Debreceni Egyetem
Informatikai Kar

A JAVA NYELV TANÍTÁSA KÖZÉPISKOLÁBAN

Témavezető:

Kósa Márk
egyetemi tanársegéd

Készítette:

Fenyvesi Tibor
Szabó László
Informatika tanár szak

Debrecen
2010.

Tartalomjegyzék

Szerzők:

F.T.: Fenyvesi Tibor
Sz.L. Szabó László

1. Bevezetés

Napjaink közoktatásának hangsúlyos részévé vált az informatikai tudás. Míg korábban a számítástechnika – majd későbbi nevén informatika – tantárgy csak szakmai tárgyként szerepelt az iskolarendszerű oktatás palettáján, mára már közismereti tárggyá vált. Gyakorlatilag egyenrangú lett az alaptantárgyakkal.

Ma a mindennapi élet szinte minden területe megköveteli az informatikai alaptudást. Ez az alaptudás az informatikai eszközök és alkalmazói szoftverek legalább alapszintű használatát jelenti, amelyet a mai ifjúság a közoktatás keretein belül elsajátíthat. Van azonban az informatikai oktatásnak egy másik ága is, amely a kezdeti időszak (’80-as, ’90-es évek) egyeduralma után gyakorlatilag háttérbe szorult. Ez pedig a programozói tudás. A társadalomnak egyre több jó programozóra lenne szüksége, mert az informatika térnyerése töretlen. Nem elég a hardver, nem elég a képzett felhasználó, ha nincsenek jó szoftverek! Az internet minden képzeletet felülmúló térnyerése szintén programozók hadát igényli, mert az interaktív weboldalak nem mások, mint számítógépes programok. Ezért a leendő programozók képzését érdemes minél előbb elkezdeni, de legkésőbb a középiskolai képzés keretén belül feltétlenül.

Néhány éve lehetőség van arra, hogy középiskolában a hagyományos programozási nyelvek mellett (Pascal, Delphi, Visual Basic, C, C++) egy viszonylag új nyelv, a Java nyelv is oktatható. Egyre inkább tért hódít, az internet egyik alapnyelve, sok eszköz (pl. mosógép, mobiltelefon) vezérlőnyelve. Újdonsága abban rejlik, hogy – ellenben a korábbi eljárás-orientált programozási nyelvekkel – új szemléletet képvisel: a tisztán objektumorientált programozást. Ez a paradigma az újrafelhasználhatóság és a bezártság megvalósításával korszerű, biztonságos eszközt ad a programozók kezébe.

Szerzőtársammal, Szabó Lászlóval együtt – mint középiskolában tanító pedagógusok – úgy éreztük, hogy – élve a diplomamunka-kiírás adta lehetőséggel – csoportmunka keretében érdemes lenne foglalkozni ezzel a nyelvvel. Célul tűztük ki egy olyan oktatóprogram elkészítését, amelynek segítségével egy középiskolás diák is könnyen elsajátíthatja a Java nyelv alapjait, és megismerkedhet az objektum-orientált programozási módszerekkel is. Tartalmát tekintve szem előtt tartottuk, hogy az átadni kívánt ismeretek körének meg kell felelnie a NAT, az informatikai központi kerettanterv és az emelt szintű informatika érettségi vizsga programozási feladata által támasztott követelményeinek is.

Az elgondolást tett követte, és a témát két részre osztva a célkitűzést megvalósítottuk. Formailag éltünk az internet adta lehetőségekkel, és a webes feldolgozást választottuk. Az oktatóprogram első részében – amelyet a kollégám készített el – a Java nyelv alapelemeiről (adattípusok, adatszerkezetek, programozási és vezérlő szerkezetek) van szó, míg a második részben az I/O műveletek és a kivételkezelés ismertetése, valamint érettségi programozási feladatok megoldása mellett nagyrészt az objektum-orientált programozás alapjait dolgoztam fel.

Az elkészült tananyagban az alapismeretek mellett igyekeztünk bemutatni minél több életszerű, érdekes példát, valamint programozási trükköket, fogásokat is. Minden példaprogramot részletes magyarázattal láttunk el, külön kiemelve a lényeges részeket. Az ismertetett programokhoz mellékeltük azok forráskódját is. A fejezetek végén – az önellenőrzést szem előtt tartandó – azonnal kiértékelődő tesztkérdéseket és programozási feladatokat helyeztünk el. Természetesen a feladatok megoldása is megtalálható az oldalon. Tehát oktatóprogramunk felépítése a klasszikus új anyagot közlő-, gyakoroltató- és ellenőrző funkciókat követi. A webes elérhetőség pedig mindenki számára lehetővé teszi, hogy akár otthonról, egyéni ütemben is feldolgozhassa az anyagot.

A Java nyelvet alapvetően könnyű elsajátítani. Azonban irányítás nélkül könnyen elveszhetünk a számtalan utasítás, metódus és azok opciói között. A nyelv rendkívül kifinomult eszközökkel rendelkezik szinte minden programozási feladat elvégzésére, de ezen eszközök megtalálása és célszerű felhasználása nem könnyű. Ezzel a projekttel megpróbáltuk a Java nyelvet egy középiskolai tanuló számára is érthetővé tenni úgy, hogy – a nyelvben kellő rutint szerezve – minden tőle elvárható programozási feladatot könnyedén megoldjon.

2. A Java programozási nyelv

2.1. A Java programozási nyelv története

A kezdetek 1991-re nyúlnak vissza, a Sun Microsystem egy önálló fejlesztői csoportot hozott létre „Green Team” fantázia néven. Azt a feladatot kapták, hogy olyan háztartási eszközöket készítsenek, amelyek beépített mikrocsippel (mikrochip) rendelkeznek, és képesek az egymás közötti kommunikációra is. Ebben a csoportban dolgozott James Gosling, Patrick Naugthon és Mike Sheridan. Kezdetben a C++ programozási nyelvvel próbálkoztak, de Gosling felismerte, hogy ez a nyelv alkalmatlan erre a feladatra, és a csapat megtervezte a saját nyelvét, az Oak-ot (Tölgy). A név állítólag úgy jutott Gosling eszébe, hogy éppen egy könyvtárat hozott létre az új nyelvnek, és akkor az ablakon kinézve meglátott egy tölgyfát. A nyelvvel szemben támasztott alapvető elvárás az volt, hogy platformfüggetlen legyen.

A kezdeti sikerek ellenére sem kaptak nagyobb megrendeléseket, így 1993 tavaszára a projekt végveszélybe került. Ekkor döntöttek úgy a csoport vezetői, hogy meg kell próbálkozni az internettel.

A grafikus böngészők elterjedésével az internet egyre nagyobb népszerűségre tett szert. Az Oak technológia a platformfüggetlenségének köszönhetően tökéletesen megfelelt az internet számára, hiszen nem okozott gondot a hálózatba kapcsolt gépek inhomogenitása. 1995 elejére a csapat kifejlesztett egy grafikus böngészőt „Webrunner” néven. Később ez a böngésző lett az őse a HotJava böngészőnek, és itt jelent meg a Java kabalafigurája a Duke figura. Az Oak nevet a Sun-nak nem sikerült levédenie, mert kiderült, hogy már használják egy programozási nyelv elnevezéseként. A programnyelv kifejlesztésének ideje alatt „rengeteg” kávét fogyasztottak a fejlesztők, így a kávé hazájának emléket állítva kapta a technológia a Java nevet.

Az interneten lehetővé tették a források ingyenes letölthetőségét, és így bárki számára kipróbálhatóvá, tesztelhetővé vált. A letöltések száma rohamosan növekedett, 1995 novemberében már letölthető volt a nyelv béta verziójának forráskódja és fejlesztőkészlete. Néhány év alatt a Java nyelv a programozók egyik legfontosabb eszközévé vált. Köszönhető mindez annak a lelkes kis csapatnak, amely meglátta a platformfüggetlenség iránti igényt, és az ebben rejlő lehetőségeket.

2.2. A Java programozási nyelv bemutatása

[14] A Java nyelv fejlesztői hivatalos kiadványban (white paper) tették közre tervezési céljaikat és eredményeiket. Ez a kiadvány az alábbi szavakkal jellemzi a Java nyelvet:

- **Egyszerű** (simple): A Java nyelv a C++ leegyszerűsített változata. Leegyszerűsödött a szintaktika (eltűntek a mutatók, automatikus lett a memória felszabadítása). Java programot írni vagy olvasni sokkal egyszerűbb, mint egy C++ programot.
- **Objektum orientált** (object-oriented): Majdnem tiszta OO nyelv, egy kevésbé hibrid nyelv. Egy Java programot osztályok készítésével és újrafelhasználásával építünk össze.
- **Elosztott** (distributed): Egy Java program képes az internet bármely pontján található, URL-lel azonosított objektumot elérni és feldolgozni.
- **Robusztos** (robust): Másképpen hibatűrő, megbízható. A nyelv tervezői nagy gondot fordítottak a hibák korai felismerésére, még a fordítás idején történő kiszűrésére.
- **Biztonságos** (secure): A Java nyelvet elsősorban internetes környezetben való működésre tervezték, ezért biztonsági korlátokat kellett bevezetni, nehogy egy ilyen program kárt tehessen a másik felhasználó számítógépében.
- **Architektúra semleges** (architecture neutral): A fordítóprogram géptípustól független bájtkódot (.class fájl) generál, és ez a kód a különböző gépek processzorain futtatható, ha biztosítva van a megfelelő futtató környezet (Java Virtual Machine, JVM). Az adott gép a bájtkódot futás közben értelmezi a virtuális gép segítségével. A JVM által véglegesre fordított kód a natív kód, ez a kód már ténylegesen fut az adott gépen.
- **Hordozható** (portable): A nyelvnek nincsenek implementáció függő elemei, azaz nem fordulhat elő olyan eset, hogy egy nyelvi elem vagy osztály az egyik környezetben másképp legyen specifikálva, mint a másikban.
- **Interpretált** (interpreted): A célgépen futó natív kódot az értelmező hozza létre utasításonként a bájtkód (.class fájl) értelmezésével. Ha a célgépen installálnak egy Java értelmezőt, akkor bármilyen Java kódot értelmezhet.
- **Nagy teljesítményű** (high performance): Ez most még egy elérendő cél, a fejlesztők azon munkálkodnak, hogy a jelenlegi fordítás és futtatás során jelen lévő jelentős időigényt lerövidítsék.

- **Többszálú** (multithreaded): A többszálú programozás lényegében azt jelenti, hogy ugyanabban az időben több programrész fut egymással párhuzamosan, több szálon. Így jobban kihasználható a számítógép központi egysége.
- **Dinamikus** (dynamic): A Javát úgy tervezték, hogy könnyedén tovább lehessen fejleszteni. Az osztálykönyvtárak szabadon bővíthetők anélkül, hogy azok hatással lennének az őket használó kliensekre.

A Java tartalmaz olyan eszközöket, amelyek nem objektumok, ezért, „majdnem tiszta" OO programnyelvnek tekintjük. (pl. a primitív adattípusok is ilyen eszközök)

Nézzük meg, hogy mit is értünk az *objektumorientáltság* kifejezés alatt?

- Egy objektumorientált program együttműködő objektumok (`object`) összessége.
- A program alap-építőkövei az objektumok. Ezek olyan, a környezetüktől jól elkülöníthető, viszonylag független összetevők, amelyeknek saját viselkedésük, működésük és lehetőleg rejtett, belső állapotuk van. Egy objektumra a környezetben lévő egyéb objektumok hatnak, és ennek hatására saját állapotukat megváltoztathatják.
- Minden objektum valamilyen osztályba (`class`) tartozik. Az osztályok megfelelnek az absztrakt adattípusoknak, minden objektum valamely típus példánya, egyede (`instance`). Az osztályok definiálják az egyes objektumok állapotát leíró adatszerkezetet, és a rajtuk végezhető műveleteket, az úgy nevezett módszereket (`method`). Az egyes egyedek csak az állapotukat meghatározó adatszerkezet tényleges értékeiben különböznek egymástól, a módszerekkel definiált viselkedésük közös.
- Az egyes osztályokat az öröklődés hierarchiába rendezi. Az öröklődés az az eljárás, amely segítségével egy osztály felhasználhatja a hierarchiában felette álló osztályokban definiált állapotot (adatszerkezeteket) és viselkedést (módszereket). Így az ismétlődő elemeket elegendő egyszer, a hierarchia megfelelő szintjén definiálni.

A Java programok osztályokból épülnek fel. Java nyelven programot írni annyit jelent, mint elkészíteni az osztály programját, illetve felhasználni a rendelkezésre álló készletből (API[1]), a feladat megoldásához szükséges osztályt.

[1] Application Programming Interface

A Java nyelv az elmúlt néhány évben állandóan fejlődött, átalakult. Napjainkig az alábbi változatok jelentek meg:

1.0 Ez volt az első verziója a Java virtuális gépnek és az osztálykönyvtáraknak. (1996)

1.1 Itt jelent meg először a „belső osztály” fogalom, ami lehetővé teszi több osztály egymásba ágyazását. (1997)

1.2 Ez a verzió számottevő mérföldkő volt a nyelv evolúciójában. Hogy ezt kihangsúlyozza, a Sun hivatalosan ***Java 2***-nek nevezte el. (1998)

1.3 Csak néhány kisebb változtatást végeztek el rajta. (2000)

1.4 Ez ma a legelterjedtebb változat. (2002)

5 Belső számozás szerint 1.5, kódneve Tiger, újdonsága például a továbbfejlesztett ciklusmegoldások, az adattípusok automatikus objektummá alakítása (autoboxing), a generic-ek. (2004)

6 Belső számozás szerint 1.6.0, kódneve Mustang. Decemberben jelent meg a végleges változat kiterjesztett nyomkövetési és felügyeleti megoldásokkal, szkriptnyelvek integrációjával, grafikusfelület-tervezést támogató kiegészítésekkel. (2006)

7 Kódneve Dolphin (?)

3. A programozási nyelv tanításának követelményei

3.1. A Nemzeti Alaptanterv előírásai

A középiskolai informatika tantárgy oktatásának követelményeit legmagasabb szinten a Nemzeti Alaptanterv (NAT) rögzíti. Ebben az alapdokumentumban az információs társadalom kihívásainak való megfelelés kiemelt célként szerepel. A tanulónak *„…el kell sajátítania a megfelelő információszerzési, -feldolgozási, adattárolási, -szervezési és -átadási technikákat…”*. [1] Képessé kell válnia a valós világ modellezésére, amelyhez az informatika az egyik alapvető eszközt biztosítja. A fejlesztési feladatok között megtalálható az algoritmizálás és az adatmodellezés, mint elsajátítandó tudás.

3.2. Kerettantervi előírások

Következő szintet képvisel a kerettanterv, amely a NAT iránymutatása alapján konkretizálja a célokat és a követelményeket. A két fő (középiskolai) iskolatípus lehetőségeit összehasonlítva óriási különbséget tapasztalunk! Gyakorlatilag csak a szakközépiskolai szakmacsoportos alapozó oktatás (11.-12. évfolyamok) keretén belül van mód megfelelő óraszámban informatikát tanítani (heti 8 óra!), a gimnáziumi óraszám (heti 1-2 óra) már önmagában is tragikusan kevés, a programozásra pedig gyakorlatilag idő sem jut [3]. Ez utóbbi iskolatípusban csak fakultáció vagy szakkör keretén belül van lehetőség behatóbban foglalkozni a témával. Nézzük, mit ír elő a szakközépiskolai kerettanterv a programozással kapcsolatban [2]:

A számítógép-programozás célja, hogy a tanulók megismerjék

- az alapvető programozási elveket és tételeket
- a programozási elmélet alapjait
- a strukturált programozás fogásait
- az egyszerűbb programtervezési módszereket
- az objektum-orientált programozás alapjait

Fontosabb programozással kapcsolatos témakörök:

11. évfolyam:

- a programozás eszközei
    - programkód, programnyelv, kódolás
    - fejlesztői környezet, fordítás, szerkesztés
    - utasítások, adatok, függvények, eljárások és objektumok

- programozás-technikai alapismeretek, programtervezés
    - modellezés
    - algoritmuskészítés
    - kódolás, tesztelés
- értékadás, változók, konstansok
- elágazások és iterációk szervezése, logikai feltételvizsgálat
- alapfüggvények, függvényhívások, paraméterátadás
- egyszerű és összetett adatszerkezetek
- rendezési algoritmusok
- alapvető I/O műveletek (konzol, fájl)

12. évfolyam:

- az objektum-orientált programozás elve és alapjai
- osztály és objektum, konstruktor
- objektumok felépítése és tulajdonságai (adattag, metódus)
- láthatóság, hatáskör
- objektumok hierarchiája, öröklődés
- hibakezelési eljárások

Ezeknek a tartalmaknak meg kell jelenniük az iskolák helyi tantervében, majd az erre épülő tanmenetekben is.

3.3. Az emelt szintű informatika érettségi vizsga követelményei

A középfokú oktatás lezárását adó érettségi vizsgának – választhatóan – az informatika tantárgy is része lehet. A programozás azonban csak az emelt szintű érettségi vizsgán szerepel, ahol is pontszámban a gyakorlati feladatsor több mint egyharmad részét adja. Az érettségi vizsga vizsgaszabályzata – az informatika érettségi vizsga részeként – szintén rögzíti a programozási ismeretek követelményrendszerét [4]:

- egy programozási nyelv részbeni (specialitások nélküli) ismerete
- algoritmizálás, adatmodellezés
- elemi és összetett adatok
- adatstruktúrák, fájlszervezés
- elemi algoritmusok (rendezések, rekurzió)
- programkészítés (tervezés, kódolás, tesztelés, hibakeresés)

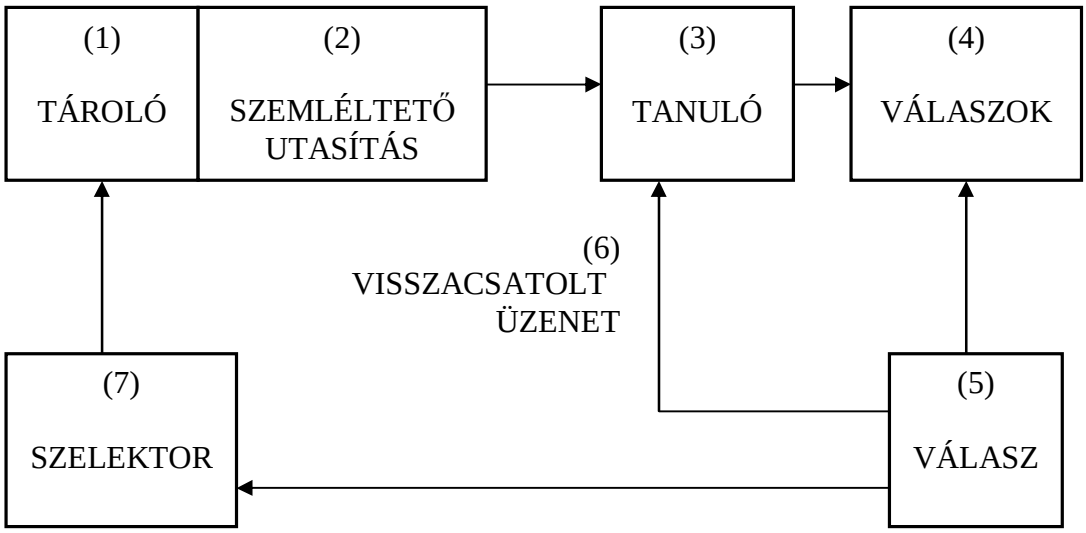
Mindezek az előírások egyértelműen meghatározzák a programozási nyelvek középiskolai oktatásával kapcsolatos célokat és követelményeket, amelyeket az oktatóprogramunk kifejlesztésénél mindig szem előtt tartottunk.

4. A számítógép jelentősége és felhasználása a tanítási-tanulási folyamatban

1981-ben Ershov professzor az alábbi megállapítást tette: „***a számítógép-programozás néhány éven belül olyan lesz, mint egy második ábécé, s ugyanolyan nélkülözhetetlen lesz, hogy tudjunk számítógéppel dolgozni, mint amilyen ma az, hogy tudjunk olvasni és írni.***”

4.1. A számítógépes oktatás elmélete

[15] A System Development Corporation az alábbi számítógépes oktatási rendszert dolgozta ki:



A rendszer egyszerűsége lehetővé teszi az elméleti meggondolások bemutatását.

A tananyagot (1) bemutatjuk a tanulónak (3) a tananyaghoz fűződő kérdéssel vagy problémával (2) együtt. A válasz (4) értékelése (5) és automatizálása „közölt” visszacsatolt üzenet (6) jelentik a következő lépéseket. Ezután új anyagrész következik (7). Az utolsó választól függően az új anyagrész vagy korrigáló anyagot tartalmaz, vagy bonyolultabb kérdéseket tárgyal. Az egész ciklust addig folytatjuk, amíg a leckét be nem fejeztük.

Erre a rendszerre épülve fejlesztették ki a Computer-based Laboratory for Automated School Systems-t (számítógépes laboratórium automatizált oktatási rendszerek számára) és **CLASS**-nak nevezték.

Ez a rendszer egyidejűleg 20 tanuló számára rendelkezett perifériával. A húszas létszámot elsősorban a helyhiány és a perifériák száma szabta meg, nem pedig a számítógép kapacitása.

A fejlődés következő lépése, a terminálhálózattal összekapcsolt számítógépes oktatórendszer volt.

Az oktatórendszerek fejlődése napjainkban is lankadatlanul folyik tovább. Az internet megjelenése, térhódítása újabb lendületet adott a fejlesztéseknek (Megjelentek a web alapú oktatási anyagok, e-learning rendszerek stb.).

4.2. A számítógépek a tanítási-tanulási folyamatban

A számítógépeknek a tanítási-tanulási folyamatban való felhasználási módja alapján az alábbi típusok alakultak ki:

1. **CAI** (Computer Assisted Instruction) számítógéppel segített oktatás – a tanuló közvetlen kapcsolatban áll a számítógéppel. (1. ábra)
2. **CMI** (Computer Managed Instuction) számítógéppel szervezett oktatás. Itt a számítógép általában a tanár közvetítésével szervezi, irányítja a tanulást.
3. **CBI** (Computer Based Instuctions) számítógépre alapozott oktatás. Az előző két módszerre alapozva építi fel a tananyagokat.

A CAI programtípusok főbb csoportjai:

a) **Begyakorló** (drill and practike) **programok**

Ezek a programok főként matematikai, fizikai, technikai és verbális fogalmak begyakorlását szolgálják. A tanuló addig kapja a begyakorlandó fogalmakkal, műveletekkel stb. kapcsolatos feladatokat a számítógéptől, amíg a kívánt begyakorlási szintet el nem éri.

b) **Konzultációs** (tutorial) **programok**

A konzultációs programok diagnosztizáló teszteket tartalmaznak. A tanuló a program feldolgozása során igen sok irányba mehet el, a kérdésekre adott válaszoktól függően.

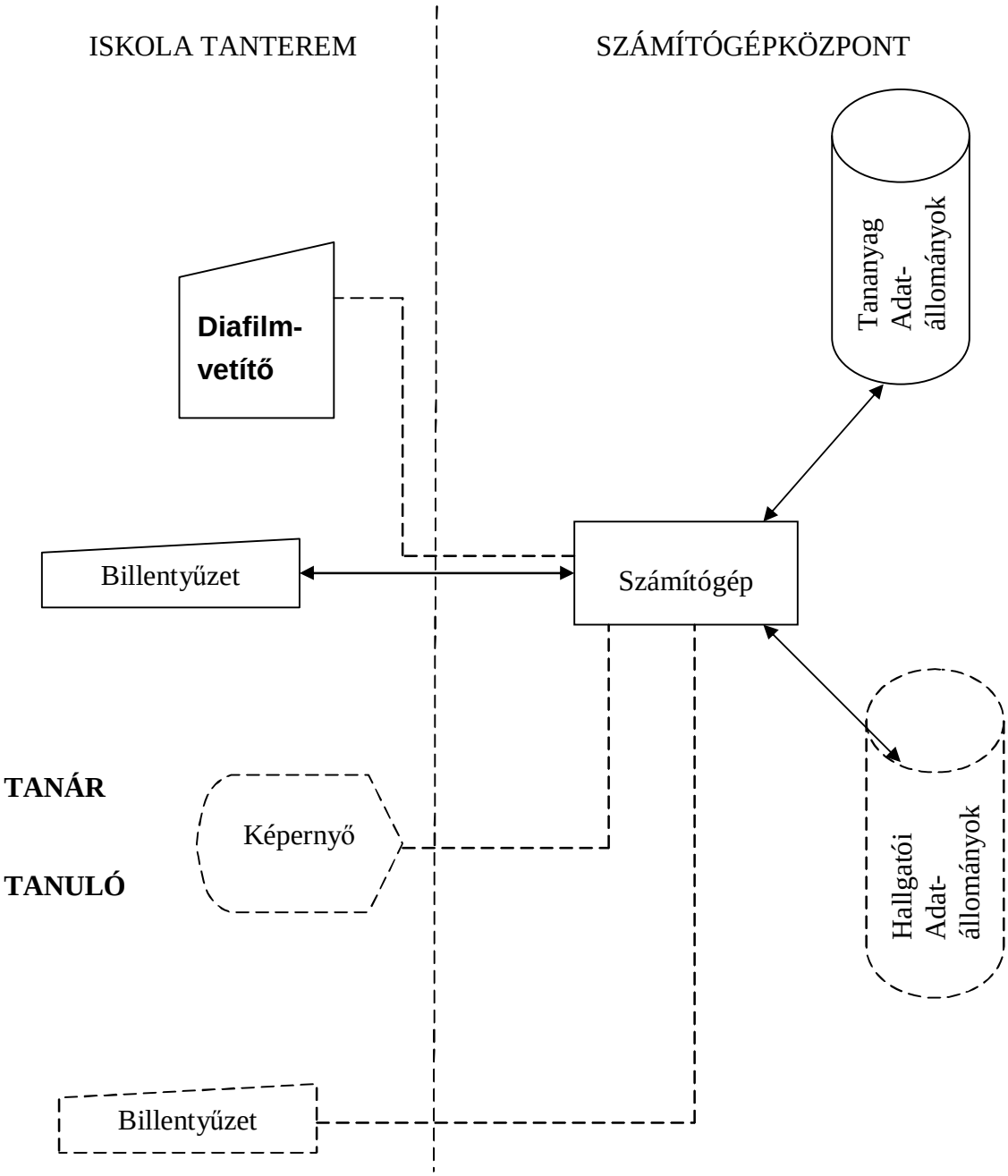
c) **Problémamegoldási** (problem solving) **programok**

Amíg az előző két programtípusnál a feldolgozás algoritmusa előre meghatározott, addig itt a tanuló feladata az, hogy adott probléma megoldásához algoritmust dolgozzon ki, és ezt tesztelje. A feladat megoldása során a számítógépnek kérdéseket tehet fel, illetve számításokat végeztethet vele.

d) **Szimulációs** (simulation) **programok**

Ezek a programok abban segítik a tanulót, hogy egy folyamat, mechanizmus stb. összetevőit és ezek különböző szituációkban működő hatásait, azaz modelljét felfedezze. A tanuló által bevitt adatok alapján a számítógép szimulálja a folyamatot és bemutatja, hogy mi lenne az

eredménye. A tanuló az eredmények ismeretében újra választhat paramétereket, és újra megismerheti döntései következményeit.



1. ábra

4.3. Az oktatóprogramok hatása az oktató-nevelő munkára

Napjainkban egyre-másra árasztanak el bennünket a különböző oktató-bemutató (multimédiás) programok. Egyes számítógépes szoftverek egyre nagyobb hatással vannak az oktatás folyamatára. A számítógépek egyre inkább átveszik a hagyományos oktatástechnikai eszközök szerepét is, így nem hagyhatjuk kiaknázatlanul a számítógép nyújtotta lehetőségeket. Bár csodát sem várhatunk feltétlenül a számítógéptől (oktatóprogramoktól), hiszen a pedagógus emberi jelenlétét, az oktatásban betöltött szerepét soha nem pótolhatja, valamint nem nélkülözhető a tanuló aktív közreműködése sem a tanítási-tanulási folyamatban.

Az oktatóprogramok használatával a tanárok könnyebben és szemléletesebben tudják a tananyagot átadni a tanulóknak. A diákok amellett, hogy elsajátítják a számítógép kezelését aktívan részt is vesznek a tanítási-tanulási folyamatban. A programok segítségével egyszerre több érzékszervre hatva történik az információk átadása, így válik hatékonyabbá az ismeretek átadása-elsajátítása.

A számítógép alkalmazásának másik óriási jelentősége abban áll, hogy az ismeretek átadása során alkalmazkodik a diákok egyéni tempójához. Lehetőséget biztosít a feladatmegoldások során az azonnali értékelésre, megmutatja a hibákat és útmutatást ad a hibák kijavításhoz.

Az internet térhódításával a különböző oktatóprogramok, segédanyagok, videók, stb. otthonról, a lakásból ki sem mozdulva is elérhetővé váltak. Mindenki az érdeklődési körének megfelelő információ birtokába juthat „szempillantás” alatt. A világháló nyújtotta lehetőségeket kihasználva a diákok kicserélhetik egymással a tapasztalataikat, segítséget kérhetnek egymástól, fórumokat hozhatnak létre stb. Az internet tehát egy kifogyhatatlan információs „kincsesbánya”, csak tudni kell a lehetőségeket használni.

5. Az oktatóprogram bemutatása

5.1. Fejlesztőeszközök

Az oktatóprogram és a szakdolgozat elkészítéséhez az alábbi szoftvereket használtuk:

- operációs rendszer: **Microsoft Windows XP Professional SP3 HU**
  (licenc: Tisztaszoftver Program)

- weblapszerkesztő: **Microsoft Office FrontPage 2003 SP3 HU**
  (licenc: Tisztaszoftver Program)

- szövegszerkesztő: **Microsoft Office Word 2003 SP3 HU**
  (licenc: Tisztaszoftver Program)

- webböngésző: **Mozilla Firefox v3.6.3 HU**
  (licenc: Mozilla Public License, GNU)

- Java futattórendszer: **Java SE Development Kit v6u19**
  (licenc: GPL v2 + Classpath exception)

- Java fejlesztő: **NetBeans IDE v6.8**
  (licenc: CDDL és GNU General Public License)

- Tesztgenerátor: **eXe XHTML editor v1.04.1**
  (licenc: GNU General Public License)

5.2. Rendszerterv

Az oktatóprogram készítése során a bevezetőben ismertetett megfontolások vezettek bennünket. Egy olyan segédeszközt kívánunk a középiskolás diákok kezébe adni, aminek a használatával elsajátíthatják a Java programok készítésének a menetét, valamint képesek lesznek az emelt szintű érettségin a programozási feladat megoldására is.

A követelmények (3. fejezet) áttanulmányozása után, valamint a korábbi években szerzett szaktanári tapasztalatainkat (Pascal programozási nyelv tanítása) figyelembe véve a tananyagot 15 hétre bontottuk, heti 4 tanítási óra keretben. Úgy látjuk, hogy ez a megfelelő forma arra, hogy átadjuk az ismereteket, valamint készségszintre hozzuk az alapvető programozástechnikai elemek elvégzését.

A tanítási egységeket a következők szerint alakítottuk ki:

1. hét A kezdetek
2. hét Nyelvi alapelemek
3. hét Nyelvi eszközök
4. hét Numerikus- és egyéb adattípusok

5. hét Karakteres adattípusok
6. hét Elágaztató utasítások
7. hét Ciklusszervező utasítások
8. hét Tömbök
9. hét Input-output műveletek
10. hét Érettségi feladatok megoldása
11. hét Az objektum-orientált paradigma
12. hét Osztály és csomag
13. hét Öröklődés
14. hét Kivételkezelés
15. hét Objektum-orientált feladat megoldása

Minden egyes tanítási egység a következő módon épül fel:

1. Tartalom
2. Kidolgozás
3. Tesztkérdések
4. Feladatok

Tartalom: A fejezetek elején található, útmutatót nyújt a diákok számára az adott tanítási egység felépítéséről, tananyag tartalmáról.

Minta:

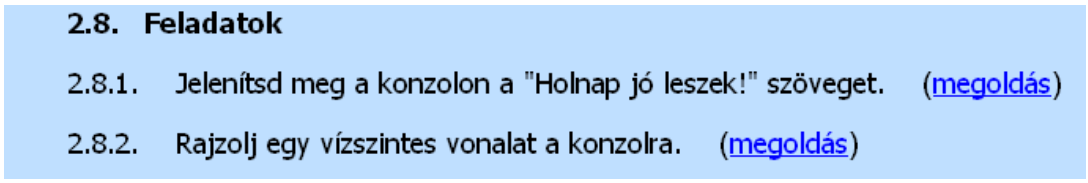
Kidolgozás: Az adott téma részletes, példaprogramokkal szemléltetett bemutatása található itt.

Tesztkérdések: A tananyag végén ellenőrző kérdések segítségével próbáljuk az elsajátított ismereteket feleleveníteni. A tanulók válaszát az oktatóprogram azonnal értékeli.

Minta:

Feladatok: Az itt található feladatok elkészítésével a tanulók, elmélyítik a lecke ismereteit, begyakorolják az adott egységhez tartozó programozói fogásokat.

Minta:



A feladatok megoldása is megtalálható a tananyagban, így a diákok önállóan képesek a munkájukat leellenőrizni.

A tananyag elérhető az interneten, így biztosítjuk azt, hogy esetleges lemaradás esetén a diákok pótolni tudják a kimaradt ismereteket, valamint így a saját egyéni tempójukhoz igazodva tudnak haladni a tananyag feldolgozásában.

A fent említett szempontokat követve készítettük el az oktatóprogramot.

Az oktatóprogram egy webhely, amely elérhető a http://java2.uw.hu címen. Tartalmazza a tananyagot, teszteket, feladatokat a feladatok megoldását, valamint kiegészítettük egy fogalomtárral is.

A program nyitóképernyője:

5.3. Az oktatóprogram részletes bemutatása

Az alábbiakban bemutatjuk a fejezetekre bontott oktatóprogramot, amelynél egy-egy fejezet – megfelelő óraszám esetén (kb. 4-6 óra) – 1 hét alatt feldolgozandó ismeretanyagot tartalmaz. A szakdolgozat szűkre szabott terjedelme miatt az egyes fejezetek végén található tesztkérdések, gyakorló feladatok és azok megoldásai nem kerültek a dolgozatba, de a weboldalon természetesen elérhetők.

5.3.1. A KEZDETEK

5.3.1.1. Általában a Javáról

[16] A Java egy objektum-orientált programozási nyelv, amelyet a C++ programnyelvből fejlesztettek ki. Elsődleges célja az volt, hogy legyen egy olyan programozási nyelv, amelyet az interneten keresztül is használhatunk, biztonságos és lehetőleg platformfüggetlen.

A platformfüggetlenség azt jelenti, hogy nemcsak UNIX-on, PC-n, ... stb, lehet futtatni az adott programot, hanem bármilyen géptípuson. Ez annyiban sántít csupán, hogy „bármilyen géptípuson” futtatható egy Java program, amely rendelkezik Java futtatóval, az ún. Java Virtual Machine-nal (JVM), azaz egy virtuális, Java programot futtatni képes környezettel.

Az interneten keresztül történő használat azt jelenti, hogy a Web-böngészőben megjelenő Weblapjainkról indulhatnak ún. Java appletek, amelyek azonos módon fognak megjelenni és futni, minden platformon futó Web-böngészőben. A Java appletek a Java programoktól abban különböznek, hogy az appletnek korlátozottak a képességei. Például nem írhatja, olvashatja a helyi fájljainkat, míg egy Java programból ugyanúgy beolvashatom, írhatom mintha azt egy másik programozási nyelven tettem volna meg.

A Java programozási nyelv jellemzői:

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

A Java fordítóprogramos programnyelv, de a fordítás folyamata az alábbiak alapján történik:

- Először a forrásprogramot a Java-fordító (compiler) egy közbülső nyelvre fordítja > Java bájtkódot kapunk eredményül (ez a bájtkód hordozható)
- Ezt a kódot értelmezi és futtatja a JVM. A JVM tehát egy interpreternek tekinthető.

A Java programok készítéséhez az alábbiakra lesz szükségünk:

- **Java integrált fejlesztőkörnyezet** (Java SE + NetBeans).
  Letöltés: http://java.sun.com/javase/downloads/index.jsp
  Telepítés: A telepítés a Windowsban megszokott egyszerűséggel történik, általában elegendő a *Next* gombra kattintani.
- **Java dokumentációk**
  A dokumentációk letölthetők a http://java.sun.com oldalról.

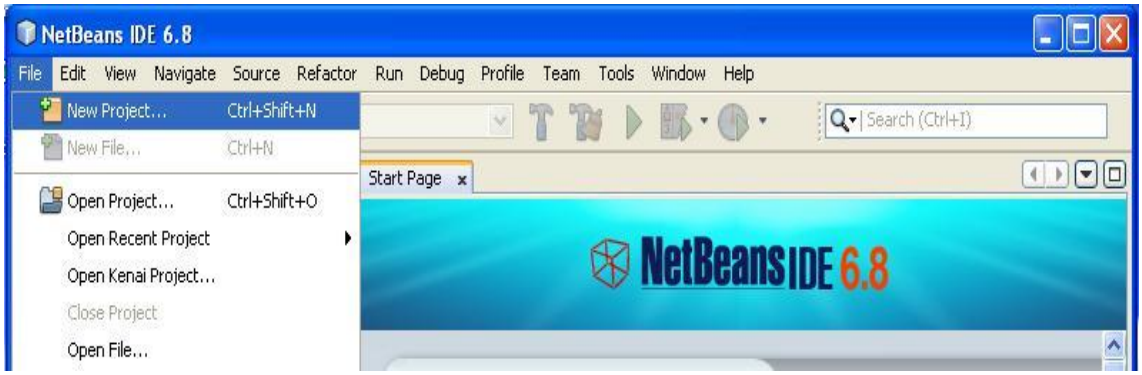
5.3.1.2. Ismerkedés a fejlesztőkörnyezettel (NetBeans)

A programok készítésénél a Sun saját fejlesztőkörnyezetét a NetBeanst fogjuk használni. Ez egy integrált fejlesztőkörnyezet (Integrated Development Environment, IDE). A NetBeans lehetővé teszi a programozók számára a programok írását, fordítását, tesztelését, valamint hibakeresés végezését, majd a programok profilozását és telepítését is.

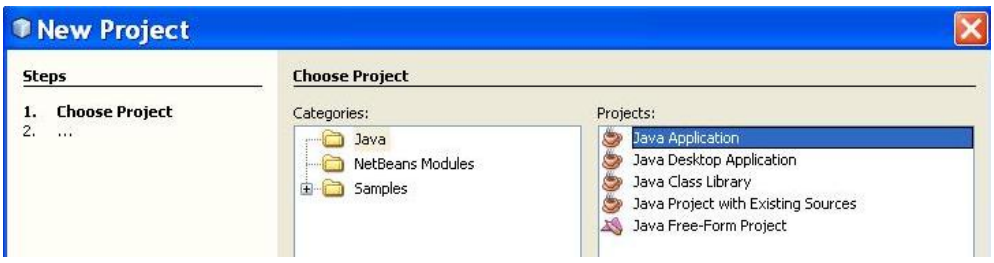
Ismerkedjünk meg egy egyszerű projekt készítésének a folyamatával! (Hogyan lehet Java alkalmazást készíteni?)

5.3.1.2.1. Projekt létrehozása

A NetBeans indítása után `File > New Project`

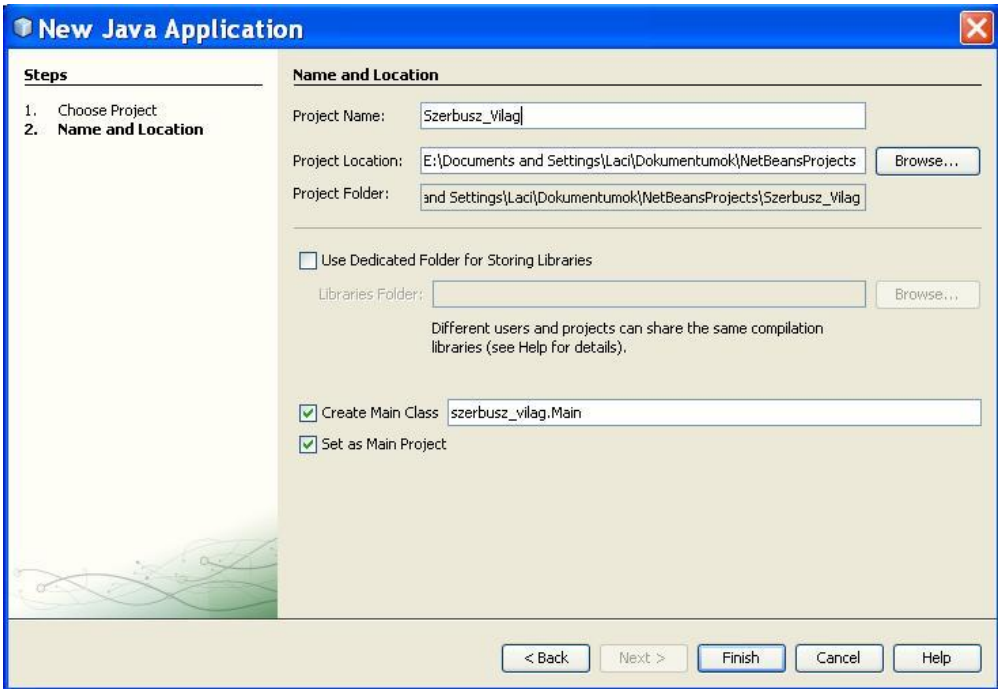


A Java kategórián belül a Java Application projektet válasszuk ki.



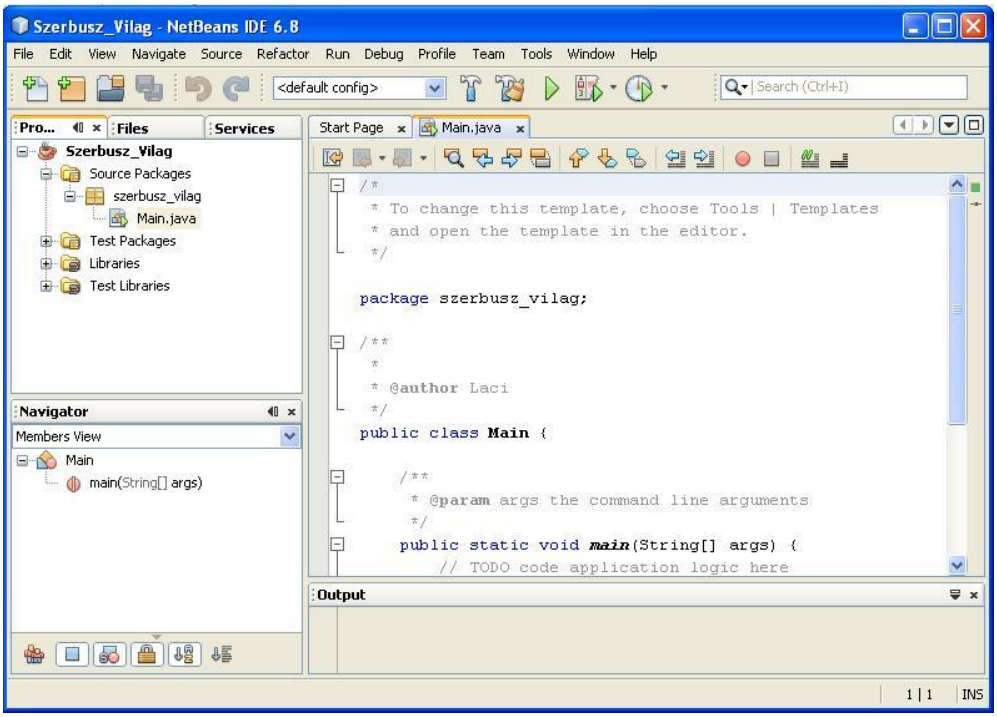
A következő lépésben a projekt nevét kell megadnunk.

Kapcsoljuk be a *Create Main Class*, valamint a *Set as Main Project* opciót.



A `Finish` gomb lenyomásával indul a varázsló és elkészíti a projektünket.

A varázsló futtatásának eredménye:



Az előző lépéseket végrehajtva elkészítettünk egy Szerbusz_Világ nevű projektet, amely tartalmaz egy szerbusz_vilag nevű csomagot, valamint egy `Main` nevű osztályt. Ezekről a későbbiek során fogunk tanulni.

5.3.1.3. Az első program elkészítése

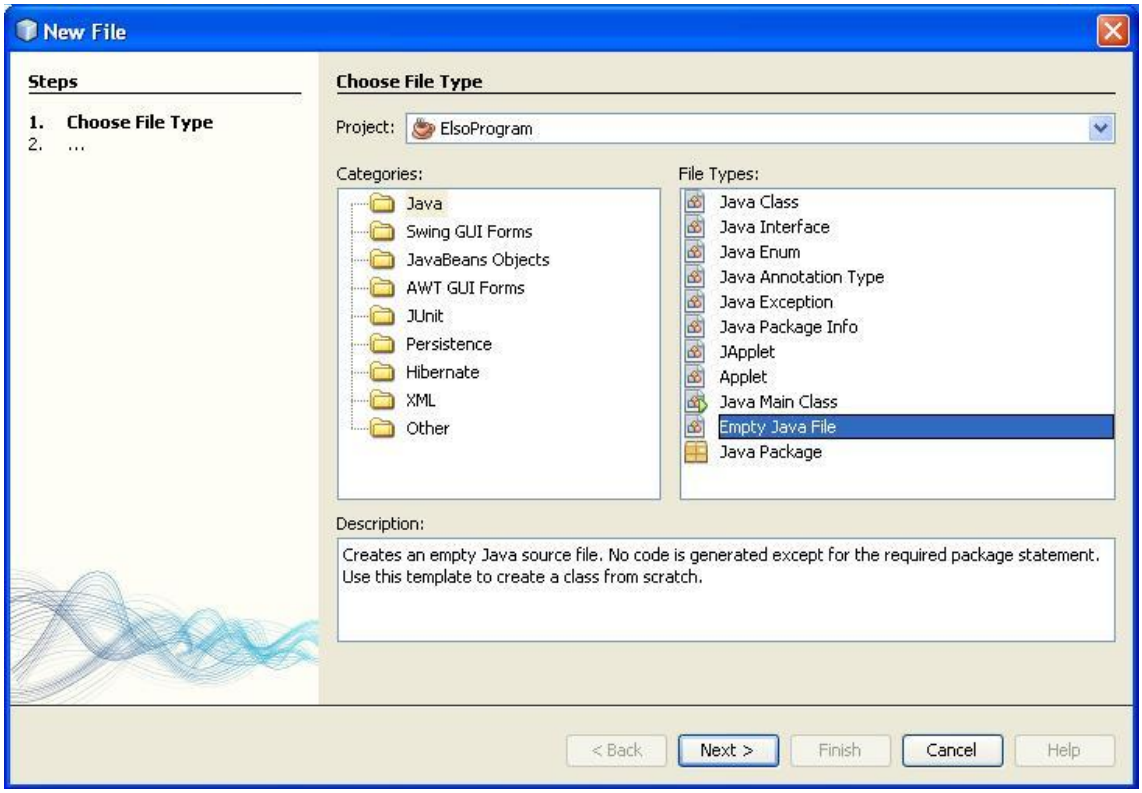
Feladat: Üdvözlő szöveg kiírása a képernyőre. („Szerbusz kedves Idegen!”)

Fontos tudnunk, hogy a Java nyelvben megkülönböztetjük a kis- és nagybetűket. Ezenkívül a forrásprogram fájlneve és a benne definiált `public` típusú osztály neve azonos kell legyen, még a betűállást tekintve is.

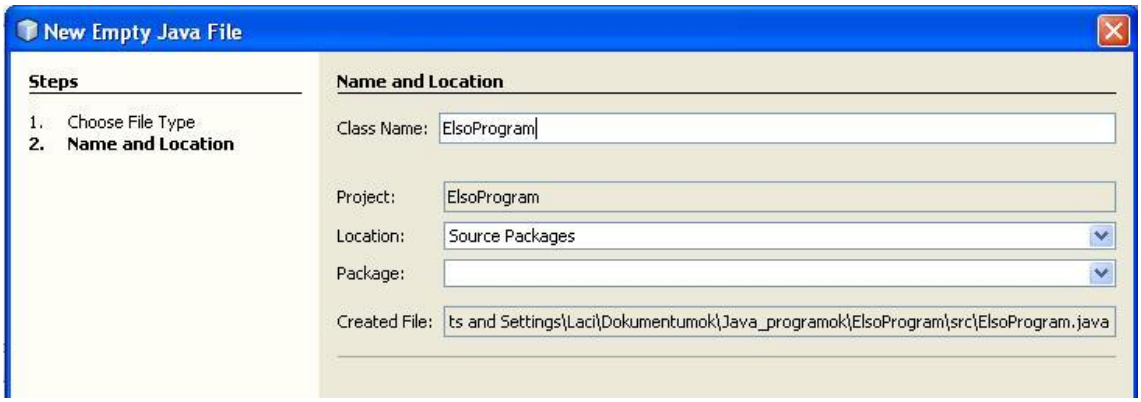
Készítsünk egy új projektet ElsoProgram néven.

A készítés menete az előző részben leírtak alapján történik, azzal az eltéréssel, hogy most ne hagyjuk bekapcsolva a Create Main Class, valamint a Set as Main Project opciót.

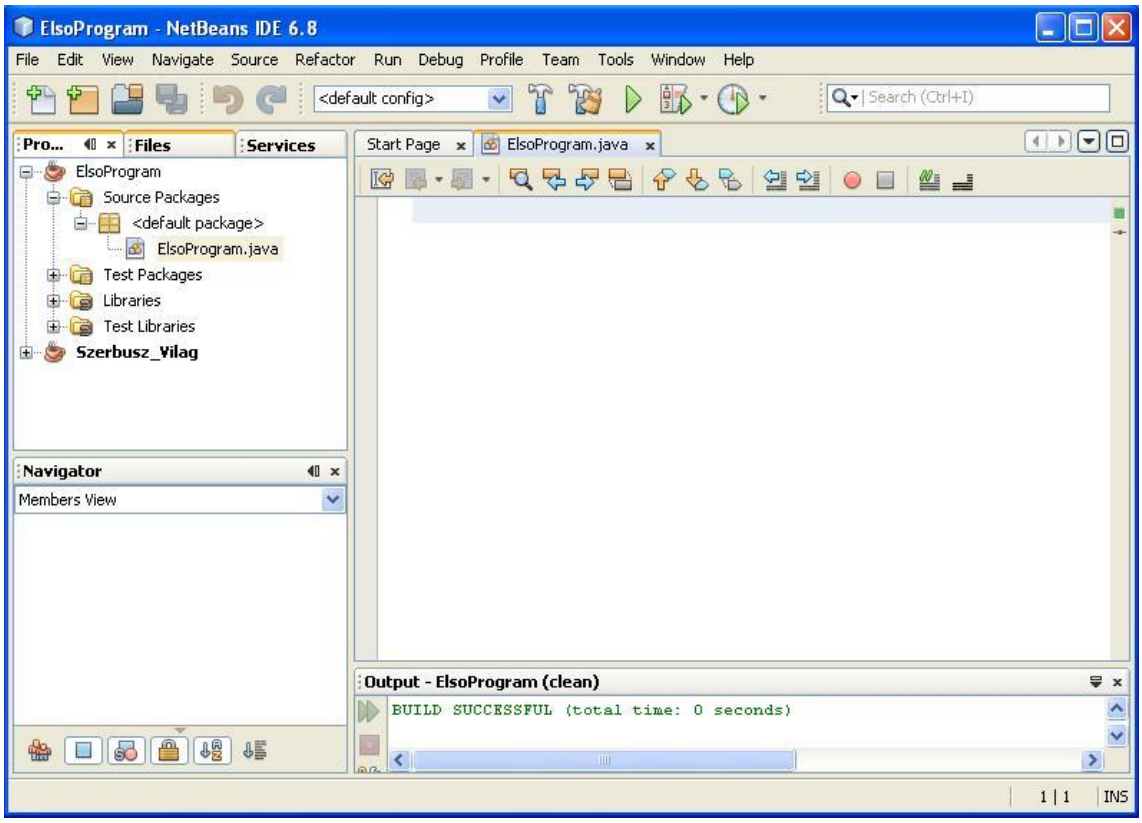
Szúrjunk be a projektbe egy üres Java fájlt.



Az osztály neve egyezzen meg a projekt nevével.



Elkészült az ElsoProgram projektünk, egy ugyanolyan nevű osztállyal.



Gépeljük be a forrásprogramot a fejlesztőrendszerbe!

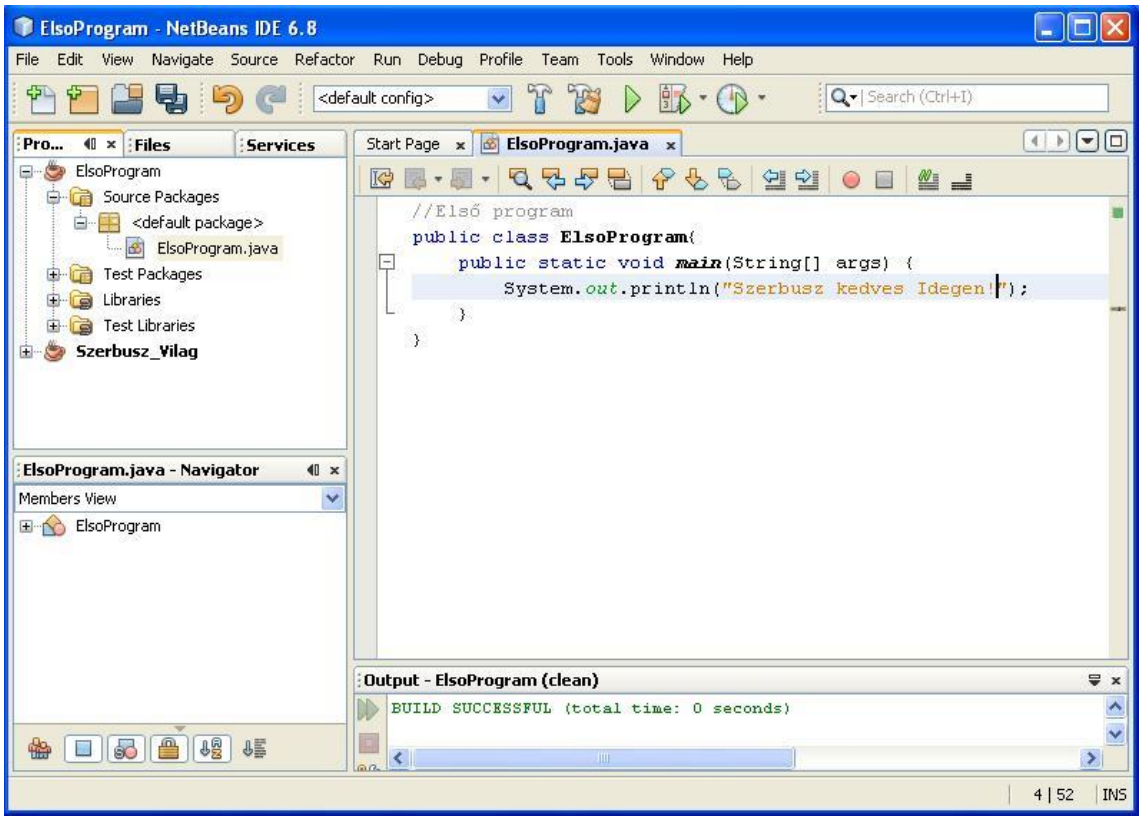
```
//Első program
public class ElsoProgram{
    public static void main(String[] args) {
        System.out.println("Szerbusz kedves Idegen!");
    }
}
```

Különösen figyeljünk oda minden zárójelre, pontosvesszőre, mert egy jel hiánya is szintaktikai hibát eredményez, és nem fogja a fordító elfogadni. A beíráskor a példa szerinti

tagolásban vigyük be a programot. Ez azért fontos, mert így könnyen áttekinthető lesz a programunk, és a hibákat gyorsan felfedezhetjük.

A parancsok begépelésénél használhatunk rövidítéseket, általában a parancs első két karakterének a begépelése után nyomjuk meg a *TAB* billentyűt. pl.

pu + <*TAB*>	public
cl + <*TAB*>	class
psvm + <*TAB*>	public static void main(String[] args{})
sout + <*TAB*>	System.out.println("");



Vizsgáljuk meg soronként a program felépítését!

1. A // kezdetű sorok megjegyzések, és nem kerülnek feldolgozásra. Itt a program érthetőségét növelő egyéni megjegyzéseket helyezhetünk el.
2. Ebben a sorban adjuk meg az osztály definícióját, amelyben meghatározzuk a láthatóságát (`public`), a nevét (ElsoProgram) és egyebeket, melyekről a későbbiekben lesz szó. Ezt követi egy **blokk**, amelyet kapcsos zárójelek közé zárunk.
3. Programunk egy metódust tartalmaz, amelynek neve „`main`”, vagyis „főmetódus”. Az íves zárójelben egy karakterlánc-tömböt definiálunk „`String args[]`”, amelyben

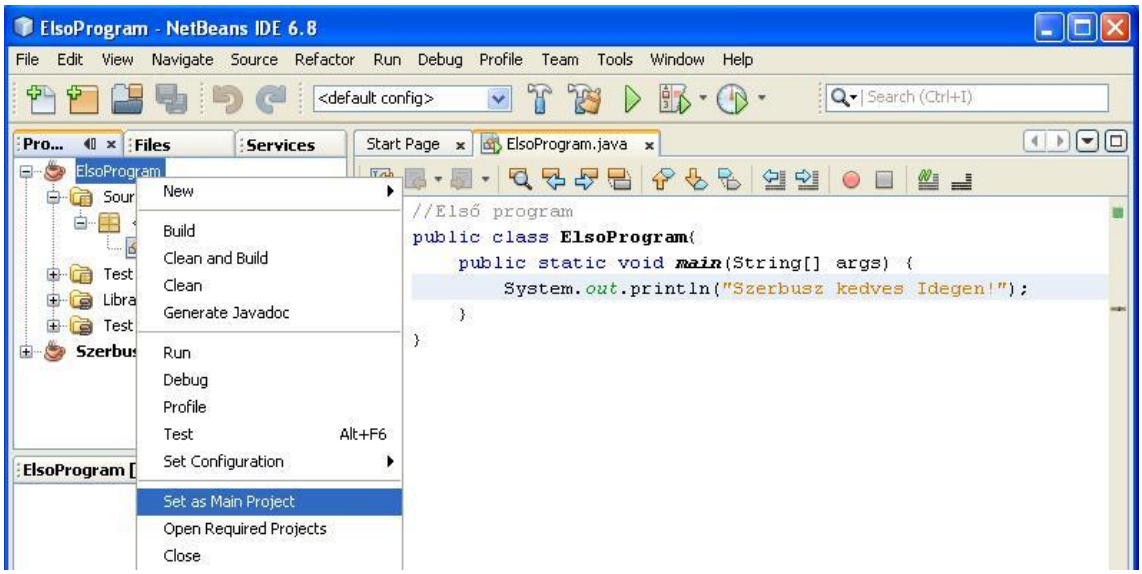
a parancssori paramétereket kapjuk meg. A `main` metódus tartalmát is kapcsos zárójelek közé zárjuk.

4. A „`System.out.println`” az íves zárójelben megadott karakterláncot írja ki a képernyőre.

Fordítás, hibakeresés

Ha sikeresen begépeltük a forrásprogramunkat, mentsük el, és fordítsuk le.

Figyelem! Előtte állítsuk be a következőt: Jobb egérgombbal kattintsunk a Projekt ablakban található ElsoProgram projektünkre. A megjelenő menükből válasszuk ki a `Set as Main` Project opciót!



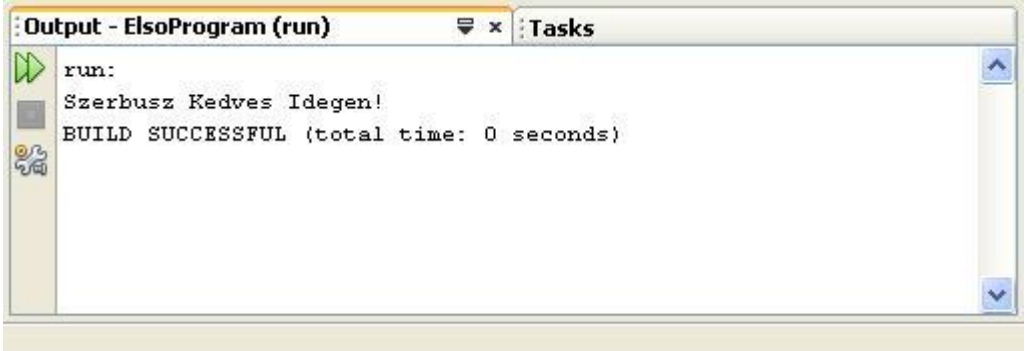
A fordítást a `Run` menüben, vagy az eszköztáron érhetjük el  , ahol kétféle lehetőség közül választhatunk. (Build Main Project, Clean and Build Main Project)

Az Output ablakban kapunk tájékoztatást a fordítás eredményéről, és az előforduló hibákról.

Futtatás

A sikeresen lefordított programot futtathatjuk. 

A futás eredményét az Output ablakban tekinthetjük meg.



5.3.2. NYELVI ALAPELEMEK

5.3.2.1. Karakterkészlet

A program forrásszövegének legkisebb alkotórészei a karakterek.

Az UNICODE 16 bites kódtáblára épül, ahol betű minden nyelv betűje lehet. Tartalmazza az összes nemzetközi abc-t. Megkülönbözteti a kis- és nagybetűt.

Kategóriák:

- betűk (minden nyelv betűje)
- számjegyek (0-9 decimális számjegy)
- egyéb karakterek (műveleti jelek, elhatároló jelek, írásjelek, speciális karakterek)

5.3.2.2. Szimbolikus nevek

5.3.2.2.1. Azonosító

Olyan karaktersorozat, amely betűvel kezdődik, és betűvel vagy számjeggyel folytatódhat. Arra használjuk, hogy a programozói eszközeinket megnevezzük vele, és ezután bárhol a program szövegében hivatkozhassunk rá. Betűnként nem csak a nemzeti karaktereket, hanem az _ és $ jelet is használhatjuk.

Azonosító képzési szabályok:

- Az azonosító bármilyen hosszú Javában használható unikód karaktersorozat lehet.
- Egy azonosító nem kezdődhet számjeggyel, nem lehet kulcsszó, és nem lehet a nyelv egy előre definiált konstansa (true, false vagy null).
- Két azonosító csak akkor egyezik meg, ha unikód karaktereik rendre megegyeznek.

Példák az azonosítóra:

Szabályos Java azonosítók:	Nem azonosítók:
x	x+y
ár	x*y+3
diak_kod	12 öt

Néhány gyakorlati hasznos tanács az azonosítók készítéséhez:

- Próbáljunk meg jól olvasható, beszédes azonosítókat választani. (minSebesseg, karakterekSzama)
- Az osztályok azonosítóját nagybetűvel kezdjük. (Személy, Alakzat)
- A változók és metódusok azonosítóját kisbetűvel kezdjük. (sebesség, darab)
- A konstansokat csupa nagybetűvel írjuk. (ALAPAR, NYUGDIJKORHATAR)
- Az azonosítókat úgy tagoljuk, hogy az összetett szavak kezdőbetűit nagybetűvel, a többit kisbetűvel írjuk. (maxSebessegHatar)

5.3.2.2.2. Kulcsszó

Olyan karaktersorozat, amelynek az adott nyelv tulajdonít jelentést, és ez a jelentés a programozó által nem megváltoztatható.

A Java kulcsszavai a következők:

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

A `const` és a `goto` szintén foglalt szó, de nincs implementálva, egyik sem használatos.

5.3.2.3. Címke

A végrehajtható utasítások megjelölésére szolgál, bármely végrehajtható utasítás megcímkézhető. Azért alkalmazzuk, hogy a program egy másik pontjáról hivatkozni tudjunk rá.

A címke egy azonosító, az utasítás előtt áll kettősponttal elválasztva.

Szintaktika:

```
cimke: {     }, vagy cimke: utasítás
```

A címkét a `break` vagy a `continue` paranccsal együtt használjuk úgy, hogy megadjuk a parancs után azt, hogy melyik egységre vonatkozik.

Példa:

```
public class Cimke {
    public static void main(String[] args) {
        kulso://külső ciklus cimkéje
        for (int i = 5; i < 10; i++)//ciklus 5-től 9-ig
        {
            belso://belső ciklus cimkéje
            for (int j = 10; j > 0; j--) {//ciklus 10-től visszafelé 1-ig
                if (i == j) {
                    System.out.println("i=" + i);
                    break kulso;
                }
            }
        }
    }
}
```

5.3.2.4. Megjegyzések

A megjegyzés olyan programozási eszköz, amely arra szolgál, hogy a program írója a kód olvasójának valamilyen információt szolgáltasson.

A megjegyzést a forrásszövegben háromféle módon helyezhetjük el:

- //-től sorvégig megjegyzés.
- /* */ zárójelek között tetszőleges hosszan.
- /** */ dokumentációs megjegyzés. A fejlesztőkörnyezet segítségével automatikusan lehet generálni hypertext (HTML) dokumentációt, ami felhasználja a dokumentációs megjegyzéseket is. (Run > Generate Javadoc (Projekt név))

Példa a megjegyzésekre:

```
/*
 * Itt egy többsoros megjegyzés található
 * A program kiírja az i változó értékét
 */
public class Megjegyzes{//Itt definiáljuk a Megjegyzes nevű osztályt
    public static void main(String[] args) {
        int i=1;//i változó deklarálása és kezdőértékének megadása
        System.out.println("i="+i);//kiírjuk az i változó értékét
    }
}
```

5.3.2.5. Literálok

A literál olyan programozási eszköz, amelynek segítségével fix, explicit értékek építhetők be a program szövegébe. Két komponense van: típus és érték.

Fajtái:

- **egész**: egy pozitív vagy negatív egész szám, vagy nulla.(pl. +200, -3, 1256987)
- **valós**: egy tizedesekkel leírható szám. Két formában adható meg: tizedes forma és lebegőpontos forma (pl. 6.32, 100.0, 0.0006, 6.3E2, 5E-4)
- **logikai**: két logikai konstans létezik, a true (igaz) és a false (hamis).
- **karakter**: egy unicode karakter (szimpla aposztrófok közé tesszük). (pl. 'a', 'B')
- **szöveg**: akármilyen hosszú, unicode karakterekből álló sorozat (idézőjelek közé tesszük). (pl. „Szia”)
- **null**

5.3.2.6. Változó

A programok adatokon végeznek különféle manipulációkat, ezek az adatok a program futása alatt változtatják értéküket. A kezdeti adatokat a program eltárolja a változókba, az értékek az algoritmusnak megfelelően módosulnak a program futása folyamán. Az algoritmus végén a változó a végeredmény értékével rendelkezik. A változó olyan adatelem, amely azonosítóval van ellátva. Az egyik legfontosabb programozási eszközünk.

A változókat használat előtt deklarálni kell. Ez azt jelenti, hogy meg kell adni a típusát, nevét, esetleg értékét is. Amikor egy változónak kezdeti értéket adunk, akkor a változót inicializáljuk.

```
pl. byte a; int x = 0;
```

Egy változót a program tetszőleges részén deklarálhatunk.

A változó érvényességi tartománya a programnak az a része, ahol a változó használható. A változódeklaráció helye határozza meg az érvényességi tartományt.

A változó lehet (a deklarálás helyétől függően.):

- tagváltozó: Az osztály vagy objektum része, az osztály egészében látható (alkalmazható). Az osztályváltozók deklarálásánál a `static` kulcsszót kell használni.
- lokális változó: Egy kódblokkon belül alkalmazzuk. A láthatósága a deklaráció helyétől az őt körülvevő blokk végéig tart.
- A metódusok formális paraméterei az egész metóduson belül láthatók.
- A kivételkezelő paraméterek hasonlók a formális paraméterekhez.

Példa a változókra:

```
public class Valtozok{
   static int a=0;//Az a változó tagváltozóként lett deklarálva.
    public static void main(String[] args) {
        int b=1;//A b változó lokális változó.
        System.out.println("a = "+a);
        System.out.println("b = "+b);
    }
}
```

A változót lehet véglegesen is deklarálni (konstans változó), a végleges változó értékét nem lehet megváltoztatni az inicializálás után.

A végleges változók deklarációnál a `final` kulcsszót kell használni:

pl. `final int aKonstans = 0;`

5.3.3. NYELVI ESZKÖZÖK

5.3.3.1. Kifejezések (operandusok és operátorok)

A kifejezések szintaktikai eszközök, arra valók, hogy a program egy adott pontján ismert értékekből új értékeket határozzunk meg. Két komponense van: típus és érték.

Kétféle feladata van: végrehajtani a számításokat, és visszaadni a számítás végeredményét.

Összetevők:

- **Operandusok**: Az értéket képviselik. Lehet literál, nevesített konstans, változó vagy függvényhívás az operandus.
- **Operátorok**: Műveleti jelek (pl. `*, /, +, -`).
- **Kerek zárójelek**: A műveletek végrehajtási sorrendjét befolyásolják.

Példa a kifejezésre:

a + **sin(0)**

Operandus Operátor Operandus

Attól függően, hogy egy operátor hány operandussal végez műveletet, beszélhetünk:

- **egyoperandusú** (unáris pl. `(+2)`),
- **kétoperandusú** (bináris pl. `(2+3)`) vagy
- **háromoperandusú** (ternáris pl. `(a*b*c)`) operátorokról.

5.3.3.1.1. Operátorok:

(Az operanduson hajtanak végre egy műveletet.)

5.3.3.1.1.1. Aritmetikai operátorok: Alapvető matematikai műveletek végzésére használjuk őket.

+	(összeadás),
-	(kivonás),
*	(szorzás),
/	(osztás),
%	(maradékképzés).

Példa az aritmetikai operátorok használatára:

```
public class AritOperatorok{
    public static void main(String[] args) {
       int i=3, j=11;
       double a=3.5,b=12.65;
        System.out.println("Összeadás:");
        System.out.println(" i + j = "+(i + j));
        System.out.println(" a + b = "+(a + b));
        System.out.println("Kivonás:");
        System.out.println(" i - j = "+(i - j));
        System.out.println(" a - b = "+(a - b));
        System.out.println("Szorzás:");
        System.out.println(" i * j = "+(i * j));
        System.out.println(" a * b = "+(a * b));
        System.out.println("(egész)Osztás:");
        System.out.println(" i / j = "+(i / j));
        System.out.println(" a / b = "+(a / b));
        System.out.println("Egész osztás maradéka:");
        System.out.println(" i % j = "+(i % j));
        System.out.println(" a % b = "+(a % b));
    }
}
```

Implicit konverzió: Amikor egy aritmetikai operátor egyik operandusa egész, a másik pedig lebegőpontos, akkor az eredmény is lebegőpontos lesz. Az egész érték implicit módon lebegőpontossá konvertálódik, mielőtt a művelet végrehajtódna.

Példa: Egy egész számot osztunk egy valós számmal, a végeredmény automatikusan valós lesz.

```
public class Implicit{
    public static void main(String[] args) {
        int x=10;//egész szám
        double y=3.23;//lebegőpontos szám
        System.out.println("x/y = "+x/y);//itt írjuk ki a hányadost
    }
}
```

A konverziót ki is „kényszeríthetjük” **explicit konverzióval**. A programozó ebben az esetben a kifejezés értékére „ráerőltet” egy típust.

Szintaxis:

```
(<típus>) <kifejezés>
```

Példa:

6/4 >> eredmény: **1** (Itt két egész számot osztottunk egymással.)

(double) 6/4 >> eredmény: **1.5**

5.3.3.1.1.2. Relációs operátorok: Összehasonlítanak két értéket, és meghatározzák a köztük lévő kapcsolatot.

> (nagyobb),

>= (nagyobb vagy egyenlő),

< (kisebb),

<= (kisebb vagy egyenlő),

= = (egyenlő),

! = (nem egyenlő).

Példa a relációs operátorok használatára:

```
public class RelacOperatorok{
    public static void main(String[] args) {
       int a=13, b=11;
       System.out.println(" a > b = "+(a > b));//a nagyobb mint b
       System.out.println(" a >= b = "+(a >= b));//a nagyobb vagy egyenlő mint b
       System.out.println(" a < b = "+(a < b));//a kisebb mint b
       System.out.println(" a <= b = "+(a <= b));//a kisebb vagy egyenlő mint b
       System.out.println(" a == b = "+(a == b));//a egyenlő b-vel
       System.out.println(" a != b = "+(a != b));// a nem egyenlő b-vel
    }
}
```

5.3.3.1.1.3. Logikai operátorok:

&& (logikai és),

|| (logikai vagy),

! (logikai nem),

& (bitenkénti és),

| (bitenkénti vagy),

^ (bitenkénti nem),

~ (Bitenkénti tagadás (negáció)).

Példa a logikai operátorok használatára:

```
public class LogOperatorok{
    public static void main(String[] args) {
      boolean a=true,b=false,c=true, d=false;
        System.out.println("Logikai és:");
        System.out.println("a && b = "+(a && b));
        System.out.println("a && c = "+(a && c));
        System.out.println("Logikai vagy:");
        System.out.println("a || b = "+(a || b));
        System.out.println("a || c = "+(a || c));
        System.out.println("b || d = "+(b || d));
        System.out.println("Logikai nem:");
        System.out.println("!a = "+(!a));
        System.out.println("!b = "+(!b));
        System.out.println("Bitenkénti és:");
        System.out.println("a & b = "+(a & b));
        System.out.println("a & c = "+(a & c));
        System.out.println("Bitenkénti vagy:");
        System.out.println("a | b = "+(a | b));
        System.out.println("b | d = "+(b | d));
        System.out.println("Bitenkénti nem:");
        System.out.println("a ^ b = "+(a ^ b));
        System.out.println("a ^ c = "+(a ^ c));

    }
}
```

5.3.3.1.1.4. Léptető operátorok: A léptető operátorok bitműveleteket végeznek, a kifejezés első operandusának bitjeit jobbra vagy balra léptetik.

<< (op1 << op2; op1 bitjeit op2 értékével balra lépteti, jobbról nullákkal tölti fel)

>> (op1 >> op2; op1 bitjeit op2 értékével jobbra lépteti, balról a legnagyobb helyértékű bitet tölt fel)

>>> (op1 >>> op2; op1 bitjeit op2 értékével jobbra lépteti, balról nullákkal tölt fel)

5.3.3.1.1.5. Értékadó operátorok

= Alap értékadó operátor, arra használjuk, hogy egy változóhoz értéket rendeljünk.

Rövidített értékadó operátorok: + =, - =, * =, / =, % =, & =, | =, ^ =, << =, >> =, >>> = (pl. `i = i + 1;` rövidítve: `i + = 1;`)

Példa az értékadó operátorok használatára:

```
public class ErtekaOperatorok {
    public static void main(String[] args) {
        int i = 1, j = 0;
        System.out.println("i = " + i);
        i = i + 1;//i változó értékét megnöveljük egyel
        System.out.println("i = " + i);
        System.out.println("j =" + j);
        j += 2;// j változó értékét növeljük 2-vel, rövidített értékadó op.
        System.out.println("j = " + j);
    }
}
```

5.3.3.1.1.6. Egyéb operátorok:

?: Feltételes operátor

[ ] Tömbképző operátor

. Minősített hivatkozás

new Új objektum létrehozása

A kifejezés alakja lehet:

- **prefix**: Az operátor az operandusok előtt áll (`* 2 5`).
- **infix**: Az operátor az operandusok között áll (`2 * 5`).
- **postfix**: Az operátor az operandusok mögött áll (`2 5 *`).

Azt a folyamatot, amikor a kifejezés értéke és típusa meghatározódik, a **kifejezés kiértékelésének** nevezzük. A kiértékelés során adott sorrendben elvégezzük a műveleteket, előáll az érték és hozzárendelődik a típus.

A Javában a következő módon megy végbe a kiértékelés:

- Zárójelezéssel meghatározhatjuk, hogy egy kifejezés hogyan értékelődjön ki. pl. `(x + y) / 2` Először a zárójelben szereplő kifejezés hajtódik végre.
- Ha nem jelezzük, hogy milyen sorrendben akarjuk az összetett kifejezést kiértékelni, akkor a sorrendet az operátorok precedenciája fogja meghatározni. A magasabb precedenciával rendelkező operátorok hajtódnak végre elsőként.

Pl. `x + y / 2` Ebben a kifejezésben, mivel az osztás precedenciája magasabb, mint az összeadásé, ezért először ez hajtódik végre, majd az összeadás következik. A következő kifejezés egyenértékű vele: `x + (y / 2).`

Operátor precedencia szintek:

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

A táblázat a Java platformon használt operátorok precedencia szintjeit mutatja be. Az operátorok precedencia szint szerint vannak rendezve, mégpedig úgy, hogy legfelül a legnagyobb precedenciájú, legalul a legkisebb precedenciájú található. Az azonos szinten elhelyezkedő operátorok azonos precedenciával rendelkeznek. Ha azonos precedenciájú operátorok szerepelnek egy kifejezésben, akkor balról jobbra történik a végrehajtás, kivéve az értékadó operátorokat, amelyek jobbról balra hajtódnak végre.

Azt a kifejezést, amelynek értéke fordítási időben eldől, **konstans kifejezésnek** nevezzük. Létrehozására a `final` módosítót használjuk, így érjük el, hogy az adattag értékét nem változtathatja meg senki.

pl. `final double PI=3.14;`: Pi értékét jelenti.

5.3.3.2. Deklarációs és végrehajtható utasítások

A **deklarációs utasítások** mögött nem áll tárgykód. A programozó a névvel rendelkező saját programozási eszközeit tudja deklarálni.

pl. `int i=10;`

A **végrehajtható utasításokból** generálja a fordító a tárgykódot.
Melyek ezek?

- Üres utasítás (pl. `;`)
- Kifejezés utasítás (pl. `8;  a = b;`)
- Vezérlő utasítások (szekvencia, feltételes utasítás, többszörös elágaztatás, ciklusok, vezérlésátadó utasítások)
- Módszerhívások (metódushívások)
- Objektumot létrehozó kifejezések

5.3.3.3. Alprogramok

A programot célszerű kisebb egységekre (alprogram) bontani, melyeket a program bizonyos pontjairól aktivizálhatunk. Így áttekinthetőbbé, olvashatóbbá válik a forrásprogram, valamint a többször előforduló tevékenységeket elegendő egyszer megírni (elkészíteni).

A Javában **metódus**nak nevezzük az alprogramot. A metódus utasítások összessége, melyet meghívhatunk a metódus nevére való hivatkozással.

Egy metódus lehet eljárás vagy függvény aszerint, hogy van-e visszatérési értéke.

5.3.3.3.1. Eljárás

Az eljárásnál nincsen visszatérési érték, egyszerűen végrehajtódik az eljárás nevére való hivatkozással. A végrehajtás után a program azzal az utasítással folytatódik, amelyik az eljárást meghívó utasítást követi. Az eljárás visszatérési típusa void (semleges), ami azt jelenti, hogy a visszatérési típus nincs definiálva.

Példa az eljárásra:

(Az eljárás egy vonalat „rajzol” a képernyőre.)

```
public class MetodusPelda {
    public static void main(String[] args) {
        System.out.println("Következik a vonal:");
        vonalhuz();//itt történik a metódus meghívása
    }

    private static void vonalhuz() {//vonalhúzó metódus
        System.out.println("_____");
    }

}
```

5.3.3.3.2. Függvény

A függvénynél van visszatérési érték. Függvényt is egyszerűen a nevére való hivatkozással hívunk meg, de visszaad egy értéket, melyet a függvény neve képvisel.

Példa a függvényre:

(A függvény megduplázza az x változó értékét.)

```
public class Fuggveny {
    public static void main(String[] args) {
        int x = 2;
        System.out.println("x= " + x);
        x = duplaz(x);//itt történik a függvény hívása
        System.out.println("x= " + x);
    }
    private static int duplaz(int x) {//duplázó függvény
        return x = 2 * x;
    }
}
```

5.3.3.4. Blokk

A blokk nulla vagy több utasítás kapcsos zárójelek között, amely használható bárhol, ahol az önálló utasítások megengedettek.

Jellemzői:

- { } zárójelek között szerepel.
- Címkézhető.
- Tetszőleges mélységben egymásba skatulyázható.
- A változó a blokk lokális változójaként deklarálható.
- A blokkon belül tetszőleges a deklarációs és végrehajtható utasítások sorrendje.

5.3.4. NUMERIKUS- ÉS EGYÉB ADATTÍPUSOK

A Java nyelvben az adattípusoknak két fajtája van: **primitív** és **referencia** típusok. A primitív adattípusok egy egyszerű értéket képesek tárolni: számot, karaktert vagy logikai értéket.

Primitív típusok:

- Egész típus
- Valós típus
- Karakter típus
- Logikai típus

Referencia típusok:

- Tömb típus
- Osztály típus
- Interfész típus

5.3.4.1. Egész és valós számtípusok

5.3.4.1.1. Egészek

Típus	Leírás	Méret / formátum
byte	bájt méretű egész	8 bit kettes komplemens (-128 - 127)
short	rövid egész	16 bit kettes komplemens (-32768 - 32767)
int	egész	32 bit kettes komplemens (-2147483648 - 2147483647)
long	hosszú egész	64 bit kettes komplemens (-9223372036854775808 - 9223372036854775807)

5.3.4.1.2. Valós számok

Típus	Leírás	Méret / formátum
float	egyszeres pontosságú lebegőpontos	32 bit IEEE 754
double	dupla pontosságú lebegőpontos	64 bit IEEE 754

5.3.4.2. Egyéb típusok

Típus	Leírás	Méret / formátum
char	karakterek	16 bit Unicod karakter
boolean	logikai érték	***true*** vagy ***false***

5.3.4.3. Szám és szöveg közötti konvertálások

5.3.4.3.1. Szövegből számmá konvertálás

Előfordulhat, hogy a sztringként rendelkezésre álló adatot számként kell kezelnünk. pl. A szöveges állományból beolvasott adatok sztringként állnak a rendelkezésünkre, és ha szeretnénk velük valamilyen matematikai műveletet végezni, akkor át kell alakítani szám típusúvá.

Az átalakítást a `valueOf` metódussal tudjuk elvégezni.

Példa: (Két szöveg típusú változó számként történő összeadása.)

```
public class SzovegKonvert {
   public static void main(String[] args) {
      String a = "213.4";//a sztring deklarálása
      String b = "110.6";//b sztring deklarálása
      double x = Double.valueOf(a);//a sztring átalakítása x változóba,számként
      double y = Double.valueOf(b);//b sztring átalakítása y változóba,számként
      System.out.println("a+b = " + (a + b));//két sztring összefűzése, kiírása
      System.out.println("x+y = " + (x + y));//két szám összegének kiírása
   }
}
```

5.3.4.3.2. Számból szöveggé konvertálás

Az a helyzet is előfordulhat, hogy a számból szövegbe konvertálásra van szükségünk. pl. A program által szolgáltatott számértékeket szeretnénk egy szöveges állományba kiírni.

Az átalakítást a `toString()` metódussal tudjuk elvégezni.

Példa: (Egy egész típusú szám számjegyeinek megszámlálása.)

```
public class SzambolSzoveg {
    public static void main(String[] args) {
        int x = 1042;//deklaráltunk egy egész számot
        String s = Integer.toString(x);//a számból sztringet készítünk
        System.out.println("Számjegyek száma = " + s.length());/*a szöveg
        hosszának kiírása, így egyszerűsítjük le a számlálás problémáját*/
    }
}
```

5.3.4.4. A Math osztály

Az alapvető aritmetikai számításokon (`+, -, /, %`) túl a Java nyelv biztosítja számunkra a Math osztályt. Az osztály metódusai magasabb rendű matematikai számítások elvégzését is lehetővé teszik. (pl. egy szög szinuszának a kiszámítása.)

A Math osztály metódusai osztály metódusok, így közvetlenül az osztály nevével kell őket meghívni. pl. `Math.abs(-32);`

Nézzük meg a legfontosabb metódusokat!

5.3.4.4.1. Alapvető metódusok:

`double abs(double)`
`float abs(foat)`
`int abs(int)`
`long abs(long)`
	abszolút érték
	A kapott paraméter abszolút értékével tér vissza.

`double ceil(double)`	**felfelé kerekítés** A legkisebb double értékkel tér vissza, ami nagyobb vagy egyenlő a megkapott paraméterrel.
`double floor(double)`	**lefelé kerekítés** A legnagyobb double értékkel tér vissza, ami kisebb vagy egyenlő a megkapott paraméterrel.
`double rint(double)`	**a legközelebbi egészhez kerekít** A megkapott paraméterhez legközelebb álló double értékkel tér vissza.
`long round(double)` `int round(float)`	**kerekítés** A legközelebbi long vagy int értéket adja vissza.

`double min(double, double)`
`float min(float, float)`
`int min(int, int)`
`long min(long, long)`
	A két paraméter közül a kisebbel tér vissza.

`double max(double, double)`
`float max(float, float)`
`int max(int, int)`
`long max(long, long)`
	A két paraméter közül a nagyobbal tér vissza.

A következő program bemutatja a fenti táblázatban ismertetett metódusok használatát:

```
public class MatDemo1 {
    public static void main(String[] args) {
        double szam1 = -25.64;
        double szam2 = 221;
        System.out.println("Az első szám =" + szam1);
        System.out.println("A második szám =" + szam2);
        System.out.println("A szám abszolút értéke =" + Math.abs(szam1));
        System.out.println("A szám értéke felfelé kerekítve =" + Math.ceil(szam1));
        System.out.println("A szám értéke lefelé kerekítve =" + Math.floor(szam1));
        System.out.println("A legközelebbi egészhez kerekítve =" + Math.rint(szam1));
        System.out.println("A round eredménye " + Math.round(szam1));
        System.out.println("A kisebbik szám =" + Math.min(szam1, szam2));
        System.out.println("A nagyobbik szám =" + Math.max(szam1, szam2));
    }
}
```

5.3.4.4.2. Hatványozással kapcsolatos metódusok:

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

A következő program bemutatja a fenti táblázatban ismertetett metódusok használatát:

```
public class MatDemo2 {
    public static void main(String[] args) {
        double x = 16.00;
        double y = 2.32;
        System.out.println("x =" + x);
        System.out.println("y =" + y);
        System.out.println("Az e értéke =" + Math.exp(1));
        System.out.println("ln(x)=" + Math.log(x));
        System.out.println("x az y hatványon =" + Math.pow(x, y));
        System.out.println("x négyzetgyöke =" + Math.sqrt(x));
    }
}
```

5.3.4.4.3. Trigonometrikus függvények:

`double sin(double)`	Egy szám szinuszával tér vissza.
`double cos(double)`	Egy szám koszinuszával tér vissza.
`double tan(double)`	Egy szám tangensével tér vissza.
`double asin(double)`	Egy szám arc szinuszával tér vissza.
`double acos(double)`	Egy szám arc koszinuszával tér vissza.
`double atan(double)`	Egy szám arc tangensével tér vissza.
`double toDegrees(double)`	A paramétert fokká konvertálja.
`double toRadians(double)`	A paramétert radiánná konvertálja.

A következő program bemutatja a fenti táblázatban ismertetett metódusok használatát:

```
public class MatDemo3 {
    public static void main(String[] args) {
        double szog = 60.0;
        double szogRad = Math.toRadians(szog);
        System.out.println("A szög =" + szog + " fok");
        System.out.println("Radiánba megadva =" + szogRad);
        System.out.println("sin(" + szog + ") = " + Math.sin(szogRad));
        System.out.println("cos(" + szog + ") = " + Math.cos(szogRad));
        System.out.println("tan(" + szog + ") = " + Math.tan(szogRad));
        System.out.println("arc sin(0.5) = " + Math.toDegrees(Math.asin(0.5)));
    }
}
```

5.3.4.4.4. Véletlen szám készítés:

Gyakran szükségünk van véletlen számok előállítására, ezt a `double random()` metódussal tehetjük meg. A metódus egy véletlen számot ad [0.0;1.0] intervallumban.

Példa:(Generáljunk egy véletlen egész számot 1-től 10-ig terjedő intervallumban.)

```
public class Veletlen1 {
    public static void main(String[] args) {
        int szám = (int) ((Math.random() * 10) + 1);
        System.out.println("A véletlen szám = " + szám);
    }
}
```

Figyeljük meg a bekeretezett kifejezést!

A `random()` metódus 0 és 1 közötti `double` típusú számot állít elő úgy, hogy a 0-t igen, de az 1-t soha nem veszi fel. Mi 1 és 10 közötti egész számot szeretnénk, ezért a véletlen számot meg kell szorozni 10-el és még hozzá kell adni 1-t is. Azt, hogy a végeredmény egész típusú legyen „ki kell kényszerítenünk” (`int`).

5.3.5. KARAKTERES ADATTÍPUSOK

Karaktereket definiálhatunk egyszerűen a char típus segítségével, (pl. `char a='a';`) de létrehozhatunk karakter objektumokat is.

5.3.5.1. A Character osztály

Az egyéb típusoknál ismertetett char típus helyett van, amikor szükségünk van arra, hogy a karaktert objektumként használjuk, pl. amikor egy karakter értéket akarunk átadni egy

metódusnak, ami megváltoztatja ezt az értéket. A `Character` típusú objektum egyetlen karakter értéket tartalmaz.

Készítsünk egy karakter objektumot! pl. `Character a = new Character('a');`

5.3.5.1.1. A Character osztály fontosabb konstruktorai, metódusai

`Character(char)`	A Character osztály egyetlen konstruktora, amely létrehoz egy Character objektumot.
`compareTo(Character)`	Összehasonlít két Character objektumban tárolt értéket. Visszaad egy egész számot, ami jelzi, hogy az objektum értéke kisebb, egyenlő, vagy nagyobb, mint a paraméterben megadott érték.
`equals(Object)`	Két karakter objektumot hasonlít össze, true értékkel tér vissza, ha a két érték egyenlő.
`toString()`	Sztringé konvertálja az objektumot. A sztring 1 karakter hosszú lesz.
`charValue()`	Megadja az objektum értékét egyszerű char értékként.
`boolean isUpperCase(char)`	Meghatározza, hogy az egyszerű char érték nagybetű-e.
`boolean isLowerCase(char)`	Meghatározza, hogy az egyszerű char érték kisbetű-e.
`boolean isLetter(char)`	Meghatározza, hogy az egyszerű char érték ékezetes betű-e
`boolean isDigit(char)`	Meghatározza, hogy az egyszerű char érték számjegy-e.
`boolean isSpaceChar(char)`	Meghatározza, hogy az egyszerű char érték szóköz-e.

A következő program bemutatja a fenti táblázatban ismertetett metódusok használatát:

```
1  public class Karakterek{
2      public static void main(String[] args) {
3          char c='c';//c egy char típusú változó
4          Character a = new Character('a');
5          Character a1 = new Character('a');
6          Character a2 = new Character('b');
7          Character b = new Character('b');
8          Character d = new Character ('D');
9          Character s = new Character (' ');// itt egy space van
10         Character l = new Character ('1');//itt egy egyes szám van
11         System.out.println("a egyenlő a1? "+(a.equals(a1)));
12         System.out.println(" b nagyobb mint a? "+(b.compareTo(a)));
13         System.out.println("A d az nagybetű? "+Character.isUpperCase(d));
14         System.out.println("A d az kisbetű? "+Character.isLowerCase(d));
15         System.out.println("s egy szóköz? "+Character.isSpaceChar(s));
16         System.out.println("Az l az számjegy? "+Character.isDigit(l));
17
18     }
19 }
```

A Java nyelvben három osztály áll rendelkezésünkre, amelyekkel tárolhatunk, illetve manipulálhatunk sztringeket, ezek a `String`, a `StringBuffer` és a `StringBuilder`. A `StringBuilder` osztállyal nem foglalkozunk. A `String` osztályban olyan sztringeket tárolunk, amelyek értéke nem fog változni. A `StringBuffer` osztályt akkor alkalmazzuk, ha a szövegen szeretnénk módosítani.

5.3.5.2. A String osztály

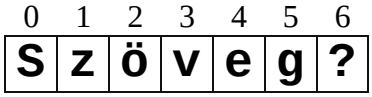
5.3.5.2.1. String objektumok létrehozása

A String objektumok létrehozása kétféle módon történhet:

- Készíthetjük a sztringet egy sztring konstansból, egy karaktersorozatból.
  pl. `String str1 = "Ez egy szöveg";`
- Vagy mint minden más objektumot, a `new` operátorral hozhatjuk létre.
  pl. `String str2 = new String ("Ez is egy szöveg");`

5.3.5.2.2. A karakterlánc indexelése:

Figyeljük meg a következő ábrát!



Az ábrán egy „Szöveg?” objektum látható, amely egy 7 karakter hosszú szöveg (String). Az első betű az S indexe 0, a v betű indexe 3, a ? karakteré pedig a 6.

Figyeljünk arra, hogy az indexelés nullával kezdődik!

5.3.5.2.3. A String osztály fontosabb konstruktorai, metódusai:

`String()`	A létrehozott objektum az üres karakterláncot reprezentálja.
`String(String value)`	A létrehozott objektum a paraméterben megadott szöveget tartalmazza.
`int length()`	Visszaadja a szöveg hosszát.
`char charAt(int index)`	Visszaadja az index indexű karaktert.
`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 maga másolatát.
`String replace(char oldChar,char newChar)`	A metódus visszaad egy karakterlánc-objektumot, amelyben minden oldChar karaktert newChar-ra cserél.
`String substring(int beginIndex)`	Visszaadja az objektum részláncát beginIndex-től a végéig.
`String substring(int beginIndex,endIndex)`	Visszaadja az objektum részláncát beginIndex-től endIndex-1-ig.
`boolean equals(Object anObject)`	Összehasonlítja az objektumot a paraméterként megadott másik objektummal.
`boolean equalsIgnoreCase(String str)`	Összehasonlítja az objektumot a paraméterül megadott másik String objektummal. A kis- és nagybetű között nem tesz különbséget.
`int compareTo (String str)`	Összehasonlítja az objektumot a paraméterül megkapott másik String objektummal. A visszaadott érték 0, ha a két objektum megegyezik. Ha a szöveg nagyobb, mint a paraméter, akkor pozitív, ellenkező esetben negatív.
`String concat (String str)`	Összefűzi a paraméterül megadott sztringet az objektummal. (A + operátorral is el lehet végezni ezt a műveletet.)

5.3.5.3. A StringBuffer osztály

A `String` osztállyal ellentétben a `StringBuffer` osztály szövege manipulálható.

A `StringBuffer` osztály fontosabb konstruktorai, metódusai:

`StringBuffer()`	A létrehozott objektum az üres karakterláncot reprezentálja.
`StringBuffer(int length)`	A létrehozott objektum az üres karakterláncot reprezentálja, kezdeti kapacitása length karakter.
`StringBuffer(String str)`	A létrehozott objektum a paraméterben megadott szöveget tartalmazza.
`int capacity()`	Megadja a kezdeti kapacitást, ennyi karakter fér az objektumba.
`int length()`	Megadja a szöveg aktuális hosszát.
`char charAt(int index)`	Visszaadja az index indexű karaktert.
`StringBuffer append(<Type> value)`	Bővítés, a Type itt egy osztályt reprezentál.

`StringBuffer append(<type> value)`	Bővítés, a type itt egy tetszőleges primitív típus.
`StringBuffer insert( in offszet, <Type> value)`	Beszúrás offszet pozíciótól kezdve.
`StringBuffer insert( in offszet, <type> value)`	Beszúrás offszet pozíciótól kezdve.
`StringBuffer deleteCharAt(int index)`	Adott indexű karakter törlése a szövegből.
`StringBuffer delete(int start, int end)`	Részlánc törlése a szövegből, start indextől end index-1-ig.
`StringBuffer replace(int start, int end, String str)`	A start és end-1 közötti részlánc cseréje str-rel.
`StringBuffer reverse()`	Megfordítja az objektum szövegét.
`String substring (int start, int end)`	Visszaad egy start és end-1 index közötti részláncú új String objektumot.
`String substring (int start)`	Visszaad egy start indexel kezdődő részláncú új String objektumot.

5.3.5.4. Sztringekkel végezhető műveletek

5.3.5.4.1. Szöveg hosszának meghatározása, adott karakterek megjelenítése:

`int length()` >> **Szöveg hossza**

Példa:

```
String str="Laci";
int hossz=str.length(); >>   4
```

`char charAt(int index)` >> **Adott indexű karakter**

Példa:

```
String str="Laci";
char c=str.charAt(0);   >>   L
```

5.3.5.4.2. Manipulálás a szöveggel:

`String toLowerCase()` **>>** **Kisbetűs átalakítás**

Példa:

```
String str="KICSI";
String kstr=str.toLowerCase();    >>   kicsi
```

`String toUpperCase()` >> **Nagybetűs átalakítás**

Példa:

```
String str="nagy";
String nstr=str.toUpperCase();    >>   NAGY
```

```
String replace(char oldChar,char newChar)   >>   Karakterek kicserélése
```

Példa:

```
String str1="Remek eme Mekk Mester";
String str2=str1.replace(e, a);   >>   Ramak ama Makk Mastar
```

```
String substring(int beginIndex)  >>   Részsztring készítés
String substring(int beginIndex, int endIndex)
```

Példa:

```
String str1="Pálinka";
String str2=str1.substring(3);    >>   inka
String str3=str1.substring(0,3);  >>   Pál
```

5.3.5.4.3. Összehasonlítás (egyenlőségvizsgálat):

```
boolean equals(Object anObject)   >>   Objektumok összehasonlítása
int compareTo(String str)
```

Példa:

```
String str1="Laci";
String str2="Gizi";
str1.equals(str2); >>   false
str1.compareTo(str2)    >>   5
```

Figyelem! Az = = operátor nem az egyenlőséget, hanem az azonosságot vizsgálja!

5.3.5.4.4. Keresés:

```
int indexOf(int ch)     >>   Karakter keresés
int indexOf(int ch, int fromIndex)
```

```
int indexOf(String str) >>   Részsztring keresés
int indexOf(String str, int fromIndex)
```

Példa:

```
String str1="Laci";
char c = 'a';
String str2="ci";
```

```
str1.indexOf(a);    >>    1
str1.indexOf(str2;0)  >>    2
```

5.3.5.4.5. Konkatenáció (összefűzés):

```
String concat(String str)   >>   Hozzáfűzés
```

Példa:

```
String str1="La";
String str2="ci";
str1.concat(str2); >>   Laci
```

5.3.5.4.6. Bővítés:

```
StringBuffer append(<Type|type> value) >>   Bővítés a végén
```

Példa:

```
StringBuffer szoveg = new StringBuffer("érték");
szoveg.append("12.3");  >>   érték12.3
```

```
StringBuffer insert(int offset,<Type|type> value)    >>
```

Bővítés adott pozíciótól.

Példa:

```
szoveg.insert(5, "=");  >>   érték=12.3
```

5.3.5.4.7. Törlés:

```
StringBuffer deleteCharAt(int index)   >>   Adott indexű
```

karakter törlése.

```
StringBuffer delete(int start, int end)    >>   Részlánc
```

törlése.

Példa:

```
StringBuffer szoveg = new StringBuffer("Labdarúgás");
szoveg.delete(5,10);    >>   Labda
szoveg.deleteCharAt(0)  >>   abda
```

5.3.6. ELÁGAZTATÓ UTASÍTÁSOK

A program készítése során adódnak olyan helyzetek, ahol a folyamatos utasítás végrehajtás menetét meg kell törnünk, feltételektől függő elágazásokat kell a programba beépítenünk. Így tudjuk megoldani azt a feladatot, hogy ez a programrész csak akkor kerüljön végrehajtásra, ha arra szükség van.

5.3.6.1. Kétirányú elágaztató (feltételes) utasítás: if..else

Két alakjában használhatjuk: rövid alak, ha hiányzik az `else` ág , egyébként hosszú alakról beszélünk.

Szintaktika:

> `if(<feltétel>) <utasítás vagy blokk>`: **rövid alak**
> `[else <utasítás vagy blokk>]`: **hosszú alak**, a teljes utasítás

Működési elv:

Először kiértékelődik a feltétel.

- Ha igaz, akkor végrehajtódik a feltétel utáni tevékenység és a program az `if` utasítást követő utasításon folytatódik.
- Ha a feltétel nem igaz, akkor az `else` ágban megadott tevékenység hajtódik végre, majd a program az `if` utasítást követő utasításon folytatódik. Amennyiben nincs `else` ág, akkor ezt egy üres utasításnak vehetjük.

Példa:

```
public class Ha {
    public static void main(String[] args) {
        int szam = 21;
        if (szam < 20) {
            System.out.println("Kicsi");
        } else {
            System.out.println("Nagy");
        }
    }
}
```

5.3.6.2. Többirányú elágaztató utasítás

Ha olyan feladattal állunk szemben, hogy több tevékenység közül egyet kell kiválasztanunk, akkor a többirányú elágaztató utasítást alkalmazzuk. A Javában ezt kétféle vezérlőszerkezettel tehetjük meg: `switch..case` és az `else..if`.

5.3.6.2.1. switch..case

Főleg akkor használjuk, ha egy kifejezés jól meghatározott, különböző értékeire szeretnénk bizonyos utasításokat végrehajtani.

Szintaktika:

```
switch(<kifejezés>){
    case <konstans kifejezés> : <utasítás vagy blokk>
    [case <konstans kifejezés> : <utasítás vagy blokk>]
    ...
    [default:<utasítás v. {blokk}>]
}
```

Működési elv:

Először kiértékelődik a kifejezés.

A kifejezés értéke a felírás sorrendjében összehasonlításra kerül a `case` ágak értékeivel.

- Ha van egyezés, akkor végrehajtódik az adott ágban megadott tevékenység, majd a program a következő ágakban megadott tevékenységeket is végrehajtja.
- Ha nincs egyezés, akkor a `default` ágban megadott tevékenység hajtódik végre.

Példa:

```
public class Menu{
    public static void main(String[] args) {
        byte c=3;
        System.out.println("Az alábbi lányokat választottad:");
        switch(c){
            case 4 : System.out.println("Ági");
            case 3 : System.out.println("Judit");
            case 2 : System.out.println("Kati");
            case 1 : System.out.println("Mari");
            default : System.out.println("");
        }
    }
}
```

Output:

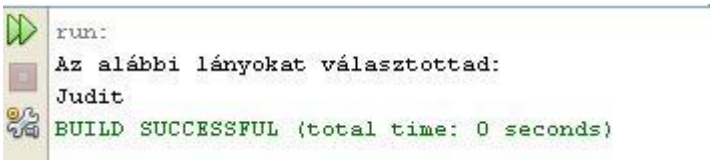
```
run:
Az alábbi lányokat választottad:
Judit
Kati
Mari
```

Ha rögtön ki szeretnénk lépni a `switch` utasításból az adott tevékenység végrehajtása után, akkor külön utasítást kell alkalmaznunk (`break`).

Példa:

```
public class Menu1{
    public static void main(String[] args) {
        byte c=3;
        System.out.println("Az alábbi lányokat választottad:");
        switch(c){
            case 4 : System.out.println("Ági");break;
            case 3 : System.out.println("Judit");break;
            case 2 : System.out.println("Kati");break;
            case 1 : System.out.println("Mari");break;
            default : System.out.println("");break;
        }
    }
}
```

Output:



5.3.6.2.2. else..if

Szintaktika:

```
if (<feltétel1>)
     <utasítás . blokk>
else if (<feltétel2>)
     <utasítás v. blokk>
     ...
else
     <utasítás v. blokk>
```

A működési elvet az alábbi példa szemlélteti:

```
public class Menu2{
    public static void main(String[] args) {
        byte c=3;
        System.out.println("Az alábbi lányokat választottad:");
        if(c==4){System.out.println("Ági");}
        else if(c==3){System.out.println("Judit");}
        else if(c==2){System.out.println("Kati");}
        else if(c==1){System.out.println("Mari");}
        else {System.out.println("");}
    }
}
```

A forráskód az előző példában (Menu1) ismertetett végeredményt szolgáltatja. A két kód futtatásának eredménye egyenértékű egymással. A programozótól függ, hogy melyiket használja az adott feladat megoldása során.

5.3.6.3. Elágaztató utasítások egymásba ágyazása

Az elágaztató utasítások tetszőlegesen egymásba ágyazhatók. A program áttekinthetősége érdekében célszerű a forrásprogram szerkezetét jól strukturáltan elkészíteni. Figyeljünk a „csellengő `else`”-re! Mit is jelent ez? Ha rövid `if` utasításokat írunk, akkor az `else` utasítás vajon melyik `if`-hez tartozik? A Java azt mondja, hogy a legutóbbi `if`-hez, ha csak blokkok képzésével felül nem bíráljuk ezt. A probléma másik megoldása az, hogy mindig hosszú `if` utasításokat alkalmazunk.

Nézzük végig példákon keresztül a fent említett eseteket!

(Azt vizsgáljuk, hogy az adott szám belül van-e egy tartományon ([100;1000]), és ha ott van, akkor páros vagy páratlan?)

Megtévesztő kód: (Strukturálatlan, hova tartozik az `else`?)

```
public class Csellengo1{
    public static void main(String[] args) {
        int szam= 20;
        if(szam>=100 && szam<=1000)System.out.println("tartományon belül van");
        if(szam%2==0)System.out.println("páros");
        else System.out.println("páratlan");
    }
}
```

A kód helyesen strukturálva:

```
public class Csellengo1{
    public static void main(String[] args) {
        int szam= 20;
        if(szam>=100 && szam<=1000)System.out.println("tartományon belül van");
            if(szam%2==0)System.out.println("páros");
            else System.out.println("páratlan");
    }
}
```

Alakítsuk át úgy, hogy az első `if`-hez tartozzon az `else`:

```
public class Csellengo3{
    public static void main(String[] args) {
        int szam= 20;
        if(szam>=100 && szam<=1000){System.out.println("tartományon belül van");
            if(szam%2==0){System.out.println("páros");}
            else {System.out.println("páratlan");}
        }
        else {System.out.println("tartományon kívül van");}
    }
}
```

Blokk készítésével oldottuk meg ezt a feladatot.

Próbáljuk meg hosszú `if`-utasításokkal!

```
public class Csellengo4{
    public static void main(String[] args) {
        int szam= 20;
        if(szam>=100 && szam<=1000){System.out.println("tartományon belül van");
            if(szam%2==0){System.out.println("páros");}
            else {System.out.println("páratlan");}
        }
        else {System.out.println("tartományon kívül van");}
    }
}
```

5.3.6.4. A billentyűzetről bevitt adatok vizsgálata

A programok készítése során gyakran kerülünk olyan feladat elé, amikor a programnak a továbbhaladáshoz szüksége van a felhasználó beavatkozására, adatokat vár a billentyűzetről. pl. menürendszernél a felhasználó választásától függően fog a vezérlés a megfelelő programrészre kerülni.

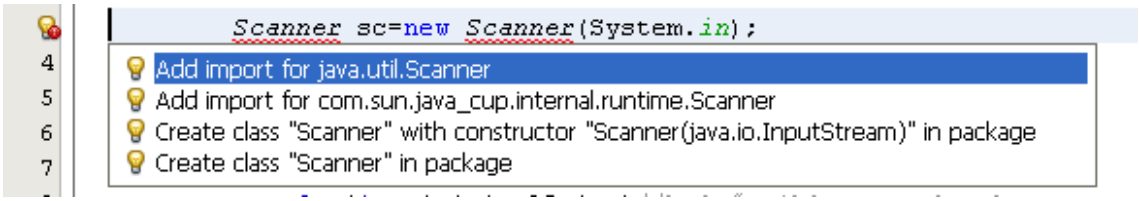
Hogyan olvassuk be az adatokat a billentyűzetről? Az adatok kezelésével a 9. héten fogunk foglalkozni részletesen, most elégedjünk meg egy kis programrészlettel, aminek a segítségével megoldható a beolvasás problémája.

A kódot másoljuk be a programunkba, a kívánt helyre!

```
Scanner sc=new Scanner(System.in);
     System.out.println("Kérem a számot!");
     int a=sc.nextInt();
```

Ne feledkezzünk meg a `Scanner` osztály importálásáról!

Kattintsunk bal gombbal a lámpácskára és válasszuk az `Add import for java.util.Scanner opciót`:



Példa: (A program a felhasználó választásától függő darabszámú csillagot jelenít meg.)

```
import java.util.Scanner;
public class CsillagMenu{
  public static void main(String[] args) {
        Scanner sc=new Scanner(System.in);
    System.out.println("Hány csillagot szeretnél? [1..9]");
    int a=sc.nextInt();//Az a változóba kerül a beolvasott érték.
    switch(a){
        case 1:System.out.println("*");break;
        case 2:System.out.println("**");break;
        case 3:System.out.println("***");break;
        case 4:System.out.println("****");break;
        case 5:System.out.println("*****");break;
        case 6:System.out.println("******");break;
        case 7:System.out.println("*******");break;
        case 8:System.out.println("********");break;
        case 9:System.out.println("*********");break;
        default:System.out.println("");;break;
    }
  }
}
```

5.3.7. CIKLUSSZERVEZŐ UTASÍTÁSOK

Ha egy bizonyos tevékenységet a programban egymásután többször végre kell hajtani, akkor ciklusokat alkalmazunk. A ciklusok végrehajtását, vagy az abból való kilépést feltételekhez kötjük.

Egy ciklus általános felépítése:

- fej
- mag
- vég

Az ismétlésre vonatkozó információk vagy a fejben vagy a végben találhatók.

A magban helyezzük el azokat az utasításokat, amelyeket többször végre akarunk hajtani.

A ciklusok működése során két szélsőséges esettel is találkozhatunk:

- Üres ciklus: A ciklusmag egyszer sem fut le.
- Végtelen ciklus: Az ismétlődés soha nem áll le.

5.3.7.1. Előfeltételes ciklus: while

Szintaktika:

```
while (feltétel)
    <utasítás v. blokk>
```

Működési elv:

A program a ciklusba való belépés előtt megvizsgálja a feltételt (belépési feltétel), és ha ez teljesül, akkor a ciklusmag végrehajtásra kerül, egyébként nem. Ennél a ciklusfajtánál kialakulhat üres ciklus, abban az esetben, ha a belépési feltétel soha nem teljesül.

Példa: (Írjunk ki a konzolra egymás mellé 6 db csillagot!)

```
public class Ciklus1{
    public static void main(String[] args) {
        int i=1;
        while(i<=6){//belépési feltétel
            System.out.print("*");
            i++;//itt léptetjük i változó értékét eggyel
        }
    }
}
```

5.3.7.2. Előfeltételes ciklus: for

Szintaktika:

```
for (inicializálás;feltétel;növekmény)
     <utasítás v. blokk>
```

Működési elv:

- *inicializálás*: Itt deklaráljuk a ciklusváltozót, és adjuk meg annak kezdeti értékét.
- *feltétel*: Belépési feltétel, minden ciklus elején kiértékelődik, ha igaz, akkor a ciklus végrehajtásra kerül, egyébként a vezérlés a `for` utáni utasításra ugrik.
- *növekmény*: Itt változtatjuk meg a ciklusváltozó értékét.

Példa:

```
public class Ciklus2{
    public static void main(String[] args) {
        for(int i=1;i<=6;i++)
            System.out.print("*");
    }
}
```

A `for` fejének tagjai hagyhatók üresen is, de a pontosvesszőket ki kell tenni. pl. `for(;;)...`

5.3.7.3. Végfeltételes ciklus: do..while

Szintaktika:

```
do
     <utasítás v. blokk>
while (feltétel);
```

Működési elv:

A ciklus magja egyszer mindenképpen végrehajtódik, majd a ciklus végén, egy feltételvizsgálat következik, amely eldönti, hogy bent maradunk-e a ciklusban, vagy nem. Ha a feltétel igaz, akkor a ciklusmag újból végrehajtásra kerül. Ez a folyamat addig folytatódik, amíg a feltétel hamissá nem válik (bennmaradási feltétel).

Üres ciklus a végfeltételes ciklusnál nem fordulhat elő, hiszen a ciklus magja legalább egyszer lefut.

Példa:

```
public class Ciklus3{
    public static void main(String[] args) {
        int i=1;
        do{
            System.out.print("*");
            i++;
        }
        while(i<=6);//bentmaradási feltétel
    }
}
```

Az előző három ciklus mindegyike ugyanazt a feladatot oldotta meg, kirajzolt 6 db csillagot a képernyőre.

5.3.7.4. Ciklusok egymásba ágyazása

A ciklusainkat egymásba is ágyazhatjuk.

Nézzünk meg egy egyszerű példát két ciklus egymásba ágyazására!

Példa: (Jelenítsük meg a konzolon következő alakzatot:

****
****
**********)

Forrásprogram:

```
public class Ciklus4 {

    public static void main(String[] args) {

        for (int i = 1; i <= 3; i++) {//külső ciklus, sorok
            for (int j = 1; j <= 20; j++)//belső ciklus, oszlopok
            {
                System.out.print("*");
            }
            System.out.println("");//sortörés megoldása
        }

    }
}
```

A program elemzése:

A csillagok egy 3 soros, 20 oszlopos táblázatot alkotnak. A kirajzolást úgy tudjuk egyszerűen elképzelni, mintha a táblázat celláiba helyeznénk el a csillagokat.

A külső ciklusban megyünk végig a sorokon, a belső ciklusban pedig az oszlopokon. Először az 1. sor elemeit jelenítjük meg egymás után, majd a 2. sor elemeit, és végül a 3. sor következik. Ha egy teljes sort megjelenítettünk, akkor meg kell oldani a sorváltás problémáját. Ezt egyszerűen egy `println("")` utasítással is megtehetjük.

5.3.7.5. Alapvető algoritmusok

5.3.7.5.1. Megszámlálás

Ezzel az algoritmussal egy sorozat, adott tulajdonsággal rendelkező elemeit számoljuk meg.

Az algoritmus menete a következő:

1. A számlálót nullára állítjuk.
2. Végig lépkedünk a sorozat elemein, és ha az aktuális elem tulajdonsága megegyezik az adott tulajdonsággal, akkor a számláló értékét eggyel megnöveljük.
3. Megjelenítjük a számláló értékét, ez a szám adja az adott tulajdonsággal rendelkező elemek számát.

Példa: („Ma süt a nap.” sztringben megszámoljuk az 'a' karakterek számát.)

```
public class Szamlall{
  public static void main(String[] args) {
    int db=0;//ebben a változóban fogjuk az a karakter darabszámát tárolni
    String str="Ma süt a nap.";
    System.out.println(str);
      for (int i=0;i<=(str.length()-1);i++)/*végig lépkedünk a karaktereken
                                           * egyesével, a szöveg végéig
                                           */
        if (str.charAt(i)=='a')db=db+1; /*egyezés esetén 1-el növeljük db-t,
                                         * ha az aktuális karakter éppen az a
                                         */
           System.out.println("Az a betűk száma = "+db);

  }
}
```

5.3.7.5.2. Összegzés, átlagszámítás

Az olyan feladatokat, amelyekben a sorozat elemeit valamilyen módon gyűjtenünk kell (pl. göngyölítés), összegzéses feladatoknak nevezzük. Ebbe a feladatcsoportba sorolható a különbség-, illetve a szorzatképzés is.

Átlag kiszámításakor egyszerre két dolgot is végzünk: összegzünk, s közben számlálunk is. Végül a két érték hányadosát vesszük.

Példa: (Addig olvassuk be a számokat, amíg 0-t nem ütünk, majd írjuk ki a számok összegét és átlagát!)

```
import java.util.Scanner;
public class Atlag{
  public static void main(String[] args) {
    int összeg=0,db=0;
    Scanner sc=new Scanner(System.in);
    System.out.print("Kérem a számot! (0-ra vége!)");
    int a=sc.nextInt();
    while(a!=0){
        összeg=összeg+a;db=db+1;//itt folyik a göngyölítés és a számlálás
        System.out.print("Kérem a következő számot! (0-ra vége!)");
        a=sc.nextInt();

    }
      System.out.println("A számok összege = "+összeg);
      System.out.println("A darabszám = "+db);
      if(db!=0)System.out.println("A számok átlaga = "+((double)összeg/db));
      else System.out.println("Nincs átlag!");

  }
}
```

Figyelem: Az átlag számításakor vigyázni kell a 0-val való osztásra!

5.3.7.5.3. Minimum- és maximum kiválasztás

Minimum kiválasztása esetén a sorozat legkisebb, maximum kiválasztása esetén a sorozat legnagyobb elemét kell meghatároznunk.

Az algoritmus menete a következő:

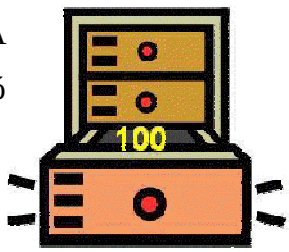
1. A sorozat első elemét elhelyezzük egy minimum (maximum) változóba.
2. Sorban végiglépkedünk az elemeken és, ha az adott elem kisebb (nagyobb) a minimum (maximum) változóban lévőnél, akkor a változóban lévő elemet kicseréljük vele.
3. Megjelenítjük a változóban található elemet, ez az elem lesz a sorozat minimuma (maximuma).

Példa: (A program bekér 5 db valós számot, majd megjeleníti a legkisebbet!)

```
import java.util.Scanner;
public class Minimum {

    public static void main(String[] args) {
        double minimum, a;
        Scanner sc = new Scanner(System.in);
        System.out.print("Kérem az 1. számot! ");
        a = sc.nextDouble();
        minimum = a;//az 1. számot elhelyezzük a változóba

        for (int i = 2; i <= 5; i++) {
            System.out.print("Kérem az " + i + ". számot! ");
            a = sc.nextDouble();
            if (a < minimum) {//ha kisebb számot találunk, akkor kicseréljük
                minimum = a;
            }
        }

        System.out.println("A legkisebb szám = " + minimum);
    }
}
```

5.3.8. TÖMBÖK

A tömb egy olyan változó, amely több azonos típusú adatot tartalmaz. A tömb hossza a létrehozáskor dől el, és attól kezdve a tömb egy állandó méretű adatszerkezet.



A tömböket egy fiókos szekrényhez lehetne hasonlítani, amelyben az egyes fiókokban egy-egy adatot helyezünk el. A fiókokra a sorszámukkal (a tömbben elfoglalt helye) hivatkozunk. A sorszámozást nullával kezdjük! Ha valamelyik fiók tartalmára szükségünk van, akkor megadjuk, hogy hányadik fiókról van szó, és kiolvassuk a tartalmát.

Példa: (Ha szükségünk van a 100-as számra, akkor a 3-as indexű fiókot kell „kihúzni”!)

0	1	2	3	4	5	6	7	8	9
10	2	34	**100**	1	0	5	67	76	99

Lehetőségünk van arra is, hogy olyan tömböket hozzunk létre, amelyek tömböket tartalmaznak (tömbök tömbjei).

5.3.8.1. Tömbök deklarálása, kezdőértékük beállítása

A tömb deklarálása a többi változóéhoz hasonlóan két részből áll: meg kell adni a tömb típusát és a tömb nevét.

Szintaktika:

```
<elemtípus>[] <tömbAzonosító>; pl. int[] szamok;
```

A **[ ]** tömbképző operátort a tömbAzonosító után is tehetjük.

```
<elemtípus> <tömbAzonosító>[]; pl. int szamok[];
```

A deklarálás során nem hoztuk még létre a tömböt, egyelőre csak a referenciának (memóriacímnek) foglaltunk helyet. A tömböt a Javában külön létre kell hozzuk a `new` operátorral!

Szintaktika:

```
new elemtípus[<méret>];
```

pl. `new int [10];`//Itt egy olyan integer típusú adatelemeket tartalmazó tömböt hoztunk létre, amely 10 db elemet tartalmaz.

A deklarációt és a létrehozást egy lépésben is elvégezhetjük.

Például:

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

A deklarálás során (inicializáló blokkal) a tömb elemeinek kezdeti értékek is adhatók.

Szintaktika:

```
<elemtípus>[] <tömbAzonosító> = {<érték0>, <érték1>, ...};
```

pl. Készítsünk egy String típusú tömböt, amely a hét napjait tartalmazza!

```
String[] napok =
{"Hétfő","Kedd","Szerda","Csütörtök","Péntek","Szombat",
"Vasárnap"};
```

5.3.8.2. Tömbelemek elérése

A tömb egyes elemeire az indexükkel hivatkozhatunk.

Szintaktika:

```
<tömbAzonosító>[index];
```

pl. `szamok[3] = 2;` // A 4. fiókba (3-as indexű) betettük a 2-es számot.

5.3.8.3. Tömb méretének meghatározása

Minden tömb objektumnak van egy `length` konstansa, amely megadja a tömb hosszát. (Egyszerűen azt, hogy hány adat helyezhető el benne?)

```
tömbAzonosító.length
```

pl. (A napok tömbnél, amely a hét napjait tartalmazza.)

```
System.out.println(napok.length); >> 7
```

5.3.8.4. Objektumtömbök

A tömbökben tárolhatunk referencia típusú elemeket is. A létrehozásuk olyan, mint a primitív típusú elemeket tartalmazó tömböké.

Nézzünk meg egy egyszerű példát, ahol a tömbben három `String` objektumot tárolunk:

```
public class Otomb {
    public static void main(String[] args) {
        String[] Felsorolas = {"répa", "retek", "mogyoró"};
    }
}
```

A tömb elemeinek a bejárásához használhatunk egy speciális programozói eszközt, az ún. `for-each` ciklust.

Szintaktika:

```
for(<típus> <változó> : <objektum>)
    <utasítás vagy blokk>
```

Példa:

```
public class Otomb2 {
    public static void main(String[] args) {
        String[] Felsorolas = {"répa", "retek", "mogyoró"};
        for (String s : Felsorolas) {
            System.out.println(s);
        }
    }
}
```

for-each ciklus

5.3.8.5. Tömbök tömbjei

A tömbök tartalmazhatnak tömböket, tetszőleges mélységben egymásba ágyazva.

Szintaktika:

Deklarálás: `<elemtípus>[][] ... [] <tömbAzonosító>;`

Létrehozás: `new <elemtípus> [méret0][méret1]...[méretn-1];`

Tömb elemeinek elérése: `<tömbAzonosító>index0][index1]...[indexn-1]`

Példa: (Egy tömbben elhelyezünk másik 3 tömböt.)

```
public class Tomb {
    public static void main(String[] args) {
        String[][] mondoka = {
            {"Egy", "Megérett a meggy."},
            {"Kettő", "Csipkebokor vessző."},
            {"Három", "Te leszel a párom."}
        };
    }
}
```

Ezt úgy képzelhetjük el a legegyszerűbben, mintha egy „3 soros 2 oszlopos táblázatban” helyeznénk el az adatokat:

Egy	Megérett a meggy.
Kettő	Csipkebokor vessző.
Három	Te leszel a párom.

Írjuk ki a konzolra a tömbből a „Te leszel a párom szöveget”!

```
public class Tomb {
    public static void main(String[] args) {
        String[][] mondoka = {
            {"Egy", "Megérett a meggy."},
            {"Kettő", "Csipkebokor vessző."},
            {"Három", "Te leszel a párom."}
        };
        System.out.println(mondoka[2][1]);
    }
}
```

A megadott szöveg a „táblázat” 3. sorának 2. oszlopában található, az indexelése pedig a következő lesz [2] [1].

Figyelem: **Ne felejtsük el, hogy az indexelés 0-val kezdődik!**

5.3.9. INPUT-OUTPUT MŰVELETEK

5.3.9.1. Egyszerű konzol I/O műveletek

Minden programozási nyelv alapfeladatai közé tartozik a külvilággal való kommunikáció, amely a legtöbb esetben az adatok olvasását jelenti egy bemeneti eszközről (pl. billentyűzet), ill. az adatok írását egy kimeneti eszközre (pl. képernyő, fájl). A Java nyelv az I/O műveleteket **adatfolyamokon** (ún. streameken) keresztül, egységesen valósítja meg. Egy dologra kell csak figyelni, hogy bemeneti vagy kimeneti csatornát kezelünk-e. A csatornák kezelése Java osztályokon keresztül valósul meg, azonban ezek nem részei a Java alap eszközkészletének, ezért importálni kell őket a `java.io` csomagból [11], [12], [13].

A konzol kezelésére a Java három adatfolyamot biztosít. Ezeket az adatfolyamokat nem kell megnyitni vagy bezárni, e műveletek automatikusan történnek. A standard adatfolyamok a következők:

- standard bemenet: `System.in`
- standard kimenet: `System.out`
- standard hiba: `System.err`

A konzol I/O műveletek használata esetén a `java.util` csomagból importálnunk kell a megfelelő osztályokat.

Az alábbi kis egyszerű program a standard be- és kimeneti konzol használatát mutatja be. A program egy egész számot vár a billentyűzetről, amit utána megjelenít a képernyőn:

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Kérek egy egész számot!");
        int a = sc.nextInt();
        System.out.println("A beolvasott szám: " + a);
    }
}
```

Az sc nevű objektum a `Scanner` osztály példánya lesz, amely osztály a beolvasáshoz a `System.in` (billentyűzet input) paramétert használja. Az egész típusú `a` adattag az `sc`

példány `nextInt()` metódusa által megkapja a beírt számot, a standard `System.out.println()` metódus pedig megjeleníti a képernyőn.

Több szám bevitele esetén a számok közé alapesetben szóközöket kell beiktatni, amely jel a Java-ban felülbírálható. Az alábbi program az előző továbbfejlesztése. Több (akárhány!) egész számot vár – vesszővel elválasztva – a billentyűzetről, majd kiírja a darabszámukat és az összegüket:

```
public static void main(String[] args) {
    int sum = 0;
    Scanner sc = new Scanner(System.in);
    System.out.println("Kérek több egész számot vesszővel elválasztva!");
    String sor = sc.nextLine(); // sor beolvasása

    // darabolás
    StringTokenizer st = new StringTokenizer(sor,","); // elválasztójel a vessző!
    int db = st.countTokens(); //adatok (tokenek) száma
    while (st.hasMoreTokens()) {
        sum += Integer.parseInt(st.nextToken());
    }
    System.out.println("A beolvasott számok (" + db + " db) összege: " + sum);
}
```

Ebben az esetben a teljes input sort beolvassuk a sor nevű stringbe, majd a osztály segítségével a megadott határolójel mentén adategységekre (tokenekre) feldarabolva feldolgozzuk azt. Nézzünk egy programfutási eredményt is:

```
run:
Kérek több egész számot vesszővel elválasztva!
1,2,3,4,5
A beolvasott számok (5 db) összege: 15
BUILD SUCCESSFUL (total time: 5 seconds)
```

5.3.9.2. Karakteres fájlok kezelése

A Java a fájlokat is streamként kezeli, ezért most is importálni kell a `java.io` csomag néhány osztályát. Az I/O típusok **karakteresek** (a Java az UTF-8-as Unicode karakterkódolást használja) és **binárisak** (8 bites bájtformátum) lehetnek. A karakteres állományok kezelését három fő részre osztjuk:

- megnyitás
- műveletek
- lezárás

A megnyitás művelete során valamely stream osztály objektuma jön létre (példányosítással), amelyhez hozzárendeljük a megnyitandó állományt. A fájlkezelő műveleteket `try-catch-finally` kivételkezelő blokkszerkezetbe kell foglalni, mert sok probléma adódhat használatuk során (pl. nem létező állomány, betelt háttértár, stb.). A következő példaprogramok mindegyikében lesz ilyen szerkezet, de a jobb áttekinthetőség miatt e forráskódok csak a minimálisan kötelező kivételkezelést tartalmazzák. A kivételkezelésről bővebben a 14. heti tananyagban lesz szó.

5.3.9.2.1. Karakteres fájlok olvasása

Karakteres állományok olvasására a `FileReader` osztály `read()` metódusa áll rendelkezésre. A metódus használatához implementálni kell a `FileNotFoundException` (a fájl nem található) kivételkezelő osztályt, és ennek ősét, az `IOException` (általános I/O hiba) osztályt is.

A következő példa egy szöveges állományból egyenként beolvassa a karaktereket, majd megjeleníti őket a képernyőn. Ha nem jelölünk ki pontos elérési utat, akkor a fájlnak a Java projekt főkönyvtárában kell lennie. A fájl végét a -1-es értékkel érzékeli a `read()` metódus.

```
public static void main(String[] args) {
    try {
        FileReader fr = new FileReader("szöveg.txt"); // fájl megnyitása
        fr.skip(1); // Az UTF-8 azonosító átugrása
        int c;
        while ((c = fr.read()) != -1) { // olvasás a fájl végéig
            System.out.print((char)c); // beolvasott karakter kiírása a képernyőre
        }
        fr.close(); // a fájl bezárása
    }
    catch (IOException ioe){ // általános I/O hiba
        System.err.println(ioe.getMessage());
    }
}
```

A forrásállomány (`szöveg.txt`) az UTF-8-as Unicode kódolású szöveg mentését is lehetővé tevő Jegyzettömb programmal készült. A 2. és 3. sorban használt kis- és nagybetűs tesztszövegben („árvíztűrő tükörfúrógép”) minden magyar ékezetes betű szerepel, így könnyen ellenőrizhető a fájlkezelő metódusok betűhelyes működése. A forrásállomány Jegyzettömb által mutatott tartalma:

```
szöveg eleje
árvíztűrő tükörfúrógép
ÁRVÍZTŰRŐ TÜKÖRFÚRÓGÉP
Negyedik sor
end
```

A képernyőn megjelenő szöveg:

```
run:
szöveg eleje
árvíztűrő tükörfúrógép
ÁRVÍZTŰRŐ TÜKÖRFÚRÓGÉP
Negyedik sor
end
BUILD SUCCESSFUL (total time: 0 seconds)
```

5.3.9.2.2. Karakteres fájlok írása

Karakteres állományok írására a `FileWriter` osztály `write()` metódusa áll rendelkezésre. Ehhez a metódushoz az IOExcception (általános I/O hiba) osztályt kötelező implementálni.

Az alábbi program egy string típusú változóban tárolt kétsoros szöveget – karakterenként feldolgozva – ír ki egy szöveges állományba. Ha nem jelölünk ki pontos elérési utat, akkor a fájl a Java projekt főkönyvtárában jön létre. Mivel könnyen lekérdezhető a string hossza, a karakterek feldolgozása nem okoz gondot.

```
public class Main {
    public static void main(String[] args) {
        String s = "árvíztűrő tükörfúrógép"+"\r\n"+"ÁRVÍZTŰRŐ TÜKÖRFÚRÓGÉP";
        try {
            FileWriter fw = new FileWriter("mentés.txt"); // fájl megnyitása írásra
            for ( int i = 0 ; i < s.length(); i++) {
                int c = (int)s.charAt(i); // karakter beolvasása a stringből
                fw.write(c); // beolvasott karakter fájlba írása
            }
            fw.close(); // a fájl bezárása
        }
        catch (IOException ioe){ // általános I/O hiba
            System.err.println(ioe.getMessage());
        }
    }
```

Nézzük a létrejött állomány (`mentés.txt`) tartalmát! Szerencsére a fájlba írás karakterkódolásával semmi gond, és a Jegyzettömb is helyesen jeleníti meg:

```
árvíztűrő tükörfúrógép
ÁRVÍZTŰRŐ TÜKÖRFÚRÓGÉP
```

5.3.9.3. Bináris fájlok feldolgozása

A bináris adatokat tartalmazó állományok kezelését a `FileInputStream` és a `FileOutputStream` osztályok biztosítják. Mindkét osztályt importálni kell a `java.io` csomagból. Metódusaik a szokásos `read()`, `write()` és `close()`. Használatuk hasonló a karakteres állományoknál látottakhoz.

Hozzunk létre egy `binary.dat` nevű bináris állományt! Bájtjainak értéke decimálisan 65-től 90-ig terjedjen!

```
public static void main(String[] args) {
    try {
        FileOutputStream fstr = new FileOutputStream("binary.dat");
        for (int i = 65; i <= 90; i++) {
            fstr.write(i);
        }
        fstr.close();
    }
    catch (IOException ioe) {
        System.err.println(ioe.getMessage());
    }
}
```

Bővítsük előző programunk Main metódusát! A keletkezett állományt olvassuk be, és írjuk ki a bájtjai által reprezentált karaktereket a képernyőre szóközzel elválasztva! Ne feledjük, hogy az importáló utasításokat bővíteni kell a `java.io.FileInputStream` osztállyal!

```
try {
    FileInputStream fis = new FileInputStream("binary.dat");
    int b;
    while ((b = fis.read()) != -1) {
        System.out.print((char)b + " ");
    }
    System.out.println();
    fis.close();
}
catch (IOException ioe) {
    System.err.println(ioe.getMessage());
}
```

A képernyőn a várt eredmény látható:

```
run:
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
BUILD SUCCESSFUL (total time: 0 seconds)
```

5.3.9.4. Szöveges állományok kezelése

Szöveges állományok feldolgozása gyakran nem karakterenként, hanem nagyobb részenként, pl. soronként történik. Ehhez a Java ún. pufferelt fájlkezelési támogatást nyújt. Külön osztályok támogatják a byte- és string-szintű műveleteket:

Byte-szintű osztályok:

- `BufferedInputStream` (olvasás)
- `BufferedOutputStream` (írás)
- `PrintStream` (írás)

String-szintű osztályok:

- `BufferedReader` (olvasás)
- `PrintWriter` (írás)

Számunkra most a stringek kezelése fontosabb, ezért nézzünk rá egy egyszerű példát! Írjunk ki egy sima szöveges állományba (`text.txt`) néhány UTF-8-as kódolású sort, majd az állományt visszaolvasva jelenítsük meg azokat a képernyőn is! Írás a `PrintWriter` osztály `println()` metódusával:

```
// szöveges állomány létrehozása
try {
    System.setProperty("file.encoding", "UTF8"); //karakterkódolás beállítása
    PrintWriter pw = new PrintWriter("text.txt");
    pw.println("Első sor");
    pw.println("árvíztűrő tükörfúrógép");
    pw.println("ÁRVÍZTŰRŐ TÜKÖRFÚRÓGÉP");
    pw.println("Utolsó sor");
    pw.close();
}
catch (IOException ioe) {
    System.err.println(ioe.getMessage());
}
```

Olvasás a `BufferedReader` osztály `readLine()` metódusával:

```
// szöveges állomány beolvasása és kiírása a képernyőre
try {
    System.setProperty("file.encoding", "UTF8"); //karakterkódolás beállítása
    BufferedReader br = new BufferedReader(new FileReader("text.txt"));
    String sor;
    while ((sor = br.readLine()) != null) {
        System.out.println(sor);
    }
    br.close();
}
catch (IOException ioe) {
    System.err.println(ioe.getMessage());
}
```

Figyeljük meg, hogy a `BufferedReader` osztály nem tud egy fájlt közvetlenül megnyitni, hanem csak a `FileReader` osztály egy példányát. Az eredmény meggyőző:

```
run:
Első sor
árvíztűrő tükörfúrógép
ÁRVÍZTŰRŐ TÜKÖRFÚRÓGÉP
Utolsó sor
BUILD SUCCESSFUL (total time: 0 seconds)
```

5.3.9.5. Állományok közvetlen elérése

Az előző alfejezetek adatfolyam-kezelő műveletei mind soros hozzáférésűek voltak, tehát az állományok írása ill. olvasása az elejüktől a végükig sorban történt. Azonban lehetőség van arra is, hogy egy fájl tartalmához közvetlenül is hozzáférhessünk. Ezt a `java.io` csomag `RandomAccessFile` osztálya biztosítja. Ez az osztály egyszerre alkalmas olvasásra és írásra is, csupán a példányosítás során egy paraméterrel jelezni kell, hogy milyen műveletet kívánunk alkalmazni az objektumra:

- csak olvasás: `RandomAccessFile("input.txt", "r")`
- olvasás+írás: `RandomAccessFile("input.txt", "rw")`

Az így megnyitott állományokon használhatók a szokásos `read()` és `write()` metódusok, valamint a közvetlen elérést támogató metódusok:

- `void seek(long pos)`: megadott pozícióra ugrás a fájl elejétől
- `int skipBytes(int n)`: aktuális pozíció mozgatása n bájttal
- `long getFilePointer()`: aktuális pozíció lekérdezése

Fontos tudni, hogy a `read()` és `write()` metódusok meghívása a fájlmutató (aktuális pozíció a fájlon belül – file pointer) értéket automatikusan megnöveli 1 egységgel, tehát nem szükséges manuálisan léptetni. Ezek a metódusok és a `close()` (fájl bezárása) is megkövetelik az `IOException` kivételkezelő osztály használatát.

Hozzunk létre egy állományt (`random.txt`), amely az angol ABC nagybetűit tartalmazza!

```
// random fájl létrehozása, és szekvenciális feltöltése
try {
    RandomAccessFile raf = new RandomAccessFile("random.txt", "rw");
    for (int i = 65; i <= 90; i++) {
        raf.write(i);
    }
    raf.close();
}
catch (IOException ioe) {
    System.err.println(ioe.getMessage());
}
```

Majd minden ötödik karaktert – a fájlban elfoglalt pozíciójukkal együtt – jelenítsük meg a képernyőn!

```
// random fájl pointerének mozgatása, és olvasás fájlból
try {
    RandomAccessFile raf = new RandomAccessFile("random.txt", "r");
    long hossz = raf.length();
    long poz = 4; // 5. pozíció (a pointer értéke 0-tól kezdődik!)
    while (poz < hossz) {
        raf.seek(poz);
        System.out.println(raf.getFilePointer()+1+ " - "+(char)raf.read());
        poz += 5;
    }
    raf.close();
}
catch (IOException ioe) {
    System.err.println(ioe.getMessage());
}
```

A `raf.length()` metódussal lehet lekérdezni a fájl méretét. Figyeljük meg, hogy a kiírás során a `getFilePointer()` pozíciólekérdező metódus megelőzi az olvasó metódust, így a helyes értéket adja, de mégis meg kell növelni az értékét 1-gyel, mivel a pozíciók számozása 0-tól kezdődik. Az eredmény:

```
run:
5 - E
10 - J
15 - O
20 - T
25 - Y
ABCDeFGHIjKLMNoPQRStUVWXyZ
BUILD SUCCESSFUL (total time: 0 seconds)
```

A kimenet utolsó sorát az alábbi kiegészítés állítja elő. A fájl minden ötödik karakterét cseréljük le annak kisbetűs változatára, majd jelenítsük meg a fájl összes karakterét!

```
// random fájlba írás
try {
    RandomAccessFile raf = new RandomAccessFile("random.txt", "rw");
    long hossz = raf.length();
    long poz = 4; // 5.(kezdő) pozíció (a pointer értéke 0-tól kezdődik!)
    int kód;
    while (poz < hossz) {
        raf.seek(poz); // ugrás minden 5. pozícióra
        kód = raf.read(); // karakterkód beolvasása
        raf.seek(poz); // az olvasás miatt újrapozícionálni kell
        raf.write(kód+32); // kisbetűs karakter kiírása
        poz += 5;
    }

    // eredmény kiírása a képernyőre is
    raf.seek(0); // ugrás a fájl elejére
    while ((kód = raf.read()) != -1) {
        System.out.print((char)kód);
    }
    System.out.println("");

    raf.close();
}
catch (IOException ioe) {
    System.err.println(ioe.getMessage());
}
```

A betűcserét végrehajtó ciklus minden ötödik betűre pozícionál, majd az ott található betű kódjához 32-t adva éri el, hogy kisbetű lesz belőle. A kiolvasás eltolja az aktuális pozíciót, ezért szükséges visszalépni egyet. A betűk képernyőn való megjelenítését megelőzi a fájl kezdőpozíciójába ugrás, majd egy sima (fájlvégjel-figyeléses) feldolgozó ciklus kiírja a karaktereket.

5.3.10. ÉRETTSÉGI FELADATOK MEGOLDÁSA

Az alábbiakban két emelt szintű érettségi feladatot fogunk megoldani. Az eddig tanultak elegendő nyelvi eszközt biztosítanak minden részfeladat megoldásához [6]. Nagyban megkönnyíti a munkánkat az a fontos körülmény, hogy sem az inputállományok, sem a felhasználó által bevitt adatok helyességét ill. érvényességét nem kell ellenőrizni. Így lényegesen csökken a kódolási idő, de meg kell jegyezni, hogy a valóságban az ilyen programok használhatósága nagyban korlátozott.

A feladatok megoldása fájlműveleteket is igényel, amelyek metódusai megkövetelik a kivételkezelést, ezért a programok main metódusának fejében a `throws IOException` záradékkal a kivételeket „továbbdobjuk” a Java futtató rendszere felé [7].

5.3.10.1. Foci

Első érettségi feladatunk lényege a következő: egy labdarúgó-bajnokság mérkőzéseinek adatait tartalmazó fájlt kell feldolgoznunk, és az adatok alapján a megadott kérdésekre válaszolnunk.

Mint majd látjuk, a program sok metódusa igényli a `java.io` és a `java.util` osztályok alosztályait, amelyek alapértelmezésben nincsenek implementálva, ezért a program elején importálnunk kell őket:

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

1. feladat:

Olvassuk be a `meccs.txt` fájlban található adatokat! Tárolásukra két tömböt fogunk használni, mivel az adatok egy része egész számokat, másik része szöveget tartalmaz. Alkalmazzuk a „tömb a tömbben” módszert, mivel adategységenként (az inputfájl egy-egy sora) 5 szám- és 2 szövegadat tárolásáról kell gondoskodnunk. A tömböket statikus osztályváltozókként definiáljuk, mivel a későbbiekben egy saját metódussal – `MaxMin()` – fogunk hivatkozni rájuk.

```
static int[][] T1; // mérkőzések számadatai
static String[][] T2; // mérkőzések szöveges adatai
```

A megoldás a mérkőzések számának beolvasásával kezdődik, majd soronként feldolgozzuk az inputfájlt. Ez a `StringTokenizer` osztály metódusainak segítségével (a sort a szóközök mentén feldarabolva) könnyen megvalósítható. Figyeljük meg, hogy a tömbök indexelése 0-tól kezdődik!

```
// 1. feladat
int i = 0;
int msz = 0; // mérkőzések száma
BufferedReader br = new BufferedReader(new FileReader("meccs.txt"));
msz = Integer.parseInt(br.readLine());
T1 = new int [msz][6];
T2 = new String [msz][2];
String sor;
while ((sor = br.readLine()) != null) {
    StringTokenizer st = new StringTokenizer(sor," ");
    for (int j = 0; j<5; j++) {
        T1[i][j] = Integer.parseInt(st.nextToken());
    }
    T1[i][5] = 0; //statisztikához
    T2[i][0] = st.nextToken(); // hazai csapat neve
    T2[i][1] = st.nextToken(); // vendégcsapat neve
    i++;
}
br.close();
```

A `nextToken()` metódus stringet ad vissza, ezért a számoknál a konvertáláshoz igénybe kell venni az `Integer` osztály `parseInt()` metódusát.

2. feladat:

Egy forduló sorszámát bekérve írjuk ki a képernyőre az adott forduló mérkőzéseinek adatait a megadott formában.

```
System.out.println("2. feladat:");
Scanner sc = new Scanner(System.in);
System.out.println("Kérem a forduló számát!");
int fsz = sc.nextInt();
for (int j = 0; j < msz; j++)
    if (T1[j][0] == fsz)
        System.out.println(T2[j][0]+"-"+T2[j][1]+": "+T1[j][1]+"-"+T1[j][2]
                +" ("+T1[j][3]+"-"+T1[j][4]+")" );
System.out.println("");
```

A billentyűzet inputhoz jól használható a `Scanner` osztály. A kiírás formátumának beállítása aprólékos munkát igényel, de szerencsére a `System.out.println()` metódus támogatja a szám- ill. szövegváltozók vegyes használatát.

3. feladat:

Azokat a csapatokat (és a forduló sorszámát) kell kiíratni a képernyőre, amelyek megfordították a mérkőzés állását, tehát a félidőben még vesztésre állva a mérkőzést a végén megnyerték.

```
System.out.println("3. feladat:");
for (int j = 0; j < msz; j++)
    if ((T1[j][3] < T1[j][4]) && (T1[j][1] > T1[j][2]))
        System.out.println(T1[j][0]+" "+T2[j][0]);
    else if ((T1[j][4] < T1[j][3]) && (T1[j][2] > T1[j][1]))
        System.out.println(T1[j][0]+" "+T2[j][1]);
System.out.println("");
```

Az összetett feltételvizsgálat oka az, hogy mind a hazai-, mind a vendégcsapat szempontjából meg kell vizsgálnunk a mérkőzések félidei- és végeredményét.

4. feladat:

Egy csapat nevét kell bekérni a felhasználótól, majd (ez esetben egy `csn` változóban) eltárolva felhasználni a következő feladatok megoldásához.

```
System.out.println("4. feladat:");
sc = new Scanner(System.in);
System.out.println("Kérem egy csapat nevét!");
String csn = sc.next();
System.out.println("");
```

5. feladat:

Az adott csapat által lőtt és kapott gólokat kell összesítve megjeleníteni, tehát a lekérdezés szűrőfeltétele a csapat neve. Ne feledjük, hogy egy csapat hazaiként és vendégként is szerepelhet egy mérkőzésen, és csak a mérkőzések végeredményét kell összesíteni!

```
System.out.println("5. feladat:");
int lőtt = 0;
int kapott = 0;
for (int j = 0;j < msz; j++)
    if  (csn.matches(T2[j][0])) {
        lőtt += T1[j][1];
        kapott += T1[j][2];
    } else if (csn.matches(T2[j][1])) {
        lőtt += T1[j][2];
        kapott += T1[j][1];
    }
System.out.println("lőtt: "+lőtt+", kapott: "+kapott);
System.out.println("");
```

Figyeljük meg a `string matches()` metódusát! Nagyon jól használható összehasonlításokhoz.

6. feladat:

Feladatunk az adott csapatról meghatározni, hogy otthon melyik fordulóban kapott ki először, és ki volt a legyőzője. Azt is jelezni kell, ha veretlen maradt.

```
System.out.println("6. feladat:");
boolean kikapott = false;
int j = 0;
while ((! kikapott) && (j < msz)) {
    if (csn.matches(T2[j][0]) && (T1[j][1] < T1[j][2]))
        kikapott = true;
    j++;
}
if (! kikapott)
    System.out.println("A csapat otthon veretlen maradt.");
else
    System.out.println(T1[j-1][0]+" "+T2[j-1][1]);
System.out.println("");
```

A mérkőzések adatait tartalmazó tömbök feldolgozását végző `while` ciklus csak addig fut, amíg nem talál egy otthoni vereséget az adott csapatnál. Ezt a kikapott nevű logikai változó használatával érjük el.

7. feladat:

Statisztikát kell készítenünk a bajnokságban előforduló végeredményekről, amelyet egy fájlban kell eltárolnunk. Fontos feltétel, hogy a fordított eredményeket egyezőknek kell tekinteni és a nagyobb számot mindig előre kell írni.

A feladat megoldásához érdemes írni egy saját metódust, amely a végeredmények gólszámából egy olyan számot állít elő, amelynek tízes helyiértékét a nagyobb-, egyes helyiértékét pedig a kisebb gólszámok adják. Így a végeredményeket egységesítve sokkal könnyebb a feldolgozásuk.

```
public static int MaxMin(int z) {
    return Math.max(T1[z][1], T1[z][2]) * 10 + Math.min(T1[z][1], T1[z][2]);
}
```

A végeredmények számának tárolására a `T1` tömb egy üres oszlopa szolgál. A tömb az összes végeredmény-fajtát tartalmazza, tehát külön adatstruktúrát nem szükséges létrehozni hozzájuk. Az algoritmus lényege az, hogy az egyes végeredményeket – függetlenül attól, hogy milyen sorrendűek (pl. 2-1 vagy 1-2) – az első eredmény-előfordulás helyén tárolva számláljuk meg. A kimeneti fájl létrehozásához kiválóan alkalmas az egyszerűen használható `PrintWriter` osztály.

```
// 7. feladat
PrintWriter pw = new PrintWriter("stat.txt");
for (i = 0; i < msz; i++)
    for (int k = 0; k <= i; k++)
        if (MaxMin(i) == MaxMin(k)) {
            T1[k][5]++;
            break;
        }
// kiíratás
for (i = 0; i < msz; i++)
    if (T1[i][5] != 0)
        pw.println(Math.max(T1[i][1], T1[i][2]) + "-" + Math.min(T1[i][1], T1[i][2])
                + ": " + T1[i][5] + " darab");
pw.close(); // állomány zárása
```

Figyeljük meg a második – beágyazott – `for` ciklus felépítését, amely az eredmények eltárolását végzi! Mindig csak addig fut – a tömb elejétől kezdve a vizsgálatot –, amíg nem talál egy olyan végeredményt, mint amilyet el akar tárolni. A kiíratásnál arra kell figyelni, hogy a teljes tömböt fel kell dolgozni, mivel egy eredmény akár a tömb utolsó sorában is szerepelhet! A gólszámok sorrendhelyes megjelenítéséhez a `Math` osztály `max()` és `min()` függvényei hathatós segítséget nyújtanak.

Összességében a feladatról elmondható, hogy – bár egyszerű fájlkezelő műveleteket igényel, de – néhány részfeladata magabiztos algoritmizáló képességet feltételez, és az adatok tárolásához a tömb adatstruktúra alapos ismerete feltétlenül szükséges.

5.3.10.2. Robot

Második feladatunk egy kis robot programozásáról szól. Négy irányba tud mozogni az egybetűs – égtájakat, mint irányokat jelölő – E, D, K és N parancsok hatására. Feladataink az ilyen utasításokból álló programsorok feldolgozásával kapcsolatosak.

Most is szükségünk van az `io` és `util` osztályokra, valamint egy számérték formázott megjelenítése miatt a `DecimalFormat` osztályra is:

```
import java.io.*;
import java.text.DecimalFormat;
import java.util.*;
```

1. feladat:

Első lépésként olvassuk be a `Program.txt` állományt! Első sora a programok számát tartalmazza, a többi sor pedig egy-egy programot. Utóbbiakat egy egyszerű string típusú `P` tömbben tároljuk.

```
int i = 0;
int psz = 0; // programok száma
String[] P; // programok utasításai
System.out.println("1. feladat");
BufferedReader br = new BufferedReader(new FileReader("program.txt"));
psz = Integer.parseInt(br.readLine());
P = new String [psz];
String sor;
while ((sor = br.readLine()) != null) {
    P[i] = sor;
    i++;
}
br.close();
```

2/a. feladat:

Az összetett feladat egy utasítássor számának bekérésével kezdődik, majd el kell döntenünk az adott sorról, hogy egyszerűsíthető-e. Akkor egyszerűsíthető, ha közvetlenül egymás után két olyan lépés van előírva, amelynek eredményeképpen a robot egy helyben marad (pl. ED, NK). A kérdéses betűpárok megtalálásához a string `contains()` metódusát használjuk fel, amely igaz értéket ad, ha betűpárok bárhol előfordulnak a vizsgált stringben.

```
System.out.println("2. feladat:");
Scanner sc = new Scanner(System.in);
System.out.println("Kérem az utasítássor sorszámát!");
int us = sc.nextInt()-1;

// 2.a.
if (P[us].contains("ED") || P[us].contains("DE") ||
    P[us].contains("KN") || P[us].contains("NK"))
    System.out.println("egyszerűsíthető");
else
    System.out.println("nem egyszerűsíthető");
```

2/b. és 2/c. feladat:

Ezt a két részfeladatot érdemes egyszerre megoldanunk, mert mindkettő ugyanazt a ciklust igényli az utasítások feldolgozása során, így futási időt takaríthatunk meg.

A b. feladatban azt kell meghatároznunk, hogy a kiválasztott utasítássor végrehajtása után legkevesebb hány lépésben lehetne a robotot a kiindulási helyére visszajuttatni. Ehhez érdemes a lépéseket „lejátszani” úgy, hogy közben külön számoljuk a két tengelyen (észak-dél ill. kelet-nyugat) megtett lépéseket (`ed` és `kn` változók). Az egyik lépést pozitív, az ellentétes irányú párját negatív értékkel vesszük figyelembe. A végpozíciót elfoglalva az összesített lépésszámok abszolút értéke adja a megoldást.

```
// 2.b. és 2.c.
int ed = 0; // az észak-déli tengely lépésszáma
int kn = 0; // a keleti-nyugati tengely lépésszáma
double max_távolság = 0; // a robot maximális távolsága a startponttól
double akt_távolság = 0; // a robot aktuális távolsága a startponttól
int max_lépés = 0; // a maximális távolsághoz tartozó lépések száma
for (i = 0; i < P[us].length(); i++) {
    // 2.b.
    switch (P[us].charAt(i)) {
        case 'E': ed++; break;
        case 'D': ed--; break;
        case 'K': kn++; break;
        case 'N': kn--;
    }
```

A c. pont arra kíváncsi, hogy hány lépés után kerülünk a kiindulási ponttól a legtávolabbra, és ekkor mennyi ez a távolság. Utóbbi meghatározásához a Pitagorasz-tételt használjuk fel, és az eredményt valós számtípusú változóban tároljuk.

```
    // 2.c.
    akt_távolság = Math.sqrt(Math.pow(ed,2)+Math.pow(kn,2));
    if (akt_távolság > max_távolság) {
        max_lépés = i+1;
        max_távolság = akt_távolság;
    }
}
```

Figyeljük meg, hogy a négyzetgyökvonó- és a hatványozó függvények a `Math` osztály metódusai! Az eredmény kiírásánál formázott megjelenítést kell alkalmaznunk, amelyet a `DecimalFormat` osztály `format()` metódusa biztosít. A maszk # és 0 jele egy-egy számjegyet jelöl, és az utóbbi kötelező megjelenítést ír elő. Így lehet 3 tizedes pontosságú számot kiíratni.

```
// 2.b.
System.out.println(Math.abs(ed) + " lépést kell tenni az ED, "
                   + Math.abs(kn) + " lépést a KN tengely mentén.");
// 2.c.
DecimalFormat htf = new DecimalFormat("###.000");
System.out.println("A robot " + max_lépés + " lépést követően volt a "
                    + "legmesszebb, amikor is ");
System.out.println("a központtól mért távolsága légvonalban "
                    + htf.format(max_távolság) + " egység volt.");
System.out.println("");
```

3. feladat:

A kis robot tevékenységei (elindulás, lépés, irányváltás) különféle mértékű energiafelhasználással járnak. Meg kell határoznunk, hogy mely programok végrehajtása igényel maximum 100 egységnyi energiafelhasználást.

```
System.out.println("3. feladat:");
int akt_en = 0;
for (i = 0; i < psz; i++) {
    akt_en = 2 + P[i].length(); // fix energia: indulás + lépések száma
    for (int k = 0; k < P[i].length()-1; k++)
        if (P[i].charAt(k) != P[i].charAt(k+1))
            akt_en += 2;
    if (akt_en <= 100)
        System.out.println("A(z) " + (i+1) + ". utasítássor " + akt_en
                           + " egység energiát fogyaszt.");
}
System.out.println("");
```

Figyeljük meg, hogy minden program rendelkezik egy fix energiaigénnyel, ami az indulásból és a megtett lépések számából adódik. Ezeken kívül már csak az irányváltásokat kell figyelni, amit két szomszédos utasítás különbözőségéből könnyen meghatározhatunk. Az utasítássor egy-egy elemét a `String` osztály `charAt()` metódusával könnyen meghatározhatjuk. Mivel mindig az aktuális utasítás kódját hasonlítjuk a sorban következőhöz, így a feldolgozó ciklus csak az utolsó előtti elemig fut!

4. feladat:

Új formátumúra kell alakítanunk és egy fájlba kiírnunk az utasítássorokat. A konverzió lényege az, hogy az egymás után ismétlődő parancsokat azok számával jelölve rövidítsünk az utasítássoron. Pl. az EEEDKNNNN sorból állítsuk elő a 3EDK4N (hivatalos nevén futáshossz-tömörítésű) utasítássort. Az egyedüli utasításokat változatlanul kell leírni.

```
System.out.println("4. feladat");
PrintWriter pw = new PrintWriter("ujprog.txt");
String tsor; // technikai sor
int usz;
for (i = 0; i < psz; i++) {
    usz = 1; //utasításszámláló
    tsor= P[i]+"*"; // zárókarakter hozzáadása (csak technikai okokból)
    for (int k = 1; k <= tsor.length()-1; k++)
        if (tsor.charAt(k) == tsor.charAt(k-1))
            usz++;
        else {
            if (usz > 1) pw.print(usz);
            pw.print(tsor.charAt(k-1));
            usz = 1;
        }
    pw.println(""); // soremelés
}
pw.close(); // állomány zárása
```

A kimeneti fájl létrehozására a `PrintWriter` osztály most is kézenfekvő megoldás. Az átkódolás menete viszont már nem olyan egyszerű! Hozzunk létre minden utasítássorból egy „technikai sort”, amely egy * karakterrel való kibővítéssel jön létre. Ez ahhoz kell, hogy a következő – a vizsgálatot a 2. karaktertől indító! – ciklus az adott karaktert az előzőhöz hasonlítva megszámlálhassa (az `usz` változóban) az egymás után következő azonos utasításkaraktereket.

5. feladat:

A cél egy új formátumú utasítás visszaalakítása a régi formátumra. A visszakódolandó utasítássort a felhasználótól kérjük be, amelyet az `uts` változóban helyezünk el. Figyelnünk kell arra is, hogy az ismétlődések száma maximum 200 lehet, azaz akár 3 számjegyből is állhat! Ez a körülmény alapvetően meghatározza az átalakító algoritmus felépítését, ugyanis az ismétlődésszámot adó karaktereket is gyűjtenünk kell (`ism_kar` változó).

```
System.out.println("5. feladat:");
sc = new Scanner(System.in);
System.out.println("Kérek egy új formátumú utasítássort!");
String uts = sc.next();
int ism_szám = 0; // betűismétlések száma
String ism_kar = "0"; // számkarakterek gyűjtője
for (i = 0; i < uts.length(); i++) {
    if (uts.substring(i,i+1).matches("E|D|K|N")) {
    ism_szám = Integer.parseInt(ism_kar);
        if (ism_szám == 0) ism_szám = 1;
        for (int k = 1; k <= Math.min(ism_szám, 200); k++)
            System.out.print(uts.charAt(i));
        ism_kar = "0";
    } else
          ism_kar += uts.charAt(i);
}
System.out.println("");
```

Figyeljük meg, hogy az utasítássor elemeinek „darabolásához” a `charAt()` helyett a `substring()` metódust használjuk, ugyanis az utóbbinak van egy olyan `matches()` metódusa, amely segítségével nagyon tömör kóddal eldönthetjük, hogy az adott karakter utasításkód-e. A maszkban szereplő `|` jel a logikai vagy megfelelője. A szöveg-szám konverzióhoz most is az `Integer` osztály `parseInt()` metódusát használjuk. Abban az esetben, ha az ismétlődések száma nulla, akkor gondoskodnunk kell arról, hogy 1 legyen, mivel a kiíró programrész ciklusának legalább egyszer le kell futnia az utasításkarakter megjelenítéséhez. A 200-as határ vizsgálatához gyors, kényelmes és elegáns megoldást kínál a `Math` osztály `min()` függvénye.

5.3.11. AZ OBJEKTUM-ORIENTÁLT PARADIGMA

5.3.11.1. Osztály és objektum, konstruktor

A Java nyelv alapját az objektum-orientált paradigma (módszer, alapelv, elmélet) képezi. Ez azt jelenti, hogy a korábbi programozási nyelvek által használt strukturális, moduláris programozási módszert a Java nyelv esetében felváltotta egy új szemléletmód, a megoldandó problémák objektum-orientált módon való megközelítése. E módszer központi elemei az **objektumok**, amelyek segítségével a megoldandó problémát leírjuk, modellezzük. Egy Java program lényegét az objektumok egymás közötti együttműködése, kommunikációja alkotja [8]. Nézzük, mi is lehet objektum? Pl. egy autó vagy egy ember.

Minden ilyen objektum két fő jellemzővel rendelkezik:

- **tulajdonságokkal** (attribútum, adattag, változó) – pl. az autó színe, aktuális sebességfokozata, vagy az ember neve, testmagassága, aktuális tevékenysége
- **viselkedési jellemzőkkel** (metódus, módszer) – pl. az autó sebességet vált, fékez, ill. az ember beszél, dolgozik

A Java nyelv e két jellemzőt egységesen kezeli:

Az azonos adattagokkal és metódusokkal rendelkező objektumokat egy közös ún. **osztályba** soroljuk. Az objektumok mindig egy ilyen osztály tagjai, példányai. Pl. az Autók nevű osztály egy példánya lehet a HHT-376 frsz.-ú Opel Astra, vagy Kis István tanuló az Ember osztálynak. Egy objektum létrehozását **példányosításnak** is nevezzük, amelyhez a „tervrajzot” az osztály adja.

Egy objektumpéldány létrehozását (inicializálását) az adott osztály ún. **konstruktora** végzi el. Ennek során az újonnan létrejövő objektum adattagjai kezdőértéket kapnak. A konstruktort a `new` operátorral hívjuk meg, amely memóriaterületet foglal le az újonnan létrehozandó objektumpéldány számára.

Minden osztálynak van legalább egy konstruktora (neve ugyanaz, mint az osztályé), amelyet ha mégsem hozunk létre, akkor a Java rendszer megteszi ezt helyettünk. Ez a konstruktor viszont paraméter nélküli lesz, így a létrejövő objektumnak nem adhatunk közvetlenül kezdőértéket. Egy osztálynak több konstruktora is lehet, amelyek azonban a paramétereik száma ill. típusa alapján megkülönböztethetőknek kell lenniük.

Az adattagok (változók) és a metódusok is lehetnek **osztály-** vagy **példányszintűek**. Pl. az autók színe különbözik, tehát példányváltozókban tároljuk őket. De ha minden autó 5 sebességes, akkor ezt a jellemzőt elég osztályszinten definiálni. Ilyen speciális osztályszintű változó lehet az osztály éppen aktuális objektumpéldányainak száma is.

Azt, hogy egy objektumpéldány egy adott osztálynak tagja-e, az `instanceof` – logikai értéket adó – operátorral kérdezhetjük le:

```
(autó1 instanceof Autó)
```

5.3.11.2. Üzenet, interfész, bezárás és láthatóság

Egy Java program objektumai üzenetküldéssel kommunikálnak egymással, amely során az egyik egy konkrét feladat elvégzésére kéri a másikat. Ez a megszólított objektum egy – a kezdeményező által látható, elérhető – metódusának meghívását jelenti. Egy objektum azon metódusainak összességét, amelyeken keresztül meghívható, az objektum **interfészének** nevezzük. Minden objektumnak van interfésze, egyébként nem lenne értelme a létezésének.

Az objektumok tervrajzát adó osztályok definíciójánál gondoskodhatunk arról, hogy a majdan létrejövő objektumpéldányok mely **adattagjai** legyenek láthatók, ill. mely **metódusai** és **konstruktorai** legyenek meghívhatók más objektumok által, valamint melyek legyenek csak az adott objektum által hozzáférhetők. Ezt az eszközrendszert nevezzük **bezárásnak**.

A Java nyelvben a bezárási szinteket ún. láthatósági módosítókkal állíthatjuk be. Fajtái:

- publikus (public)
- védett (protected)
- privát (private)
- módosító nélküli (alapértelmezett, csomagszintű láthatóság)

Az alábbi táblázat mutatja a módosítók hatását (● látható, ● nem látható):

Elérési szintek (láthatóság)				
Módosító	**Osztály**	**Csomag**	**Leszármazott**	**Összes osztály**
`public`	●	●	●	●
`protected`	●	●	●	●
módosító nélküli	●	●	●	●
`private`	●	●	●	●

Az első oszlop (Osztály) azt mutatja, hogy maga az osztály elérheti-e az adott módosítóval megjelölt adattagjait metódusait. Természetesen minden esetben igen. A második oszlop azt jelzi, az eredeti osztállyal azonos csomagban lévő más osztályok (függetlenül a szülői kapcsolatoktól) elérhetik-e a tagokat. A harmadik oszlop mutatja, hogy az adott osztály leszármazottai (függetlenül attól, hogy mely csomagban vannak) látják-e tagokat. A negyedik oszlopban az összes többi osztály számára biztosított elérhetőség látható.

Az interfész és a bezárás eszközével biztosítható az objektumok integritása, vagyis adattagjaik ellenőrzött körülmények közötti megváltoztathatósága.

5.3.11.3. Öröklődés és polimorfizmus

Egy osztály legegyszerűbben adattagjainak és metódusainak felsorolásával hozható létre. Azonban az objektum-orientált paradigma lehetőséget ad egy másik, hatékonyabb módszerre is, az **öröklődésre**. Az öröklődés az újrafelhasználhatóságot szem előtt tartva arra ad lehetőséget, hogy már meglévő (**szülő-, ős-**) osztályból kiindulva hozzunk létre új (**gyermek-, leszármazott-, al-**) osztályt. Az öröklés két osztály között fennálló olyan kapcsolat, amely során a leszármazott osztály rendelkezik a szülő osztály összes tulajdonságával (adattagjait és metódusait sajátjaként kezeli), s ezeket újabbakkal egészítheti ki. Az így létrehozott osztály is lehet más osztályok őse, így ezek az osztályok egy öröklési hierarchiába szerveződnek. Attól függően, hogy egy osztálynak egy- vagy több őse van, beszélünk **egyszeres-** ill. **többszörös** öröklődésről. A Java az egyszeres öröklődést támogatja.

A Java-ban az osztályhierarchia legfelső eleme az **Object** osztály, amelyből minden más osztály (közvetve vagy közvetlenül) származik.

Egy alosztály az örökölt metódusokat újraimplementálhatja. Ilyenkor az adott metódus ugyanolyan néven, de más, módosított (alosztályra specifikált) tartalommal kerül megvalósításra. Az ilyen metódusokat **polimorfnak** nevezzük.

Polimorfizmusról beszélünk akkor is, amikor egy osztályon belül több, azonos nevű metódus létezik. Minden ilyen metódus implementációja különbözik egymástól, amelyet a paramétereik sorrendje vagy típusa biztosít. Ekkor beszélünk egy metódus többalakúságáról.

5.3.11.4. Osztály és objektum létrehozása

Definiáljunk egy egyszerű, általánosan elérhető `Autó` nevű osztályt, amelynek legyen három megfelelő típusú – minden más osztály elől rejtett – adattagja a rendszám, a teljesítmény és az automata váltó meglétének tárolására, valamint egy osztályszintű példányszámlálója:

```
public class Autó {
    private static int példányszám;
    private String rendszám;
    private int teljesítmény;
    private boolean automata;
```

Példánkban az osztály láthatóságának módosítója a `public` kulcsszó, az adattagok elrejtését pedig a `private` biztosítja. A `static` minősítő állítja be a példányszám változót ún. osztályváltozónak.

Hozzunk létre egy olyan konstruktort, amely segítségével már az inicializáláskor kezdőértéket adhatunk az adattagoknak, ill. példányosításkor az automatikus számlálás is történjen meg! Ezt az osztálydeklaráció alábbi kiegészítésével érhetjük el:

```
public Autó(String rendszám, int teljesítmény, boolean automata) {
    Autó.példányszám++;
    this.rendszám = rendszám;
    this.teljesítmény = teljesítmény;
    this.automata = automata;
}
```

Miután van egy „működőképes” osztályunk, példányosítsuk! Ezt az `Autó` osztály `main` metódusában kezdeményezhetjük:

```
public static void main(String[] args) {
    Autó a1=new Autó(new String("HHT-376"),75,false);
    Autó a2=new Autó(new String("GER-786"),60,false);
    Autó a3=new Autó(new String("BGJ-391"),95,true);
}
```

Az osztálydeklaráció részleteivel a következő fejezetekben ismerkedünk meg.

5.3.12. OSZTÁLY ÉS CSOMAG

5.3.12.1. Osztály létrehozása

A Java nyelven való programozás az adattagokkal és metódusokkal rendelkező objektumok programozását jelenti. Nem beszélhetünk klasszikus értelemben vett eljárásokról és függvényekről, csak objektumok együttműködéséről, amely a közöttük lévő kommunikációval valósul meg [9].

Objektumot egy osztály példányosításával hozhatunk létre. Ha nem áll rendelkezésünkre megfelelő osztály, akkor nekünk kell definiálni egyet. Azaz először a „tervrajzot” kell elkészíteni, majd az alapján lehet objektumokat létrehozni.

Az osztálydefiníció általános szintaxisa a következő:

Osztály fej:

```
[módosítók] class <osztály_neve> [extends <szülő_osztály_neve>]
[implements <interfészek_neve>]
```

Osztály törzs:

```
{
  [adattag deklarációk];
  [inicializáló blokkok];
  [konstruktor deklarációk];
  [metódus deklarációk];
}
```

Az osztály fejrészének módosítói nemcsak hozzáférési szinteket takarhatnak, hanem opcionálisan egyéb attribútumokat is:

Osztály-fejrész módosítók:

[hozzáférési szint]: public | üres (csomagszintű)

[abstract]: nem példányosítható osztály, amely öröklődés kiindulópontjaként használható; legalább egy absztrakt metódust tartalmazó osztály kötelező minősítője

[final]: végleges osztály, metódusai nem módosíthatók, az öröklődés okafogyott

Az `extends` (leszármazott osztály létrehozása) és az `implements` (interfészek osztályhoz fűzése) kulcsszavak a későbbi fejezetek témáját képezik.

5.3.12.2. Adattagok

Egy objektum tulajdonságait (aktuális állapotát) az adattagjainak értékei reprezentálják. Egy adattagnak mindig van azonosítója (neve), valamint típusa, amely lehet primitív típus (pl. egész szám, string, logikai), vagy akár osztálytípus is. Az adattagok a típusuknak megfelelő értékek tárolására szolgálnak, de értéküket az objektum életciklusa során megváltoztathatják (kivéve a konstansokat).

Az adattagok lehetnek **példányszintűek**, azaz minden objektumpéldányban egyedileg létezők, valamint **osztályszintűek**, amelyek egy osztályon belül csak egyszer léteznek, és magához az osztályhoz kapcsolódnak. Utóbbi az adattag-definíciót bevezető `static` kulcsszóval deklarálható.

Egy példányszintű adattag mindig egy objektumhoz tartozik. Ha a deklarációs részben nem adunk kezdőértéket egy adattagnak, a Java automatikusan feltölti a típusának megfelelő „nullaszerű” kezdőértékkel: pl. boolean – false, számok – 0, char – nullás kódú karakter, osztály típusú adattagok – null referencia.

Az adattag neve lehet minden megengedett azonosító, amelyet szokás szerint kisbetűvel kezdünk. Két adattagnak nem lehet azonos neve egy osztályon belül.

Adattagok deklarációja:

```
[módosítók] <típus> <adattag_neve> [=<kezdőérték>][, ...];
```

Az adattagok deklarációja az osztálytörzsben, bármilyen konstruktor- ill. metódusdeklaráción kívül történik. A módosítók itt sem kizárólag hozzáférési szinteket takarhatnak, hanem más eszközöket is:

Adattag-deklarációs módosítók:

[hozzáférési szint]: `public|protected|üres (csomagszintű)|private`
[static]: osztályváltozó deklarálása
[final]: az adattag értéke végleges, konstans

Az alábbi példában definiálunk egy kezdőérték nélküli adattagokkal rendelkező `Bankbetét` nevű osztályt:

```
public class Bankbetét {
    String név;
    int betétösszeg;
    int adó;
    boolean ismétlődő;
}
```

Lássuk ugyanezt az osztályt két kezdőértékadással rendelkező adattaggal definiálva:

```
public class Bankbetét {
    String név;
    int betétösszeg;
    int adó = 20;
    boolean ismétlődő = true;
}
```

A tömörebb kódok elérése miatt megengedett, hogy egy adattag-deklaráción belül több azonos típusú adattagot is megadjunk, megjelölve a típust, majd az egymástól vesszővel elválasztott neveket felsorolva:

```
public class Bankbetét {
    String név;
    int betétösszeg, adó;
    boolean ismétlődő;
}
```

5.3.12.3. Inicializáló blokkok

Az osztály adattagjainak kezdőértékét osztály- ill. példány-inicializáló blokkban is beállíthatjuk. Az inicializáló blokkok speciális, fej nélküli metódusok, melyeket csak a blokk-képző kapcsos zárójelpárok határolnak, és `return` utasítást sem tartalmaznak. Az osztály-inicializáló blokkot a blokk előtti `static` kulcsszóval kell jelölni. Fontos tudni, hogy az inicializáló blokk csak egyszer fut le: típusától függően az osztály létrehozásakor, ill. a példányosítások alkalmával egyszer-egyszer példányonként.

5.3.12.4. Konstruktorok, példányosítás

Egy osztály példányosítása mindig valamelyik konstruktorának meghívásával történik. A konstruktor egy konkrét objektum életciklusa alatt pontosan egyszer fut le, létrehozva ezzel az objektumot, s beállítva az adattagjainak kezdeti értékét. A konstruktor lefutása után az objektum kész az üzenetek fogadására, feladatának elvégzésére.

A konstruktor neve kötött (mindig megegyezik az őt hordozó osztály nevével), így egy osztálynak csak akkor lehet több konstruktora (konstruktor túlterhelés), ha azok eltérő formális paraméterlistával rendelkeznek. A konstruktor egy speciális, típus nélküli metódusként fogható fel, amelynek nincs visszatérési értéke. Más megfogalmazásban a konstruktor egy – az osztály típusával visszatérő – névtelen metódus.

Egy osztályhoz mindig tartoznia kell legalább egy konstruktornak, hiszen nélküle az osztály nem példányosítható, vagyis nem hozhatók létre az osztály típusának megfelelő objektumpéldányok. Ha nem adunk meg konstruktort, akkor a Java fordító mindig létrehoz egy alapértelmezett, paraméter nélküli konstruktort. Ennek meghívása olyan példányt hoz létre, amelynek adattagjai „nullaszerű" értékekkel rendelkeznek: a numerikus típusok nulla, a karakteres típusok „`null`", a logikai típusok `false` értéket vesznek fel. Ha azonban készítünk valamilyen konstruktort, akkor a paraméter nélküli konstruktor „elvész". Így ha mégis szükségünk van rá, definiálnunk kell!

Csak olyan paraméterlistát adhatunk át egy konstruktornak, amelynek megfelelő konstruktort készítettünk, egyébként a fordító hibát jelez.

Konstruktor deklarációja:

```
[módosítók] <osztály_neve>([<típus> <adattag_neve>[, ...]]) {
  [this.]<adattag_neve> = <adattag_neve>[, ...];
}
```

A metódusokhoz hasonlóan ha egy konstruktorban olyan változó- vagy paraméternevet alkalmazunk, amely egyben az osztály egy adattagjának is az azonosítója, akkor a `this` kulcsszóval minősítve hivatkozhatunk az adattagra, míg minősítés nélkül a változóra ill. a paraméterre. Figyelem! A `this` minősítő ezen alkalmazása nem tévesztendő össze a `this()` alakú konstruktorhívással!

Nézzük az alábbi konstruktorfajtákat!

```
public Bankbetét(String név, int betétösszeg, int adó, boolean ismétlődő) {
    this.név = név;
    this.betétösszeg = betétösszeg;
    this.adó = adó;
    this.ismétlődő = ismétlődő;
}
```

```
public Bankbetét() {
    név = "Lópici Gáspár";
    betétösszeg = 100000;
    adó = 25;
    ismétlődő = false;
}

public Bankbetét(String név) {
    this(név, 50000, 35, false);
}
```

Az első konstruktor a paraméterek alapján hozza létre az objektumpéldányt, a második esetben nincs paraméterátadás, mégis minden adattag kezdőértéket kap. Az ilyen üres paraméterlistájú konstruktorban a `this` minősítő felesleges az adattagok elé, mivel ilyenkor nincsenek őket eltakaró lokális nevek vagy paraméterek. A harmadik esetben csak egy paraméterrel hívunk meg egy újabb konstruktorfajtát, a többi paramétert maga a konstruktor állítja be a `this([<aktuális paraméterlista>])` – osztályon belüli konstruktorhívást kezdeményező – kulcsszó használatával.

Egy konstruktor meghívása a `new` kulcsszóval történhet:

Példányosítás:

```
<osztály_neve> <objektumpéldány_neve>;
<objektumpéldány_neve> =
new <osztály_neve>([aktuális paraméterlista]);
```

vagy

```
<osztály_neve> <objektumpéldány_neve> =
new <osztály_neve>([aktuális paraméterlista]);
```

A háromféle konstruktor meghívásának módja:

```
Bankbetét b1 = new Bankbetét(new String("b1"), 10000, 20, true);
Bankbetét b2 = new Bankbetét();
Bankbetét b3 = new Bankbetét("Mekk Elek");
```

Lássuk a konstruktorhívások eredményeképpen létrejövő objektumok adattagjainak tartalmát is (az osztály `toString()` metódusának megfelelő módosítása után):

```
run:
Név:       bl
Betét:     10000
Adó %:     20
Ismétlődő: true

Név:       Lópici Gáspár
Betét:     100000
Adó %:     25
Ismétlődő: false

Név:       Mekk Elek
Betét:     50000
Adó %:     35
Ismétlődő: false
```

Egy konstruktorból az osztály egy másik konstruktorát, ill. a közvetlen ősosztály egy konstruktorát a `this()` ill. a `super()` kulcsszó alkalmazásával érhetjük el.

5.3.12.5. Metódusok

Az osztályokhoz tartozó objektumok viselkedését (vagyis az objektumokhoz küldhető üzeneteket) az osztály definíciójában felsorolt metódusok határozzák meg. A metódus egy olyan programrutin, amely egy objektum egy konkrét feladatának algoritmusát valósítja meg.

A Java nyelvben a metódusoknak két nagy csoportját különböztethetjük meg a visszatérési értéküknek megfelelően. Az egyik a visszatérési értékkel nem rendelkező **eljárások**, míg a másik az ezzel rendelkező **függvények** csoportja. Függvények esetén meg kell adni a visszatérési érték típusát, míg a másik csoport esetén a `void` kulcsszó helyettesíti azt.

A metódusdefiníció általános szintaxisa a következő:

Metódus fej:

```
[módosítók] <típus> <metódus_neve>([formális paraméterlista])
[throws <kivételnévlista>]
```

Metódus törzs:

```
{
  [lokális változó-deklarációk];
  [utasítások];
  [return[<kifejezés>]];
}
```

A metódus fejrészének módosítói nemcsak hozzáférési szinteket takarhatnak, hanem opcionálisan egyéb attribútumokat is:

Metódus-fejrész módosítók:

> [**hozzáférési szint**]: `public|protected|üres (csomagszintű)|private`
>
> [**abstract**]: öröklődés céljából készült, üres metódus: nincs metódusblokk (kapcsos zárójelpár), kifejtésére egy leszármazott osztályban kerül sor, a metódusfejet pontosvessző zárja
>
> [**final**]: végleges metódus, a leszármazott osztályokban a metódus nem módosítható
>
> [**static**]: osztálymetódus, csak statikus osztályszintű adattagra és metódusra hivatkozhat

Ha a `static` kulcsszó nem szerepel a fejrész módosítói között, akkor példánymetódusról van szó, amely példánymetódus példányszintű adattagokra és más példánymetódusokra (összefoglaló néven példánytagokra), valamint nyilvános (`public`) osztálytagokra (azaz osztályszintű adattagokra és metódusokra) hivatkozhat.

A `<típus>` a metódus visszatérési értékének típusát határozza meg. Az eljárások visszatérési típusa `void`, a függvényeké pedig bármilyen primitív típus, vagy hivatkozási típus lehet.

A formális paraméterlistát egy típusokból és szintaktikailag lokális változónak minősülő adattagokból álló párosok alkotják. Egy metódushíváskor az aktuális- és a formális paraméterlisták elemeinek páronként meg kell egyezniük, azaz értékadás szerint kompatibilisnek és páronként megfelelő sorrendűeknek kell lenniük. A Java-ban csak érték szerinti paraméterátadás létezik.

A `throws` (kivételek továbbküldése) kulcsszó egy későbbi fejezet témája lesz.

Egy metódust a neve és a paraméterlistája azonosít egyértelműen, tehát egy osztályon belül megadható két azonos nevű metódus, ha azok paraméterlistája (szignatúrája) vagy a listában szereplő paraméterek típusaiban, vagy azok sorrendjében, esetleg mindkettőben eltérnek. Hivatkozáskor a futtató környezet a megadott paraméterek típusából és sorrendjéből el tudja dönteni, hogy a két azonos nevű metódus körül melyiket kell végrehajtania. Az így létrehozott szerkezetet a polimorfizmus egyik formájának tekinthetjük (metódusnevek túlterhelése – overload).

Ha egy metódusban olyan változó-, vagy paraméternevet alkalmazunk, amely egyben az osztály egy adattagjának is az azonosítója, akkor a `this` kulcsszóval minősítve hivatkozhatunk az adattagra, míg minősítés nélkül a változóra ill. a paraméterre.

A metódusok visszatérési értékének típusa a metódus deklarációjakor adható meg. A metóduson belül a `return` – feltétel nélküli vezérlésátadó – utasítással lehet a visszaadott értéket előállítani. A `void`-ként deklarált metódusok nem adnak vissza értéket, ezért `return` utasítást sem szükséges tartalmazniuk. Viszont minden olyan metódusnak, amely nem `void`-ként lett deklarálva, kötelezően tartalmaznia kell a `return` utasítást. Sőt, a Java fordító azt is kikényszeríti, hogy `return` utasítással végződjön minden lehetséges végrehajtási ág. A visszaadott érték adattípusának meg kell egyeznie a metódus deklarált visszatérési értékével. Ezért nem lehetséges pl. egész számot visszaadni olyan metódusból, amelynek visszatérési értéke logikai típusú.

Az alábbi két metódus egyike eljárás, amely eldönti, hogy pozitív összeg-e a bankbetét, a másik pedig egy függvény (a paraméterként kapott egész számmal megnöveli az adó mértékét):

```
public boolean jóbetét() {
    if (betétösszeg>0)
        return true;
    else
        return false;
    //rövidebb alak:
    //return betétösszeg>=0;
}

public void Adónövel(int mérték) {
    adó += mérték;
}
```

A két metódus meghívása és futásuk eredménye:

```
System.out.println("Jó betét? "+b1.jóbetét());
System.out.println("");

b2.Adónövel(3);
System.out.println(b2);
```

```
Jó betét? true

Név:       Lópici Gáspár
Betét:     100000
Adó %:     28
Ismétlődő: false
```

Speciális metódus a `main` metódus, amelyet célszerű az osztályon belül utolsóként deklarálni. Egy alkalmazás végrehajtása során a Java futtatórendszere először mindig meghívja az alkalmazás valamelyik osztályának `main` metódusát, és ez a metódus fogja az összes többit meghívni.

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

A `public` módosító biztosítja, hogy a metódus a programon belül bárhonnan elérhető legyen, a `static` jelzi, hogy osztálymetódusról van szó, és a `void` kulcsszóból láthatjuk, hogy a metódusnak nincs visszatérési értéke. A `main` metódusnak egy string-tömb típusú paramétere van, amely a parancssori paramétereket tartalmazza.

5.3.12.6. Csomag

A csomag logikailag összefüggő osztály- és interfészdefiníciók gyűjteménye. Hierarchikus felépítésével rendszerezhetjük típusainknak, külön névtereket létrehozva számukra. A csomag elnevezése a `package <csomagnév>` utasítás segítségével történhet, amelyet a fordítási egység legelső sora kell, hogy legyen.

Ha olyan osztályokat is szeretnénk sokszor használni, amelyek más csomagokban vannak, akkor azokat (az állandó csomagnév hivatkozást elkerülendő) importálni érdemes. Formái:

- teljes csomag: `import <csomagnév>.*;`
- csomagon belüli csomag: `import <csomagnév>.<belső_csomagnév>.*;`
- csomag egy osztálya: `import <csomagnév>.<Osztálynév>;`

Az importáló utasítások kötelezően a csomagdeklarációs sor után következnek. Egy utasításban csak egy osztály (vagy a .*-gal egy teljes csomag) importálható. A Java fordító minden csomaghoz automatikusan hozzáfordítja az alapvető osztályokat (pl. `System`, `String`) tartalmazó `java.lang` csomagot, így ezt nem szükséges külön importálni. Az importálás művelete nem rekurzív, tehát egy teljes csomag importálásakor nem lesznek elérhetők az alcsomagok osztályainak definíciói.

5.3.13. ÖRÖKLŐDÉS

5.3.13.1. Az öröklődés

A 11. leckében már volt szó az öröklődés elméleti alapjairól. Ezen eszköz segítségével létező ősosztályból származtathatunk tetszőleges számú leszármazott osztályt, annak továbbfejlesztése, kibővítése céljából. Az utódosztály további osztályok őse lehet, így egész osztályhierarchiák építhetők fel. A Java-ban minden osztálynak szigorúan csak egy közvetlen szülőosztálya van. Fontos tudni, hogy egy leszármazott osztály minden olyan üzenetre tud reagálni, amelyre az őse is tudna. Az utódosztály örökli összes ősének minden olyan adattagját és metódusát, amely nem `private` minősítésű.

Az öröklődéssel egy – már létező – osztály adattagjait (tulajdonságait) és metódusait (viselkedésmódját) bővíthetjük, módosíthatjuk:

- új adattag hozzáadásával
- új metódus hozzáadásával
- létező adattag elfedésével
- létező metódus felülírásával

Alapértelmezésben egy újonnan létrehozott osztálynak az `Object` nevű osztály lesz az őse, amely a Java osztályhierarchiájának csúcsán áll, és minden Java-beli osztály közös őse. Ezt kihasználva olyan általános programokat írhatunk, amelyek nem kötik ki, hogy milyen osztályú objektumokra van szükségük. Pl. implementálhatunk egy általános felhasználású verem adatszerkezetet, amely bármilyen objektumot tud kezelni. Az `Object` osztály az egyetlen külön nem importálandó csomag, a `java.lang` csomag része.

Ha létrehozandó osztályunk számára nem megfelelő (pl. túl általános) az `Object` osztály, akkor az `extends` kulcsszót használva más – specifikusabb – ősosztályból is örököltethetjük az új osztályunkat.

Az öröklődés deklarációja:

```
[módosítók] class <utódosztály_neve> extends <ősosztály_neve>
[implements <interfészek_neve>];
```

5.3.13.2. Adattagok elrejtése

Ha egy osztály adattagja ugyanazt a nevet viseli, mint a szülőosztálya, akkor elrejti azt (még akkor is, ha különböző típusúak). Ekkor a szülőosztály adattagjára nem hivatkozhatunk egyszerűen a nevével, ezért általában nem javasolt az adattagok elrejtése.

5.3.13.3. Metódusok felülírása és elrejtése

Ha egy leszármazott osztály metódusának ugyanaz a szignatúrája (azaz neve, paramétereinek száma és típusa), valamint visszatérési értéke, mint az ősosztály metódusának, akkor a leszármazott osztály **felülírja** az ősosztály metódusát. Ez a képesség lehetővé teszi, hogy egy elég közeli viselkedésű osztályból öröklődve – annak viselkedését szükség szerint megváltoztatva – új, leszármazott osztály jöhessen létre. Például az `Object` osztály tartalmaz egy `toString()` nevű metódust, amely visszaadja az objektumpéldány szöveges reprezentációját. Ezt a metódust minden osztály megörökli. Az `Object` e metódusának végrehajtása azonban nem túl hasznos a leszármazott osztályok számára, ezért a metódus felülírása célszerű, hogy használhatóbb információt nyújthasson az objektum saját magáról. Erről bővebben e lecke további részében lesz szó.

A felülíró metódus láthatósága lehet bővebb, mint a felülírt metódusé, de szűkebb nem. Mivel egy leszármazott osztály objektuma bárhol használható, ahol egy ősosztálybeli objektum, ezért a leszármazott egyik tagjának láthatósága sem szűkülhet, hiszen akkor az ilyen használat lehetetlen lenne.

Egy leszármazott osztály nem tudja felülírni az olyan metódusokat, amelyek az ősosztályában `final` (végleges) minősítéssel definiáltak. Azonban felül kell írni azokat a metódusokat, amelyeket a felsőbb osztályban `abstract`-nak nyilvánítottak, vagy magának a leszármazott osztálynak is absztraktnak kell lennie.

Ha egy leszármazott osztály egy osztálymetódust ugyanazzal az aláírással (szignatúrával) definiál, mint a felsőbb osztálybeli metódus, akkor a leszármazott osztály metódusa **elrejti** (elfedi) a szülőosztálybelit. Fontos megkülönböztetni a felülírást az elrejtéstől! Az alábbi példaprogram jól szemlélteti a különbséget:

Adott egy **Állat** ősosztály, melynek osztály- és példánymetódusa is van:

```
public class Állat {
    public static void o_elrejt() {
        System.out.println("Az o_elrejt() osztálymetódus " +
                           "az Állat ősosztályban.");
    }

    public void p_felülír() {
        System.out.println("A p_felülír() példánymetódus " +
                           "az Állat ősosztályban.");
    }
}
```

Származtassunk belőle egy **Macska** utódosztályt, amelyben az `o_elrejt()` osztálymetódust elrejtjük, és a `p_felülír()` példánymetódust pedig felülírjuk.

```
public class Macska extends Állat {
    public static void o_elrejt() {
        System.out.println("Az o_elrejt() osztálymetódus " +
                           "a Macska utódosztályban.");
    }

    @Override
    public void p_felülír() {
        System.out.println("A p_felülír() példánymetódus " +
                           "a Macska utódosztályban.");
    }

    public static void main(String[] args) {
        Macska Tóni = new Macska();
        Állat állat = (Állat)Tóni;
        állat.o_elrejt();
        állat.p_felülír();
    }
}
```

Ebben az osztályban a `main` metódus létrehoz egy **Macska** példányt Tóni néven, majd ezt a példányt típuskonverzióval átteszi egy **Állat** példányba állat (kisbetűvel!) néven. Utóbbi példányra hívjuk meg mindkét metódust! Az eredmény:

```
run:
Az o_elrejt() osztálymetódus az Állat ősosztályban.
A p_felülír() példánymetódus a Macska utódosztályban.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Az `o_elrejt()` metódus a szülőosztályból került meghívásra, míg a `p_felülír()` a leszármazottból.

Osztálymetódushoz a futtatórendszer azt a metódust hívja meg, amely a hivatkozás szerkesztési idejű típusában van definiálva. A példában az `állat` példány szerkesztési idejű típusa az **Állat**. Így a futtatórendszer az **Állat**-ban definiált, rejtett metódust hívja meg. A példánymetódusnál a futtatórendszer a hivatkozás futásidejű típusában meghatározott metódust hívja meg. A példában az `állat` példány futásidejű típusa a **Macska**, vagyis a futtatórendszer a **Macska** osztályban definiált felülíró metódust hívja meg.

Fontos még tudni, hogy egy példánymetódus nem tud felülírni egy osztálymetódust, és egy osztálymetódus nem tud elrejteni egy példánymetódust.

5.3.13.4. A `super()` kulcsszó használata

Az öröklődés során csak az adattagok és a metódusok kerülnek át a leszármazott osztályba, a konstruktorok nem. Mégis van lehetőség a leszármazott osztály konstruktorából az ősosztály egy konstruktorát meghívni. Ezt a `super()` kulcsszóval tehetjük meg. A `super()` egy hivatkozás az ősosztályra, működése hasonló a `this()` kulcsszóhoz. Segítségével az ősosztály egy konstruktorára vagy egy felüldefiniált metódusára hivatkozhatunk. Ha nem írunk konstruktort, akkor a fordító automatikusan beilleszt egy paraméter nélkülit, amelynek első sora a `super()` konstruktorhivatkozás. Ezért ha az ősosztályunk nem rendelkezik paraméter nélküli konstruktorral, hibaüzenetet kapunk. Ilyen esetben írnunk kell egy konstruktort, amelyben meg kell adnunk, hogy melyik őskonstruktort szeretnénk használni, s milyen paraméterekkel.

Az előző lecke `Bankbetét` osztályából örököltetjük a `KamatosBetét` osztályt, majd az új osztályt bővítjük egy új adattaggal, amely a kamatösszeget tárolja:

```
public class KamatosBetét extends Bankbetét{
    int kamat;

    public KamatosBetét(String név, int betétösszeg, int adó,
        boolean ismétlődő, int kamat) {
        super(név, betétösszeg, adó, ismétlődő);
        this.kamat = kamat;
    }
```

Figyeljük meg a `this` minősítő használatát is! A sima kamat azonosító a konstruktor lokális változóját jelenti, viszont a `this.kamat` az új osztály adattagját hivatkozza.

5.3.13.5. Az Object osztály `toString()`és `equals()` metódusa

Van az `Object` osztálynak két olyan alapvető metódusa, amelyeket a leszármazott osztályok példányain sokszor alkalmazunk. Érdemes és hasznos minden új osztály definíciója során felülírni őket, hogy a `toString()` megfelelő értékeket reprezentálhasson, ill. a saját igényeinknek megfelelő módon tudjunk az `equals()` segítségével objektumokat összehasonlítani, hogy egyenlők-e.

5.3.13.5.1. A `toString()` metódus

Gyakori eset, hogy tájékoztatást kívánunk kapni egy objektum pillanatnyi állapotáról, amelyet az adattagjai reprezentálnak. Erre a Java-ban a `toString()` névtelen metódus szolgál. Azonban ez a metódus alapesetben nem azt jeleníti meg, amit várnánk. Nézzük, mi lesz a `System.out.println(b1)` utasítás eredménye:

```
bankbetét.Bankbetét@190d11
```

Amit látunk, az nem az objektumpéldány adattagjainak tartalma, hanem az azonosítója (OID - Object ID), amely az alábbi felépítésű:

```
<csomagnév>.<osztálynév>@<objektumazonosító>
```

Minden objektumpéldánynak külön azonosítója van. Definiáljuk felül a nem statikus `toString()` metódust úgy, hogy a mi elvárásunknak megfelelő kimenetet produkáljon!

Az Object osztály `toString()` metódusának felüldefiniálása:

```
@Override
public String toString() {
  return <objektumjellemzőket leíró karakterlánc>
}
```

Az alábbiakban látható egy lehetséges megoldás a `Bankbetét` osztály példányainak szöveges formában való megjelenítésére. Figyeljük meg a `System.getProperty ("line.separator")` soremelő metódust, amely biztosítja az adattagok külön sorba tördelését!

```
@Override
public String toString() {
    return "Név:        " + név
                          + System.getProperty("line.separator")+
           "Betét:      " + betétösszeg
                          + System.getProperty("line.separator")+
           "Adó %:      " + adó
                          + System.getProperty("line.separator")+
           "Ismétlődő:  " + ismétlődő
                          + System.getProperty("line.separator");
}
```

5.3.13.5.2. Az `equals()` metódus

Két – azonos osztályhoz tartozó – objektum akkor egyenlő, ha az adattagjaik által reprezentált állapotuk megegyezik. Ezt összehasonlító operátorral nem tudjuk megállapítani, ugyanis a `(<objektumnév1> == <objektumnév2>)` logikai kifejezés az érintett objektumok mutatóját hasonlítja össze, ami természetesen csak akkor lesz azonos, ha egy objektumot önmagához hasonlítunk. Az `equals()` metódus viszont képes úgy összehasonlítani két objektumot, hogy az összehasonlítás alapját képező adattagok körét mi határozhatjuk meg a metódus felüldefiniálásával.

Az Object osztály `equals()` metódusának felüldefiniálása:

```
@Override
public boolean equals(Object obj) {
   if (obj == null || !(obj instanceof <Osztálynév>)
       return false;
   if (obj == this) return true;
return <adattag_összehasonlító_logikai_kifejezések>
}
```

Figyeljük meg a metódus törzsének első utasítását, amely azonnal kiszűri az üres és az „osztályidegen” objektumokat, ill. a második utasítás felismeri azt, hogy az objektumot önmagához hasonlítjuk.

Az alábbi példában azokat a Bankbetét objektumokat tekintjük egyenlőnek, amelyek tulajdonosának neve megegyezik:

```
@Override
public boolean equals(Object obj) {
    if (obj == null || !(obj instanceof Bankbetét))
        return false;
    if (obj == this) return true;

    Bankbetét bb = (Bankbetét)obj;
    return név.equals(bb.név);
}
```

5.3.13.6. A `compareTo()` metódus

Ha azt szeretnénk elérni, hogy egy osztály példányai valamely adattagjuk alapján összehasonlíthatók legyenek, akkor a `compareTo()` metódust kell ehhez az osztályba implementálni. A metódus összehasonlítja az aktuális objektumot az átvett objektummal, és a visszatérési értéke attól függően, hogy az aktuális objektum kisebb, egyenlő, vagy nagyobb az átvettnél, rendre negatív egész, nulla ill. pozitív egész érték lesz.

A `compareTo()` metódust akkor érdemes – és kötelezően kell is – implementálni egy osztályba, ha célunk az osztály objektumainak sorba rendezése valamely adattagjuk alapján. Ilyenkor az osztálydefiníció fejrészét az `implements Comparable` utasítással kell kiegészíteni.

A következő példában kiegészítjük a KamatBetét osztály definícióját úgy, hogy példányai kamatösszegük alapján összehasonlíthatók legyenek:

```
public int compareTo(KamatosBetét kb) {
    return this.kamat-kb.kamat;
}
```

5.3.13.7. Végleges osztályok és metódusok

A `final` kulcsszóval deklarált adattagok értéke inicializálásuk után már nem módosíthatók, és ezt nem teheti meg a leszármaztatott osztály sem. Ez egy fontos szempont a rendszer biztonsága és az objektum-orientált tervezés szempontjából. Ugyanez a helyzet az osztály metódusaival is. Ha nem akarjuk, hogy egy származtatott osztályban felüldefiniáljanak egy metódust, „véglegesíteni” kell. Az Object ősosztálynak is vannak `final` típusú metódusai, de nem mindegyik az, lásd `toString()`, `equals()`.

Ha magát az osztályt a `final` kulcsszóval deklaráljuk, akkor belőle leszármaztatott osztályt egyáltalán nem tudunk létrehozni.

5.3.14. KIVÉTELKEZELÉS

5.3.14.1. A Java kivételkezelése

A Java kivételkezelésének célja a programfutás során keletkezett hibák kiszűrése és megfelelő kezelése. Az ilyen hibákat a Java platformon Exception-nek (kivételnek) nevezik. Két fő csoportjuk van: a futási időben és a nem futási időben keletkezett kivételek.

Futásidejű kivételek az aritmetikai (pl. nullával való osztás), az indexeléssel kapcsolatos (pl. tömb nem létező eleméhez való hozzáférés), és a referenciával kapcsolatos (pl. objektumokra való hivatkozás) kivételek. Ezeket a kivételeket nem kötelező implementálni, de erősen ajánlott. Nem futásidejű kivételek a Java rendszerén kívül keletkeznek. Ilyenek az I/O műveletek során keletkező hibák (pl. a fájl nem található). Utóbbiakat kötelező lekezelni.

A felmerülő hibák sokféleségének kezelését a Java az objektum-orientált paradigma lehetőségeinek felhasználásával oldja meg: a kivételeket osztályok (ill. objektumaik) reprezentálják. Minden beépített – vagy általunk létrehozott – kivétel közös ősosztálya a `Throwable` osztály.

Amikor egy metódus futása során valamilyen hiba lép fel (pl. nullával való osztás, veremtúlcsordulás, indexhatár túllépése, vagy a háttértároló megtelik, stb.), akkor egy kivételobjektum (egy kivételosztály példánya) jön létre. Ez az objektum olyan információkat tartalmaz a kivétel fajtájáról és a program aktuális állapotáról, amelyeket a kivétel lekezelésekor felhasználhatunk. A kivételobjektum létrehozását és a futtatórendszer által történő lekezelését **kivételdobásnak** hívjuk. Ezeket ellenőrzött kivételeknek is nevezzük. Az ellenőrzött kivételeket kötelező lekezelni, amit a fordító már fordítási időben ellenőriz is.

Nézzünk egy egyszerű példát a nullával való osztás kivételkezelésére! Az alábbi program alaphelyzetben semmiféle ellenőrzést nem végez az osztó értékére vonatkozóan:

```
Scanner sc = new Scanner(System.in);
System.out.println("Kérem az osztandót!");
int osztandó = sc.nextInt();
System.out.println("Kérem az osztót!");
int osztó = sc.nextInt();

double hányados = (double)(osztandó / osztó);
    System.out.println(osztandó + " / " + osztó + " = " + hányados);
```

A program futtatása ennek megfelelően 0 osztónál hibát jelez:

```
run:
Kérem az osztandót!
12
Kérem az osztót!
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at osztás.Osztás.main(Osztás.java:15)
Java Result: 1
BUILD SUCCESSFUL (total time: 3 seconds)
```

Hagyományos megoldásként az osztás művelete előtt ellenőrizzük az osztó értékét!

```
if (osztó == 0)
    System.out.println("Nullával nem lehet osztani!");
else {
    double hányados = (double)(osztandó / osztó);
    System.out.println(osztandó + " / " + osztó + " = " + hányados);
}
```

Ez a megoldás bonyolultabb ellenőrzések esetén az `if` feltételek garmadáját vonhatja magával, ami nagymértékben rontja a kód olvashatóságát és az esetleges programjavítás lehetőségét.

5.3.14.2. A `try` blokk

Kivétel mindig egy metódus törzsében keletkezhet. Ha ezeket a hibákat mind ugyanitt kezelnénk le, a programkód áttekinthetősége jelentősen romlana, és nem is lehet minden hibára előre felkészülni. Ezt szem előtt tartva olyan megoldás született, amelyben a hibás, kivételes programállapotot eredményezhető sorokat összefogva, s hozzá egy úgynevezett kivételkezelőt megadva a probléma elegánsan megoldható. Ekkor az adott blokkban fellépő kivételeket egységesen, egy helyen kezelhetjük, jól elválasztva egymástól a program fő feladatát végző, illetve a hibák lekezeléséért felelős kódrészt. Ennek megvalósítására a `try` - `catch` - `finally` blokkszerkezet használható. Sorrendjük szigorúan kötött!

A `try` blokk utasításokat zár közre, amelyeket futási időben végig felügyelete alatt tart. Lehetőleg minél kevesebb utasítást tegyünk egy ilyen blokkba, mert kivétel keletkezése esetén csak a kivétel keletkezési helyéig hajtódnak végre a blokkbeli utasítások.

A `try-catch-finally` kivételkezelő kódblokk felépítése:

```
try {
     <utasítások>
}
catch (<kivételtípus_1> <változónév_1>) {
     <utasítások>
}
[catch (<kivételtípus_2> <változónév_2>) {
     <utasítások>
}]
[finally {
     <utasítások>
}]
```

Néhány fontosabb kivételosztály:

- `Exception` (általános hiba, ez az osztály minden más kivételosztályt magába foglal)
- `IOException` (általános I/O hiba)
- `FileNotFoundException` (a fájl nem található)
- `EOFException` (olvasás a fájl vége után)
- `IndexOutOfBoundsException` (indextúlcsordulás)
- `NumberFormatException` (számformátum hiba)
- `ArithmeticException` (aritmetikai hiba)
- `IllegalArgumentException` (érvénytelen argumentum)
- `InputMismatchException` (az input adat nem megfelelő típusú)

5.3.14.3. A `catch` blokk

A `catch` blokkok a `try` blokkban keletkező – típusuknak megfelelő – kivételeket kezelik le. Minden `try` blokkhoz megadható tetszőleges számú (ha nincs `finally` blokk, akkor legalább egy) `catch` ág, amelyek az esetlegesen fellépő hibák feldolgozását végzik. Semmilyen programkód nem lehet a két blokk között! A `catch` ágak sorrendje sem mindegy, mert az első olyan `catch` blokk elkapja a kivételt, amelynek típusa megegyezik a kiváltott kivétellel, vagy őse annak. Ezért érdemes a specifikus kivételtípusoktól az általánosabb típusok felé haladva felépíteni a `catch` blokkokat. A hierarchia tetején álló

`Exception` osztály nem előzhet meg más (leszármazott) kivételosztályokat, mert azok sosem hajtódnak végre. A fordító ilyen sorrend esetén hibát jelez.

Ha valahol kivétel keletkezik, akkor a futtató rendszer megpróbál olyan `catch` ágat találni, amely képes annak kezelésére. Az az ág képes erre, amelynek paramétere megegyező típusú a kiváltott kivétellel (vagy annak ősével), valamint amelynek a hatáskörében a kivétel keletkezett.

A `catch` ág egy szigorúan egyparaméteres metódusként fogható fel, amely paraméterként megkapja a fellépő kivételt, és ezt tetszőlegesen felhasználhatja a hibakezelés során. A kivételobjektumot nem kötelező felhasználni, mert sokszor a típusa is elég ahhoz, hogy a programot a hibás állapotból működőképes mederbe tereljük.

Ha a `catch` ágak egyike sem tudja elkapni a kivételt, akkor a beágyazó kivételkezelő blokkban folytatódik a keresés. Ha egyáltalán nincs megfelelő `catch` blokk, a program befejeződik.

5.3.14.4. A `finally` blokk

A `finally` ág akkor is lefut, ha volt kivétel, akkor is, ha nem. Ebben a tetszőlegesen felhasználható blokkban kezdeményezhetjük pl. a nyitott fájlok bezárását, amit – függetlenül attól, hogy a megelőző `try` blokkban volt-e kivétel, vagy sem – mindig illik megtennünk. Ezt az ágat nem kötelező létrehozni, kivéve, ha nincs egyetlen `catch` ág sem. A kivétel lekezelése után a program végrehajtása a kivételkezelő kódblokk utáni utasításon folytatódik.

A kivételkezelő blokkok megismerése után alakítsuk át példaprogramunkat! Az osztás művelete kerüljön egy `try` blokkba, a hibát egy `catch` blokkban aritmetikai hibaként kezeljük le, valamint a `finally` blokkban jelezzük az osztás sikerességét!

```
Scanner sc = new Scanner(System.in);
System.out.println("Kérem az osztandót!");
int osztandó = sc.nextInt();
System.out.println("Kérem az osztót!");
int osztó = sc.nextInt();

boolean sikeres = false;
double hányados = 0.0;
```

```
try {
    hányados = (double)(osztandó / osztó);
    System.out.println(osztandó + " / " + osztó + " = " + hányados);
    sikeres = true;
}
catch (ArithmeticException ae) {
    System.err.println(ae.getMessage());
}
finally {
    if (sikeres)
        System.out.println("Az osztás rendben lezajlott.");
    else
        System.out.println("Az osztás sikertelen!");
}
```

A futás eredménye nulla osztó esetén:

```
run:
Kérem az osztandót!
12
Kérem az osztót!
0
/ by zero
Az osztás sikertelen!
BUILD SUCCESSFUL (total time: 3 seconds)
```

A vörös színben megjelenő „`/ by zero`” üzenet a kivételkezelő osztály hibaüzenete. Ebben a példában a `finally` ágra nincs is igazán szükség, így ezt elhagyva és a hibaüzenetet átírva egyszerűsíthető a program.

```
double hányados = 0.0;
try {
    hányados = (double)(osztandó / osztó);
    System.out.println(osztandó + " / " + osztó + " = " + hányados);
}
catch (ArithmeticException ae) {
    System.err.println("Nullával nem lehet osztani!");
}
```

A kimenet is barátságosabb:

```
run:
Kérem az osztandót!
12
Kérem az osztót!
0
Nullával nem lehet osztani!
BUILD SUCCESSFUL (total time: 4 seconds)
```

5.3.14.5. A `throws` használata

Ha egy metódus végrehajtása közben kivétel keletkezik, és ezt nem akarjuk, vagy nem tudjuk helyben – az adott metóduson belül – lekezelni, akkor tovább kell küldenünk egy magasabb szintre, az adott metódust hívó metódus felé. Ez mindaddig folytatódhat, amíg el nem érjük azt a szintet (metódust), amely már elegendő információval rendelkezik a megfelelő intézkedések elvégzéséhez. Ezt a metódus fejlécében a `throws` kulcsszóval tehetjük meg, utána felsorolva azokat a kivételosztályokat (vagy egy ősüket), amelyeket nem kívánunk (vagy nem tudunk) helyben lekezelni. Legkésőbb a program `main` metódusában az ilyen kivételeket le kell kezelni.

5.3.14.6. Saját kivételek létrehozása a `throw` utasítással

A **Java** kivételkezelése nyitott, ami azt jelenti, hogy bárki létrehozhat saját névvel és funkcionalitással ellátott kivételosztályokat, amelyeket célszerű az `Exception` osztályból származtatni. Az új kivételosztályok nevében célszerű az „Exception” szót is szerepeltetni, hogy utaljon annak szerepére.

Saját kivételobjektum létrehozása:

```
throw new <Saját_Kivételosztály_név> ("Saját hibaüzenet");
```

Az alábbi program egy 3 db egész szám tárolására szolgáló verem adatszerkezetet modellez [5]. Megvalósítjuk a verembe helyezést és a veremből való kivételt úgy, hogy e műveletek hibájának lekezelésére saját kivételosztályt használunk, és a hiba okát is megjelenítjük.

Első lépésként a saját kivételosztályunkat definiáljuk, amely hibaüzenet átvételére is alkalmas.

```
public class Verem_Exception extends Exception {

    public Verem_Exception(String hibaüzenet) {
        super(hibaüzenet);
    }
}
```

Majd következik a verem implementálása. A verem fontos jellemzője a mérete és a veremmutató. A `betesz()` metódus a paraméterként megadott számot a verem – mutató által jelzett – tetejére helyezi, a `kivesz()` – paraméter nélküli – metódus pedig a verem tetején levő számot „emeli ki”. A szám helyének „kinullázása” nem kötelező, mert a verem telítettségét a mutató állása jelzi, nem a tartalma.

```
public class Verem {
    private final static int MÉRET = 3; // a verem mérete
    private int [] verem = new int [MÉRET]; // a verem
    private int mutató = 0; // veremmutató

    public void betesz(int i) throws Verem_Exception {
        if (mutató < MÉRET) {
            verem[mutató] = i; // a szám elhelyezése a veremben
            System.out.println("A szám (" + i + ") a verembe helyezve!");
            mutató++; //a mutató növelése
        } else
            throw new Verem_Exception("A verem megtelt!");
    }

    public int kivesz() throws Verem_Exception {
        if (mutató == 0)
            throw new Verem_Exception("A verem üres!");
        mutató--;
        int i = verem[mutató]; // a szám kivétele a veremből
        System.out.println("A szám (" + i + ") a veremből kivéve!");
        return i;
    }
}
```

Próbáljunk háromnál több elemet elhelyezni a veremben, majd ezután a megtelt veremből háromnál többet kivenni!

```
public static void main(String[] args) {
    Verem v = new Verem();
    try {
        v.betesz(21);
        v.betesz(52);
        v.betesz(77);
        vár(msec);
        v.betesz(99); // gond lesz!
    } catch (Verem_Exception ve){
        System.err.println(ve);
    }
    System.out.println(); // üres sor a tagoláshoz

    try {
        v.kivesz();
        v.kivesz();
        v.kivesz();
        vár(msec);
        v.kivesz(); // gond lesz!
    } catch (Verem_Exception ve){
        System.err.println(ve);
    }
}
```

Az eredmény kiírása kissé rapszodikus sorrendben történik, mivel a kivételek kezelése külön programszálon fut, így nem a várt időpillanatban írják ki a hibaüzenetüket. Ezért a hibát kiváltó művelet előtt alkalmazzunk egy 2 mp-es késleltetést, amelyet a `vár()` nevű saját készítésű metódussal állítunk elő. Az `msec` osztályváltozó tárolja a késleltetés idejét milliszekundumban.

```
public static int msec = 2000;

public static void vár(int n){
    long t0, t1;
    t0 =  System.currentTimeMillis();
    do {
        t1 = System.currentTimeMillis();
    } while ((t1 - t0) < (n / msec));
}
```

Az eredmény:

```
run:
A szám (21) a verembe helyezve!
A szám (52) a verembe helyezve!
A szám (77) a verembe helyezve!
verem.Verem_Exception: A verem megtelt!

A szám (77) a veremből kivéve!
A szám (52) a veremből kivéve!
A szám (21) a veremből kivéve!
verem.Verem_Exception: A verem üres!
BUILD SUCCESSFUL (total time: 2 seconds)
```

Jól látható, hogy a 4. szám elhelyezése és az üres veremből való 4. kivétel is hibát okozott. Figyeljük meg a verem működését is! Az utoljára bekerült elem elsőként lett kivéve (LIFO adatszerkezet: Last In – First Out, utoljára be – elsőként ki).

Térjünk vissza egy rövid kiegészítés erejéig a 9. heti tananyag I/O műveleteire! Az ott alkalmazott forráskódok a jobb áttekinthetőség miatt nem tartalmazták a fájl bezárását megvalósító `close()` metódus biztonságos kivételkezelését, pedig ez a művelet is okozhat kivételt (pl. időközben megszűnt a kapcsolat a fájllal), így ezt az utasítást is érdemes `try-catch` szerkezetbe foglalni a következők szerint:

```
// random fájl létrehozása, és írás fájlba
RandomAccessFile raf = null;
try {
    raf = new RandomAccessFile("random.txt", "rw");
    for (int i = 65; i <= 90; i++) {
        raf.write(i);
    }
}
catch (IOException ioe) {
    System.err.println(ioe.getMessage());
}
finally {
    if (raf != null) {
        try {
            raf.close();
        }
        catch (IOException ioe) {/* hibaelnyelés */}
    }
}
```

Figyeljük meg, hogy a `close()` utasítást „védő” `try-catch` szerkezet a fájlkezelő műveletek `finally` ágában helyezkedik el, tehát minden körülmények között lefut. Viszont a 9. fejezetben szereplő példák mindegyikében hiába volt a `close()` utasítás a `try` blokk által védett, ha előtte bekövetkezett egy kivétel (pl. a fájlba írás során), akkor soha nem került rá a vezérlés, tehát a fájl nyitva maradhatott.

5.3.15. OBJEKTUM-ORIENTÁLT FELADAT MEGOLDÁSA

5.3.15.1. Számítógép

Az alábbiakban egy olyan összetett feladatot fogunk megoldani, amely a Java nyelv igazi erősségét, az objektumközpontú programozást helyezi előtérbe [10].

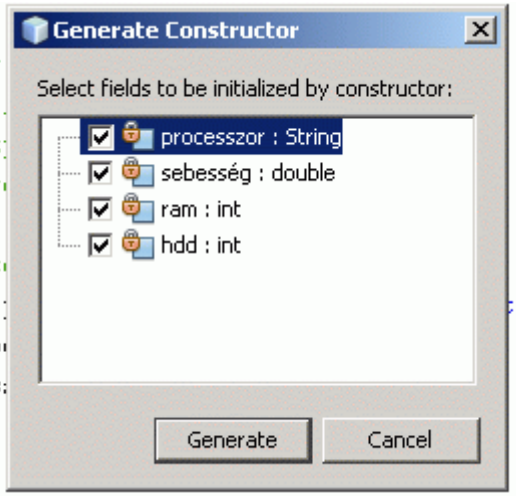
Adott az alábbi osztály:

```
public class Számítógép {
    private String processzor; // a processzor neve
    private double sebesség; // a processzor sebessége (GHz)
    private int ram; // a központi memória nagysága (MB)
    private int hdd; // a merevlemez tárolókapacitása (GB)
}
```

1. feladat: Definiáljunk egy olyan konstruktort az osztályhoz, amellyel az adattagoknak kezdőérték adható!

A megoldásban segít a NetBeans fejlesztőrendszer. A szerkesztőterület helyi menüjének „Insert code” menüpontjából kezdeményezhetünk konstruktor-generálást, kijelölve azokat az adattagokat, amelyeknek kezdőértéket kívánunk adni.

```
public class Számítógép {
    private String processzor
    private double sebesség;
    private int ram; // a köz
    private int hdd; // a mer

    // 1. feladat - konstrukt
    public Számítógép(String
        this.processzor = pro
        this.sebesség = sebes
        this.ram = ram;
        this.hdd = hdd;
```

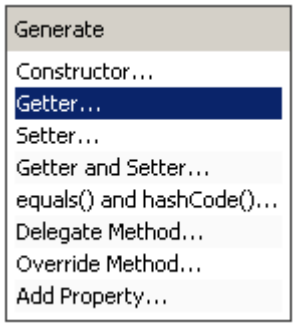


```
public Számítógép(String processzor, double sebesség, int ram, int hdd) {
    this.processzor = processzor;
    this.sebesség = sebesség;
    this.ram = ram;
    this.hdd = hdd;
}
```

Figyeljük meg, hogy az adattagok nevei önmagukban mint paraméterek jelennek meg, az osztály „igazi” adattagjaira a `this` kulcsszóval hivatkozhatunk!

2. feladat: Írjuk meg az adattagok lekérdező és beállító metódusait!

Ismét használható a helyi menü, ahol a metódusokat a Getter (lekérdező) és a Setter (beállító) menüpontok generálják le.



A létrejövő kódrészletek:

```
public int    getHdd()         { return hdd; }
public String getProcesszor()  { return processzor; }
public int    getRam()         { return ram; }
public double getSebesség()    { return sebesség; }
```

```
public void setHdd(int hdd)                   { this.hdd = hdd; }
public void setProcesszor(String processzor)  { this.processzor = processzor; }
public void setRam(int ram)                   { this.ram = ram; }
public void setSebesség(double sebesség)      { this.sebesség = sebesség; }
```

Figyeljük meg, hogy az automatikus generálás az adattagokat mindkét metódusnál ABC sorrendbe állította, amely utasítások természetesen visszaállíthatók az osztálydefiníció szerinti sorrendjükbe.

3. feladat: Definiáljuk felül az osztályunk `toString()` metódusát úgy, hogy az adattagjait soronként és a következő formátumban adja vissza:

`<processzor> <sebesség> GHz CPU, <ram> MB RAM, <hdd> GB HDD`
(pl. „AMD 2.4 GHz CPU, 1024 MB RAM, 500 GB HDD”)

Az `Object` osztályból örökölt `toString()` metódus átdefiniálását szintén kezdeményezhetjük a helyi menü metódus-felülíró (Override Method) menüpontjában, viszont a kapott kódot a kívánalmaknak megfelelően át kell alakítani:

```
@Override
public String toString() {
    return processzor + " " + sebesség + " GHz CPU, " + ram + " MB RAM, "
           + hdd + " GB HDD";
}
```

4. feladat: Definiáljuk felül az `equals()` metódust úgy, hogy két számítógép akkor legyen egyenlő, ha a processzoruk neve megegyezik, és a sebesség- és kapacitás-paramétereik páronként maximum 10%-kal térnek el egymástól!

Ennek a metódusnak is generáljuk a vázát, majd a `Math` osztály `abs()` metódusát felhasználva – és a kódot jelentősen átírva – előállítjuk az összetett feltételt. Az automatikusan generálódó `hashCode()` metódus átdefiniálására most nincs szükség, akár törölhető is.

```
@Override
public boolean equals(Object obj) {
    if (obj==null || !(obj instanceof Számítógép))
        return false;

    Számítógép sz = (Számítógép)obj;
    return (processzor.equals(sz.processzor) &&
            Math.abs(sebesség / sz.sebesség - 1) <= 0.1 &&
            Math.abs(ram / sz.ram - 1) <= 0.1 &&
            Math.abs(hdd / sz.hdd - 1) <= 0.1);
}
```

Figyeljük meg, hogy null (üres) és nem számítógép objektum esetén nincs átdefiniálás!

5. feladat: Egészítsük ki a `Számítógép` osztály definícióját úgy, hogy az objektumai processzoruk sebessége alapján összehasonlíthatók legyenek! Implementáljuk a `Comparable` interfészt is!

A feladat megoldásához a `compareTo()` metódust kell felüldefiniálni, hogy a megadott szempont szerint tudjuk az osztálypéldányokat összehasonlítani.

```
public int compareTo(Számítógép sz) {
    return (int)(this.sebesség - sz.sebesség);
}
```

A `Comparable` interfész implementálását az osztály fejében jelezni kell. Definíciója csak egyetlenegy metódust tartalmaz.

```
public class Számítógép implements Comparable<Számítógép>{
```

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

6. feladat: Származtassunk a `Számítógép` osztályból egy `Notebook` osztályt! Az új osztály objektumai a számítógép tulajdonságain kívül egy új – valós értékű – `üzemidő` adattaggal is rendelkeznek. Készítsünk az új adattaghoz lekérdező metódust, valamint az osztályhoz egy konstruktort, amellyel mindegyik adattagjának kezdőérték adható!

Az új osztályt az `extends` kulcsszóval származtathatjuk a szülőosztályból. A konstruktor és az új adattag lekérdező metódusának előállítása a generátor használatával egyszerű. Figyeljük meg a `super()` metódus használatát! Segítségével a szülőosztály adattagjaira hivatkozhatunk, míg a `this` a származtatott osztály új adattagját hivatkozza.

```
public class Notebook extends Számítógép {
    double üzemidő; // a notebook akkumulátoros üzemideje órában

    public Notebook(String processzor, double sebesség, int ram, int hdd, double üze
        super(processzor, sebesség, ram, hdd); // öröklés a szülőosztálytól
        this.üzemidő = üzemidő; // új adattag
    }

    public double getÜzemidő() { return üzemidő; }
}
```

7. feladat: Egészítsük ki a Notebook osztályt egy olyan konstruktorral is, amely a RAM méretét 2048 MB-ra állítja be, a többi adattag értékét pedig paraméterek határozzák meg!

A `Notebook` osztály új konstruktorában a `ram` adattag nem szerepel paraméterként, mert konkrét értéket kap, amit a `super()` metódus meghívásakor állítunk be.

```
public Notebook(String processzor, double sebesség, int hdd, double üzemidő) {
    super(processzor, sebesség, 2048, hdd);
    this.üzemidő = üzemidő;
}
```

8. feladat: Definiáljuk felül a `Számítógép` osztálytól örökölt `toString()` metódust úgy, hogy a notebook a számítógéptől örökölt adattagjai után az üzemidőt is jelenítse meg egy tizedes pontossággal (pl. ..., 3,5 óra üzemidő)!

A korábban már megismert `java.text.DecimalFormat` osztályt importálva a feladat megoldása nem okoz gondot.

```
@Override
public String toString() {
    DecimalFormat df = new DecimalFormat("##.0");
    return super.toString() + ", "+ df.format(üzemidő) + " óra üzemidő";
}
```

9. feladat: Készítsünk egy `main` nevű osztályt, amely a főprogramot tartalmazza. Hozzunk létre két-két `Számítógép` típusú objektumot az alábbi adatokkal, majd írassuk ki őket a képernyőre!

- `Számítógép szg1:` AMD, 3.2 GHz, 4096 MB, 400 GB
- `Számítógép szg2:` Intel, 2.6 GHz, 2048 MB, 500 GB

```
Számítógép szg1 = new Számítógép("AMD", 3.2, 4096, 400);
Számítógép szg2 = new Számítógép("Intel", 2.6, 2048, 500);

System.out.println(szg1);
System.out.println(szg2);
```

A két objektumpéldány adattagjainak megjelenése a `toString()` metódus átdefiniálásának megfelelő formátumú:

```
run:
AMD 3.2 GHz CPU, 4096 MB RAM, 400 GB HDD
Intel 2.6 GHz CPU, 2048 MB RAM, 500 GB HDD
BUILD SUCCESSFUL (total time: 0 seconds)
```

10. feladat: Hozzunk létre két `Notebook` objektumot is! Az első minden adattagját paraméterként kapja, a második viszont 2048 MB-os memóriával jöjjön létre. Adatok:

- `Notebook nb1:` AMD Turion X2, 1.8 GHz, 3072 MB, 250 GB, 3.55 óra
- `Notebook nb2:` Intel Atom, 1.6 GHz, 120 GB, 2.2 óra

```
Notebook nb1 = new Notebook("AMD Turion X2", 1.8, 3072, 250, 3.55);
Notebook nb2 = new Notebook("Intel Atom", 1.6, 120, 2.2);

System.out.println(nb1);
System.out.println(nb2);
```

Figyeljük meg, hogy a két objektumpéldányt más-más konstruktor hozta létre!

```
run:
AMD Turion X2 1.8 GHz CPU, 3072 MB RAM, 250 GB HDD, 3,6 óra üzemidő
Intel Atom 1.6 GHz CPU, 2048 MB RAM, 120 GB HDD, 2,2 óra üzemidő
BUILD SUCCESSFUL (total time: 0 seconds)
```

5.4. Az oktatóprogram használata

Oktatóprogramunkat webes környezetre terveztük, így még telepíteni sem kell. Bármely olyan számítógépről elérhető, amely internetkapcsolattal rendelkezik és telepítve van rá egy webböngésző. Platformfüggetlen, azaz nemcsak Windows környezetből, hanem gyakorlatilag bármely más operációs rendszer alól is használható. A Java programok fejlesztésére és futtatására alkalmas komponenseket (JDK, NetBeans) a gyártók több operációs rendszer (Mac OS X, Linux, Solaris) alá is elkészítették, így a példaprogramok is feldolgozhatók, és a feladatok is elvégezhetők bárki számára. A weblapot az ingyenes tárhelyet biztosító UltraWEB Kft. (http://www.ultraweb.hu) szerverére töltöttük fel.

Elérhetősége: http://java2.uw.hu

A feldolgozandó anyagot féléves oktatási időszakra terveztük, heti bontásban tárgyalva egy tanegységet. Egy tanegység feldolgozásához kb. 4-5 óra szükséges, de hosszabb oktatási időszak is alkalmazható, így csökkenthető a heti óraszám.

A weblap felépítése nagyon egyszerű. Egy tanegység az új ismeretek tárgyalásával indul, szinte kivétel nélkül gyakorlati példaprogramokkal illusztrálva, amely programok forráskódja és az esetleges inputállományaik is letölthetők. Majd elméleti tesztkérdések, és gyakorlati programozási feladatok következnek. A tesztek azonnal kiértékelésre kerülnek, a feladatok – magyarázatokkal ellátott – megoldása szintén letölthető. A tesztek és a feladatok külön menüpontokba szervezve együtt is elérhetők. Rendelkezésre áll egy fogalomtár is, amely elsősorban az objektum-orientált paradigma fogalomrendszerét foglalja össze.

6. A továbbfejlesztés lehetőségei

A tananyagot már jelenlegi állapotában is tanításra alkalmasnak véljük, de ez nem jelenti azt, hogy nyugodtan ülhetünk a babérjainkon. A puding próbája az evés! Egyrészt hiányzik egy teljes tanítási ciklus tapasztalata, amely minden oktatási anyag használhatóságának alapvető mércéje. Olyan kérdésekre kell még keresnünk a választ, hogy jó-e a tanegységek sorrendje, egyáltalán azok tartalma, és a tanulók számára érthető-e a Java nyelv fogalomrendszere? Vagy ott van az objektum-orientált programozási szemléletmód és annak eszközrendszere, amely eléggé komplex ahhoz, hogy ne legyünk biztosak abban: egy középiskolai tanulónak mindez könnyen megérthető.

Úgy érezzük, hogy a tesztkérdések és a gyakorló feladatok köre bővítésre szorul, jelenlegi mennyiségüket érdemes és szükséges is növelni. Hiányzik a tanulói önértékelés és tanári értékelés rendszere is. A tesztkérdésekre adott válaszok azonnali kiértékelése önmagában hasznos, de komplex értékelésre nem ad módot.

A továbbfejlesztés másik területe a mai számítógépes oktatási technológiák ismeretében szinte tálcán kínálja magát. Oktatóprogramunk új generációjú eTananyag platformra való átültetése minőségi ugrást jelentene. E rendszerekben már nem a tanár van a központban, hanem a tanuló. A tanulói passzivitást („csak olvasás”), felváltja az aktivitás („olvasás-írás”). Ez az új szemléletmód egyéni előrehaladást, aktív közreműködést, gyűjtő- és csoportmunkát, valamint a teljesítmény egyéni mérését és értékelését is lehetővé teszi. Természetesen ekkor sem maradhat el a tanári közreműködés, a tanulás folyamatának felügyelete elengedhetetlen. A tanulmányaink során megismert Moodle ingyenes virtuális oktatási környezet ilyen felületet kínál. Tananyagunk felépítése és webes formája miatt könnyen átültethető lenne, és a tesztkérdések is olyan szoftverrel készültek, amely kompatibilis a Moodle-lel.

Oktatási anyagot készíteni nem könnyű. Magabiztos szakmai felkészültséget, szintetizáló-képességet, speciális – elektronikus környezetben használható – módszertani ismereteket, az állandó továbbfejlesztéshez kitartó motivációt, és nem utolsósorban sok időt és munkát igényel. De a tanulóifjúság előrehaladásáért mindezt érdemes felvállalni!

7. Összefoglalás

Az oktatóprogram készítése során a bevezetőben említett célok vezettek bennünket. Munkánkkal egy olyan oktatóprogram készítését céloztuk meg, amely segítséget nyújt a tanulóknak, kollégáknak, és a téma iránt érdeklődőknek a Java programozási nyelv alapjainak elsajátításához, valamint az objektum-orientált szemléletmód kialakításához.

Áttanulmányoztuk a NAT, az informatika központi kerettanterve és az emelt szintű informatika érettségi programozásra vonatkozó előírásait. A követelményeket szem előtt tartva állítottuk össze az elsajátítandó tananyagot. A tananyag részekre bontásánál heti ciklusokban gondolkodunk, heti 4 tanítási óra keretben. A 15 lecke elsajátítása kb. egy féléves, közös tanár-diák munka eredménye lehet. A tananyag készségszintre történő emelését követően, az emelt szintű informatika érettségin bátran állhatnak a diákok a programozási feladat elkészítése elé.

Mind a ketten gyakorló pedagógusok vagyunk, éveken át a Pascal ill. a Delphi programozási nyelv rejtelmeit próbáltuk a diákokkal megszerettetni. Az itt szerzett gyakorlati tapasztalatainkat is beépítettük az oktatóprogramba.

A sikeres érettségi vizsgán túl, a tananyag elsajátítása kellő alapot biztosít szakirányú továbbtanulás esetén (pl. műszaki informatikus, mérnök informatikus, informatikatanár, stb.) az egyetemen, főiskolán oktatott programozás tantárgy sikeres teljesítéséhez. A felsőoktatási intézményekben – a C++ programozási nyelv mellett – egyre több helyen alkalmazzák laborgyakorlatokon a Java programozási nyelvet.

Természetesen az oktatás nem fejeződik be a tanítási órán, szükség van a tanuló otthoni egyéni munkájára is. Egy programozási nyelvet elsajátítani csak minél több feladat önálló elkészítésével, a programok futtatásával, tesztelésével lehet. A tananyag elérhető az interneten, így a diákok az esetleges lemaradás esetén akár otthon is bepótolhatják a hiányosságokat. A leckék végén található feladatokat a diákok önállóan, otthon készítik el. A megoldások is elérhetők a „neten”, amennyiben gondjuk akadna a program elkészítése során, vagy szeretnék ellenőrizni az elgondolásuk helyességét.

Az oktatóprogram lehetővé teszi a tananyag egyéni tempóban történő elsajátítását, a tesztek segítségével elmélyíti a megszerzett ismereteket, felhívja a figyelmet a hiányosságokra. A feladatok elvégzésével begyakorolják a diákok a különböző programozási

eszközök használatát, és megfelelő jártasságra tesznek szert a fejlesztőkörnyezet használatában.

A program természetesen magában hordozza a továbbfejlesztés igényét, melyeket az előző fejezetben részleteztünk.

A következő 2010/2011-es tanévben azt tervezzük, hogy az eddig iskolánkban oktatott Free Pascal ill. Delphi programozási nyelvet lecseréljük, és áttérünk a Java programozási nyelv tanítására. Az elkészített oktatóprogram segítségével fogjuk a diákokat a Java nyelv rejtelmeibe bevezetni.

8. Irodalomjegyzék

[1] Nemzeti Alaptanterv:
http://www.okm.gov.hu/letolt/kozokt/nat_070926.pdf (2010. április)

[2] Kerettanterv a gimnázium 9-12. évfolyamára:
http://www.okm.gov.hu/letolt/kozokt/kerettanterv/melleklet1/3_gimnazium/08_informatika.rtf (2010. április)

[3] Szakközépiskolai szakmacsoportos alapozó oktatás kerettanterve:
https://www.nive.hu/fejlesztes/kerettantervek/Mk28tart/Mk028-2-200.pdf
(2010. április)

[4] Az érettségi vizsga vizsgaszabályzata:
http://net.jogtar.hu/jr/gen/getdoc.cgi?docid=99700100.kor (2010 április)

[5] Kósa Márk: Adatszerkezetek és algoritmusok jegyzet (a verem és a sor):
https://infotech.inf.unideb.hu/honlap/adatszerk?action=AttachFile&do=view&target=adatszerkezetek-05-20090326.pdf (2010. április)

[6] Juhász István: Programozás 1. jegyzet:
http://www.inf.unideb.hu/~panovics/programozas12009.pdf (2010. április)

[7] Juhász István: Programozás 2. jegyzet:
http://www.inf.unideb.hu/~panovics/Programozas220040330.pdf (2010. április)

[8] Debreceni Egyetem, IK: Programozás 1. és 2. kurzusok órai jegyzetei

[9] Java 6 online dokumentáció: http://java.sun.com/javase/6/docs/api/ (2010. április)

[10] Java vizsgafeladat-sorok:
http://www.inf.unideb.hu/~panovics/beugrok.pdf (2010. április)

[11] Java Basics: http://leepoint.net/JavaBasics/index.html (2010. április)

[12] Java Programming Notes: http://leepoint.net/notes-java/index.html (2010. április)

[13] Nagy Gusztáv: Java programozás v1.3 (Creative Commons licence):
http://nagygusztav.hu/sites/default/files/Java_programozas_1.3_0.pdf (2010. április)

[14] Angster Erzsébet: Objektum-orientált tervezés és programozás, Java első kötet
4KÖR Bt., 2003

[15] Illés Lajosné: Számítógépek a pedagógiában, Tankönyvkiadó, Budapest, 1972

[16] Móricz Attila: Java programozási nyelv, Alapismeretek, LSI Oktatóközpont, 1997

9. Függelék

9.1. Feladatok

1.1. Töltsd le, telepítsd, és ismerkedjél a fejlesztőkörnyezettel!

2.1. Jelenítsd meg a konzolon a „Holnap jó leszek!” szöveget!

2.2. Rajzolj egy vízszintes vonalat a konzolra!

2.3. Mi lesz a következő program outputja?

```
1   public class Feladat3{
2       public static void main(String[] args) {
3           System.out.print("Első sor: ");
4           System.out.println("Én");
5           System.out.print("Második sor: ");
6           System.out.println("Te");
7       }
8   }
```

2.4. Rajzolj egy négyzetet a konzolra!

3.1. Készíts egy eljárást, amely egy fenyőfát rajzol a konzolra!

3.2. Készíts egy függvényt, amely összead két számot!

3.3. Mi lesz a következő program outputja:

```
public class Feladat7{
    public static void main(String[] args) {
        int x=1;
        int y=4;
        System.out.println(x+"+"+y+"="+z);
    }
 }
```

3.4. Add meg a kifejezés (a / 2) + 2 * b zárójel nélküli alakját!

3.5. Készíts programot, ami bemutatja a normál és a maradékos osztás közötti különbséget!

4.1. Készíts programot, amely kiszámítja és kiírja az 5 egység sugarú kör kerületét és területét!

4.2. Készíts programot, amely kiszámítja és kiírja a képernyőre a 30 fokos szög szögfüggvényeit!

4.3. Készíts programot, amely 10-től 20-ig generál két véletlen egész számot! Majd kiírja a két szám összegét, valamint szorzatát.

4.4. A következő sorok közül melyik fog hiba nélkül lefordulni?

a) `byte a=256;`
b) `int i=10,`
c) `char c="a";`
d) `double d=1.33;`

4.5. Készíts programot, ami ha megadunk két nem negatív számot sztring formátumban, akkor a kisebbik négyzetgyökét kiírja a képernyőre.

5.1. Készíts programot, amely az „Indul a görög aludni” sztring tartalmát megfordítja és kiírja!

5.2. Készíts programot, amely két adott sztringet megjelenít, az egyiket kisbetűs, a másikat nagybetűs formában! A második sztringben található összes 'e' karaktert kicseréli az első sztring második karakterével.

5.3. Mi lesz a következő program outputja:

```
1   public class KarakterFeladat{
2       public static void main(String[] args) {
3           Character a = new Character('a');
4           Character b = new Character('b');
5           Character b2 = new Character('b');
6           Character c = new Character ('c');
7           System.out.println("különbség = "+(c.compareTo(a)));
8           System.out.println("b = b2? "+b.equals(b2));
9
10      }
11  }
```

5.4. A következő sorok közül melyik fog hiba nélkül lefordulni?

a) `"pá"+"linka"`
b) `3+"pá"`
c) `3+5`
d) `3+5.5`

5.5. Adj meg egy valós számot. A számot úgy jelenítsd meg, hogy a tizedespont helyén egy tizedesvessző jelenjen meg!

6.1. Készíts programot, amely bekér egy számot, majd kiírja, hogy osztható-e 2-vel vagy 3-mal!

6.2. Készíts programot, amely bekér két számot és kiírja, hogy melyik a nagyobb illetve ha egyenlők, akkor azt.

6.3. Készíts programot, ami generál két véletlen egész számot [0; 100] intervallumban. A nagyobbik számból vonja ki a kisebbet, és írja ki a végeredményt.

6.4. Készíts programot, ami egy számformátumban megadott érdemjegyet szövegesen jelenít meg! Az érdemjegyet a billentyűzeten kell bevinni! (pl. 1 = elégtelen; 2 = elégséges; stb.)

6.5. Készíts programot, ami bekéri egy bankbetét összegét, valamint azt, hogy hány hónapig lesz lekötve az összeg. Majd megjeleníti, hogy mennyi lesz a betét összege a lekötés végén, ha a kamat évi 12%!

7.1. Készíts programot, amely egymás alá hússzor kiírja a „Jó napot kívánok!” szöveget!

7.2. Készíts programot, amely 1-től 10-ig kiírja egymásmellé, vesszővel elválasztva a számok négyzetét!

7.3. Írjál programot, ami 50 db kockadobást szimulál, és kiírja a dobásokat egymás mellé, szóközzel elválasztva!

7.4. Írjuk ki az 1-200 közötti számok közül azokat az 5-tel oszthatóakat, amelyek nem oszthatók 25-tel!

7.5. Kérd be n értékét és készítsd el az alábbi n soros háromszöget!

```
1
1 2
1 2 3
...
1 2 3 n
```

8.1. Írjál programot, amely a következő tartalmú tömböt hozza létre, majd ki is írja a képernyőre!

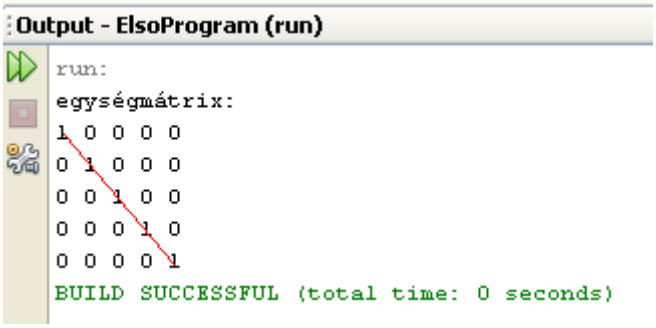
```
0    1    2
3    4    4
0    0    0
```

8.2. Készíts programot, amely egy 20 elemű tömböt feltölt 'a' karakterekkel, majd a tömb minden 2. elemét kicseréli 'b' karakterre, majd kiírja egymásmellé az elemeket, szóközzel elválasztva!

8.3. Készíts programot, amely feltölt 1-től 10-ig véletlen valós számokkal egy 6x6-os tömböt, majd megjeleníti a tömb tartalmát! Írja ki a program a számok átlagát is!

8.4. Írjál programot, amely az 5x5-ös egységmátrixot hozza létre! (Az egységmátrixban a főátlóbeli elemek 1-t, míg az ezen kívüli elemek 0-t tartalmaznak.)

Minta:



8.5. Készíts programot, amely bekér a billentyűzetről 5 db számot és elhelyezi egy megfelelő tömbben! A program írja is ki a tömb tartalmát a képernyőre, valamint jelenjen meg külön a legkisebb szám is.

9.1. Kérjünk be két egész számot a billentyűzetről, és írjuk ki a képernyőre a szorzatukat!

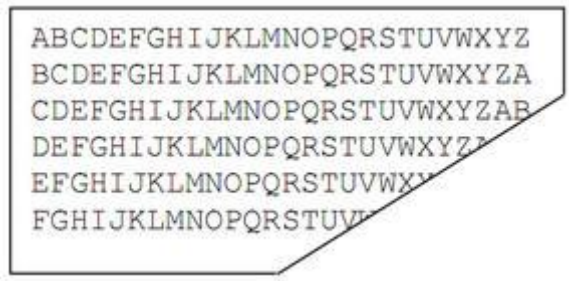
9.2. A `FileWriter` és a `FileReader` osztályok segítségével írjuk ki az `abc.txt` fáljba az angol abc kisbetűit! Utána olvassuk be a fájlt, és az abc-t – nagybetűsre alakítva – írjuk ki a képernyőre!

9.3. A `PrintWriter` és a `BufferedReader` osztályokat felhasználva írjuk ki a hét napjait soronként a `napok.txt` szöveges fájlba! Ezután olvassuk be a fájlt, és a napokat egy sorban, vesszővel elválasztva jelenítsük meg a képernyőn!

9.4. A `RandomAccessFile` osztály metódusai segítségével cseréljük le az `abc.txt` fájl minden 3. betűjét nagybetűsre, majd a teljes fájlt írjuk ki a képernyőre!

10.1. Jelenítsük meg a képernyőn vesszővel elválasztva a Fibonacci számok első 20 elemét ciklussal és a 21-30. elemét rekurzióval! (A Fibonacci-számok első két eleme a 0 és az 1, a következő elemek pedig az előző elemek összege, tehát 0, 1, 1, 2, 3, 5, 8, ...)

10.2. Készítsük el a Vigenere táblát! Ez a 26*26-os tábla az angol abc betűit tartalmazza a következő elrendezésben:



10.3. Kérjünk be a felhasználótól egy magyar nyelvű (ékezetes betűket is tartalmazó) szöveget! A beolvasott szöveget kódoljuk át az alábbiak szerint, majd írjuk ki a képernyőre:

- a magyar ékezetes betűket „ékezetmentesíteni” kell (pl. á - a, ű - u)
- csak az angol abc 26 betűje és a számjegyek szerepelhetnek a kódolt szövegben, szóköz vagy írásjelek nem
- minden átalakított betű nagybetűs legyen

10.4. A `szavak.txt` szöveges fájl első sora tartalmazza a fájlban lévő szavak számát. Olvassuk be az összes szót, majd írjuk ki a képernyőre a legrövidebb ill. a leghosszabb szavak listáját! Készítsünk statisztikát a szavak hosszúságának gyakoriságáról!

11.1. Hozzunk létre egy `Dolgozó` nevű osztályt a következő tulajdonságok reprezentálására:

- `név` (szöveg típus)
- `születési_év` (egész szám típus)
- `munkakör` (szöveg típus)
- `fizetés` (egész szám típus)
- `adójóváírás` (logikai típus)

11.2. Hozzunk létre az osztályunkhoz egy olyan paraméteres konstruktort, amely segítségével a példányosítás során minden adattagnak kezdőértéket adhatunk!

11.3. Példányosítsuk az előző osztályunkat! Hozzuk létre az alábbi objektumokat, majd a megadott kódrészletet beszúrva írassuk ki őket a képernyőre!

- Kovács Péter, 1966, igazgató, 500000, nem
- Nagy Árpád, 1954, portás, 100000, igen
- Kiss P. Eszter, 1980, főelőadó, 250000, nem

Beszúrandó kódrészlet:

```
@Override
public String toString() {
    return "Név:           " + név + System.getProperty("line.separator") +
           "Születési év:  " + születési_év + System.getProperty("line.separator") +
           "Munkakör:      " + munkakör + System.getProperty("line.separator") +
           "Fizetés:       " + fizetés + " Ft" + System.getProperty("line.separator") +
           "Adójóváírás:   " + (adójóváírás ? "jár" : "nem jár");
}
```

11.4. Készítsünk a pályakezdő dolgozóknak olyan konstruktort, ahol a fizetés a minimálbérrel egyezik meg (2010-ben ez 73.500 Ft), és az adójóváírás alanyi jogon járjon! Ezt a konstruktort felhasználva hozzuk létre és írassuk ki a következő tulajdonságokkal rendelkező objektumot: Soós Elemér, 1990, gyakornok.

12.1. Bővítsük tovább az előző fejezet `Dolgozó` osztályát `Dolgozó2` néven! Hozzunk létre egy – csak ebből az osztályból látható – osztályváltozót, amely a dolgozók számát tárolja! Gondoskodjunk arról is, hogy példány létrejöttekor automatikusan növekedjen az értéke! Ellenőrizzük, hogy eddig hány dolgozó van!

12.2. Készítsünk – más osztályból nem látható – metódusokat a fizetés emeléséhez! Az alábbi típusú fizetésemelések lehetségesek:

- 1. típus: rendkívüli – fix 10.000 Ft-os emelés
- 2. típus: százalékos – emelés a meglévő fizetés százalékában
- 3. típus: jogszabály szerinti – emelés a minimálbér 10%-ával
- 4. típus: általános – adott összegű fizetésemelés

A minimálbér összegét definiáljuk végleges osztályváltozóként, és ennek megfelelően módosítsuk a „minimálbéres" konstruktort! A dolgozókat (`d1`, ... `d4`) rendre rendkívüli, 20%-os, jogszabály szerinti, ill. 5.000 Ft-os általános fizetésemelésben részesítsük, majd jelenítsük meg képernyőn a nevüket és az új fizetésüket! Mennyi lett az új átlagfizetés?

12.3. Hozzuk létre az osztály paraméter nélküli konstruktorát, amely segítségével példányosítsunk egy újabb dolgozót! Adattagjai milyen értékeket vettek fel? Írjuk felül őket a `d4`-es dolgozó adataival, kivéve a nevet és a születési évet, amely Molnár Attila (szül. 1985) legyen! Írjunk egy példánymetódust a nettó fizetés kiszámítására! A levonások összesen 45%-ot tesznek ki. Mennyi a nettó bére az új dolgozónak?

12.4. Írjuk meg a példányváltozók lekérdező és beállító metódusait! Hozzunk létre osztály- és példány-inicializáló blokkokat, amelyekben jelezzük, hogy a program futása közben milyen inicializálás történik! Készítsünk egy osztálymetódust, amellyel a minimálárat adott százalékkal megnövelhetjük!

13.1. Készítsünk megfelelő osztályhierarchiát a hasáb és a gömb felszínének és térfogatának reprezentálására. Közös ősosztályuk neve legyen `Test`! A testek jellemző adatai (élek, sugár – cm-ben megadva) egész típusúak legyenek, a számított értékek pedig double típusúak!

13.2. Írjuk felül a két leszármazott osztály `toString()` metódusát úgy, hogy a példányaik adattagjai és a két metódusuk eredménye az alábbi formában jelenjenek meg (figyeljünk a `Gömb` osztálynál az 1 tizedes kerekítésre!):

```
Hasáb (élei: 5, 10, 15 cm)
 - felszíne  550 cm2
 - térfogata 750 cm3
Gömb (sugara: 10 cm)
 - felszíne  1256,6 cm2
 - térfogata 3141,6 cm3
```

13.3. Definiáljuk felül a `Hasáb` osztály `equals()` metódusát úgy, hogy két hasáb csak akkor legyen egyenlő, ha éleik hossza – függetlenül azok sorrendjétől – megegyezik!

13.4. Definiáljuk felül a `Gömb` osztály `equals()` metódusát úgy, hogy két gömb csak akkor legyen egyenlő, ha sugaruk megegyezik. Ellenőrizzük a metódus működését! Egészítsük

ki ennek az osztálynak a definícióját úgy, hogy példányaik – sugaruk nagysága alapján – összehasonlíthatók legyenek!

14.1. Olvassunk be egy számot a billentyűzetről és írjuk ki a négyzetgyökét! A kritikus műveleteket tegyük `try - catch` blokkokba, és negatív szám bevitele esetén „Negatív számból nem lehet négyzetgyököt vonni!” hibaüzenet jelenjen meg! A helyes eredmény 3 tizedes pontosságú legyen!

14.2. Egészítsük ki a 9. fejezet `io_token.java` programját úgy, hogy hibás inputadatok bevitele esetén is működjön, és csak az egész számokat adja össze! A hibás adatok `NumberFormatException` kivételt váltanak ki. Elválasztójel a szóköz legyen! Pl. a 23 44,4 12 k 10 bemenő adatokból kiszűrhető egész számok összege 23+12+10=45 jelenjen meg a képernyőn! A végeredmény mellett a hibás adatok is kerüljenek kiírásra!

14.3. Készítsünk programot, amely a `vissza.txt` szövegfájlból beolvassa a leghosszabb magyar szót, majd mind előre, mind hátrafelé olvasva kiírja képernyőre! A fájlműveleteket lássuk el megfelelő kivételkezeléssel (`FileNotFoundException`, `IOException`)!

14.4. Írjunk programot egy 3 db egész szám tárolására szolgáló sor adatszerkezet modellezésére. Típusa „fix kezdetű” legyen, azaz a sor első eleme mindig a sor első tárhelyén helyezkedjen el! Implementáljuk a sorba való behelyezést és a sorból való kivételt úgy, hogy e műveletek hibájának lekezelésére saját kivételosztályt használjunk, és a hiba okát is jelenítsük meg [5]!

15.1. Egészítsük ki a `Számítógép` osztályt egy olyan metódussal, amely a következő feltételek alapján eldönti egy objektumról, hogy korszerű-e! Korszerű, ha a processzor sebessége minimum 1.6 GHz-es, a memória legalább 2048 MB-os, valamint a merevlemez nagyobb, mint 160 GB. Ezek a számítógépek korszerűek?

- `Számítógép szg3:` AMD, 2.8 GHz, 3072 MB, 500 GB
- `Számítógép szg4:` Intel, 2.6 GHz, 1024 MB, 320 GB

15.2. Származtassunk a `Notebook` osztályból egy `Pda` osztályt! Az új osztály objektumai a notebook tulajdonságain kívül egy új – egész értékű – `súly` adattaggal is rendelkeznek, amely a `pda` súlyát tárolja grammban kifejezve. Készítsünk az új adattaghoz lekérdező és beállító metódust, valamint az osztályhoz egy konstruktort, amellyel mindegyik adattagjának kezdőérték adható!

15.3. Definiáljuk felül a `Pda` osztály `Notebook` osztálytól örökölt `toString()` metódusát úgy, hogy az örököltek mellett a `súly` adattagot is jelenítse meg, valamint a `hdd` adattag neve „HDD” helyett „háttértár” legyen! Az alábbi `pda` példányt hozzuk létre, és jelenítsük meg a képernyőn!

- `Pda p1:` Samsung, 0.4 GHz, 512 MB, 64 GB, 3 óra, 125 g

15.4. Módosítsuk a `Számítógép` osztály `korszerű()` metódusát úgy, hogy az alábbi feltételek teljesülése esetén a `Notebook` és a `Pda` osztály példányait is – a kategóriájuknak megfelelő paraméterekhez viszonyítva – minősíteni tudjuk! Jelenítsük meg a képernyőn a már létező `nb1`, `nb2` és `p1` példányok minősítését!

Korszerűek, ha teljesítik az alábbi paramétereket:

	CPU sebesség (GHz)	RAM (MB)	Háttértár (GB)	Üzemidő (óra)	Súly (g)
notebook	>1	>=1024	>=120	>=3	
pda	>0.3	>=128	>=16	>=2.5	<150

9.2. Fogalomtár

Absztrakt osztály: Olyan osztály, amely nem példányosítható, mert van olyan metódusa, amely nincs az adott osztályban implementálva. Az implementálás – származtatás után – a utódosztályban történik meg.

Adatmező: Lásd adattag!

Adattag (attribútum, adatmező): Az osztálydefiníció része, az osztály objektumainak (példányainak) egyedi jellemzőit, tulajdonságait tároló adattípus. Lehetnek példányszintűek és osztályszintűek is.

Aktuális paraméterlista: Kifejezések vesszővel ellátott felsorolása. Paraméterátadásnál az aktuális paraméterek típusának rendre – tehát megfelelő sorrendben, páronként – meg kell egyeznie a formális paraméterlistában megadottakkal.

Attribútum: Lásd adattag!

Destruktor: A konstruktor által lefoglalt erőforrások felszabadítását végzi. A Java-ban automatikusan is megtörténik, ha egy objektumra minden hivatkozás megszűnik.

Dinamikus (késői) kötés: A hívott metódus futásidejű hozzárendelése az objektumhoz.

Egységbezárás: Adattagok és metódusok osztályba történő összezárását jelenti. Egy objektum (példány) állapotát (adattagjait) csak metódusai által módosíthatjuk.

Eljárás: Olyan metódus, amelynek nincs visszatérési értéke.

Formális paraméterlista: Típusokból és szintaktikailag lokális változónak minősülő adattagokból álló párosok, amelyek az aktuális paraméterlista felépítését meghatározzák.

Függvény: Olyan metódus, amelynek van visszatérési értéke.

Kivételkezelés: A program futása során keletkező hibák kiszűrését megvalósító olyan programszerkezet, amelyben a tevékenységeket végző programkód határozottan elkülönül az azokban előforduló hibák lekezelését végző programkódtól.

Konstruktor: Egy osztály olyan speciális metódusa, amely a példányosítást végzi. Egy osztálynak több – szigorúan azonos nevű – konstruktora is lehet.

Metódus (módszer): Az osztálydefiníció része, az osztály objektumainak viselkedését leíró programrutin. Lásd még osztályszintű metódus!

Metódus felülírása: Azonos nevű, de különböző formális paraméterlistával rendelkező metódusok. Dinamikus kötéssel jön létre.

Metódus túlterhelése: Azonos nevű, de különböző formális paraméterlistával rendelkező metódusok. Statikus kötéssel jön létre.

Módszer: Lásd metódus!

Objektum: Olyan programozási eszköz, amelynek neve, adattagjai által reprezentált állapota, és metódusaival jellemzett viselkedése van. Egy osztály konstruktor általi példányosítása során jön létre.

Objektum-orientált programozás: Olyan programozási paradigma, amely a programot objektumokból építi fel. Az objektumok viselkedése adja a program működését.

Osztály: Olyan felhasználói típus (tervrajz), amely alapján objektumok (példányok) hozhatók létre. Legfontosabb részei a neve, adattagjainak és metódusainak definíciója.

Osztályszintű adattag: Olyan adattag, amely egy osztályon belül csak egyszer létezik, és magához az osztályhoz kapcsolódik. Lásd még példányszintű adattag!

Osztályszintű metódus: Olyan metódus, amely közvetlenül az osztályhoz kapcsolódik, nem pedig a példányokhoz. Az osztálymetódust az objektumok közösen használhatják.

Osztályváltozó: Lásd osztályszintű adattag!

Öröklődés: Az újrafelhasználás eszköze. Leszármazott (gyermek-, al-) osztály létrehozása már meglévő (szülő-, ős-) osztályból, amely során a leszárazott osztály örökli az ősosztály adattagjait és metódusait.

Paradigma: Személet- és gondolkodásmód, egy tudományos közösség tagjai által elfogadott értékek és módszerek összessége.

Példány: Lásd objektum!

Példányosítás: Egy osztály objektumának előállítása konstruktor segítségével.

Példányszintű adattag: Olyan adattag, amely egy osztály minden példányában egyedileg létezik, tehát a példányhoz kapcsolódik. Lásd még osztályszintű adattag!

Példányváltozó: Lásd adattag, példányszintű adattag!

Polimorfizmus: Egyrészt jelenti a metódusok felülírását (dinamikus polimorfizmus, újraimplementálás), amely során egy leszármazott osztályban felülírjuk (módosítjuk) az ősosztály egy metódusát, másrészt jelenti a metódusok túlterhelését (statikus polimorfizmus, többalakúság), amikor egy osztályon belül ugyanolyan néven több metódus is létezik, és csak formális paraméterlistájuk alapján különböztethetők meg. Statikus polimorfizmust valósítanak meg az osztályok konstruktorai is.

Statikus (korai) kötés: A hívott metódus fordításidejű hozzárendelése az objektumhoz.

Üzenet: Egy objektum metódusának másik objektum által történő meghívását jelenti. A megszólított objektum a kérés végrehajtásával válaszol az üzenetre.

Végleges osztály: Olyan osztály, amelynek metódusai nem definiálhatók felül, így az öröklődésük sem valósítható meg.

10. Köszönetnyilvánítás

Köszönetet mondunk mindazoknak, akik munkánkat segítették, és tanácsokkal láttak el bennünket.

Köszönjük családtagjainknak mindazt a megértést és türelmet, amit a dolgozat készítése során tanúsítottak.