikötés, hogy az objektumoknak legyen `compareTo()` vagy egyéb hasonlító metódusuk:

```
private void rendez() {
 for (int i=0; i<=size-2; i++) {
 int minIndex = i;
 for (int j=i+1; j<=size-1; j++)
 if (tomb[j].compareTo(tomb[minIndex])<0)
 minIndex = j;

 if (i != minIndex) {
 Hordo seged = tomb[i];
 tomb[i] = tomb[minIndex];
 tomb[minIndex] = seged;
 }
 }
}
```

A saját készítésű osztályok objektumait ugyanúgy keressük, mint más objektumokat. Egyetlen kikötés, hogy az objektumoknak legyen `compareTo()`, `equals()` vagy egyéb hasonlító metódusuk. Itt egy adott méretű (átmérőjű és magasságú) hordót keresünk, a keresési feltétel szempontjából rendezetlen tömbben:

```
public int indexOf(Hordo hordo) {
 for (int i=0; i<size; i++) {
 if (hordok[i].equals(hordo))
 return i;
 }
 return -1;
}
```

Adott kapacitású hordó keresése előlről. Meghívja a másik `indexOf()` metódust:

```
public int indexOf(int kap) {
 return indexOf(kap, 0);
}
```

Adott kapacitású hordó keresése a megadott indextől. A keresés a kapacitás szerint rendezett tömbben történik:

```
public int indexOf(int kap, int index) {
 for (int i=index; i<size; i++) {
 int iKap = tomb[i].kapacitas();
 if (iKap == kap)
 return i;
 if (iKap > kap)
 return -1;
 }
 return -1;
}

public class HordoKereses {
 private Kontener kontener;
```

A konstruktörben létrehozzuk a hordó objektumokat tartalmazó konténert, aztán először adott méretű, majd adott kapacitású hordókat keresünk:

```
public HordoKereses() {
 kontener = new Kontener();
 kontener.kiir();
 meretKereses();
 kapacitasKereses();
}
```

Adott méretű hordók keresése. A bekért adatok alapján létrehozunk egy `hasonlitoHordo` nevű objektumot hasonlítás céljára. Ezt a hordót megkeressük a konténerben a `Kontener` osztályban megírt `indexOf(Hordo)` függvénytel:

```
void meretKereses() {
 int atmero;
 Hordo hasonlitoHordo;

 atmero = Console.readInt("\nÁtmerő: ");
 while (atmero!=0) {
 hasonlitoHordo=
 new Hordo(atmero,Console.readInt("Magasság: "));
 int index = kontener.indexOf(hasonlitoHordo);
 if (index >= 0)
 System.out.println("Indexe: "+index);
 else
 System.out.println("Nincs ilyen elem");
 atmero = Console.readInt("\nÁtmerő: ");
 }
}
```

Adott kapacitású hordók keresése. Most nem kell hasonlító objektum, mert az egyes hordók primitív típusú kapacitásait hasonlítjuk össze sorban a keresendő értékkel. A keresést a Kontener osztályban megírt indexOf(int) függvénytel végezzük el:

```
void kapacitasKereses() {
 int kap;

 while ((kap=Console.readInt("Keresendő kapacitás: "))!=0){
 int index = kontener.indexOf(kap);
 if (index >= 0)
 System.out.println("Indexe: "+index);
 else
 System.out.println("Nincs ilyen hordó");
 }

 public static void main(String[] args) {
 new HordoKereses();
 } // main
} //HordoKereses
```

Figyelje meg, hogy a Kontener osztályban háromféle indexOf() metódus szerepel! Az adott indextől induló keresést a feladatban nem használtuk ki. A metódus azonban kapóra jöhét, ha meg szeretnénk számolni az adott kapacitású hordók számát, vagy ki szeretnénk írni azok adatát.

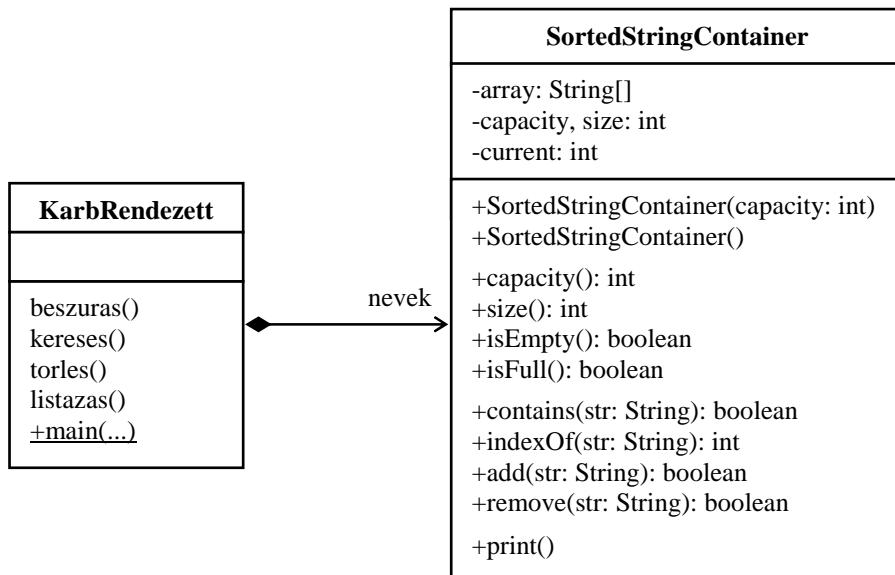
## 19.7. Szövegek rendezett karbantartása

### Feladat – Karbantartás, rendezett

Neveket szeretnénk nyilvántartani. Készítsünk olyan programot, amely lehetővé teszi a nevek rendezett karbantartását és listázását. Két egyforma név nem szerepelhet a nyilvántartásban. A következő funkciókat menüből lehet kérni:

- Név felvitele
- Név keresése
- Név törlése
- Névsor rendezetten

Készítsünk egy `SortedStringContainer` osztályt, amely felelős a karbantartási funkciók elvégzéséért. A konténerbe be lehet tenni egy nevet, ki lehet onnan egy adott nevet törlni, ki lehet listázni a benne levő összes nevet stb. A program terve a 19.8. ábrán látható.



19.8. ábra. KarbRendezett program osztálydiagramja

### Forráskód, a program elemzése

```
import extra.*;
```

A SortedStringContainer szövegeket rendezetten tároló osztály. A szövegeket az array tárolja; capacity a tároló kapacitása (nem lehet növelni); size a tároló aktuális mérete; current pedig egy globális mutató (index): azt a contains állítja, és a keresés után más metódusok felhasználják értékét:

```
class SortedStringContainer {
 private String[] array; // konténer tömb
 private int capacity; // konténer kapacitása
 private int size; // konténer aktuális mérete
 private int current; // keresés után felhasználható mutató
```

Konstruktorok. Ha nem adunk meg kapacitást, akkor az 100 lesz:

```
public SortedStringContainer(int capacity) {
 this.capacity = capacity;
 size = 0;
 array = new String[capacity];
}

public SortedStringContainer() {
 this(100);
}
```

Lekérdező metódusok:

```
public int capacity() { return capacity; }
public int size() { return size; }
public boolean isEmpty() { return size == 0; }
public boolean isFull() { return size == capacity; }
```

Megmondja, van-e már ilyen elem. Állítja az osztálysintű current indexet: értéke az a pozíció, ahol az elem van, illetve ahova be kellene szúrni:

```
public boolean contains(String str) {
 for (current=0; current<size; current++) {
 if (array[current].compareTo(str) == 0)
 return true;

 if (array[current].compareTo(str) > 0)
 return false;
 }
 return false;
}
```

Ha van a tárolóban str, akkor annak indexét, egyébként -1-et ad vissza:

```
public int indexOf(String str) {
 if (contains(str))
 return current;
 else
 return -1;
}
```

Beszúrás. A contains megkeresi azt a pozíciót, aholá az elemet be kellene szúrni. Be is szúrjuk az str-t, és a felette álló elemeket feljebb toljuk:

```
public boolean add(String str) {
 if (isFull())
 return false;
 if (contains(str))
 return false;
 else { // current-et a contains adja
 for (int i=size; i>current; i--)
 array[i] = array[i-1];
 array[current] = str;
 size++;
 return true;
 }
}
```

Törlés. A törlött elemre húzzuk a felette álló elemeket. A current. elem megszűnik úgy, hogy azt felülírjuk a current+1. elemmel, a current+1. elemet felülírjuk a current+2. elemmel stb. A tároló méretét eggyel csökkentjük:

```
public boolean remove(String str) {
 if (contains(str)) {
 size--;
 for (int i=current; i<size; i++)
 array[i] = array[i+1];
 return true;
 }
 else
 return false;
}
```

Elemek listázása:

```
public void print() {
 for (int i=0; i<size; i++)
 System.out.println(array[i]);
} // SortedStringContainer
```

**A főosztály.** Nevek rendezett karbantartása és listázása:

```
public class KarbRendezett {
 private SortedStringContainer nevek =
 new SortedStringContainer(20);
```

**Beszúrás:** Bekérünk egy nevet. Ha a konténer tele van, vagy létezik már ilyen név, sajnálkozunk, egyébként a nevet beszúrjuk:

```
void beszuras() {
 if (nevek.isFull()) {
 System.out.println("Nincs több hely");
 return;
 }
}
```

```
String nev = Console.readLine("Új név: ");
if (nevek.contains(nev))
 System.out.println("Van már ilyen név");
else
 nevek.add(nev);
}
```

**Keresés:** Bekérjük a nevet. Ha a név indexe -1, akkor kiírjuk, hogy nincs ilyen név, egyébként kiírjuk, hogy hányadik a rendezettségben (0-tól számítva):

```
void kereses() {
 String nev = Console.readLine("Keresendő név: ");
 int n = nevek.indexOf(nev);
 if (n == -1)
 System.out.println("Nincs ilyen név");
 else
 System.out.println(n+". név");
}
```

**Törlés:** Bekérjük a nevet. Ha van ilyen név (indexe>=0), akkor rákérdezünk, hogy a felhasználó biztosan törölni akarja-e, és ha igen, akkor törljük:

```
void torles() {
 String nev = Console.readLine("Törlendő név: ");
 if (nevek.indexOf(nev)>=0) {
 if (Character.toUpperCase(Console.readChar
 ("Biztosan törölni akarja? (I/N) "))=='I')
 nevek.remove(nev);
 }
 else
 System.out.println("Nincs ilyen név!");
}
```

**Listázás:** Meghívjuk a konténer print() metódusát:

```
void listazas() {
 System.out.println("Nevek:");
 nevek.print();
}
```

**Menü:** Az előbbi funkciókat menüből hívjuk:

```
void menu() {
 char menu;
 do {
 System.out.print("\nU(j) K(eres) T(töröl) L(ista) V(ége)? ");
 menu = Character.toUpperCase(Console.readChar());
 switch (menu) {
 case 'U' : { beszuras(); break; }
 case 'K' : { kereses(); break; }
 case 'T' : { torles(); break; }
 case 'L' : { listazas(); break; }
 }
 } while (menu != 'V');
```

```
public static void main(String[] args) {
 new KarbRendezett().menu();
}
```

A `SortedStringContainer` osztályt megírhatnánk sokkal általánosabbra is. A konténer alaposztálya nyugodtan lehetne `Object`, ekkor az `Object` bármilyen leszármazottját is belethetnék. Figyelje meg, hogy a `contains()` metódust kivéve teljesen mindegy, hogy `String` vagy `Object` típusú elemmel dolgozunk. A `contains()` metódus viszont kihasználja, hogy a konténerbe beletett objektumban van `compareTo()` metódus, ez `Object`-ben hiányzik. A teljesen általános konténert a következő fejezet tárgyalja.

*Megjegyzés:* A Java osztálykönyvtárában vannak kész konténerek. A továbbiakban természetesen ezeket a kész konténereket fogjuk felhasználni programjaink készítésekor. A következő fejezet ilyen konténerekről szól.

## Tesztkérdések

- 19.1. Jelölje meg az összes igaz állítást a következők közül!
  - a) A Javában az objektumok operátorokkal összehasonlíthatók.
  - b) Ha egy sorozatot direkt módon rendezünk, akkor a rendezés után visszanyerhető a sorozat eredeti, rendezés előtti állapota.
  - c) Ha egy sorozatot indexesen rendezünk, akkor a rendezés után visszanyerhető a sorozat eredeti, rendezés előtti állapota.
  - d) A minimumkiválasztásos rendezés esetén nem kell elemeket cserélni.
- 19.2. Adva van egy növekedés szerint rendezett tömb. Keresünk egy olyan elemet, amely benne van a tömbben. Jelölje meg az összes igaz állítást a következők közül!
  - a) A tömb végéig kell keresni.
  - b) Abba hagyhatjuk a keresést, ha találtunk az elemnél kisebb elemet.
  - c) Abba hagyhatjuk a keresést, ha találtunk az elemnél nagyobb elemet.
  - d) Ha az elemek objektumok, akkor azok osztályában kötelezően szerepelnie kell egy olyan metódusnak, amely eldönti, melyik objektum van előbb a rendezettségben.
- 19.3. Jelölje meg az összes igaz állítást a következők közül!
  - a) Karbantartás során az objektum kulcsát nem szabad módosítani.
  - b) A beszűrás rendezett, illetve rendezetlen konténerbe ugyanúgy történik.
  - c) A törlés rendezett, illetve rendezetlen konténerből ugyanúgy történik.
  - d) Ha egy rendezetlen sorozatban szekvenciálisan keresünk, akkor az elemet addig keressük, amíg meg nem találjuk, vagy a sorozat végére nem érünk.

- 19.4. Jelölje meg az összes igaz állítást a következők közül!
- Csak olyan objektumok rendezhetők, amelyek összehasonlíthatók.
  - A `compareTo()` metódust két objektum sorrendiségeinek megállapítására szokták alkalmazni.
  - Az `Object.equals()` metódus visszatérési értéke `int`.
  - Az `equals(Object)` metódussal történő kereséshez össze kell állítanunk egy hasonlító objektumot, és a metódusnak ezt az objektumot kell átadni aktuális paraméterként.
- 19.5. Mi lesz az elemek sorrendje a következő programrészlet lefutása után? Jelölje meg az egyetlen jó választ!
- ```
int[] tomb = {2,4,0,4,55,3};  
for (int i=0; i<tomb.length-1; i++)  
    for (int j=i+1; j<tomb.length; j++)  
        if (tomb[j] > tomb[i]) {  
            int seged = tomb[i];  
            tomb[i] = tomb[j];  
            tomb[j] = seged;  
        }
```
- 0 2 3 4 4 55
 - 0 2 4 4 55 3
 - 55 4 4 3 2 0
 - 55 4 4 2 0 3

Feladatok

- 19.1. Generáljon 1000 darab -100 és 100 közötti véletlen valós számot! Rendezze a számokat
- (A) növekvő sorrendbe!
 - (A) csökkenő sorrendbe!
 - (C) úgy, hogy előbb álljanak a nem negatív számok növekvő sorrendben, aztán a negatív számok csökkenő sorrendben!
- (*VeletlenSzamok.java*)
- 19.2. Kérjen be a konzolról neveket végjelig! Rendezze a neveket
- (A) növekvő sorrendbe!
 - (A) csökkenő sorrendbe!
 - (C) keresztnév (a második szó) szerint növekvő sorrendbe!
- Tegyük fel, hogy 100-nál több nevet biztosan nem visznek be. (*Nevsor.java*)
- 19.3. (B) Kérjen be a konzolról neveket. Ha olyan nevet ütneki be, amelyik már szerepelt, akkor kérje be az illető születési évét, és tegye azt a név után (pl. Olajos Olga 1986). Ha ilyen is létezik, kérjük be újra a nevet! Végül írja ki a neveket név, azon belül születési év szerint rendezetten! (*RendNevek.java*)

- 19.4. **(A)** Táplálja be a programba a világ legnagyobb tengereinek neveit és terület adatait! Listázza ki az adatokat a tenger területe szerint csökkenő rendezettségen!
(Jeges:10512, Korall:4791, Arab:3683, Földközi:2969, Weddel:2890, Karib:2754...)
(*Tengerek.java*)
- 19.5. **(B)** A 19. fejezet 6. pontban található *HordoKereses.java* programot változtassa meg úgy, hogy a rendezés
- a hordó kapacitása és azon belül az átmérő szerint növekvő legyen!
 - a hordó kapacitása szerint növekvő, azon belül az átmérő szerint csökkenő legyen!
 - a hordó kapacitása és azon belül a magasság szerint csökkenő legyen!
- (*HordoKereses.java*)
- 19.6. **(C)** Készítsen egy olyan szövegeket tároló osztályt, melyben egyáltalán nem fontos a tárolás sorrendje, fontos viszont a gyors törlés! Tipp: Új elemet az utolsó elem után vigyünk fel, törléskor az utolsó elemet ugrasszuk be a törlött elem helyére!
(*KarbRendezetlen.java*)
- 19.7. **(C)** Egészítse ki a *SortedStringContainer* osztályt egy *increaseCapacity(int)* metódussal, mely a paraméterben megadott plusz elemszámmal megnöveli a konténer kapacitását! Új elem felvitelénél a konténer automatikusan bővüljön!
(*KarbKapNovel.java*)

20. A Vector és a Collections osztály

A fejezet pontjai:

1. A konténer funkciói általában
 2. Vector osztály
 3. Az equals metódus szerepe
 4. A konténer elhagyása az UML diagramról
 5. Interfészek – Collection, List, Comparable
 6. Collections osztály
 7. Feladat – Nobel díjasok
 8. Feladat – Városok
 9. Feladat – Autóeladás
-

Konténer objektumnak nevezzük az egy–sok kapcsolatot megvalósító objektumot. A tömb is konténer, de nem osztály, nincsen viselkedése: a tömbben tárolt objektumok karbantartására és az elemek keresésére az egyes funkciókat külön meg kell írni. Egy konténer osztály az elemek tárolásán kívül a különböző karbantartási, keresési és bejárási funkciókat is megvalósítja. A Java kollekció keretrendszer (Collections Framework) egy API konténergyűjtemény, mely különböző konténer- és más, kiegészítő osztályokat, interfészeket kínál fel a programozó számára. A fejezetben a kollekció-keretrendszer két osztályát ismerjük meg részletesen: a `Vector` és a `Collections` osztályokat. A `Vector` egy olyan kollekció, amely objektumok rendezetlen tárolására képes. A `Collections` osztály statikus metódusai kollekciókon értelmezett rendező, kereső és egyéb kisegítő algoritmusokat implementálnak.

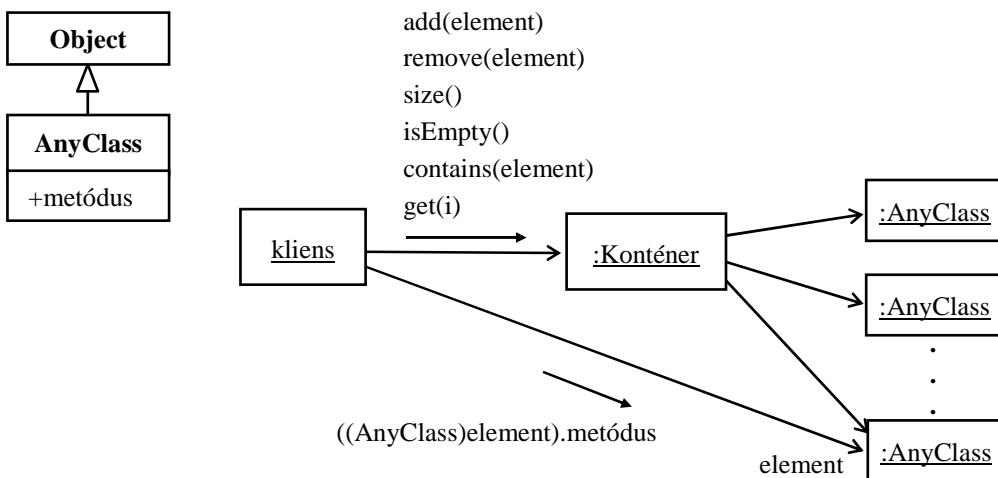
20.1. A konténer funkciói általában

A **konténer** olyan objektum, amely objektumokat tárol, és alkalmas különböző karbantartási, keresési és bejárási funkciók megvalósítására.

A 20.1. ábrán látható konténerbe az `add(element)` metódussal objektumot tehetünk be, a `remove(element)` segítségével pedig egy konkrét objektumot távolíthatunk el. A `size()` megadja a konténer aktuális méretét, az `isEmpty()` metódussal megkérdezhetjük, hogy a

konténer üres-e, a `contains(element)` pedig megmondja, hogy a konténer tartalmaz-e a paraméterben megadott objektummal egyenlő elemet. A `get(i)` visszaadja a konténer i. elemét, a berakás sorrendjében.

Megjegyzés: A konténerekre általában jellemző az a képesség is, hogy az elemeket sorban egymás után is ki tudják adni. A `first()` például kiadja a konténer első objektumát, a `next(element)` egy adott objektumhoz képest a következőt, a `previous(element)` pedig az előzőt. Ebből az első két funkciót a `Vector` osztály iterátora teljesíti. Az iterátorokról a 2. kötetben lesz szó.



20.1. ábra. A konténer funkciói általában

A Javában jó néhány konténer osztályt implementáltak. A `java.util` csomagban helyet foglaló Java kollekció keretrendszer (Collections Framework) egy általános konténereket tartalmazó osztálygyűjtemény, melyben a **konténerek kollekciók vagy leképezések**:

- ◆ **Kollekció** (collection): A kollekció konténer egyszerű tárolást valósít meg. Egyes kollekciókba kizártlag egyedi elemek tehetők be, másokban az elemek duplikálhatók. Egyes kollekciók rendezetten tárolják az elemeket, mások rendezetlenül. A kollekció keretrendszer kollekcióosztályai a `HashSet`, a `TreeSet`, a `Vector`, az `ArrayList` és a `LinkedList`.
- ◆ **Leképezés** (map): Egy leképezés konténerben a nyilvántartás kulcsobjektumok szerint történik. A kulcsokhoz információhordozó objektumok rendelhetők. Kulcsok alapján a keresés és a karbantartás sokkal hatékonyabb. A kollekció keretrendszer leképezésosztályai a `HashMap`, a `HashTable` és a `TreeMap`.

A programozó feladata azt a konténert kiválasztani, amelyik az adott feladat megoldására a legalkalmasabb. Ha nincs a feladatra alkalmas konténer, akkor két lehetőségünk nyílik: vagy egy már meglevő osztályt fejlesztünk tovább, vagy egy teljesen újat írunk. Ez utóbbi megoldásra ritkán van csak szükség.

A Java osztálykönyvtár konténerei általánosak, azokba bármilyen objektumot betethetünk. Egyetlen feltétel, hogy az objektum osztálya az `Object`-ből származzon, de ez a Javában természetes. A konténer általánosságának azonban ára van: a betett objektumok elveszítik „osztálytudatukat”, hiszen **egy konténer minden objektumot egységesen Object-ként kezel**. A konténer az objektum referenciáját a kliens számára `Object` típusúként adja át (`element: Object`). Ha a konténertől megkapott objektumnak üzenni akarunk, akkor **rá kell kényszerítenünk annak osztályát**:

```
((AnyClass)element).metódus
```

A belső zárójel az `AnyClass` típus rákényszerítése az `element`-re. A külső zárójel azért szükséges, mert a pont operátor erősebb prioritású, mint a típuskényszerítő operátor.

E fejezet a `Vector` osztályt fogja bemutatni, amely a `java.util` csomagban található Java kollekció keretrendszer része. A `Vector` osztállyal az egyszerűbb karbantartási feladatok többsége megoldható. A teljes kollekció keretrendszerről a könyv 2. kötetében lesz szó.

20.2. Vector osztály

A `Vector` osztály a 20.2. ábrán látható. Az osztályban nincs feltüntetve az összes lehetséges metódus, csak a számunkra lényeges képességeket tárgyaljuk.

A vektor egy konténer objektum; egy „változtatható méretű tömb”, amely rendelkezik karbantartási és keresési funkciókkal.

Általános jellemzők:

- **A vektor mérete az elemek hozzáadásával automatikusan bővül.** A vektor méretének fizikailag csak az index `int` típusa és a memóriaméret szab határt. A vektor ugyan egy fix méretű tömbben tárolja az elemeket (`elementData`), de ez a tömb szükség esetén automatikusan lecserélődik. Bővüléskor a rendszer létrehoz egy új, nagyobb tömböt, melybe átmásolja a régi vektor elemeit. A tömb mérete a vektor kapacitása: `capacity()`, a tömbben tárolt aktuális elemek száma pedig a vektor mérete: `size()`. Létrehozáskor megadható a vektor kezdeti kapacitása: `initialCapacity`. Ha a kapacitást nem adjuk meg, akkor annak értéke 10. Megadható továbbá a növekedés mértéke: `capacityIncrement`, ennyivel fog a tömb kapacitása automatikusan megnöni, amikor a tömb betelik. Ha nem adjuk meg ezt az értéket, akkor a vektor a kétszeresére bővül. A kapacitás és a bővülés mértékének megadásakor figyelembe kell venni, hogy a gyakori

- bővítés feltétlenül a hatékonyság rovására megy, a túl nagy kapacitás megadása viszont indokolatlan memóriafoglalással járhat.
- **A vektor elemei indexelhetők.** Az index egy 0 és `size()-1` közötti érték. Az `i`. elemet a `get(i)` függvény adja meg. Hibás index megadásakor `ArrayIndexOutOfBoundsException` kivétel keletkezik.

| «Konténer» | |
|---|--|
| Vector | |
| #elementData: Object[] | |
| #elementCount: int | |
| #capacityIncrement: int | |
| +Vector() | |
| +Vector(initialCapacity: int) | |
| +Vector(initialCapacity: int, capacityIncrement: int) | |
| +Vector(c: Collection) | |
| +size(): int | |
| +capacity(): int | |
| +isEmpty(): boolean | |
| +get(index:int): Object | |
| +contains(element : Object) : boolean | |
| +containsAll(c: Collection) : boolean | |
| +equals(obj: Object): boolean | |
| +toString(): String | |
| +setSize(newSize: int) | |
| +set(index: int, element: Object): Object | |
| +add(element: Object): boolean | |
| +add(index: int, element: Object) : boolean | |
| +addAll(c: Collection): boolean | |
| +addAll(index: int, c: Collection): boolean | |
| +remove(element: Object): boolean | |
| +remove(index: int) : Object | |
| +removeAll(c: Collection): boolean | |
| +retainAll(c: Collection): boolean | |
| +clear() | |
| +indexOf(obj: Object): int | |
| +indexOf(element: Object, index: int): int | |
| +lastIndexOf(obj: Object): int | |
| +lastIndexOf(obj: Object, index: int): int | |

20.2. ábra. Vector osztály

- **A vektor elemei rendezetlenek.** A keresések és a törlések **mindig a megadott hasonlító objektummal egyenlő objektumra vonatkoznak.** Két objektum (egyik és másik) egyenlő, ha az egyik.equals(másik) értéke true.

Karbantartó funkciók:

- Beszúrás (add): Új elemet vagy egy egész kollekciót beszúrhatunk akár a vektor végére, akár egy adott objektum előtt.
- Törlés (remove): Akár egy adott objektum, akár egy adott indexű objektum törölhető. Egy másik kollekció összes eleme is törölhető a vektorból.
- Módosítás (beállítás, set): Egy adott objektum felülírható (kicserélhető) egy másik objektummal, illetve egy adott objektum adatai megváltoztathatók.

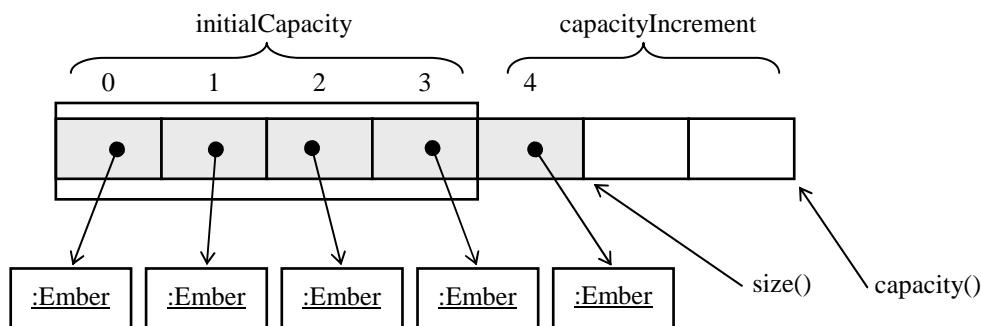
Kereső funkciók:

- Keresés (indexOf): Rákereshetünk egy adott objektumra a vektor elejétől előrefelé vagy a vektor végétől visszafelé, az első (utolsó) vagy a megadott indextől kezdődően.

Példaként létrehozunk egy `emberek` vektort, melynek kezdeti kapacitása (`initialCapacity`) 4, a bővítés egysége (`capacityIncrement`) pedig 3. A vektorba 5 darab, `Ember` osztályú objektumot teszünk (20.3. ábra):

```
Vector emberek = new Vector(4,3);
for (int i=0; i<5; i++)
    emberek.add(new Ember(...));
```

Négy elem betétele után `emberek.capacity()` és `emberek.size()` értéke egyaránt 4. Az 5. objektumnak (melynek indexe 4) a vektorhoz való hozzáadásakor (és minden olyan esetben, amikor betelt az aktuális tömb) a vektor kapacitása 3-mal bővül. Ekkor `emberek.capacity() == 7` és `emberek.size() == 5`.



20.3. ábra. A vektor bővítése

Megjegyzés: A bővülés szükség esetén automatikusan megtörténik (egy alapértelmezett értékkel), függetlenül attól, hogy megadtuk-e a kezdeti kapacitást és a bővülés mértékét. Ezeket a paramétereket kizárolag a hatékonyság érdekében szokás megadni.

Vector osztály – osztályleírás

Konstruktörök

- `Vector(int initialCapacity, int capacityIncrement)` //1
 - `Vector(int initialCapacity)` //2
 - `Vector()` //3
 - `Vector(Collection c)` //4
- //1-ben megadható a kezdeti kapacitás és a bővülés mértéke. //2-ben a bővülés mértéke kétszeres. //3-ban a kezdeti kapacitás 10, a bővülés mértéke kétszeres. Mindhárom esetben a vektor kezdeti mérete 0 (`size() == 0`). //4-ben a létrejövő vektor egy már meglévő vektor, illetve kollekció elemeinek átmásolásával indul. A vektor mérete c méretével lesz egyenlő, a kapacitás c-nek 110 %-a, a növekedés mértéke kétszeres.

Lekérdezések

- `int size()`
Visszaadja a vektor méretét, vagyis az éppen benne lévő objektumok számát.
- `int capacity()`
Visszaadja a vektor aktuális kapacitását.
- `boolean isEmpty()`
Visszaadja, hogy a vektor üres-e. `isEmpty` értéke `true`, ha `size() == 0`.
- `Object get(int index)`
Visszaadja az `index` indexű objektumot.
- `boolean contains(Object element)`
Megmondja, van-e a vektorban az `element`-tel egyenlő elem. A vektor elemeit sorban megvizsgálja, hogy van-e olyan elem, melyre az `equals(element)` metódus `true`-t ad vissza. Ha legalább egy ilyen van, akkor `contains` értéke `true`.
- `boolean containsAll(Collection c)`
Megmondja, hogy a vektor tartalmazza-e a paraméterben megadott vektor (kollekció) összes elemét. Akkor `true`, ha a `c` kollekció minden egyes `element` elemére igaz, hogy a `contains(element)` `true` értéket ad vissza.
- `boolean equals(Object obj)`
Visszaadja, hogy a vektor egyenlő-e a megadott `obj` objektummal (egy másik kollekcióval). Két vektor akkor egyenlő, ha méreteik megegyeznek, és az elemek páronként is

egyenlők (az `equals` metódus alapján, vagy minden elem `null`). Ez a metódus nem tévesztendő össze az elemeken dolgozó `equals` metódussal!

► `String toString()`

Visszaadja a vektor szöveges reprezentációját: a vektor elemeit vesszővel elválasztva, egy szögletes zárójelben.

Módosítások, bővítek

► `void setSize(int newSize)`

Beállítja a vektor méretét direkt módon: az új méret `newSize` lesz. Ha ez az új méret nagyobb, mint az eredeti, akkor szükség esetén a kapacitás megnövekszik. Akár nagyobb az új méret, akár kisebb, mint az eredeti, a `size()`-adik elemtől kezdve az összes elem mutatója `null` értékű lesz.

► `Object set(int index, Object element)`

Az `index` indexű objektumot átírja `element`-re. Visszaadja az ezen a helyen eddig szereplő objektumot.

► `boolean add(Object o)`

Az `o` által azonosított objektumot hozzáadja a vektorhoz, utolsó elemként. A vektor mérete eggyel növekszik, és ha a vektor telített volt, akkor a kapacitás is növekszik `capacityIncrement` értékkel. A visszaadott érték `true`, ha a hozzáadás sikeres (például volt elég memória).

► `boolean add(int index, Object element)`

Az `element` által azonosított objektumot beszúrja a vektorba az `index`-edik pozícióba. A összes többi objektum mutatója eggyel feljebb tolódik (indexeik eggyel megnőnek). A visszaadott érték `true`, ha a hozzáadás sikeres.

► `boolean addAll(Collection c)`

Hozzáadja a vektorhoz (az utolsó elem után) a paraméterben megadott `c` vektor (kollekció) összes elemét. A visszaadott érték `true`, ha a vektor megváltozott.

► `boolean addAll(int index, Collection c)`

Beszúrja a vektor `index`-edik eleme után a paraméterben megadott `c` vektor (kollekció) összes elemét. A visszaadott érték `true`, ha a vektor megváltozott.

Törlések

► `boolean remove(Object obj)`

Kitörli az első `obj` objektummal egyenlő objektumot a vektorból. A törölt objektum utáni objektumok lejjebb tolódnak (indexeik eggyel csökkennek). A vektor mérete eggyel csökken. A visszaadott érték `true`, ha a törlés sikeres. Az objektum csak akkor semmisül meg ténylegesen, ha más nem hivatkozik rá. (Az egyik vektorból kitörött objektum például egy másik vektornak még eleme maradhat.)

- ▶ `Object remove(int index)`
Kiveszi a megadott indexű elemet a vektorból. A visszaadott érték a törölt objektum referenciája.
- ▶ `boolean removeAll(Collection c)`
Kiveszi a vektorból a paraméterben megadott c vektor (kollekció) összes elemét. A c minden egyes element elemére végrehajtja a `remove(element)` metódust. A visszaadott érték `true`, ha a vektor megváltozott, azaz legalább egy elem törlésre került.
- ▶ `boolean retainAll(Collection c)`
Közös rész, illetve metszet képzése. Kiveszi a vektorból azokat az elemeket, melyek a paraméterben megadott c vektorban (kollekcióban) nincsenek benne. Másképp: a vektorban csak azokat az elemeket hagyja meg, melyek c-ben is benne vannak. A visszaadott érték `true`, ha a vektor megváltozott.
- ▶ `void clear()`
Törli a vektorból az összes objektumot. A vektor méretét nullára, a benne levő elemek mutatóit null-ra állítja. Az objektumokat majd a szemétgyűjtő megsemmisíti, ha nincs rájuk egyéb hivatkozás. A kapacitás nem változik.

Keresések

- ▶ `int indexOf(Object element)`
- ▶ `int indexOf(Object element, int index)`
Megkeresi a listában az első element objektummal egyenlő elemet, és visszaadja annak indexét. A keresés az első esetben a vektor elejétől, a második esetben az adott indextől megy végbe (beleértve az index-edik elemet is), és addig halad, míg nem a `get(i).equals(element)` a vektor valamely elemére `true` értéket ad vissza. Ha nincs ilyen elem, a visszaadott érték -1.
- ▶ `int lastIndexOf(Object element)`
- ▶ `int lastIndexOf(Object element, int index)`
Megkeresi a listában az utolsó element objektummal egyenlő elemet, és visszaadja annak indexét. A keresés visszafelé történik, az első esetben a vektor végétől, második esetben az adott indextől, és addig megy, míg nem a `get(i).equals(element)` a vektor valamely elemére `true` értéket ad vissza. Ha nincs ilyen elem, a visszaadott érték -1.

A következő mintaprogram a vektor használatát mutatja be. A vektor elemei tetszőleges objektumok lehetnek. Mi most szövegeket fogunk a vektorban tárolni.

Feladat – Vector minta

Próbáljuk ki egy `String` objektumokat tároló vektor működését! Küldjünk a vektornak különböző üzeneteket!

Az egyes műveletek könnyebb követése érdekében minden egyes művelet, illetve műveletcsoport elvégzése után kiírunk egy információs szöveget az éppen végrehajtott műveletről, és a lista() metódussal kilistázzuk a vektort. Az aktuálisan kilistázott vektor elemei, illetve a művelet eredménye megjegyzésként a megfelelő művelet mellett látható.

Forráskód

```

import extra.*;
import java.util.*;

public class VectorMinta {
    // Vektor listázása:
    static void lista(String info,Vector v) {
        System.out.println("\n"+info);
        System.out.println("méret: "+v.size());
        for (int i=0; i<v.size(); i++)
            System.out.println(i+". "+v.get(i));
    }

    public static void main(String[] args) {
        Vector v = new Vector();

        v.add(new String("Marci"));
        v.add(new String("Dani"));
        v.add(new String("Peti"));
        v.add(new String("Rudi"));
        lista("4 elemű vektor",v);    // Marci Dani Peti Rudi

        if (v.contains("Peti")) {      // true
            System.out.print("Peti benne van, ");
            System.out.println("indexe:"+v.indexOf("Peti"));
        }

        v.remove(2);
        lista("2. törölve",v);        // Marci Dani Rudi

        v.remove("Rudi"); // A két Rudi egyenlő (equals==true)
        lista("Rudi törölve",v);      // Marci Dani

        v.add(0,new String("Cili"));
        v.add(1,new String("Marci"));
        lista("Beszúrás előre",v);    // Cili Marci Marci Dani

        int n=v.size();
        System.out.println("\nUtolsó két objektum kiírása");
        System.out.println(v.get(n-2)+" "+v.get(n-1)); // Marci Dani

        v.set(1,new String("Tóni")); // Az eddigi 1. obj. elvész
        lista("1. elem átírása Tóni-ra",v); // Cili Tóni Marci Dani
        Vector vv = new Vector(v);
        lista("vv vektor:",vv);        // Cili Tóni Marci Dani
    }
}

```

```

vv.clear();
lista("A vv vektor üres lett",vv); // (üres)

vv.add(new String("Marci"));
vv.add(new String("Lili"));
vv.addAll(v);
lista("Marci, Lili és a teljes v hozzáadása vv-hez",vv);
// Marci Lili Cili Tóni Marci Dani
lista("v vektor:",v); // Cili Tóni Marci Dani

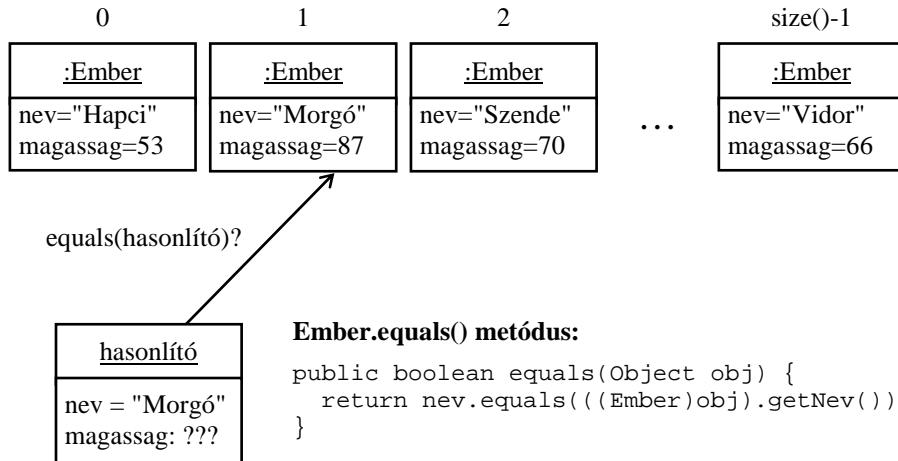
if (vv.containsAll(v)) // true
    System.out.println("\nnvv tartalmazza v elemeit");
else
    System.out.println("\nnvv nem tartalmazza v elemeit");

vv,retainAll(v);
lista("vv és v közös része:",vv);
// Marci Cili Tóni Marci Dani
}
} // VectorMinta

```

Az objektumok létrehozásakor szándékosan alkalmazzuk a new operátort: v.add(new String("Marci")); Helyes lett volna így is: v.add("Marci"); de csak az előbbi esetben tudjuk biztosítani a vektor elemeinek tényleges különbözőségét. Ennek a program működése szempontjából nincs most jelentősége, de a forráskód így jobban általánosítható. A hasonlító objektumoknak nem kell különbözniük egymástól.

20.3. Az equals metódus szerepe



20.4. ábra. Keresés a vektorban

A `Vector` osztály bizonyos metódusainak adott tulajdonságú objektumot adunk meg paraméterként. Ilyen például a `contains(Object)`, a `remove(Object)` és az `indexOf(Object)` metódus. A paraméterben megadott objektum a legtöbb esetben nem eleme a konténernek, inkább az a gyakori eset, hogy egy adott tulajdonságú objektumot keresünk. Sok esetben a keresés előtt össze kell állítanunk egy objektumot a hasonlításhoz: ebben az objektumban be kell állítanunk a keresendő állapotot vagy részállapotot (például keressük azt az objektumot, amelyben a `név=="Morgó"`). A keresés a vektor elejétől (vagy egy adott indextől) kezdődik, és akkor van vége, ha a vektor végére értünk, vagy az i. objektumra igaz, hogy `get(i).equals(hasonlító)==true`. A hasonlító objektumban csak azokat az adatokat kell megadni, amelyek befolyásolják az `equals` metódus visszatérési értékét. A keresés után a hasonlító objektumot a feladattól függően kiegészíthetjük és felhasználhatjuk, vagy akár el is dobhatjuk. A 20.4. ábra a vektorban végbemenő keresést mutatja be. A keresés az `Ember.equals()` metódus alapján történik. Eszerint két ember akkor egyenlő, ha az objektumban lévő név adatok megegyeznek, függetlenül az ember többi adatától.

A `Vector` osztály egy objektumot az elemein értelmezett `equals` metódus alapján keres meg. A kereséshez össze kell állítani egy **hasonlító objektumot**, amelyben meg kell adni az `equals()` metódus által vizsgált tulajdonságokat. Fontos, hogy az `equals` metódus akkor adjon vissza `true` értéket, amikor a vizsgált objektum részállapota megegyezik a hasonlító objektum részállapotával. **Az `equals` metódus helyes működéséről a programozónak kell gondoskodnia.**

Ha egy osztálynak nincs saját `equals` metódusa, akkor az `Object` osztály `equals` metódusa marad érvényben, amely akkor és csak akkor ad `true` értéket, ha a két objektum azonos. Az `equals` metódus feje kötött:

```
public boolean equals(Object obj) { ... }
```

A metódusnak aktuális paraméterként bármilyen objektum átadható. Mivel azonban a paraméter kötelezően `Object` típusú, ezért az `obj` objektum azon metódusaira, amelyek az `Object`ben nincsenek deklarálva, csak úgy tudunk hivatkozni, ha arra rákényszerítjük valódi osztályát.

Az `equals()` metódus a Java osztályok többségében (`Integer`, `Real`, `Character`, `String` stb.) felül van írva.

A következő feladatban saját készítésű objektumokat tárolunk vektorban.

Feladat – Törpeprogram

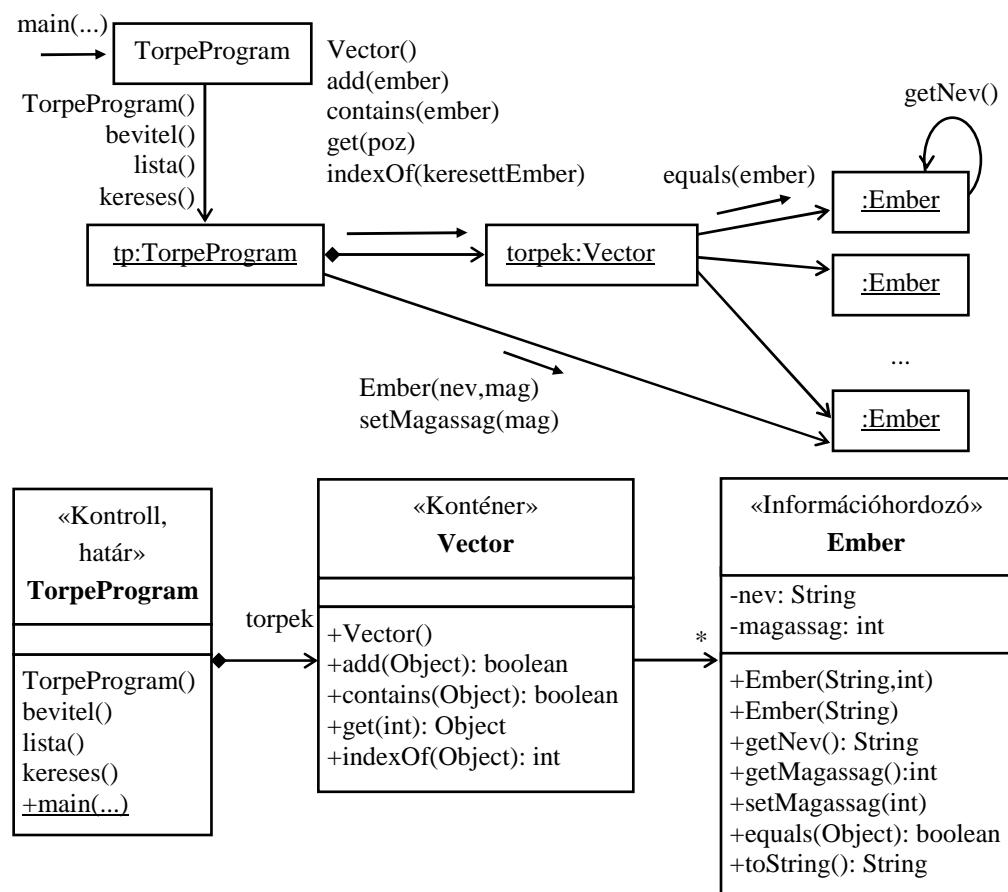
Emberek neveit és magasságait szeretnénk nyilvántartani. A nyilvántartásban két egyforma névű ember nem szerepelhet!

- Kérjük be konzolról a hét törpe nevét és magasságát (végjel: névnél Enter)!
- Listázzuk ki a törpéket a bevitel sorrendjében!
- Keressünk meg egy adott nevű törpét, és írjuk ki a magasságát!

Megjegyzés: Ne törje a fejét! Hapci, Kuka, Morgó, Szende, Szundi, Tudor, Vidor.

A program terve

A `tp:TorpeProgram` vezérlőnek van egy `torpek` nevű konténere, amelybe majd beteszi az általa létrehozott törpéket. A törpék adatait konzolról kéri be. A program fő funkcióit három részre bontjuk: bevitelre, listázásra és keresésre. A listát az érdekesség kedvéért három különböző technikai megvalósításban találjuk. A részletes magyarázatot a forráskódban találja meg.



20.5. ábra. A TorpeProgram együttműködési és osztálydiagramja

Forráskód és annak elemzése

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

class Ember {
    private String nev;
    private int magassag;
```

Konstruktorok: Az Ember osztálynak két konstruktora van. Az első (kétparaméteres) konstruktur az igazi, a második csak arra való, hogy összeállítson egy hasonlító objektumot a név szerinti kereséshez:

```
public Ember(String nev, int magassag) {
    this.nev = nev;
    this.magassag = magassag;
}

public Ember(String nev) {
    this(nev,0);
}
```

Adatokat kiolvasó és beállító metódusok:

```
public String getNev() { return nev; }
public int getMagassag() { return magassag; }
public void setMagassag(int mag) {
    if (mag>0)
        magassag = mag;
}
```

Az equals() metódus: A tárolt emberek között majd név szerint kell keresnünk, ezért meg kell írnunk a megfelelő equals() metódust. Két ember akkor egyenlő, ha nevük megegyezik:

```
public boolean equals(Object obj) {
    return nev.equals(((Ember)obj).getNev());
}

public String toString() {
    return Format.left(nev,10) + Format.right(magassag,3);
}

public class TorpeProgram {
    private Vector torpek = new Vector();
```

Bevitel: A TorpeProgram a törpék bevitelével kezdődik. A végjelig tartó olvasásban már otthonosak vagyunk. A beolvasott névről azonban el kell döntenünk, hogy van-e már ilyen névű ember a vektorban, mert a nyilvántartásban nem szerepelhet két egyforma névű ember. A hasonlító objektumban csak a nevet töltjük ki: new Ember(nev), hiszen tudjuk, hogy csak a név fog részt venni a hasonlításban. A Vector osztály contains metódusa a törpék equals

metódusai alapján keresi végig a vektort. minden egyes objektumot megszólít és megkérdezi, hogy a benne levő név megegyezik-e az obj paraméterben lévő névvel (a nev privát adat, tehát obj-ból már csak a publikus getNev() metódussal lehet azt kiolvasni). Ha van már ilyen nevű törpe, akkor erről értesítjük a felhasználót. Ha még nincs, akkor a törpe betehető a vektorba, de ehhez a magasságát is be kell kérnünk. A már félre összeállított embernek tehát megadjuk a magasságát is, és a kész törpét a torpek.add(törpe) utasítással betesszük a vektorba:

```
void bevitel() {
    Ember torpe;
    String nev = Console.readLine("\nTörpe neve: ");
    while (!nev.equals("")) {
        if (torpek.contains(törpe = new Ember(nev)))
            System.out.println("Van már ilyen törpe!");
        else {
            torpe.setMagassag(Console.readInt("magassága : "));
            torpek.add(törpe);
        }
        nev = Console.readLine("Törpe neve: ");
    }
}
```

Egyszerű lista, beépített módon: A teljes vektort egyszerűen átadjuk a println() metódusnak, amely a torpek.toString()-et kiírja a konzolra. A Vector osztály toString() metódusát úgy írták meg, hogy az meghívja a vektor minden egyes elemére a toString() metódust, és a kapott Stringeket a [] zárójelpárban vesszővel elválasztva sorolja fel. Mivel a vektor elemei itt Ember osztályúak, az Ember.toString() metódus hívódik meg, és az visszaadja az ember nevét és magasságát szöveges formában:

```
void listal() {
    System.out.println("\nBeépített lista:");
    System.out.println(torpek);
}
```

Listázás index szerint, `toString()`-gel: Az i ciklusváltozóval végigmegyünk a vektoron 0-tól size()-1-ig, és egyszerűen kiírjuk a get(i) objektumot a `toString()` implicit hívásával. Ebben az esetben is kell tehát a `toString()`, csak most megszabadtunk a [] zárójelpártól, és az elemeket külön sorba írhatjuk:

```
void lista2() {
    System.out.println("\nLista index szerint:");
    for (int i=0; i<torpek.size(); i++) {
        System.out.println(torpek.get(i));
    }
}
```

Listázás `toString()` nélkül: A listát most úgy produkáljuk, hogy nem vesszük igénybe a `toString()` metódust. Az objektumokból most üzenettel kell „kicsalni” az információt.

Mivel a `torpek.get(i)` szintaktikailag `Object` osztályú, rá kell kényszeríteni az `Ember` osztályt. Jobban járunk, ha a kiírás előtt felveszünk egy `Ember` típusú `e` objektumot, így a kiírásban már nem kell foglalkoznunk a kényszerítéssel.

```
void lista3() {
    Ember e;
    System.out.println("\nEgyéni lista:");
    for (int i=0; i<torpek.size(); i++) {
        e = (Ember)(torpek.get(i));
        System.out.println(
            "Név: "+e.getNev()+" Magasság: "+e.getMagassag());
    }
}
```

Egy törpe megkeresése: Most egy adott nevű törpét keresünk meg a konténerben. Ehhez megint összeállítunk egy hasonlító törpét nulla magassággal, és azt átadjuk az `indexOf()` metódusnak, hogy ennek alapján keresse meg a kérdéses objektumot a konténerben. Ha van ilyen objektum, akkor az `indexOf()` metódus a talált objektum `poz>=0` indexével tér vissza. A `get(poz)` tehát a keresett objektum, ettől már meg lehet kérdezni a konkrét magasságot. A magasság megkérdezéshez azonban rá kell kényszeríteni az `Ember` osztályt, hiszen a vektor az `Ember`-ként bedobott objektumot `Object`-ként adja ki. Mi azonban tudjuk, hogy ő `Ember`, és a kényszerítéssel nem okozunk bajt:

```
void kereses() {
    System.out.println("\nKeresés:");
    Ember keresettEmber = new Ember();
    Console.readLine("Törpe neve: ");
    int poz = torpek.indexOf(keresettEmber);
    if (poz >= 0)
        System.out.println("Van, magassága: "+
            ((Ember)(torpek.get(poz))).getMagassag());
    else
        System.out.println("Nincs ilyen");
}
```

Főprogram: Létrehozzuk a vezérlőt, melyet felkérünk a bevitelre, a különböző listázásokra és a keresésre:

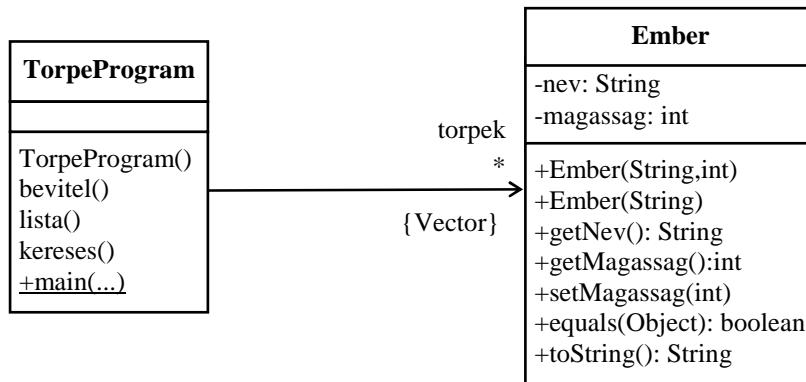
```
public static void main(String[] args) {
    TorpeProgram tp = new TorpeProgram();
    tp.bevitel();
    tp.listal();
    tp.lista2();
    tp.lista3();
    tp.kereses();
}
```

20.4. A konténer elhagyása az UML diagramról

A konténert az osztály-, illetve együttműködési diagramokon csak akkor szokás feltüntetni, ha az segít a megértésben. A konténer implementációs osztály, az egy-sok kapcsolat megvalósítását szolgálja. Ha egy osztálydiagramon az összes konténert feltüntetnénk, az zavarná az áttekinthetőséget, hiszen minden egyes egy-sok kapcsolathoz tartozik egy konténer, nem beszélve a sok-sok kapcsolatokról, amiről ebben a könyvben nem is beszélünk. A konténer kódolása rutin feladat. Egy tapasztalattabbar programozó a konténert az egy-sok kapcsolat helyén automatikusan „elképzeli”.

A 20.6. ábra a TorpeProgram leegyszerűsített osztálydiagramját mutatja. Ahogy azt a kapcsos zárójelben levő megsorításban láthatjuk, az egy-sok kapcsolatot a Vector osztály fogja megvalósítani. Ha az osztálydiagramról elhagyjuk a konténert, az még nem jelenti azt, hogy az együttműködési diagramról is el kell hagynunk. A TorpeProgramban például a karbantartó funkciók a legfontosabbak (torpek.add, torpek.contains stb.), ezért az együttműködési diagramot ebben az esetben nem ajánlatos leegyszerűsíteni.

A konténer az osztály-, illetve együttműködési diagramról elhagyható. A konténer osztályát a multiplicitás mellett megsorításként lehet megadni.



20.6. ábra. Egyszerűsített osztálydiagram – a konténer elhagyása

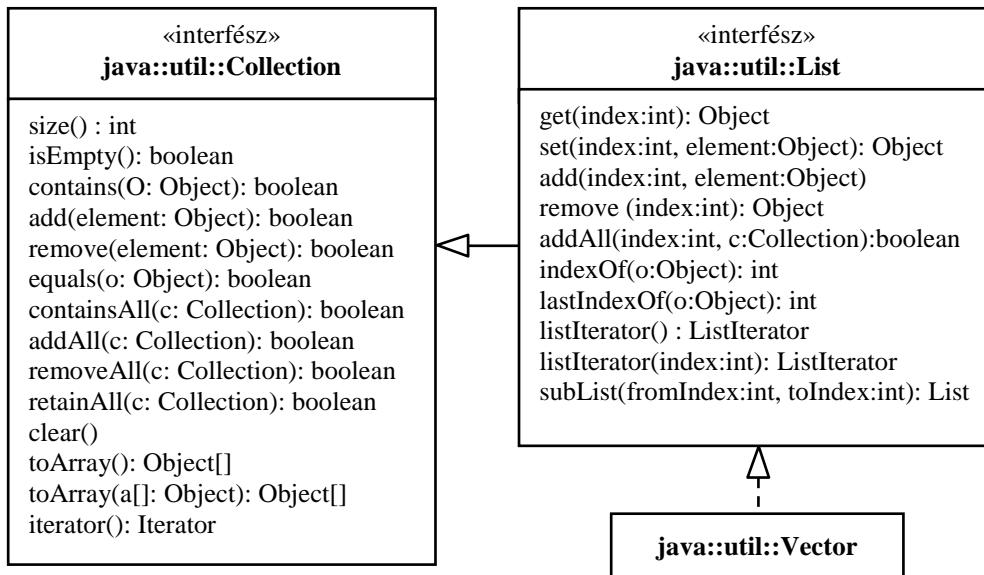
20.5. Interfészek – Collection, List, Comparable

A 7. fejezet., Öröklődés fejezetben már bevezettük az interfész fogalmát; most ismétlésként újra megadjuk a definícióját:

Az interfész (interface) **metódusfejeket definiál** abból a célból, hogy valamely osztály azt a későbbiekben implementálja, megvalósítsa. Ez egyfajta szerződés: ha valaki megtesz a szerződésben foglaltakat, vagyis egy implementáló osztályban megírja az adott interfész metódusait, akkor cserében ezt az osztályt egy adott környezetben használhatja. Szabályok:

- **Egy interfészból nem lehet példányt létrehozni**, hiszen annak nincsenek működöképes metódusai.
- **Az interfészek öröklíthetők**. Amikor egy leszármazott interfész implementálunk, akkor az interfész öröklési ágának összes metódusát implementálnunk kell az osztályban vagy annak egy ősében (még akkor is, ha a blokkot üresen hagyjuk).
- **Értékkadási kompatibilitás**: Interfész típusú változóval lehet objektumot azonosítani. Az értékkadás szerinti kompatibilitás szabályában az implementáció egyenértékű az örökléssel. Például:

```
Vector szamok = new Vector();
Collection coll = szamok;
```



20.7. ábra. A Vector osztály implementálja a List interfészt

Az interfész UML jelölése a 20.7. ábrán látható:

- ◆ Az interfészt az «interfész» sztereotípussal jelöljük.
- ◆ Ha egy osztály implementál egy interfészt, akkor az osztály egy szaggatott, nyitott hegyű nyíllal (nyílhegye olyan, mint az öröklésnél) mutat az interfészre. (Az ábrán a `Vector` osztály implementálja a `List` interfészt.)
- ◆ Az interfések közötti öröklést ugyanúgy jelöljük, mint az osztályok közötti öröklést. (Az ábrán a `List` interfész örökli a `Collection` interfészt.)

Megjegyezzük, hogy UML-ben egy osztályt úgy minősítünk annak csomagjaival, hogy az egyes csomagnevek után dupla kettőspontot teszünk.

Collection és List interfések

A `java.util` csomagban deklarált `Collection` és `List` interfések, azok a 20.7. ábrán látható üres metódusfejeket definiálják. A `List` a `Collection` interfész utódja – ez azt jelenti, hogy ha egy osztály implementálja a `List` interfészt, az egyben a `Collection` interfészt is implementálja. A `Vector` osztály implementálja a `List` interfészt: az ábrán látható `Collection` és `List` dobozában felsorolt metódusok mindenben vannak a `Vector` osztályban (a könyvben nincs az összes metódus felsorolva). A `Vector` osztály fejét így deklarálták:

```
public class Vector extends AbstractList implements List,  
    Cloneable, java.io.Serializable
```

A `Vector` tehát egy lista (`List`), klónozható (`Cloneable`) és szerializálható (`Serializable`).

Comparable interfész

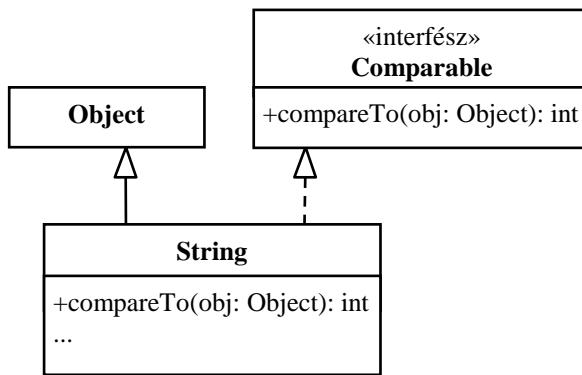
A `java.lang.Comparable` interfész mindössze egy metódusfejet definiál:

```
public int compareTo(Object obj)
```

A `Comparable` interfész megvalósító osztály objektumai összehasonlíthatók. A `compareTo()` metódus formája kötött. Az összehasonlítást végző kliens arra számít, hogy a visszaadott érték

- ◆ negatív, ha az objektum kisebb (sorrendben előrébb áll), mint a paraméterként megkapott `obj`;
- ◆ pozitív, ha az objektum nagyobb (sorrendben hátrébb áll), mint a paraméterként megkapott `obj`;
- ◆ 0, ha a két objektum egyenlő.

A 20.8. ábrán látható, hogy a `String` osztály az `Object` osztályból származik, és implementálja a `Comparable` interfész: a `String` osztályban valóban dekláráltak egy `compareTo()` metódust.



20.8. ábra. A String osztály implementálja a Comparable interfészt

Az interfész implementálása

Ahhoz, hogy egy osztály egy interfészt implementáljon, az osztály fejében meg kell adni az `implements` kulcsszó után az interfész nevét, az interfészben szereplő metódusokat pedig meg kell írni.

A `String` osztály implementálja a `Comparable` interfészt:

```

public final class String implements Comparable {

    ...
    public int compareTo(Object o) {
        return compareTo((String)o);
    }

    public int compareTo(String anotherString) {
        ...
    }
}
  
```

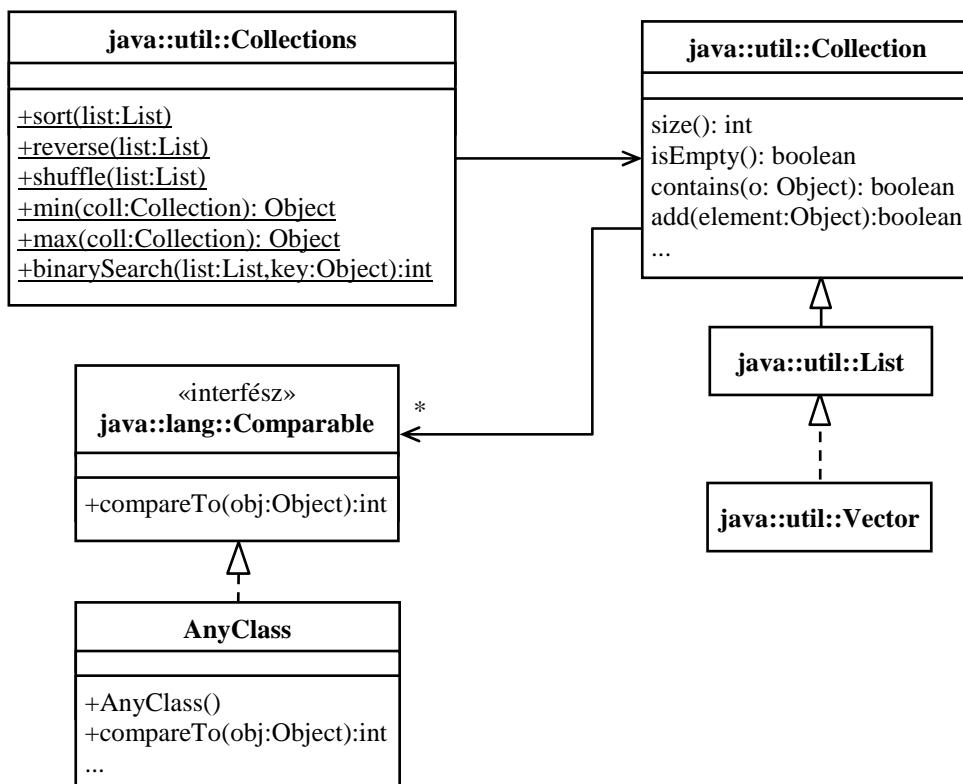
A `String` osztályban két `compareTo` metódust írtak. A `Comparable` interfész `compareTo` metódusának paramétere kötelezően `Object` osztályú. Ez a `compareTo` metódus csupán meg-hívja a `String` osztály másik, „igazi” `compareTo` metódusát, melynek paramétere `String`.

20.6. Collections osztály

A **Collections osztály** a `java.util` csomagban található, és a Java kollekció keretrendszer része. A Collections nem példányosítható, statikus metódusai általában konténereken értelmezett algoritmusok: rendezés, megfordítás, keverés, minimum- és maximumszámítás, bináris keresés stb.

Megjegyzés: A Collections osztály nem tévesztendő össze a Collection interfésszel!

A Collections osztály UML ábrája a 20.9. ábrán látható. Statikus metódusai olyan konténeken dolgoznak, amelyek implementálták a Collection interfészt vagy annak utódját, a List interfészt. Mivel a Vector osztály egy List (és így Collection is), a Collections osztály bármely metódusa paraméterként megkaphat egy vektort.



20.9. ábra. A Collections és az általa kezelt AnyClass

Az osztály algoritmusainak többsége úgy dolgozik, hogy páronként összehasonlítja a vektor elemeit. Ezek a metódusok megkövetelik, hogy a konténerbe betett objektumok összehasonlíthatók (comparable) legyenek, vagyis hogy implementálják a Comparable interfészt.

► Figyelem! Hiába írunk compareTo() metódust, ha az osztály fejéből hiányzik az implements Comparable! A Collections osztály metódusai mindenkor megkövetelik!

A Collections osztály metódusai csak akkor működnek helyesen, ha az általa kezelt vektor elemei összehasonlíthatók, vagyis implementálják a Comparable interfészt!

Összefoglalva: A vector osztály metódusai az elemek equals metódusait használják a hasonlításhoz; a Collections osztály metódusai a vektor elemeinek compareTo metódusai szerint működnek. A vektorba tett objektumoknak tehát eszerint kell megadnunk az equals, illetve compareTo metódusait. Alapértelmezett equals metódusa minden objektumnak van, legfeljebb nem az elkövetkező szerint működik; a compareTo metódus implementálásáról azonban a programozónak kell gondoskodnia!

Most nézzük sorra a Collections osztály számunkra lényegesebb metódusait!

- ▶ static void sort(List list)
Rendezi a listát (vektort) a természetes rendezettség szerint: az elemek compareTo() metódusai alapján. Az elemeknek implementálniuk kell a Comparable interfészt.
- ▶ static void reverse(List list)
Megfordítja a listát (vektort): az első elem lesz az utolsó, az utolsó pedig az első. A következő rendezettségből így csökkenő rendezettség lesz.
- ▶ static void shuffle(List list)
Összekeveri a vektor elemeit, azok rendezetlenek lesznek.
- ▶ static Object min(Collection coll)
Visszaadja a vektor, illetve kollekció legkisebb elemét. Az elemeknek implementálniuk kell a Comparable interfészt. A metódus végighalad a vektor összes elemén, és a compareTo() metódus alapján kiválasztja a legkisebb objektumot. Az egyenlők közül (ahol a compareTo visszatérési értéke 0) a legutolsót adja vissza.
- ▶ static Object max(Collection coll)
Visszaadja a vektor, illetve kollekció legnagyobb elemét. Az elemeknek implementálniuk kell a Comparable interfészt. A metódus végighalad a vektor összes elemén, és a compareTo() metódus alapján kiválasztja a legnagyobb objektumot. Az egyenlők közül (ahol a compareTo visszatérési értéke 0) a legutolsót adja vissza.

► static int binarySearch(List list, Object key)

A listában (vektorban) bináris kereséssel megkeresi a key objektumot, és visszaadja annak indexét. A vektort előzőleg rendezni kell, az algoritmus csak rendezett lista esetén működik helyesen. A keresés a compareTo() metódus alapján történik. Több egyenlő elem esetén a visszaadott objektum véletlenszerű.

Megjegyzés: A természetes rendezettségen kívül még megadhatók egyéb rendezettségek is, de ezt a lehetőséget e kötet nem tárgyalja.

Feladat – Emberek

Emberek adatait (nevét és születési évét) szeretnénk nyilvántartani. A nyilvántartásban szerepelhet két egyforma nevű ember, de akkor a születési évnek kell különböznie! Menüből lehessen választani a következő funkciókat:

1. **Felvitel:** Kérjük be konzolról egy ember nevét és születési évét! Ha van már ilyen ember, akkor írunk ki egy figyelmeztést, egyébként vigyük fel az adatait!
2. **Törlés:** Kérjük be konzolról egy ember nevét és születési évét! Ha nincs ilyen ember, akkor írunk ki egy figyelmeztést, egyébként töröljük őt a nyilvántartásból!
3. **Eredeti sorrend:** Listázzuk ki az embereket a felvitel sorrendjében!
4. **Legidősebb:** Írjuk ki a legidősebb ember adatait!
5. **Legfiatalabb:** Írjuk ki a legfiatalabb ember adatait!
6. **Növekvő sorrend:** Listázzuk ki az embereket születési év és azon belül név szerint növekvő rendezettségben!
7. **Csökkenő sorrend:** Listázzuk ki az embereket születési év és azon belül név szerint csökkenő rendezettségben!
8. **Véletlen sorrend:** Listázzuk ki az embereket véletlen rendezettségben!
9. **Vége:** Legyen vége a programnak!

Forráskód és a megoldás elemzése

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

Az osztálynak implementálnia kell a Comparable interfész, mert a Collections.sort() rendezési eljárás a compareTo() metódus alapján fog hasonlítani:

```
class Ember implements Comparable {
    private String nev;
    private int szulev;

    public Ember(String nev, int szulev) {
        this.nev = nev;
        this.szulev = szulev;
    }

    public String getNev() { return nev; }
    public int getSzulev() { return szulev; }
}
```

Az Ember osztályban kell, hogy legyen egy `equals()` metódus, hiszen a `contains()` ez alapján dolgozik. Az `equals()`akkor ad vissza igaz értéket, ha mind a név, mind a születési év megegyezik:

```
public boolean equals(Object obj) {
    return (nev.equals(((Ember)obj).getNev()) &&
            szulev == ((Ember)obj).getSzulev());
}
```

A listát majd születési év, s azon belül név szerint fogjuk rendezni – a `compareTo` metódust eszerint írjuk meg: ha a megszólított objektumban kisebb a születési év, mint a paraméterben megadott objektumban, akkor az objektum kisebb, mint a paraméter. Ha a születési év a két objektumban egyenlő, akkor a név dönt:

```
public int compareTo(Object obj) {
    if (szulev < ((Ember)obj).getSzulev())
        return -1;
    if (szulev > ((Ember)obj).getSzulev())
        return 1;
    return nev.compareTo(((Ember)obj).getNev());
}

public String toString() {
    return Format.left(nev,10) + Format.right(szulev,5);
}

public class Emberek {
    Vector emberek = new Vector();
```

A konzolról bekért adatok alapján összeállítunk egy ember objektumot, és ha ō még nem szerepel az emberek vektorban, akkor azt felvitel esetén beletesszük, törlés esetén pedig kivesszük onnan:

```
void felvitel() {
    Ember ember=new Ember(Console.readLine("\nEmber neve : "),
                           Console.readInt("születési éve: "));
    if (emberek.contains(ember))
        System.out.println("Már van ilyen");
    else
        emberek.add(ember);
}

void torles() {
    Ember ember=new Ember(Console.readLine("\nEmber neve : "),
                           Console.readInt("születési éve: "));
    if (!emberek.remove(ember))
        System.out.println("Nincs ilyen ember");
    else
        System.out.println(ember+" törölve");
}
```

A listázó metódus az objektumokat soronként írja ki a `toString()` implicit meghívásával:

```
void lista(Vector e) {
    for (int i=0; i<e.size(); i++)
        System.out.println(e.get(i));
}

void menu() {
    Vector rendEmberek; // segédvektor
    char valasz;
    do {
        System.out.print("1:felv - 2:törl - 3:eredeti - 4:legid - "+
            "5:legfiatal - 6:növek - 7:csökk - 8:véletlen - 9:vége...?");
        do
            valasz = Console.readChar();
        while (valasz<'1' || valasz>'9');

        switch (valasz) {
            case '1':           // Felvitel
                felvitel();
                break;
            case '2':           // Törlés
                torles();
                break;
            case '3':           // Eredeti lista
                lista(emberek);
                break;
        }
    }
}
```

A legidősebb és legfiatalabb embert a `Collections.min()`, illetve a `Collections.max()` metódusok meghívásával határozzuk meg (az a legidősebb ember, aki legkorábban született). A visszaadott objektumot a `toString()` implicit meghívásával írjuk ki a konzolra:

```
case '4':           // Legidősebb
    System.out.println(Collections.min(emberek));
    break;
case '5':           // Legfiatalabb
    System.out.println(Collections.max(emberek));
    break;
```

Az emberek rendezéséhez nem áldozhatjuk fel az emberek vektort, mert a bevitel sorrendjét nem felejthetjük el. Felveszünk ezért egy új, `rendEmberek` vektort az eredeti, `emberek` vektor átmásolásával. Ezzel a vektorral aztán azt teszünk, amit akarunk, hiszen az `emberekből` bármi reprodukálható. Először rendezzük a `rendEmberek` vektort, majd kilistázzuk azt:

```
case '6':           // Növekvő sorrend
    rendEmberek = new Vector(emberek);
    Collections.sort(rendEmberek);
    lista(rendEmberek);
    break;
```

Mint az előző menüpontban, itt is létrehozunk egy új vektort, hiszen azóta újabb embereket is vehettek föl vagy törölhettek ki. A vektort rendezzük a `compareTo()` szerint, majd az egész vektort megfordítjuk:

```
case '7': // Csökkenő sorrend
    rendEmberek = new Vector(emberek);
    Collections.sort(rendEmberek);
    Collections.reverse(rendEmberek);
    lista(rendEmberek);
    break;
```

Összekeverjük az `emberek` vektor elemeit (csak ez a vektor tükrözi a valós állapotot), majd kilistázzuk:

```
case '8': // Véletlen sorrend
    rendEmberek = new Vector(emberek);
    Collections.shuffle(rendEmberek);
    lista(rendEmberek);
    break;
}
} while (valasz != '9');
}

public static void main(String[] args) {
    Emberek enyv = new Emberek();
    enyv.menu();
}
}
```

Feladat – Valós számok

Kérjünk be tetszőleges sok valós számot 0 végig! Ezután írjuk ki

- a számokat a bevitel sorrendjében!
- a számokat növekvő sorrendben!
- a legkisebb és a legnagyobb számot!

Végül keressünk meg egy számot a bevittek között!

A számokat vektorba fogjuk tenni a következő okok miatt:

- ◆ a megjegyzendő számok száma elvileg korlátlan, ezért tárolására a tömb alkalmatlan;
- ◆ használni szeretnénk a Collections osztály rendező-, minimum- és maximumkereső metódusait, valamint a `Vector` osztály `indexOf` metódusát.

Mivel egy vektorba csak olyan objektumot lehetünk, melynek osztálya az `Object` leszármazottja (primitív adatokat a vektor nem tud kezelni), a számokat be kell csomagolnunk egy-egy `Double` osztályú objektumba. Ehhez felhasználjuk a `Double` osztály `valueOf` függvényét:

- `public static Double valueOf(String) throws NumberFormatException`
A karakterláncból egy `Double` típusú objektumot gyárt.

Forráskód

```
import extra.Console;
import java.util.*;

public class ValosSzamok {

    public static void main(String[] args) {
        Vector szamok = new Vector();
        final double VEGJEL = 0;
        double szam;

        // Számok bekérése:
        while (true) {
            szam = Console.readDouble("Szám: ");
            if (szam==VEGJEL)
                break;
            szamok.add(new Double(szam));
        }

        if (szamok.isEmpty()) {
            System.out.println("Nem adott meg egyetlen számot sem!");
            return;
        }

        // Számok kiírása egyenként:
        for (int i=0; i<szamok.size(); i++)
            System.out.print(szamok.get(i)+" ");
        System.out.println();

        // A vektor kiírása egyben, rendezés, újabb kiírás...
        System.out.println("A számok: "+szamok);
        Collections.sort(szamok);
        System.out.println("A számok rendezve: "+szamok);
        System.out.println("Legkisebb : "+Collections.min(szamok));
        System.out.println("Legnagyobb: "+Collections.max(szamok));

        // Egy szám keresése:
        szam = Console.readDouble("Keresendő szám: ");
        int index=szamok.indexOf(new Double(szam));

        if (index==-1)
            System.out.println("Nincs ilyen szám");
        else
            System.out.println("Van ilyen szám, indexe: "+index);
    }
}
```

20.7. Feladat – Nobel díjasok

Feladatspecifikáció

A program tartsa nyilván a világ Nobel díjasait! Egy Nobel díjhoz tartozó adatok: szakterület, évszám, név, ország. Tegyük fel, hogy ugyanabban az évben egy szakterületen (mint például béke, fizika, irodalom, kémia) legfeljebb egy Nobel díjat adnak ki. A program sorban végezze el a következő feladatokat:

- Kérje be a lehetséges szakterületeket!
- Vegye nyilvántartásba a Nobel díjasokat! Csak érvényes szakterületet engedjen megadni, és ne engedjen bevinni két egyenlő szakterületű és évszámú Nobel díjast!
- Listázza ki az összes Nobel díjast évszám szerinti rendezettségenben!
- Készítse el egy adott szakterület listáját: kérje be a felhasználótól, hogy melyik szakterületre kíváncsi, majd listázza ki e szakterület Nobel díjasait évszám szerint rendezetten!
- Írja ki évszám szerint rendezetten, hogy évszámonként összesen mennyi Nobel díjas van!

Extra ellenőrzéseket nem kell végezni.

Megoldás

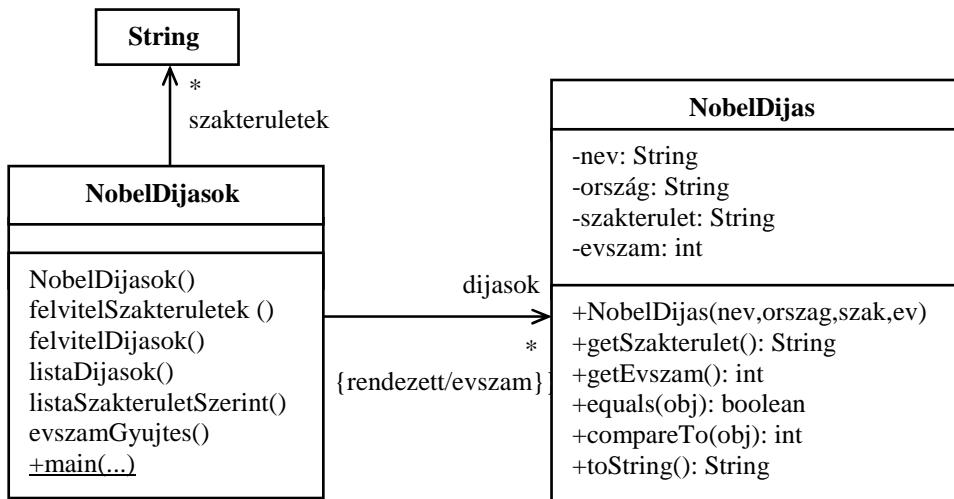
A NobelDijasok osztálydiagramját a 20.10. ábra mutatja. A NobelDijasok a vezérlő osztály, ebben minden használati esetnek felveszünk egy-egy metódust. A vezérlő egy-sok kapcsolatban áll a szakterületekkel (ezek egyszerű String objektumok) és a Nobel díjasokkal. A NobelDijas információtároló osztály, melynek az adatok kezelésén kívül biztosítania kell a helyes equals és compareTo metódusokat:

- ◆ equals: Akkor egyenlő két Nobel díjas, ha évszámuk és szakterületük is megegyezik (ilyenből nem lehet kettő a nyilvántartásban). A metódus visszatérési értéke true, ha minden két adat egyenlő.
- ◆ compareTo: A metódus évszám szerint rakja sorba az objektumokat: az egyik objektum akkor kisebb, mint a másik, ha kisebb a benne levő évszám.

Használati esetek:

- ◆ felvitelSzakteruletek: Itt felvisszük a szakterületeket, a sorrend mindegy.
- ◆ felvitelDijasok: Mivel nem vihetünk fel két olyan objektumot, melyben az évszám és a szakterület egyaránt megegyeznek, a keresést évszám+szakterületre kell elvégezni. Az equals pont erre keres rá. A létrehozott Nobel díjasokat a dijasok vektorba teszszük, melyet a felvitel végén rendezünk.
- ◆ listaDijasok: A Nobel díjasok teljes listája évszám szerinti rendezettségenben készül.
- ◆ listaSzakteruletszerint: Az évszám szerint rendezett vektoron végighaladunk, és kiválasztjuk azokat az adott szakterületű objektumokat.

- ♦ `evszamGyujtes`: Felveszünk egy tömböt, ahol minden évszámnak van egy rekesze. Végigmegyünk a tömbön (a rendezettség nem számít), és az objektum évszámának megfelelő rekeszt eggyel növeljük. Végül listázzuk a tömböt.



20.10. ábra. A NobelDijasok osztálydiagramja

Forráskód

```

import extra.*;
import java.util.*;

class NobelDijas implements Comparable {
    private String nev;
    private String orszag;
    private String szakterulet;
    private int evszam;

    public NobelDijas(String iNev, String iOrszag,
                       String iSzakterulet, int iEvszam){
        nev = iNev;
        orszag = iOrszag;
        szakterulet = iSzakterulet;
        evszam = iEvszam;
    }

    public String getSzakterulet(){
        return szakterulet;
    }

    public int getEvszam(){
        return evszam;
    }
}
  
```

```
public boolean equals(Object obj){  
    NobelDijas dijas = (NobelDijas) obj;  
    return (evszam==dijas.evszam) &&  
        (szakterulet.equals(dijas.getSzakterulet()));  
}  
  
public int compareTo(Object obj){  
    return evszam - ((NobelDijas)obj).evszam;  
}  
  
public String toString() {  
    return Format.left(nev,20)+Format.left(orszag,20)+  
        Format.left(szakterulet,20)+Format.left(evszam,10);  
}  
}  
  
public class NobelDijasok {  
    private Vector szakteruletek = new Vector();  
    private Vector dijasok = new Vector();  
  
    // Szakterületek felvitele végig. Két egyenlő nem lehet:  
    void felvitelSzakteruletek(){  
        String szakterulet;  
        while (true) {  
            szakterulet = Console.readLine("Szakterület: ");  
            if (szakterulet.equals(""))  
                return;  
            if (szakteruletek.indexOf(szakterulet)>=0)  
                System.out.println("Már van ilyen");  
            else  
                szakteruletek.add(szakterulet);  
        }  
    }  
  
    void felvitelDijasok(){  
        String nev, orszag, szakterulet;  
        int evszam;  
        NobelDijas dijas;  
  
        System.out.println("\n*** Nobel díjasok felvitele ***");  
        while (true) {  
            szakterulet = Console.readLine("\nSzakterület: ");  
            if (szakterulet.equals(""))  
                break;  
            if (szakteruletek.indexOf(szakterulet)<0) {  
                System.out.println("Nincs ilyen szakterület");  
                continue;  
            }  
  
            evszam = Console.readInt("Évszám: ");  
            dijas = new NobelDijas("", "", szakterulet, evszam);  
            if (dijasok.contains(dijas)) {  
                System.out.println("Ilyen szakterület + év már van!");  
                continue;  
            }  
        }  
    }  
}
```

```
nev = Console.readLine("Név: ");
orszag = Console.readLine("Ország: ");
dijas = new NobelDijas(nev,orszag,szakterulet,evszam);
dijasok.add(dijas);
}
Collections.sort(dijasok);
}

void listaDijasok() {
    System.out.println("\n*** Nobel díjasok ***");
    for (int i=0; i<dijasok.size();i++) {
        System.out.println((NobelDijas)dijasok.get(i));
    }
}

void listaSzakteruletSzerint(){
    NobelDijas dijas;
    String szakterulet;
    szakterulet = Console.readLine("\nSzakterület: ");
    System.out.println(Format.left("Név",20) +
        Format.left("Ország",20)+Format.left("Évszám",10));
    for (int i=0; i<dijasok.size(); i++){
        dijas = (NobelDijas)dijasok.get(i);
        if (szakterulet.equals(dijas.getSzakterulet()))
            System.out.println(dijas);
    }
}

void evszamGyujtes() {
    int[] gyujto = new int[2010];
    for (int i=0; i<dijasok.size(); i++){
        int ev = ((NobelDijas)dijasok.get(i)).getEv();
        gyujto[ev]++;
    }
    System.out.println("\nNobel díjasok száma év szerint");
    for (int ev=0; ev<gyujto.length; ev++) {
        if (gyujto[ev]!=0)
            System.out.println(Format.right(ev,4)+" : " +
                Format.right(gyujto[ev],6));
    }
}

public static void main(String[] args) {
    NobelDijasok nd = new NobelDijasok();
    nd.felvitelSzakteruletek();
    nd.felvitelDijasok();
    nd.listaDijasok();
    nd.listaSzakteruletSzerint();
    nd.evszamGyujtes();
}
}
```

20.8. Feladat – Városok

Feladatspecifikáció

Írjon programot, amely nyilvántartja a magyar városok adatait (város neve, területe km^2 -ben, lakosok száma). Két egyforma névű város nem létezhet a nyilvántartásban. Menüből választhatóan legyenek hívhatók a következő funkciók:

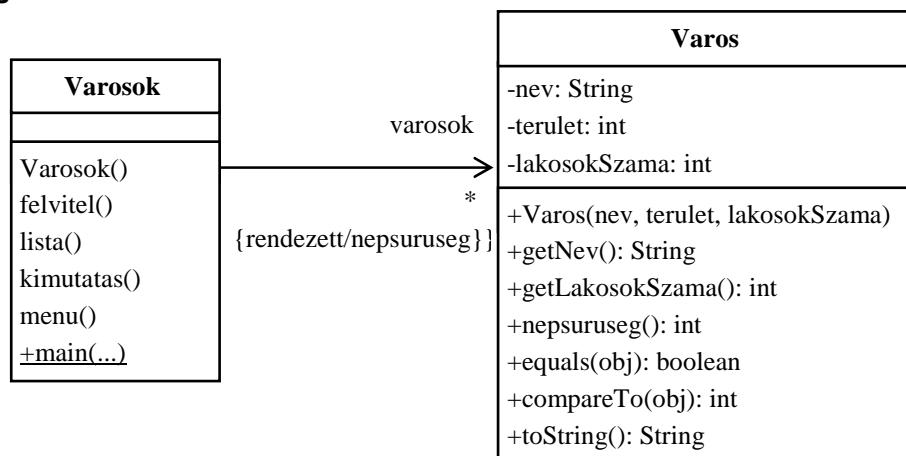
- Új városok felvitele
- Városok listája népsűrűség szerint csökkenő sorrendben: név, terület, lakosok száma, népsűrűség ($\text{fö}/\text{km}^2$).
- Kimutatás, amely megadja, hogy hány város van a nyilvántartásban a lakosok száma szerinti következő kategóriákban:

| Lakosok száma | Városok száma |
|-------------------|---------------|
| 0 – 99999 | 99 |
| 100000 – 999999 | 99 |
| 1000000 – 9999999 | 99 |
| 10000000 – | 99 |

– Vége

Extra ellenőrzéseket nem kell végezni.

Megoldás



20.11. ábra. A Varosok program osztálydiagramja

A **Varosok** osztálydiagramját a 20.11. ábra mutatja. A vezérlőben minden egyes használati esetnek egy-egy metódus felel meg, a használati eseteket menüből vezéreljük. A vezérlő kapcsolatban áll a városokkal: az egy-sok kapcsolatot egy olyan vektorral oldjuk meg, amely népsűrűség szerint rendezett. Egy városra három adat jellemző: neve, területe és a lakosok száma –

a népsűrűség ezekből számítható. A kereséshez és rendezéshez a megfelelő metódusok a következők:

- ◆ `equals`: Mivel két egyforma nevet nem engedünk meg, a vektorban pontosan a névre kell rákeresnünk. Két város tehát akkor egyenlő, ha a nevük egyenlő.
- ◆ `compareTo`: Ahhoz, hogy a népsűrűség szerint csökkenő rendezéshez használhassuk a `Collections.sort` metódust, ennek a metódusnak akkor kell `true` értéket visszatérítenie, ha a város népsűrűsége nagyobb, mint a paraméterben megadott városé.

A városok vezérlő osztály metódusai:

- ◆ `felvitel`: Sorban bekérjük a városok nevét, végigelő. Ha már létezik ilyen város, akkor kiírjuk, hogy „Már van ilyen város!”; ha még nincs ilyen, akkor bekérjük hozzá a többi adatot, és a várost betesszük a `varosok` vektorba. Végül rendezzük a vektort.
- ◆ `lista`: A vektor elemeit egyenként kiírjuk.
- ◆ `kimutatás`: Felveszünk egy 4 elemű gyűjtő tömböt. Végigmegyünk az összes városon: minden egyes esetben a lakosok számát leképezzük a tömb megfelelő elemére, és eggyel megnöveljük azt. Végül kilistázzuk a gyűjtött értékeket.
- ◆ `menu`: A metódus a felhasználó kérésétől függően meghívja az előbb megadott eseteket.

Forráskód

```
import extra.*;  
import java.util.*;  
  
class Varos implements Comparable {  
    private String nev;  
    private int terulet;  
    private int lakosokSzama;  
  
    public Varos(String nev, int terulet, int lakosokSzama) {  
        this.nev=nev;  
        this.terulet=terulet;  
        this.lakosokSzama=lakosokSzama;  
    }  
    public String getNev() {  
        return nev;  
    }  
    public int getLakosokSzama() {  
        return lakosokSzama;  
    }  
    public int nepsuruseg() {  
        return lakosokSzama/terulet;  
    }  
    public boolean equals(Object obj) {  
        return nev.equals(((Varos)obj).getNev());  
    }  
}
```

```
public int compareTo(Object obj) {
    return nepsuruseg() - ((Varos)obj).nepsuruseg();
}

public String toString() {
    return Format.left(nev,10)+ Format.right(terulet,4)+  

        Format.right(lakosokSzama,14)+  

        Format.right(nepsuruseg(),12);
}

public class Varosok {
    private Vector varosok= new Vector();

    void felvitel() {
        String nev = Console.readLine("\nVáros neve      : ");
        while (!nev.equals("")) {
            Varos varos = new Varos(nev,0,0);
            if(varosok.contains(varos))
                System.out.println("Már van ilyen város!");
            else {
                int terulet = Console.readInt("Város területe : ");
                int lakosok = Console.readInt("Város lakossága: ");
                varos = new Varos(nev,terulet,lakosok);
                varosok.add(varos);
            }
            nev = Console.readLine("\nVáros neve      : ");
        }
        Collections.sort(varosok);
    }

    void lista() {
        System.out.println
            ("\nVárosok népsűrűség szerint rendezve");
        System.out.println(
            "Név      Terület Lakosok száma Népsűrűség");
        for (int i=0; i<varosok.size(); i++) {
            System.out.println(varosok.get(i));
        }
    }

    void kimutatas() {
        int[] gyujto = new int[4];
        for (int i=0; i<varosok.size(); i++) {
            Varos varos = (Varos)(varosok.get(i));
            // Indexképzés a jegyek száma alapján:
            int index = (" "+varos.getLakosokSzama()).length();
            index = index-5;
            if (index < 0)
                index = 0;
            if (index > 3)
                index = 3;
            gyujto[index]++;
        }
    }
}
```

```

        System.out.println("\n  Lakosok száma    Városok száma");
        System.out.println("      0- 99999 " +Format.right(gyujto[0],5));
        System.out.println(" 100000- 999999 " +Format.right(gyujto[1],5));
        System.out.println(" 1000000-9999999 " +Format.right(gyujto[2],5));
        System.out.println("10000000-           " +Format.right(gyujto[3],5));
    }

    void menu() {
        char valasz;
        do {
            System.out.println("\n1- Felvitel");
            System.out.println("2- Lista");
            System.out.println("3- Kimutatás");
            System.out.print ("0- Kilépés ");
            valasz=Console.readChar();
            switch (valasz) {
                case '1': {felvitel(); break;}
                case '2': {lista(); break;}
                case '3': {kimutatas(); break;}
            }
        }
        while (valasz!='0');
    }

    public static void main(String[] args) {
        new Varosok().menu();
    }
}

```

20.9. Feladat – Autóeladás

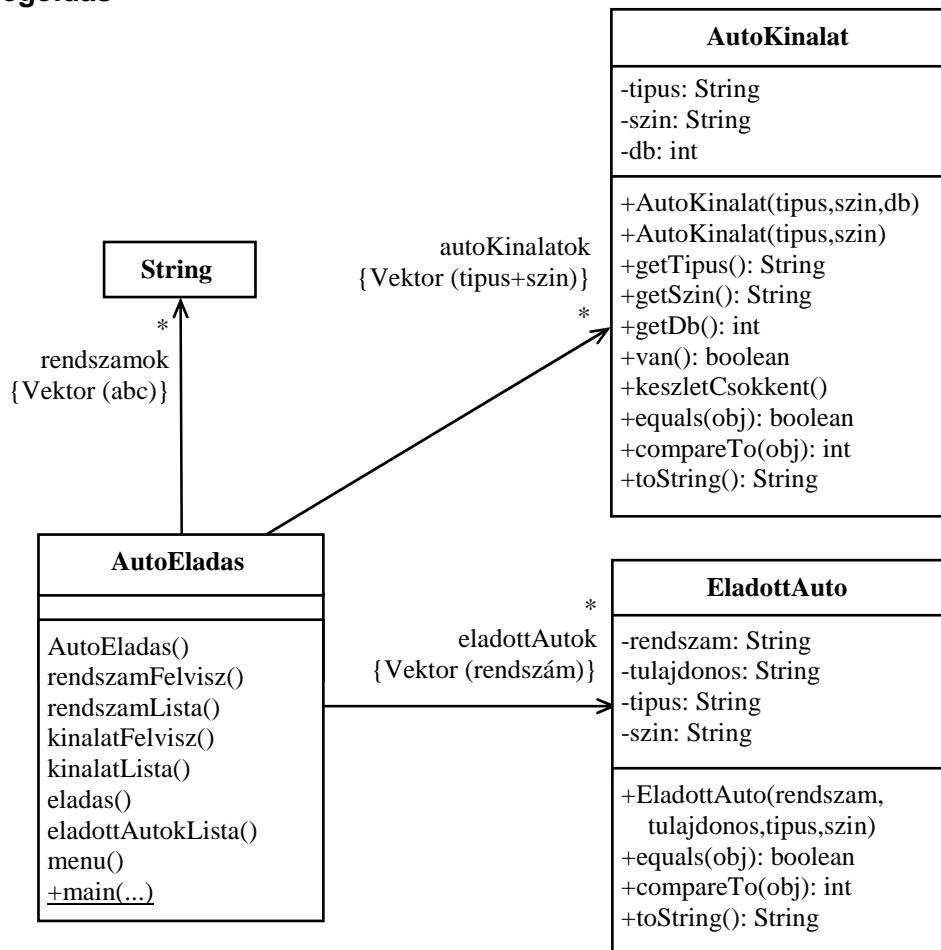
Feladatspecifikáció

Egy autókereskedés adott típusú és színű autók forgalmazásával foglalkozik. Az aktuális készletet (milyen autóból hány darab van), a rendszámtáblákat és az eladott autókat programmal tartják nyilván. A program funkciói (menüből választhatók):

- Eladás: A kuncsaft megmondja a nevét, és hogy milyen típusú és színű autót kér. Ha van ilyen autó, akkor a program az autóhoz rendszámtáblát rendel, az autót és a rendszámtáblát a készletből kiemeli, és az autót eladott autóként nyilvántartja a tulajdonos (kuncsaft) nevével együtt.
- Aktuális készlet listája (típus, szín, darab) típus, s azon belül szín szerint rendezetten.
- Még fel nem használt rendszámtáblák listája rendszám szerint rendezetten.
- Eladott autók listája (rendszám, típus, szín, tulajdonos), rendszám szerint rendezetten.

A program elején vigyük be a készlet és rendszámtábla adatokat. Az autó-, illetve rendszámkészlet bővítése nem feladata a programnak.

Megoldás



20.12. ábra. Az Autóeladás osztálydiagramja

A program tervét a 20.12. ábra mutatja. Osztályleírások:

- ◆ **AutoKinalat:** Az adott típusú és színű autók osztálya. Ezeken az adatokon kívül az jellemző rá, hogy hány darab ilyen autó van a készletben. A darabszámot eladáskor egygyel csökkentenünk kell, ezt végzi el a `keszletCsokkent` metódus. A `van` metódus megmutatja, hogy van-e egyáltalán ebből az autóból készleten. Nem lehet két egyforma objektum, ugyanazzal a típussal és színnel, ezért az `equals` metódust eszerint írjuk meg. A `compareTo` metódust a típus+szín szerinti rendezettség határozza meg.

- ◆ **EladottAuto:** Egy eladott autóra jellemző annak rendszáma, tulajdonosa, az autó típusa és színe. Két egyforma rendszámú autó nem lehetséges majd, és a rendezettség rendszám szerint lesz.
- ◆ **AutoEladas:** Ez a vezérlő osztály. Az eladas a legösszetettebb metódus: megnézi, hogy van-e egyáltalán eladni való autó és van-e hozzá rendszám. Ha van, akkor bekéri az óhajtott autó adatait. Ha van ilyen autó készleten, akkor kivesz a rendszamok konténerből egy rendszámot, csökkenti az autókínálatok megfelelő autójának készletét eggyel, összeállít egy EladottAutó objektumot, melyet betesz az eladottAutok konténerbe.

Forráskód

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

/* AutoKinalat *****/
class AutoKinalat implements Comparable {
    private String tipus;
    private String szin;
    private int db;

    public AutoKinalat(String tipus, String szin, int db) {
        this.tipus = tipus;
        this.szin = szin;
        this.db = db;
    }

    public AutoKinalat(String tipus, String szin) {
        this(tipus,szin,0);
    }

    public String getTipus() { return tipus; }
    public String getSzin() { return szin; }
    public int getDb() { return db; }
    public boolean van() { return db>0; }
    public void keszletCsokkent() {
        if (db>0) db--;
    }

    public boolean equals(Object obj) {
        AutoKinalat ak = (AutoKinalat)obj;
        return tipus.equals(ak.getTipus()) &&
            szin.equals(ak.getSzin());
    }

    public int compareTo(Object obj) {
        AutoKinalat ak = (AutoKinalat)obj;
        int comp = tipus.compareTo(ak.tipus);
        if (comp!=0 )
            return comp;
        return szin.compareTo(ak.szin);
    }
}
```

```
public String toString() {
    return Format.left(tipus,10) +
           Format.left(szin,10) + Format.left(db,6);
}
}

/* EladottAuto *****/
class EladottAuto {
    private String rendszam;
    private String tulajdonos;
    private String tipus;
    private String szin;
    public EladottAuto(String rendszam, String
        tulajdonos, String tipus, String szin) {
        this.rendszam = rendszam;
        this.tulajdonos = tulajdonos;
        this.tipus = tipus;
        this.szin = szin;
    }
    public boolean equals(Object obj) {
        return rendszam.equals(((EladottAuto)obj).rendszam);
    }
    public int compareTo(Object obj) {
        return rendszam.compareTo(((EladottAuto)obj).rendszam);
    }
    public String toString() {
        return Format.left(rendszam,10) + " " +
               Format.left(tulajdonos,10) + " " +
               Format.left(tipus,10) + " " +
               Format.left(szin,10);
    }
}

/* AutoEladas *****/
public class AutoEladas {
    private Vector rendszamok = new Vector();
    private Vector autoKinalatok = new Vector();
    private Vector eladottAutok = new Vector();
    void rendszamFelvisz() {
        String rendszam = Console.readLine("Rendszám: ");
        while (!rendszam.equals("")) {
            rendszamok.add(rendszam.toUpperCase());
            rendszam = Console.readLine("Rendszám: ");
        }
        Collections.sort(rendszamok);
    }
    void rendszamLista() {
        System.out.println("Rendszámok listája");
        for (int i=0; i<rendszamok.size(); i++)
            System.out.println(rendszamok.get(i));
    }
}
```

```
void kinalatFelvisz() {
    String tipus = Console.readLine("Autótípus: ");
    String szin;
    int db;
    while (!tipus.equals("")) {
        autoKinalatok.add(new AutoKinalat(tipus,
            Console.readLine("Szín: "),
            Console.readInt("Db: ")));
        tipus = Console.readLine("Autótípus: ");
    }
    Collections.sort(autoKinalatok);
}
void kinalatLista() {
    System.out.println("Kínálatok listája");
    for (int i=0; i<autoKinalatok.size(); i++)
        System.out.println(autoKinalatok.get(i));
}

void eladas() {
    if (autoKinalatok.isEmpty())
        System.out.println("Elfogyott az autó!");
    if (rendszerok.isEmpty())
        System.out.println("Elfogyott a rendszám!");

    kinalatLista();
    String tipus = Console.readLine("Milyen típust kér? ");
    String szin = Console.readLine("Milyen színt kér? ");
    int index = autoKinalatok.indexOf(
        new AutoKinalat(tipus,szin));
    if (index<0) {
        System.out.println("Nincs ilyen autó");
        return;
    }
    AutoKinalat autoKinalat =
        (AutoKinalat)(autoKinalatok.get(index));
    if (!autoKinalat.van()) {
        System.out.println("Elfogyott");
        return;
    }

    autoKinalat.keszletCsokkent();
    String rendszam = (String)rendszerok.remove(0);
    String tulajdonos = Console.readLine("Tulajdonos: ");
    EladottAuto eladottAuto = new EladottAuto(
        rendszam,tulajdonos,tipus,szin);
    eladottAutok.add(eladottAuto);
    System.out.println("A vásárolt autó adatai: "+eladottAuto);
}

void eladottAutokLista() {
    System.out.println("Eladott autók listája");
    for (int i=0; i<eladottAutok.size(); i++)
        System.out.println(eladottAutok.get(i));
}
```

```
public void menu() {
    rendszamFelvisz();
    rendszamLista();
    kinalatFelvisz();
    kinalatLista();
    char menu;

    do {
        menu = Character.toUpperCase(Console.readChar
            ("A(uto eladas)/K(inalatlista)/R(endszamlista)"+
            "/E(ladott autok listaja)/V(ege)? "));
        switch (menu) {
            case 'A' : { eladas(); break; }
            case 'R' : { rendszamLista(); break; }
            case 'K' : { kinalatLista(); break; }
            case 'E' : { eladottAutoLista(); break; }
        }
    } while (menu != 'V');
}

public static void main (String args[]) {
    new AutoEladas().menu();
}
```

Tesztkérdések

- 20.1. Jelölje meg az összes helyes állítást a következők közül!
 - a) A konténer ismeri a benne tárolt objektumok osztályát.
 - b) A Vector egy leképezés konténer.
 - c) A vektor mérete az elemek hozzáadásával automatikusan bővül.
 - d) A vektor tartalmazhat akár 5 milliárd objektumot is.
- 20.2. Jelölje meg az összes helyes állítást a következők közül!
 - a) A vektor elemei indexelhetők.
 - b) A vektor az elemeket rendezetten tárolja.
 - c) A vektor két objektuma (egyik és másik) egyenlő, ha az `egyik.compareTo(másik)==0`.
 - d) Egy vektor elejére beszúrható egy másik vektor összes eleme.
- 20.3. Jelölje meg az összes helyes állítást a következők közül!
 - a) A Vector osztály egy objektumot az elemein értelmezett `compareTo` metódus alapján keres meg.
 - b) A vektor átadható paraméterként a `println()` metódusnak.
 - c) A Vector osztály a List osztály utódja.
 - d) A Vector osztály a Collections osztály utódja.
- 20.4. Jelölje meg az összes helyes állítást a következők közül!
 - a) A származtatott interfész példányosítható.
 - b) Egy osztály több interfészt is implementálhat.

- c) A Collections osztály példányai képesek konténerek rendezésére.
- d) A Collection interfész statikus metódusai segítségével egy tetszőleges objektumokat tartalmazó konténer elemei rendezhetők.

Feladatok

- 20.1. (A) Kérjen be a felhasználótól egész számokat! A számok számát nem tudjuk előre (lehet nagyon sok is), és lehetnek közöttük egyenlők is. Írja meg a következő metódusokat a számsorozattal kapcsolatosan:
- a) kiírja a számokat a bevitel sorrendjében;
 - b) kiírja a számokat fordított sorrendben;
 - c) minden negatív számot nullára cserél;
 - d) visszaadja a sorozat átlagát;
 - e) visszaadja, hogy egy adott számból hány van a sorozatban;
 - f) kitörli a sorozatból a paraméterben megadott összes számot;
 - g) kitörli a sorozatból a paraméterben megadott összes számot, kivéve az elsőt!

Tipp: csomagolja be a számokat, és tegye be őket egy vektorba!

Ezután

- h) írja ki a legkisebb és a legnagyobb számot!
- i) rendezze a sorozatot növekvőleg!
- j) fordítsa meg a sorozatot!
- k) keverje össze a sorozatot!
- l) rendezze megint a sorozatot növekvőleg!
- m) bináris kereséssel keresse meg a bekért számot, és írja ki annak indexét!

A feladathoz megoldásához használja Collections osztály metódusait!

(SzamSorozat.java)

- 20.2. (A) Írja át a fejezetben található TorpeProgram.java programot: A bevitel után rendezze a törpéket név szerint! (*TorpeProgram2.java*)
- 20.3. (B) A 20.3 pontban található Torpek.java programot írja át a következőképpen: menüből hívhatóan készítse el a következő funkciókat:
- a) Vigyen fel új törpéket!
 - b) Töröljön egy adott nevű törpét!
 - c) A törpék növekedhessenek, illetve zsugorodhassanak. Tegye lehetővé egy adott nevű törpe magasságának állítását!
 - d) Írja ki a törpéket magasság szerint növekvő rendezettségen!
 - e) Írja ki a törpéket magasság szerint csökkenő rendezettségen!
 - f) Írja ki a legalacsonyabb és a legmagasabb törpe nevét és magasságát! Ha több van, akkor írja ki az összeset.
- (*TorpeProgram3.java*)
- 20.4. (B) Kérjen be a konzolról dátumokat (év, hó, nap). Rendezze a dátumokat növekvő sorrendbe! Írja ki a dátumok listáját rendezés előtt és után! (A feladatot saját dátum osztállyal oldja meg!) (*DatumRend.java*)

- 20.5. **(B)** A 20.5 pontban található `Emberek.java` programot változtassa meg a következőképpen: Ne engedjen felvinni egyforma nevű embert még akkor sem, ha születési évük különbözik! (`Emberek2.java`)
- 20.6. **(B)** Fejlessze tovább a 17. fejezet `RaktarProgram.java` mintaprogramját úgy, hogy a raktárban elvileg akárhány áru lehet. Egy adott árúból a raktárba be lehet tenni, illetve onnan ki lehet venni egy adott mennyiséget. (`RaktarProgram3.java`)
- 20.7. **(B)** Fejlessze tovább a 17. fejezet `Bank.java` mintaprogramját úgy, hogy menüből új számlát is lehet nyitni, illetve törlni lehet már meglévő számlát! A banknak tehát tetszőlegesen sok számlája lehet. (`Bank2.java`)
- 20.8. **(B)** Egy szervizben a szervizelt autókról számítógépes nyilvántartást vezetnek. Egy autóról a következő adatokat tartják nyilván: rendszám, tulajdonos és fogyasztás.
Készítsen egy konzolos alkalmazást, amelyben menüből választhatóan a következő funkciókat lehet kérni:
- Új autó adatainak felvitele (két egyforma rendszámú autót nem lehet felvinni)
 - Az összes autó adatainak kilistázása a tulajdonos neve szerinti rendezettségben
 - Egy adott rendszámú autó adatainak kiírása
(`Szerviz.java`)
- 20.9. **(C)** Egy férfi futóversenyen négy féle távon indulhatnak a versenyzők: 100, 200, 400 és 800 méteren. A program tartsa nyilván a futóverseny nevezéseit és eredményeit! Nevezéskor meg kell adni a versenyző nevét és a távot, amelyen indul. Egy adott nevű versenyző csak egy távon indulhat! minden nevezéshez majd egy eredmény fog tartozni: a mért futási idő másodpercben (tört szám is lehet). A program menüből kérhetően végezze el a következő feladatokat:
- Vegyen fel nevezéseket (az adatokat konzolról kérje be)! A távot a felkínált lehetőségek alapján számmal kell megadni (1=100m, 2=200m, 3=400m, 4=800m). A versenyző eredménye kezdetben 0, ami azt jelenti, hogy még nincs eredménye.
 - Tegye lehetővé az eredmény beírását!
 - Listázza ki az összes versenyzőt nevezési sorrendben!
 - Írja ki az első három helyezettet minden egyes táv esetén!
- Minden lista a versenyző összes adatát tartalmazza, igazítva! Extra ellenőrzéseket nem kell végeznie. (`FutoVerseny.java`)
- 20.10. **(C)** Készítsen a zöldségesnek egy nyilvántartó és számlázó programot! A program tárolja minden egyes zöldség nevét, árát, a boltban levő aktuális mennyiséget (kg-ban), valamint az ajánlott minimális és maximális mennyiséget (ez utóbbi két adat alapján történik az utánrendelés). A program funkciói a következők:
- Új áru nyilvántartásba vétele
 - Áru törlése a nyilvántartásból
 - Áru egységárának módosítása

- Vásárlás: Kilistázza a választható zöldségek neveit. A vevő választ egy árut, és megadja a vásárolni kívánt mennyiséget. A vevő egymás után több árut is kérhet. A vásárlás végén készítünk számlát: a számlán csak azok a zöldségek szerepeljenek, amelyekből történt vásárlás, mégpedig mindenki csak egyszer. A számlán zöldségenként legyen rajta annak neve, egységára, összes vásárolt mennyisége és értéke. Végül írjuk ki a fizetendő összeget!
- Utánrendelés: Egy zöldségből akkor kell rendelni, ha az aktuális mennyiség a minimális szint alá került. Ekkor annyi árut rendelünk, hogy a készlet maximális legyen. A program írja ki, hogy melyik zöldségből kell rendelni, és mennyit!
- Záras: Kilépés a programból. Írunk ki egy statisztikát az aznapi vásárlásokról: melyik zöldségből mennyit adtunk el, miből adtunk el a legtöbbet, miből nem adtunk el semmit.

(*Zoldseges.java*)

I. BEVEZETÉS A PROGRAMOZÁSBA

1. A számítógép és a szoftver
2. Adat, algoritmus
3. A szoftver fejlesztése

II. OBJEKTUMORIENTÁLT PARADIGMA

4. Mitől objektumorientált egy program?
5. Objektum, osztály
6. Társítási kapcsolatok
7. Öröklödés
8. Egyszerű OO terv – Esettanulmány

III. JAVA KÖRNYEZET

9. Fejlesztési környezet – Első programunk
10. A Java nyelvről

IV. JAVA PROGRAMOZÁSI ALAPOK

11. Alapfogalmak
12. Kifejezések, értékadás
13. Szelekciók
14. Iterációk
15. Metódusok írása

V. OSZTÁLYOK HASZNÁLATA, KÉSZÍTÉSE

16. Objektumok, karakterláncok, csomagolók
17. Osztály készítése

VI. KONTÉNEREK

18. Tömbök
19. Rendezés, keresés, karbantartás
20. A Vector és a Collections osztály



FÜGGELÉK

- A tesztkérdések megoldásai
Irodalomjegyzék
Tárgymutató

F

A tesztkérdések megoldásai

1. A számítógép és a szoftver

| | | | | | | | |
|-------|-------|-------|------|------|------|--------|-------|
| 1. ad | 2. cd | 3. bd | 4. b | 5. d | 6. b | 7. abc | 8. cd |
|-------|-------|-------|------|------|------|--------|-------|

2. Adat, algoritmus

| | | | | |
|-------|-------|-------|------|--------|
| 1. bd | 2. ac | 3. bd | 4. d | 5. abc |
|-------|-------|-------|------|--------|

3. A szoftver fejlesztése

| | | | | |
|-------|-------|--------|-------|--------|
| 1. bd | 2. ac | 3. acd | 4. ad | 5. bcd |
|-------|-------|--------|-------|--------|

4. Mitől objektumorientált egy program?

| | | |
|-------|-------|-------|
| 1. bd | 2. ad | 3. ab |
|-------|-------|-------|

5. Objektum, osztály

| | | | |
|-------|------|--------|-------|
| 1. cd | 2. c | 3. abd | 4. bc |
|-------|------|--------|-------|

6. Társítási kapcsolatok

| | | | |
|-------|--------|--------|------|
| 1. bc | 2. acd | 3. abd | 4. c |
|-------|--------|--------|------|

7. Öröklődés

| | | | | |
|------|--------|-------|-------|------|
| 1. b | 2. abd | 3. bd | 4. bc | 5. a |
|------|--------|-------|-------|------|

8. Egyszerű OO terv – Esettanulmány

| | | | | | |
|-------|-------|-------|-------|--------|-------|
| 1. ad | 2. ab | 3. ac | 4. cd | 5. abc | 6. cd |
|-------|-------|-------|-------|--------|-------|

9. Fejlesztési környezet – Első programunk

| | | | | | | | | | |
|-------|-------|--------|------|-------|--------|-------|-------|------|-------|
| 1. cd | 2. a | 3. abc | 4. b | 5. cd | 6. bcd | 7. cd | 8. cd | 9. a | 10. b |
| 11. d | 12. a | | | | | | | | |

10. A Java nyelvről

| | | | | | | | |
|------|------|------|-------|------|------|--------|--------|
| 1. b | 2. b | 3. d | 4. ab | 5. c | 6. d | 7. bcd | 8. acd |
|------|------|------|-------|------|------|--------|--------|

11. Alapfogalmak

| | | | | | | | | |
|-------|-------|-------|------|-------|--------|-------|------|------|
| 1. bc | 2. ac | 3. cd | 4. c | 5. ac | 6. acd | 7. cd | 8. c | 9. b |
|-------|-------|-------|------|-------|--------|-------|------|------|

12. Kifejezések, értékkadás

| | | | | | | | | | |
|------|------|------|-------|-------|-------|------|------|------|--------|
| 1. b | 2. b | 3. b | 4. bd | 5. ab | 6. ac | 7. b | 8. a | 9. b | 10. ab |
|------|------|------|-------|-------|-------|------|------|------|--------|

13. Szelekciók

| | | | | |
|-------|--------|------|--------|--------|
| 1. ad | 2. bcd | 3. c | 4. abc | 5. abd |
|-------|--------|------|--------|--------|

14. Iterációk

| | | | | | | | |
|--------|------|--------|------|--------|------|------|--------|
| 1. acd | 2. b | 3. abd | 4. b | 5. abd | 6. b | 7. c | 8. abd |
|--------|------|--------|------|--------|------|------|--------|

15. Metódusok írása

| | | | | | | | | | |
|-------|-------|------|-------|------|--------|------|------|------|-------|
| 1. b | 2. cd | 3. c | 4. bd | 5. b | 6. bcd | 7. c | 8. c | 9. b | 10. c |
| 11. d | | | | | | | | | |

16. Objektumok, karakterláncok, csomagolók

| | | | | | |
|-------|------|--------|--------|------|-------|
| 1. ad | 2. b | 3. abd | 4. bcd | 5. d | 6. ad |
|-------|------|--------|--------|------|-------|

17. Osztály készítése

| | | | | | | | | |
|-------|-------|------|-------|-------|------|------|--------|-------|
| 1. cd | 2. ad | 3. c | 4. bd | 5. ab | 6. d | 7. a | 8. acd | 9. bd |
|-------|-------|------|-------|-------|------|------|--------|-------|

18. Tömbök

| | | | | | | |
|-------|--------|------|--------|------|------|------|
| 1. cd | 2. acd | 3. c | 4. acd | 5. c | 6. b | 7. c |
|-------|--------|------|--------|------|------|------|

19. Rendezés, keresés, karbantartás

| | | | | |
|------|------|-------|--------|------|
| 1. c | 2. c | 3. ad | 4. abd | 5. c |
|------|------|-------|--------|------|

20. A Vector és a Collections osztály

| | | | |
|------|--------|------|------|
| 1. c | 2. acd | 3. b | 4. b |
|------|--------|------|------|

Irodalomjegyzék

Magyarul:

- [1] Angster Erzsébet: Az objektumorientált tervezés és programozás alapjai (UML, Turbo Pascal, C++), 1997
- [2] Jamsa, Kris: Java
Kossuth könyvkiadó, 1996, fordítás
- [3] Nyékyné G. Judit (szerk.) et al.: Java 2 Útikalauz programozóknak
ELTE TTK Hallgatói Alapítvány, 2000
- [4] Nyékyné G. Judit (szerk.) et al.: J2EE Útikalauz Java programozóknak
ELTE TTK Hallgatói Alapítvány, 2002
- [5] Vég Csaba - dr. Juhász István: Java - start!
Logos 2000, 1999

Angolul:

- [6] Booch, Grady: The Unified Modeling Language User Guide
Addison-Wesley, 1998
- [7] Bruce Eckel: Thinking in Java
Prentice Hall, 1999
- [8] Cay S. Horstmann, Gary Cornell: Core Java 1.2
Sun MicroSystems Press, 1999
- [9] Eriksson, Hans-Erik - Penker, Magnus: UML Toolkit
Wiley, 1998
- [10] Fowler, Martin: UML Distilled
Addison-Wesley, 1997
- [11] Gamma, Erich - Helm, Richard - Johnson, Ralph - Vlissides, John:
Design patterns
Addison-Wesley, 1997
- [12] Jacobson, Ivar: The Unified Software Development Process
Addison-Wesley, 1999

- [13] Khalid A. Mughal, Rolf W. Rasmussen: A Programmer's Guide to Java Certification
Addison-Wesley, 2000
- [14] Kruchten, Philippe: The Rational Unified Process - An Introduction
Addison-Wesley, 1999
- [15] Rumbaugh, James: OMT Insights
SIGS Books, 1996
- [16] Rumbaugh, James: The Unified Modeling Language Reference Manuel
Addison-Wesley, 1999
- [17] Y. Daniel Liang: Introduction to Java Programming
Que E&T, 1999

Tárgymutató

A, Á

absztrakció · 18, 57
adat · 23, 67, 72
adatbázis-kezelők · 171
adatok feldolgozása végjelleg · 264
additív operátorok · 217
aktor · 46, 74, 122
aktuális paraméterek · 283
alacsony szintű nyelv · 10
alaposztály · 104
algoritmus · 23
 tulajdonságai · 39
algoritmusvezérelt program · 117
alkalmazásböngésző · 136
alkalmazói
 programcsomagok · 15
állapot · 69
alozstály · 104
általánosítás · 58, 104
alulról felfelé tervezés · 19
analízis · 48, 50, 113
API csomagstruktúrája · 159
architektúra-semleges · 176
ArrayIndexOutOfBoundsException · 374
ASCII karakter · 183
assembler · 10
Assembly nyelv · 7, 10
átadás · 47
átlagszámítás · 267
automatikus befejezés · 153
automatikus konverzió · 221
automatikus
 metódusfelkínálás · 152
automatikus szemétygyűjtés · 308

azonosító · 187
elnevezési konvenciók · 188
hivatkozási kör · 355
képzési szabályok · 187

B

bájtkód · 10, 143
belépési feltétel · 251
beszúrás · 407
bezárás · 63, 81
bin könyvtár · 154
bináris keresés · 410
bináris operátor · 215
bitenkénti operátorok · 219
biztonságos · 176
blokk
 metódus · 199, 282
 utasítás · 289
Boolean osztály · 327
bottom-up · 19
böngésző · 174
bővíthetőség · 17
bővító konverzió · 221
break · 262

C

C++ · 170
CASE eszköz · 21
Character osztály · 327
ciklusok · 30, 34, 251
ciklusok egymásba ágyazása · 260
CLASSPATH · 156
Collection interfész · 454
Collections osztály · 456
Comparable interfész · 454

compareTo · 454
compiler · 12
Component osztály · 128
continue · 262

Cs

csomag szintű láthatóság · 109
csomagolás
 primitív típus · 325
csomagolók · 301

D

deklarálás · 27, 193
dekompozíció · 18
DeMorgan azonosságok · 228
dinamikus · 177
direkt rendezés · 407
do while · 253
dokumentálás · 53

E, É

editor · 11
egész literál · 190
egész típusok · 196
egyágú szelekció · 237
egydimenziós tömb · 369, 372
egyed objektum · 83
egy–egy kapcsolat · 90
egyenlőségvizsgálat
 objektum · 425, 446
 String · 314
egyenlőségvizsgáló operátor · 217

egymásba ágyazott IF · 241
 egymásba ágyazott
 szelektiók · 241
 Egységesített eljárás · 44, 49
 egy–sök kapcsolat · 92
 egyszeres öröklés · 106
 egyszerű · 176
 egyszerű OO terv · 111
 együttműködési diagram ·
 63, 113, 338
 Eiffel · 171
 eljárás · 35, 202, 280
 ellenőrizhetőség · 17
 elosztott · 176
 előjel operátor · 216
 előltesztelő ciklus · 251
 else if · 243
 Első programunk · 133
 equals metódus · 446
 erős kötés · 18
 erős tartalmazás · 87
 erősen típusos nyelv · 221
 értékkedás · 27, 213, 223
 interfész · 452
 kompatibilitás · 223
 tömb · 376
 értékkedás, objektum · 307
 értékkedő operátorok · 220
 értékkedő utasítás · 203
 értelmező · 13
 escape karakter · 184
 eseményvezérelt program ·
 117
 eszköztár · 136
 explicit típuskonverzió · 221
 extra.Console osztály · 204
 extra.Format osztály · 207

F

fehér szóköz · 186
 fejlesztési ciklus · 45
 fejlesztési dokumentáció ·
 120
 fejlesztési környezet · 133
 fejlesztőeszközök · 123
 fejlesztői dokumentáció · 53
 Feladat
 A legfiatalabb lány
 kiválasztása (1) · 24
 Angol ábécé · 259
 Átlag · 267
 Autóeladás · 470

Bank · 253, 345
 Betűgyűjtés · 384
 Cseré · 321
 Csillag · 140, 257
 Dátumbontás · 332
 Dobás–statisztika · 379
 Egy vagy két A betű ·
 218
 Életkor · 382
 Első osztható · 263
 Első öt · 259
 Emberek · 458
 Faktoriális · 291
 Fizetés · 239, 258
 Henger · 208
 Hordókeresés · 423
 Hurra · 149
 Jegyek száma · 255
 Jó szám · 241
 Kamat · 232
 Karbantartás, rendezett ·
 429
 Kis és nagy ábécé · 259
 Kocka · 246
 Kor · 244
 Kör kerülete · 264
 Következő · 286
 Következő karakter · 240
 Krumpli · 181
 Legelső · 314
 Legnagyobb · 241
 Maximális számla · 269
 Megszámol · 266
 Menü · 270
 Metódusminta · 277
 Metódusminta2 · 287
 Mi igaz · 245
 Milyen karakter? · 327
 Minimális és maximális
 kilométer · 268
 Minimumok · 288
 Nobel díjasok · 463
 Objektum paraméter ·
 322
 Osztható? · 248
 Összeg · 290
 Összegzés · 328
 Palindróma · 323
 Paraméter teszt · 392
 Prímek · 266
 Primitív rendezetlen ·
 415
 Primitív rendezett · 417
 Program paraméter teszt
 · 393
 Raktárprogram · 339
 Rendszám · 260
 Sinus · 292
 String rendezett · 473
 StringTeszt · 313
 Számbekérés · 254
 Számok · 377
 Szavak · 331
 Szavazatkiértékelés · 395
 Szorzótábla · 261
 Szökévé · 263
 Szövegek · 378
 TizedesPontCseré · 316
 Törpék · 448
 Valós számok · 145, 461
 Városok · 467
 Vector minta · 444
 Véletlen szám · 291
 felelősség · 63
 felhasználó · 44
 felhasználóbarátság · 17
 felhasználói dokumentáció ·
 53
 felmérés · 46
 feltétel · 227
 feltételes kiértékelés · 220
 felülről lefelé tervezés · 19
 felvitel · 411
 for · 256
 fordítás · 11, 139, 156
 több osztály esetén · 161
 fordítási egység · 161
 fordítóprogram · 12
 formális paraméterlista ·
 282
 forráskód · 11
 szerkezete · 197
 forráskód · 14
 forráskód strukturálása · 37
 forrásprogram · 51
 forrásprogram szerkezete ·
 197
 fő felelős · 116
 főmenü · 136
 főosztály · 104
 JFrame osztály · 128
 futás alatti hiba · 14
 futás alatti kötés · 66
 futtatás · 11, 139, 156
 független feltételek
 vizsgálata · 247

függvény · 35, 202, 280

G

gépi ábrázolás · 3
gépi kód · 5

Gy

gyenge tartalmazás · 87
gyűjtés · 379

H

hasonlítás
 objektum · 425
hasonlító objektum · 425
hasonlító operátor · 217
használati eset · 46
használati esetek · 51, 123
határ objektum · 83
hatékonyság · 17
hátróltesztelő ciklus · 253
helyesség · 17
helyzetérzékeny help · 153
hibafeltárás · 152
hibatüres · 17
hibrid nyelv · 170
hiperszöveg · 174
hivatkozási kör · 355
hordozható · 177
hordozhatóság · 17
hozzáférési mód · 108, 338

I,Í

if · 237
IF ... THEN ... ELSE IF ·
 243
if..else · 240
igazságátbla · 218, 229
implementálás · 48, 51
 interfész · 452
implicit típuskonverzió ·
 221
importálás · 199
indexelés
 String · 309
 tomb · 374
 Vector · 439
indexes rendezés · 407
indexképző algoritmus · 382

információ elrejtése · 19,
 63, 81

információhordozó

 objektum · 83

inicializáló

 blokk · 361

 kifejezés · 361

inicializáló blokk

 tömb · 375

inicializálók · 360

Insert · 151

integrált fejlesztői környezet
 · 163

integritás · 18

interfész · 106, 452

Interfész objektum · 83

Internet · 173

interpretált · 177

interpreter · 13

Interpreter · 14

ismeretségi kapcsolat · 59,
 85

ISO · 18

iteráció · 25, 48

iterációk · 30, 251

J

Java · 170
 típusok osztályozása ·
 194
 törtenete · 172
Java Fejlesztői Készlet · 153
Java nyelv
 jellemzői · 176
Java osztálykönyvtár · 159
java.applet · 160
java.awt · 160
java.io · 161
java.lang · 161
java.lang.Math osztály · 231
java.util · 161
JAVA_HOME · 155
Java-képes · 174
javalib könyvtár · 144
javaprog · 134, 146
javax.swing · 161
JBuilder · 134
JDK · 153, 165
 könyvtárstruktúrája · 153
 próbafutás · 149
jdkwork.bat · 156
JIT · 13

jre könyvtár · 154

K

kapcsolat
 egész-rész · 87
 egy–egy · 90
 egy–sok · 92
 foka · 89
 ismeretségi · 59, 86
 kötelező · 90
 objektumok között · 85
 opcionális · 90
 osztályok között · 89
 sok–sok · 93
 tartalmazási · 59, 85, 87

kapcsolat irányára · 85, 89

kapcsolat neve · 85, 89

kapcsolatok · 59

karakter literál · 191

karakter típus · 197

karakterláncok · 301

karbantartás · 411

karbantartása

 szövegek · 429

karbantarthatóság · 17

kényszerített konverzió ·

 221

képernyőterv · 121

kérélem · 62

keresés · 408

késői kötés · 66

kétágú szelekció · 240

kétdimenziós tömb · 385

kétimenziós tömb · 371

kidolgozás · 47

kifejezés

 alkotóelemei · 213

kifejezések · 213

kifejezések kiértékelése ·

 226

kis szoftver fejlesztése · 49

kiterjesztett értékkadás · 225

kiugrás a ciklusból · 260

kivételek · 200

kliens · 73

kód újrafelhasználása · 82

kódbeillesztés · 152

kódolás · 51

kódolási konvenció · 238

kollekció · 438

kompatibilitás · 17

kompozíció · 87, 88

konstans · 194

konstrukció · 47

konstruktör · 75, 305, 358
alapértelmezés szerinti · 359

túlterhelés · 360

konténer · 96, 437

konténerobjektum · 83

kontrollobjektum · 83

konverzió irányá · 221

konzol

adatbevitel · 204

megjelenítés · 206

könyv melléklete · 134

környezeti beállítások · 155

környezeti változók · 155

kötetelményfeltárás · 48,
50, 112

kötetelményspecifikáció ·
121

közvetlenős · 104

közvetlen utód · 104

kulcs · 411

kulcsszó · 189

L

láthatóság · 108, 338

laza kötés · 18

leképezés · 438

length konstans · 374

lépésenkénti finomítás · 19

léptető ciklus · 30, 256

léptető operátor · 216

leszármazott · 104

linker · 13

List interfész · 454

literál · 189

logikai operátorok · 217

logikai típus · 197

lokális változó · 289

M

magas szintű nyelv · 8

magas teljesítményű · 177

main metodús · 199

Math osztály · 231

maximumkválasztás · 268

megjegyzés · 186

megkülönböztetés · 57

megrendelő · 44

megszámlálás · 266

megvalósítás

társítási kapcsolat · 94

menükészítés · 270

metódus · 67, 278

hogyan tervezzük... · 292

metódus · 72

metódusdeklaráció · 199

metódushívás · 74, 201

metódusok írása · 277

metódusok túlterhelése ·

286

minimumkválasztás · 268

minimumkválasztásos

rendezés · 406

minőségi szoftver · 16

mintaprogram · 149, 181

mínusz · 216

módosítás · 411

módosítók · 281

tag · 354

moduláris programozás · 18

multiobjektum · 88

multiplicitás · 89

multiplikatív operátorok ·

217

munkafázis · 46

munkafolyamat · 48, 112

N

natív kód · 5

navigálás irányá · 85

navigálási irány · 89

new · 305

növekményes ciklus · 35

null referencia · 304

NullPointerException · 311

numerikus csomagoló

osztályok · 328

Ny

nyilvános · 108, 338

nyomkövetés · 164

O,Ó

Object osztály · 302

objektum · 60, 67

adat · 67

állapot · 69

azonosság · 70

deklarálása · 305

egyenlőségvizsgálat ·

308

élete · 308

értékkadás · 307

felelősségei · 122

inicializálása · 74

létrehozása · 74

létrehozása saját

osztályából · 343

metódus · 67

paraméter · 322

szemétgyűjtés · 308

sztereotípusa · 83

objektum alapú nyelv · 170

objektumdiagram · 86, 89,

113, 338

objektumorientált · 176

objektumorientált program

fő jellemzői · 59

objektumorientált

szoftverfejlesztési

módszer · 21

objektum

létrehozása · 305

operációs rendszerek · 14

operandus · 213

operátor · 192, 213

osztály meg és uralkodj · 18

osztály · 70

deklarációi · 351

felépítése · 351

inicializáló blokk · 361

készítése · 337

mintaprogram · 339

osztálydeklaráció · 199

osztálydiagram · 94, 104,

113, 338

osztályleírás · 113

osztálymetódus · 79, 353

osztályozás · 58, 64

osztálytag · 353

osztályváltozó · 79, 353

Ő,Ó

öröklés

egyszeres · 106

többszörös · 106

öröklődés · 65, 101

ősosztály · 104
összegzés · 267

P

package · 159
paraméter
 program · 393
paraméterátadás · 229, 283
 objektum · 322
 tömb · 392
parancssor-paraméter · 393
Pascal · 170
PATH · 155
példány · 70
példánydiagram · 86, 89
példánymetódus · 76, 353
példánytag · 353
példányváltozó · 76, 353
plusz · 216
polimorfizmus · 64
postfix · 215
pozicionálás · 151
prefix · 215
primitív típus · 304
 csomagolás · 325
primitív típusok · 195
primitív típusú
 paraméterátadás · 229
primitív típusú változó · 194
privát · 108, 338
program · 3, 4, 53
 alkotóelemei · 185
program strukturálása · 238
programfejlesztési
 módszertan · 21
programozási nyelv · 5, 10,
 24
programszerkesztő program
 · 13
programterv · 51, 123
projekt · 135
projektpanel · 137
prototípus · 121
pszeudokód · 33, 126

R

referencia típusú változó ·
 194
referenciatípus · 304
rendezés · 406
rendezetlen sorozat · 413

rendezett sorozat · 414

szerepnév · 89
szerkesztés · 11

return · 285
robosztus · 176
rutin · 35

S

segítség · 164
sérthetetlenség · 18
setjava.bat · 156
Smalltalk · 170
sok–sok kapcsolat · 93
sorszámozott típus · 195
specializálás · 58, 104
src.zip · 155
statikus metódus · 353
statikus változó · 353
státuszszor · 138
String
 egyenlőségvizsgálat ·
 314
String osztály · 309
StringBuffer osztály · 316
Stringek rendezése · 420
StringIndexOutOfBoundsException
 Exception · 311
StringTokenizer osztály ·
 329
strukturálás · 37, 238
strukturált algoritmus · 40
strukturált programozás · 21
struktúrapanel · 138
switch · 243

Sz

szabványosság · 18
szakterületi objektummodell
 · 51, 122
számítógépes szimuláció ·
 112
szárazteszt · 52
szekvencia · 25, 29, 33
szekvenciális keresés · 410
szelekció · 25, 238
 egyágú · 237
 egymásba ágyazott · 241
 többágú · 243
szelekciók · 29, 33, 237
szemantikai hiba · 52
szeparátor · 191

szerver · 73
szignatúra · 203, 230, 286
szintaktikai hiba · 52
szintaktikai kiemelés · 153,
 164
szoftver · 4
szoftver élete · 45
szoftverek osztályozása · 14
szoftverfejlesztés · 43, 49
szoftverfejlesztő · 44
szoftverfejlesztő rendszerek
 · 14
szoftverkrízis · 15
szövegliterál · 191
szövegszerkesztő · 151
szövegszerkesztő program ·
 11
sztereotípus · 83
szűkitő konverzió · 221

T

tag · 353
takarás
 azonosító · 355
tárgykód · 5
társítási kapcsolat · 85
 megvalósítás · 94
tartalmazási kapcsolat · 59,
 85
tartalompanel · 137
tervezés · 48, 51, 113
tesztadat · 52
tesztelés · 48, 52, 114
tevékenységsdiagram · 27
this objektumreferencia ·
 357
throws · 282
típus · 25, 192
típuskényszerítés · 221
típuskonverziók · 221
típusok osztályozása · 194
tiszta objektumorientált
 nyelv · 170
top-down · 19
többágú szelekciók · 243
többdimenziós tömb · 371,
 390
többszálú · 177

| | |
|--------------------------|-------------------------------------|
| többszörös öröklés · 106 | vizuális fejlesztőeszközök · 171 |
|--------------------------|-------------------------------------|

W

- tömb · 369
 - deklarálása · 372
 - értékkadás · 376
 - inicializáló blokk · 375
 - keresés · 408
 - kezdeti érték · 373
 - paraméterátadás · 392
 - rendezés · 406
 - szekvenciális
 - feldolgozása · 377
 - tömbök másolása · 388
- törlés · 151, 411
- túlterhelés · 203, 286
- túlterhelt metódus · 229

U,Ú

- újrafelhasználhatóság · 17
- UML · 44, 72, 301
- unáris operátor · 215
- unikód karakter · 183
- URL · 174
- utasítás · 200
- utód osztály · 104

Ü,Ű

- ügynök · 74
- üzenet · 62, 74
- üzenet · 345
- üzenetpanel · 138

V

- valós literál · 190
- valós típusok · 196
- valós világ modellezése · 57
- változó
 - kezdeti érték · 357
 - változó · 25, 192
 - deklarálása · 193
 - inicializálása · 194
 - lokális · 289
- Vector osztály · 439
- védett · 108, 338
- vezérlő · 116
- vezérlőszerkezetek · 25, 238
- visszatérési típus · 282

Vége