

Programozás Javában

Tömösközi Péter

MÉDLAINFORMATIKAI KIADVÁNYOK

Programozás Javában

Tömösközi Péter



Eger, 2013



Korszerű információtechnológiai szakok magyarországi adaptációja

TÁMOP-4.1.2-A/1-11/1-2011-0021

Nemzeti Fejlesztési Ügynökség
www.ujsechenyiterv.gov.hu
06 40 638 638



A projekt az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg.

Lektorálta:

Nyugat-magyarországi Egyetem Regionális Pedagógiai Szolgáltató és
Kutató Központ

Felelős kiadó: dr. Kis-Tóth Lajos

Készült: az Eszterházy Károly Főiskola nyomdájában, Egerben

Vezető: Kérészy László

Műszaki szerkesztő: Nagy Sándorné

Tartalom

1.	<i>Bevezetés</i>	9
1.1	Célkitűzések, kompetenciák, a tantárgy teljesítésének feltételei	9
1.1.1	Célkitűzés.....	9
1.1.2	Kompetenciák.....	9
1.1.3	A tantárgy teljesítésének feltételei	10
1.2	A kurzus tartalma	11
1.3	Tanulási tanácsok, tudnivalók	11
2.	<i>Lecke: Bevezetés a magas szintű programozási nyelvek használatába</i>	13
2.1	Célkitűzések és kompetenciák	13
2.1.1	Kompetenciák.....	13
2.2	Tananyag	14
2.2.1	Magas szintű programozási nyelvek kialakulása	14
2.2.2	Fordítás, értelmezés	17
2.2.3	Magas szintű programozási nyelvek csoportosítása	18
2.2.4	A programok hordozhatóságáról.....	21
2.2.5	Alkalmazásfejlesztés Javában	23
2.2.6	A NetBeans IDE telepítése és használata	24
2.3	Összefoglalás, kérdések	27
2.3.1	Összefoglalás	27
2.3.2	Önellenőrző kérdések.....	28
3.	<i>Lecke: Bevezetés a Java használatába. A Java nyelv alapvető elemei</i>	29
3.1	Célkitűzések és kompetenciák	29
3.2	Tananyag	29
3.2.1	Ismerkedés a NetBeans integrált fejlesztői környezettel.....	30
3.2.2	A Java nyelvű programok általános felépítése, kódolási szabályok	33
3.2.3	Hello World!	36
3.2.4	Egyszerű kiíratás	38
3.2.5	Adatok a programban.....	42
3.2.6	Nevesített konstansok	45
3.3	Összefoglalás, kérdések	46

3.3.1	Összefoglalás	46
3.3.2	Önellenőrző kérdések.....	46
4.	<i>Lecke: Vezérlési szerkezetek és utasítások.</i>	
	<i>Kivételkezelés Javában</i>	47
4.1	Célkitűzések és kompetenciák	47
4.2	Tananyag	47
4.2.1	Algoritmusból program	47
4.2.2	Üres utasítás	51
4.2.3	Értékadás, értékadó utasítás	51
4.2.4	A blokk utasítás	58
4.2.5	Feltétel nélküli vezérlésátadás	59
4.2.6	Kétirányú feltételes elágazás.....	60
4.2.7	Többágú elágazás	61
4.2.8	Előfeltételes ciklus.....	63
4.2.9	Végfeltételes ciklus.....	64
4.2.10	Számlálós ciklus.....	64
4.2.11	Bejáró (iterátor) ciklus.....	66
4.2.12	Kivételkezelés	67
4.3	Összefoglalás, kérdések	69
4.3.1	Összefoglalás	69
4.3.2	Önellenőrző kérdések.....	69
5.	<i>Lecke: Tömb, karakterlánc és adatbevitel</i>	71
5.1	Célkitűzések és kompetenciák	71
5.2	Tananyag	71
5.2.1	Az adatszerkezetekről dióhéjban	71
5.2.2	A tömb	73
5.2.3	Tömbök létrehozása	75
5.2.4	Értékadás tömbökben	76
5.2.5	A tömb mint referenciatípus	77
5.2.6	Karakter és String	81
5.2.7	A String osztály fontosabb metódusai.....	86
5.2.8	Adatbevitel	88
5.3	Összefoglalás, kérdések	93
5.3.1	Összefoglalás	93
5.3.2	Önellenőrző kérdések.....	93

6. Lecke: Objektumorientált programozás I. – osztályok és objektumok. Java osztálykönyvtár, csomagok	95
6.1 Célkitűzések és kompetenciák	95
6.2 Tananyag	96
6.2.1 Osztályok, objektumok, konstrukció	96
6.2.2 Még egyszer a main metódusról	100
6.2.3 Objektum létrehozása	102
6.2.4 Objektumok törlése	102
6.2.5 Csomagok	102
6.3 Összefoglalás, kérdések	104
6.3.1 Összefoglalás	104
6.3.2 Önellenőrző kérdések	105
7. Lecke: Objektumorientált programozás II. – hozzáférési tagok, objektumok kezelése.....	107
7.1 Célkitűzések és kompetenciák	107
7.2 Tananyag	107
7.2.1 A private hozzáférési tag	108
7.2.2 A public hozzáférési tag	109
7.2.3 Alprogramok	110
7.2.4 Paraméterátadás	114
7.2.5 Hatókör és élettartam	116
7.3 Összefoglalás, kérdések	118
7.3.1 Összefoglalás	118
7.3.2 Önellenőrző kérdések	119
8. Lecke: Objektumorientált programozás III. – öröklés, polimorfizmus, absztrakt osztályok	121
8.1 Célkitűzések és kompetenciák	121
8.2 Tananyag	121
8.2.1 Öröklés	122
8.2.2 Polimorfizmus	122
8.2.3 Absztrakt osztályok	123
8.2.4 A protected hozzáférési tag	123
8.3 Összefoglalás, kérdések	123
8.3.1 Összefoglalás	123
8.3.2 Önellenőrző kérdések	123

9.	<i>Lecke: Grafikus felhasználói felület készítése Java programokban, eseménykezelés I.....</i>	125
9.1	Célkitűzések és kompetenciák.....	125
9.2	Tananyag.....	126
9.2.1	AWT-komponensek áttekintése.....	126
9.2.2	Konténerek.....	130
9.2.3	Layout Managerek.....	131
9.2.4	Komponensek eseményeinek kezelése.....	133
9.2.5	Ablakok eseményei.....	135
9.3	Összefoglalás, kérdések.....	135
9.3.1	Összefoglalás.....	135
9.3.2	Önellenőrző kérdések.....	136
10.	<i>Lecke: Grafikus felhasználói felület készítése Java programokban, eseménykezelés II.....</i>	137
10.1	Célkitűzések és kompetenciák.....	137
10.2	Tananyag.....	137
10.2.1	A Swing használatának alapjai.....	138
10.2.2	Eseménykezelés.....	141
10.3	Összefoglalás, kérdések.....	142
10.3.1	Összefoglalás.....	142
10.3.2	Önellenőrző kérdések.....	143
11.	<i>Lecke: Appletek és fejlesztés mobil eszközökre</i>	145
11.1	Célkitűzések és kompetenciák.....	145
11.2	Tananyag.....	145
11.2.1	Applet létrehozása és futtatása.....	145
11.2.2	Alkalmazásfejlesztés mobil eszközökre.....	147
11.2.3	Grafikus felületű alkalmazások fejlesztése mobil eszközökön.....	148
11.2.4	A Java ME használata NetBeansben.....	149
11.3	Összefoglalás, kérdések.....	150
11.3.1	Összefoglalás.....	150
11.3.2	Önellenőrző kérdések.....	150
12.	<i>Összefoglalás</i>	151
12.1	Tartalmi összefoglalás.....	151

1. BEVEZETÉS

1.1 CÉLKITÚZÉSEK, KOMPETENCIÁK, A TANTÁRGY TELJESÍTÉSÉNEK FELTÉTELEI

1.1.1 Célkitűzés

A tantárgy célja egy modern, a szakmában sokak által használt, magas szintű programozási nyelv alapjainak megismertetése. A kurzus során a hallgatók megismerkednek a Java programozási nyelv alapvető lehetőségeivel, az objektumorientált és a vizuális programozás eszközrendszerével.

Számos más programozási nyelv közül választhatunk volna a programozás alapjainak bemutatásához. Választásunk mégis a Javára esett, mert egy nagyon széles körben elterjedt, a szakma által elismert és sokak által kedvelt programozási nyelvről van szó. Az interneten kimeríthetetlen mennyiségű segédlet, tutorial és kész komponens található a Java nyelvhez, melyek az informatika minden területét lefedik. A nyelv segítségével bármilyen típusú alkalmazást lehet fejleszteni, nemcsak számítógépre, de mobil eszközökre és egyéb hardverre is. A nyelv platformfüggetlen, ami azt jelenti, hogy a legtöbb hardver és operációs rendszer képes Java nyelvű programok futtatására. Ezek mellett nem elhanyagolandó tény, hogy a Java programozási nyelv ingyenes.

1.1.2 Kompetenciák

Tudás: a kurzus teljesítése után a hallgatók képesek lesznek az egyszerűbb programozási eszközökkel megoldható problémák algoritmizálására, és az algoritmus implementálására Java nyelven. Képesek lesznek a megoldáshoz szükséges programozási eszközök (vezérlési szerkezetek, osztályok és objektumok) kiválasztására és helyes alkalmazására. Az alapvető vezérlési szerkezetek használatát segédeszköz (jegyzet, referencia stb.) nélkül képesek alkalmazni. A kódolás, fordítás, futtatás, hibakeresés eszközeit megbízhatóan, önállóan tudják használni. Az implementáció során képesek lesznek a megfelelő segédeszközök (referencia, tutorial, példaprogram stb.) kiválasztására és hatékony alkalmazására.

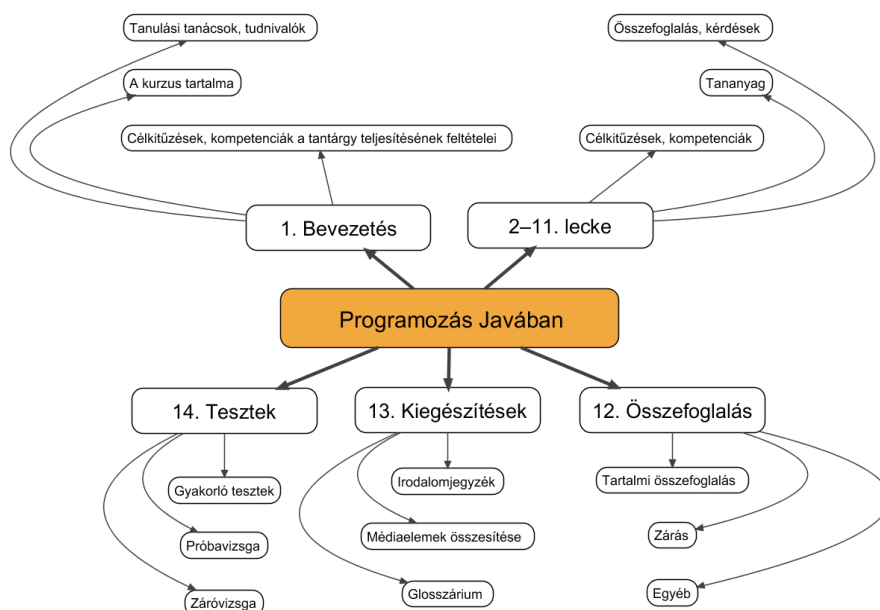
Attitűdök/nézetek: a kurzus fejleszti az algoritmikus gondolkodást, így a kurzus elvégzése után a hallgatók képesek lesznek a problémák megoldásának algoritmikus leírására, megfogalmazására. A problémák pontos leírásával, illetve azok algoritmikus megoldásával képesek lesznek a komplex feladatok részekre bontására.

Az összetett feladatok megoldása a programozók magas fokú együttműködését követeli meg. A kurzus elvégzése után a hallgatók képesek lesznek a programozásban szükséges együttműködés megvalósítására. Az együttes problémamegoldáshoz szükséges a pontos fogalmazás készsége, a segítőkészség, az együttműködésre való készség. Az összetett problémák megoldásához nem elegendő a gépies gondolkodás, a feladatok gyakran magas fokú kreativitást is igényelnek.

Képességek: a kurzus a következő képességeket fejleszti közvetlenül: áttekintő képesség, következtetési képesség, tervezési képesség, lényegfelismerés, rendszerben való gondolkodás, absztrakt gondolkodás, önállóság, stressztűrő képesség.

1.1.3 A tantárgy teljesítésének feltételei

A tanegység teljesítéséhez a hallgatók két zárthelyi dolgozatot írnak, melyeken a Java nyelv elemeinek ismeretéről adnak számot. A zárthelyi dolgozaton segédeszköz használata nem engedélyezett. A teljesítés másik feltétele egy önállóan elkészített, összetett feladat megoldása Java nyelven. A programot önállóan készítik el és a félév végén adják le. A megoldáshoz tetszőleges referencia használható. A leadott programot a hallgatók néhány perces szóbeli felellettel „védik meg”, amely során bizonyítják, hogy a leadott program a saját munkájuk eredménye.



1.2 A KURZUS TARTALMA

1. Bevezetés a magas szintű programozási nyelvek használatába
2. Bevezetés a Java programozási nyelv használatába
3. A Java nyelv alapvető elemei
4. Vezérlési szerkezetek és metódusok. Kivételkezelés
5. Tömb, karakterlánc és adatbevitel
6. Obejktumorientált programozás I. – osztályok és objektumok. Java osztálykönyvtár, csomagok
7. Obejktumorientált programozás II. – hozzáférési tagok, objektumok kezelése
8. Obejktumorientált programozás III. – öröklés, polimorfizmus, absztrakt osztályok
9. Grafikus felhasználói felület készítése Java programokban, eseménykezelés I.
10. Grafikus felhasználói felület készítése Java programokban, eseménykezelés II.
11. Java appletek. Fejlesztés mobil eszközre
12. Összefoglalás

1.3 TANULÁSI TANÁCSOK, TUDNIVALÓK

A programozás olyan tevékenység, aminek elsajátítása nem mindenki számára megy könnyedén. Talán pontosabb lenne úgy fogalmazni, hogy eleinte senki számára sem megy könnyen. Gondoljunk arra, milyen nehéz elkezdni egy új idegen nyelv megtanulását. Eleinte nagyon szokatlan a nyelv logikája. A magyarhoz képest mindent másként kell megfogalmazni, nem elegendő csak a szavakat megtanulnunk. Később, amikor már képesek vagyunk kommunikálni is az adott nyelven, nagyon sok mindent szinte észrevétlenül tanulunk meg.

A programozási nyelvek elsajátítása is hasonlóképpen történik: eleinte minden bonyolultnak tűnik, hiszen ahhoz, hogy valamit megértsünk, már tudnunk kellene olyan dolgokat is, amelyeket csak akkor érthetünk meg, ha ezt a bizonyos dolgot már értenénk. Konkrét példaként említhetjük az objektumorientált programozás nehézségeit. Senki sem úgy kezd el programozni tanulni, hogy osztályokat, virtuális metódusokat vagy épp interfészeket tervez már a legelső próbálkozáskor. Ugyanakkor muszáj már a dolgok lelegején megérteni valamelyest az osztályok és objektumok működését, hiszen ezek nélkül semmilyen programot sem készíthetünk Javában.

Ez a „róka fogta csuka, csuka fogta róka” szituáció hamar megváltozik, ahogyan az olvasó megérti a programozás, és főként a Java nyelv logikáját. Ezért talán a legcélszerűbb tanulási tanácsunk az lehet, hogy eleinte ne törekedjünk arra mindenáron, hogy a *miért*eket is megértsük. A jegyzet írásakor szándékosan követtük el itt-ott azt a hibát – ha ez egyáltalán hiba –, hogy nem mindent írtunk le precízen. Az ilyen, időnként talán felületesnek tűnő részek később újra előkerülnek, amikor már pontosabb, precízebb magyarázatokkal szolgálunk velük kapcsolatosan.

Egyetlen könyvből, de még több könyvből sem lehet megtanulni a programozást. Csak könyvekből nem. Ahogyan egy idegen nyelv esetében is a gyakorlat teszi a mestert, a programozási nyelveket is csak úgy lehet megtanulni hatékonyan, ha az ember gyakorol, kísérletezik, próbálgat. Második tanulási tanácsunk tehát az, hogy aki meg akar tanulni programozni, az programozzon minél többet!

Harmadik, egyben utolsó tanácsunk, hogy senki se adja fel az első kudarcnál! Vannak olyan nyelvek, amelyek talán könnyebben megtanulhatók, mint a Java, bár léteznek nála sokkal összetettebb, absztraktabb programozási nyelvek is. Szinte biztos, hogy eleinte érik majd kudarcok a nyelvvel most ismerkedőket, de reméljük, hogy a jól működő programok elegendő sikerélményt adnak majd nekik ahhoz, hogy a kudarcok és a nehézségek ellenére kedvüket leljék a programozásban.

2. LECKE: BEVEZETÉS A MAGAS SZINTŰ PROGRAMOZÁSI NYELVEK HASZNÁLATÁBA

2.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

Ez a lecke bemutatja a magas szintű programozási nyelvek kialakulását, típusait és jellemzőit. A cél az, hogy az olvasó megismerje a nyelvek általános jellemzőit, illetve a Java nyelv előnyeit a többi magas szintű programozási nyelvvel szemben. A lecke második felében bemutatjuk a Java sajátosságait, illetve egy olyan integrált fejlesztői környezet telepítését, amelynek segítségével gyorsan és kényelmesen tudunk Java programokat készíteni.

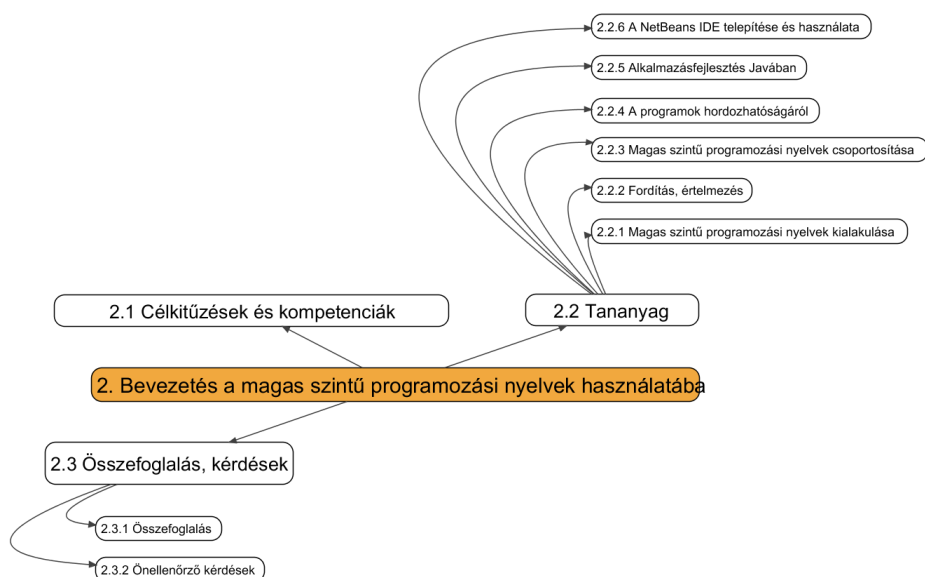
2.1.1 Kompetenciák

Tudás: a lecke végén a hallgatók ismerni fogják a ma használatos programozási nyelvek típusait, és legáltalánosabb jellemzőit. Megismerik a Java nyelv sajátosságait a többi nyelvvel szemben, és megismerik a NetBeans 7.2 nevű integrált fejlesztői környezet telepítését.

Attitűdök/nézetek: a komplex feladatok részekre bontásához a problémák pontos leírása elengedhetetlen. A programozás megtanulásához meg kell érteni a számítógép működését, elsősorban a processzor és a memória szerepét a műveletek végrehajtásában. A magas szintű programozási nyelvek előnyeinek felismeréséhez be kell látnunk, hogy egy adott probléma megoldása milyen erőforrásokat igényelne az adott nyelv eszközei nélkül, és milyen előnyökkel jár a nyelv használata.

Képességek: a lecke a következő képességeket fejleszti közvetlenül: áttekintő képesség, következtetési képesség, tervezési képesség, lényegfelismerés, rendszerben való gondolkodás, absztrakt gondolkodás.

2.2 TANANYAG



2.2.1 Magas szintű programozási nyelvek kialakulása

A számítógépes program utasítások olyan sorozata, amelyek azt mondják meg a számítógépnek, hogy annak mit kell csinálnia az adatokkal. Egy program tehát olyan eszköz, amely a bemenő adatokból meghatározott transzformációk során kimenő adatokat állít elő.

Köztudott, hogy a futtatott program utasításait a memória tárolja, a futáshoz szükséges adatokkal együtt. A program utasításait a processzor hajtja végre. A processzor azonban csak egyféle programot tud végrehajtani: olyat, amelyet az ő saját utasításkészletén írtak. Minden processzortípusnak egyedi jellemzői vannak, így egyedi a különböző processzorok utasításkészlete is. A processzorok saját utasításkészletén írt programokat nevezzük *gépi kódnak*. A gépi kód lényegében bináris számjegyek sorozataként írható fel, egy-egy utasítás 8, 16, 24 stb. jegyű bináris jelsorozat. Egy ilyen gépi kódú utasítás általában nem eredményez látványos hatást (pl. beír egy memóriaterületre egy 1 bájtos számot, vagy növeli/csökkenti egy memóriacellában található szám értékét, stb.), főleg, ha ún. magasszintű programozási nyelvek utasításaihoz hasonlítjuk őket. A gépi kódú programozás tehát azért nehéz, mert nagyon elemi szintű utasítások állnak csak rendelkezésünkre, másrészt pedig az emberi szem számára lényegében olvashatatlan egy ilyen kód: kevés olyan ember van, aki többezer bináris számjegyből álló sorozatot gépi kódként azonnal meg tud érteni. A szá-

mítástechnika hőskorában azonban ez volt az egyetlen lehetséges programozási módszer.

Az első eszköz, amely kicsit barátságosabbá tette a programozást, az *assembly* nyelv volt. Ez a gépi kód egy olvashatóbb jelölésmódját jelenti, ahol az utasításokat bináris számjegyek helyett 2-3 betűs szavak, rövidítések, ún. *mnemonikok* jelölik. Lássunk egy példát:

gépi kód

10110000 01100001

assembly

MOV AL, 061h

Az utasítás jelentése: „tedd a hexadecimális 61 értéket a processzor AL regiszterébe!”

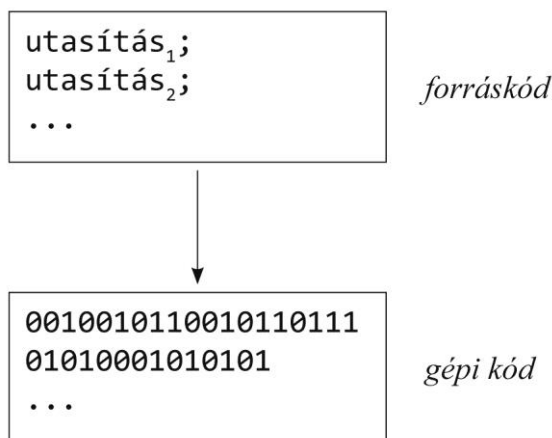
A processzorok azonban közvetlenül az assembly nyelvet sem értik, ahhoz tehát, hogy egy ilyen nyelven írott programot futtatni tudjunk, az assembly nyelvű kódot gépi kódra kell fordítani. Az ilyen fordítóprogramokat nevezzük *assemblernek*.

Az 1950-es évektől kezdtek megjelenni az ún. magas szintű programozási nyelvek¹. Ezek közös jellemzője, hogy közelebb állnak az ember beszélte (általában angol) nyelvhez, mint a gépi kódhoz. Magas szintű programozási nyelven való programozáshoz nem (feltétlenül) szükséges ismernünk a számítógép hardverének (elsősorban a processzor és a memória) felépítését, sőt, sok magas szintű programozási nyelvben nincs is eszközünk arra, hogy közvetlenül nyúljunk a memóriához vagy a processzorhoz. A magas szintű programozási nyelvekben különböző adattípusokat és adatszerkezeteket, illetve összetett hatású utasításokat használhatunk. (A nagyságrend érzékeltetéséhez: gépi kódban egy utasítással beírhatunk egy adatot egy bizonyos memóriacímre, míg egy erre alkalmas magas szintű programozási nyelvben például egyetlen utasítás segítségével a képernyő felbontásához igazítva és a videokártya sajátosságait figyelembe véve lejátszhatunk egy videoállományt.)

A probléma azonban a magas szintű programozási nyelvek esetében is ugyanaz: a processzor a magas szintű nyelvek utasításait nem képesek közvetlenül megérteni és végrehajtani. Ezért minden magas szintű programozási nyelv

¹ Az első magas szintű nyelv a FORTRAN volt, melynek leírását 1954-ben tette közzé az IBM. Az első, gyakorlatban használt programozási nyelv tíz évvel korábban készült, Konrad Zuse készítette 1944-ben Plankalkül néven a Z4 jelű számítógépéhez.

használatára során a programozónak szüksége van egy olyan eszközre, amely a magas szintű programozási nyelv utasításait gépi kódra fordítja le. A fordítás történhet közvetve vagy közvetlenül, illetve a magas szintű nyelven írott kód (más szóval a forráskód) egészéből vagy csak annak egyes utasításaiból. Vannak olyan fordítóprogramok, amelyek azonnal futtatható gépi kódot állítanak elő, míg más fordítók nem közvetlenül gépi kódra, hanem egy másik (akár magas szintű) programozási nyelvre végzik a fordítást. Az ún. értelmezőprogramok pedig nem a teljes forráskódot fordítják le, hanem utasításonként végzik a fordítást, és a számítógép minden lefordított utasítást azonnal végre is hajt.



1. ábra: Forráskód – gépi kód

Kissé pontatlanul megtehetjük az alábbi megállapításokat:

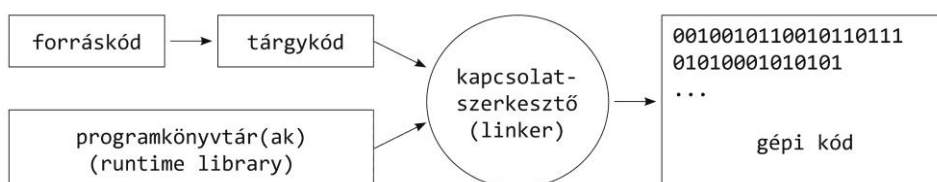
- mivel a magas szintű programozási nyelvek sokkal több kész eszközt adnak a programozó kezébe, a magas szintű programozási nyelveken egyszerűbb a programozás;
- mivel gépi kódban a programozó közvetlenül programozza a processzort, sokkal hatékonyabb kódok írhatók, mint amilyen gépi kódot a magas szintű programozási nyelvek fordítóprogramjai állítanak elő.

Mindkét állításban van igazság, de ennél azért lényegesen összetettebb kérdésről van szó. A laikus számára egy magas szintű programozási nyelven írott kód is lehet éppen annyira értelmezhetetlen, mint egy gépi kódú, másrészt a programozási nyelvek fordítóprogramjai egyre fejlettebbek, és az új fordítók képesek lehetnek közel olyan hatékonyságú gépi kód előállítására, mint amelyet egy gépi kódban jártas programozó készít.

2.2.2 Fordítás, értelmezés

Ahogy az előző fejezetben utaltunk rá, a magas szintű programozási nyelven megírt forráskódból többféle módszerrel készülhet a processzor számára értelmezhető gépi kód. A két fő módszer a fordítás és az értelmezés, előbbit a fordítóprogramok (compiler), utóbbit az értelmező programok (interpreter) végzik.

A fordítás során a fordítóprogram a teljes forráskódot lefordítja egy másik nyelvre. Ez lehet gépi kód is, de nem kizárólag csak az. Azonban az így lefordított gépi kód sem biztos, hogy azonnal futtatható a processzor számára. Gyakran szükséges egy vagy több másik eszköz alkalmazása is, pl. kapcsolatszerkesztő (linker), betöltő (loader) stb. Ezek ismerete azonban nem szükséges ahhoz, hogy programokat tudjunk készíteni, és a módszerek programozási nyelvenként, környezetenként, operációs rendszerenként stb. különbözőek is lehetnek.



2. ábra: Forráskód – tárgykód – gépi kód

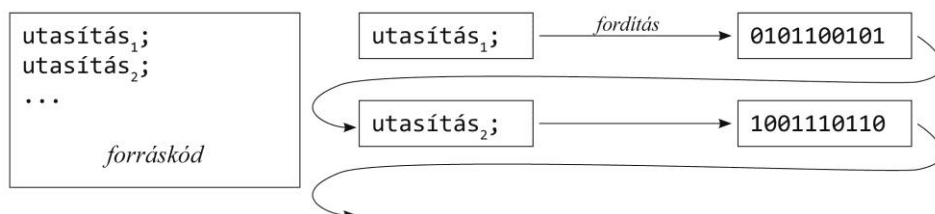
A fordítás vagy értelmezés feltétele, hogy a program szövegében ne legyenek hibák. A programozás szempontjából két alapvető csoportba soroljuk a forrásszövegben előforduló hibákat: szintaktikai hibáknak nevezzük azokat a hibákat, amelyek a nyelv „nyelvtanát” sértik. Ilyen jellegű hibát vétünk, ha például gépelési hibát ejtünk egy utasítás leírásakor. A szintaktikai hiba tehát az adott programozási nyelv szabályait sértő „helyesírási hiba”.

A hibák másik csoportjába tartoznak az ún. szemantikai hibák. A szemantikai hibák nem az adott nyelv szintaxisát sértik, ezért a fordítóprogramok nem is mindig tudják őket kiszűrni. Ezek a hibák programhelyességi hibák. Ez azt jelenti, hogy egy szemantikailag hibás program vagy utasítás ugyan átmegy a szintaktikai ellenőrzésen, de a program helytelenül fog működni, mert az adott utasítások a végrehajtáskor hibát eredményeznek. Szemantikai hiba lehet például egy x / y típusú osztási művelet, ha megengedjük, hogy az y értéke 0 is lehet.

A fordítóprogramok sajátossága, hogy mivel a teljes forráskódot próbálják lefordítani, így a fordítás során összegyűjtik az összes szintaktikai hibát (fejlettebb fordítók a szemantikai hibák egy részét is), és a programozónak egy teljes

hibalistát adnak vissza. A fordítás ilyen esetben természetesen nem történik meg.

Az értelmezőprogramok rendszerint szoros kapcsolatban állnak a futtató rendszerrel is. Az értelmezők (interpreterek) nem a teljes forráskódot fordítják, hanem mindig csak a következő utasítást. A futtató rendszer az így lefordított utasítást azonnal végre is hajtja. A végrehajtás tehát a fordítás – futtatás – fordítás – futtatás stb. séma szerint történik. Az értelmezés sajátossága az, hogy mivel nem a teljes forráskód fordítása történik meg, a program futtatása akár úgy is elkezdődhet, hogy a forráskódban szintaktikai hibák vannak. A program akár hosszú időn keresztül is futthat, amíg a vezérlés át nem adódik egy hibás utasításra. A program futása ilyenkor megáll.



3. ábra: Fordítás – futtatás – fordítás – futtatás

2.2.3 Magas szintű programozási nyelvek csoportosítása

A magas szintű programozási nyelvek kialakulása az 1950-es évek végén kezdődött. Ezt megelőzően a programozók kizárólag gépi nyelven (gépi kódban) készíthették programjaikat, amely nemcsak nagyon messze áll az emberi gondolkodástól, de a hardver maximális ismeretét is megköveteli. Az assemblerek megjelenésével a gépi kód assemblyre cserélhető volt, amely még mindig szinte gépi kódú programozást jelentett (logikáját tekintve feltétlenül), de legalább a 0-k és 1-ek végtelen sorozata helyett a programozó egy valamelyest olvashatóbb és áttekinthetőbb kóddal dolgozhatott a mnemonikok révén.

Az 1950-es évek végén megjelenő magas szintű programozási nyelvek hozták meg az igazi áttörést. Azóta több ezer magas szintű nyelvet specifikáltak és implementáltak. Ezeket a nyelveket, pontosabban a ma használatos programozási nyelveket alapvetően négy csoportba sorolhatjuk. A négy csoport négy alapvető számítási modellhez kapcsolódik: az imperatív, az applikatív, a szabály alapú és az objektumorientált programozáshoz.

Imperatív, procedurális nyelvek

Az első magas szintű programozási nyelvek, az ún. ősnelvek ebbe a kategóriába tartoznak, ez a megközelítés az 1970-es évekig volt népszerű. Az imperatív nyelveken írt programok utasításokból, illetve alprogramhívások sorozatából áll. Az adatok tárolásához változókat használhatunk, az utasítások a memóriában tárolt értékek megváltoztatását eredményezik. A programozás tulajdonképpen a megfelelő memóriaállapotok létrehozását jelenti, amely végül a megoldáshoz vezet. A procedurális nyelven írt programok általános alakja a következő:

```
utasítás1;  
utasítás2;  
...
```

A legismertebb (máig használt) imperatív programozási nyelvek a következők: FORTRAN, COBOL, PL/1, ALGOL, C, Pascal.

Applikatív, funkcionális nyelvek

A funkcionális programozási nyelvek jóval közelebb állnak a formális matematika eszközszeréhez. Az ilyen nyelvek tisztán függvényekre alapozottak, alapvetően háromféle művelet áll a programozók rendelkezésére: függvény-összehasonlítás, rekurzió és feltételes kifejezés. A funkcionális nyelveknél a *futtatható program* kifejezés helyett gyakran a *kiértékelendő kifejezés* fogalmat használjuk. Egy funkcionális program általános formája a következő:

```
függvényn (... függvény2 (függvény1) ) )
```

A legismertebbfunkcionális nyelvek a következők: LISP, Haskell, Erlang, Miranda.

Szabály alapú, logikai nyelvek

Ezek a programozási nyelvek interaktív eszközök. A nyelvek alapját a formális matematika, illetve a halmazelmélet szabályai adják. A nyelvekben a programozók szabályokat definiálnak. A program feladata egy olyan eset (megoldás) keresése, amely egy adott kérdésre a programból logikailag következik, azaz levezethető. Az ilyen nyelvekben a programozónak nem kell ismernie a megoldás algoritmusát (ha egyáltalán létezik ez az algoritmus), a megoldás keresését maga a programozási nyelv végzi.

A logikai nyelven írt programok általános alakja a következő:

```
feltétel1 -> akció1  
feltétel2 -> akció2
```

```
...
feltételn -> akción
```

A legismertebb logikai nyelv a Prolog.

Objektumorientált nyelvek

Az objektumorientált nyelvek a procedurális nyelvek továbbfejlesztéséből alakultak ki. Az objektumorientált paradigma (objektumorientált programozás, OOP) alappillérei a következők: egységbe zárás (encapsulation), öröklődés (inheritance) és sokoldalúság (polymorphism).

Az egységbe zárás lényege, hogy szemben az imperatív nyelvekkel, ahol az adatok és a rajtuk végezhető/végzett műveletek egymástól elszigetelt, majd-hogynem független egységet alkotnak, az OOP egységbe zárja az adatokat és a rajtuk értelmezett műveleteket. Az OOP alapja az objektum, amelynek mezői jelentik az objektumban tárolt adatokat, a metódusok pedig a rajtuk értelmezett és végezhető műveleteket.

Az objektumok típussal rendelkeznek, az objektumok típusát nevezzük osztálynak. Az osztály leírja egy objektum által tárolható adatokat és a rajtuk végezhető műveleteket, egy objektum (példány) pedig ténylegesen megvalósítja ezeket. Felfoghatjuk ezt úgy is, hogy az osztály a tervrajz, amely alapján elkészíthető egy konkrét dolog. Egy ház tervrajza alapján sok egyforma házat lehet építeni, amelyek azonban különbözhetnek egymástól például abban, hogy milyen színűre festették le a külső falakat. A ház tervrajza az osztály, amely alapján elkészíthetők a konkrét házpéldányok, azaz az objektumok.

Az alábbi animációban az autók osztályából példányosítunk két konkrét autót.

Az OOP második pillére az öröklődés. Ez azt jelenti, hogy ha rendelkezünk egy osztállyal, akkor abból létre tudunk hozni egy másik osztályt úgy, hogy az örökli az őosztály minden mezőjét és metódusát, de újabbakkal is ki tudjuk azt egészíteni. Ezáltal egy bonyolultabb osztály definiálását nem kell teljesen az alapoktól kezdenünk, hanem felhasználhatunk már kész osztályokat. Egyszeres öröklésről beszélünk abban az esetben, ha az új osztálynak csak egy őosztálya lehet, és többszörös öröklődésről akkor, ha egy új osztály megalkotásához egyszerre több őosztályt is bevonhatunk. Az öröklődés révén osztályhierarchia alakítható ki.

A harmadik pillér a többalakúság (polimorfizmus). Ezt azt jelenti, hogy egy adott metódus azonosítója közös lehet egy adott osztályhierarchián belül,

ugyanakkor a hierarchia minden egyes osztályában a tevékenységet végrehajtó függvény megvalósítása az adott osztályra nézve specifikus lehet.

A nyelvek fejlődése során az OOP újabb és újabb követelményeket támasztott a programozási nyelvekkel és megalkotóikkal szemben. A legelső (és máig használt) OOP nyelv a Smalltalk volt, amelyet Alan Kay alkotott meg 1969 és 1972 között. Az új nyelvek megjelenését, illetve a már létező nyelvek OOP-s eszközökkel való felvértezését illetően a következő állomásokkal találkozunk: objektumalapú nyelvek, osztályalapú nyelvek, objektumorientált nyelvek.

Az objektumalapú nyelvek nyelvi eszközként támogatják az objektumot, de az osztály fogalma még nem definiált.

Az osztályalapú nyelvek bevezetik az osztály fogalmát, nincs azonban osztályhierarchia.

Az objektumorientált nyelvek az OOP mindhárom alappillérét ismerik és támogatják. Az objektumokat osztályok alapján hozzuk létre példányosítással, definiálhatók az osztályok és az öröklődés révén osztályhierarchia hozható létre.

A Java egy tisztán objektumorientált nyelv, amely az egyszeres öröklést vallja. További OOP nyelvek: SIMULA, Smalltalk, C#.

A 6–8. fejezet a Java OOP-s eszközeit mutatja be részletesen.

2.2.4 A programok hordozhatóságáról

Az 1960-as évektől programozási nyelvek ezreit specifikálják és implementálják.² A nyelvek fejlődésével kialakult az a szokás, hogy a sikeresebb programozási nyelvekhez több változatban is készültek fordító vagy értelmező programok. A különböző számítógépek azonban hardverüket tekintve inkompatibilisek voltak, ami azt is jelenti, hogy a gépi kódjuk eltérő volt egymástól. Hiába készült egy helyes gépi kódú program egy számítógépre, azt egy másik gép biztosan nem tudta futtatni, mivel egészen más processzor és egyéb hardvereszközök voltak bennük.

Az inkompatibilitás sajnos a programozási nyelvekre is igaz volt. Az egyes gépi implementációk (fordítóprogramok) nem mindig követték a hivatkozási nyelvet, így a különféle típusú számítógépeken készült, eredetileg azonos nyelvű forráskódok általában nem voltak lefordíthatók (és futtathatók) más számítógépen. Közismert példa erre a nyolcvanas években nagyon népszerű mikro-

² A specifikáció az ún. referencia nyelv vagy hivatkozási nyelv létrehozását jelenti. A hivatkozási nyelv nem egy konkrét eszköz, hanem egy absztrakt szabvány, tulajdonképpen a nyelv elméleti leírását tartalmazza. A konkrét gépi megvalósítások elkészítése (a fordítóprogram megírása különféle hardver- és szoftverkörnyezetben) jelenti a nyelv implementációját.

számítógépek (Commodore, Videoton TV Computer, Atari, stb.) BASIC nyelvei. Ezek a számítógépek úgy kerültek forgalomba, hogy mindegyik ROM memóriájában található volt egy BASIC interpreter, azaz a számítógép bekapcsolása után a gép BASIC nyelven azonnal programozható volt (nem kellett külön betölteni lemezzről vagy kazettáról az értelmezőprogramot). BASIC-nek hívták az összes nyelvet, amelyet ezek a gépek ismertek, de ezek a BASIC-ek a legalapvetőbb szinten sem voltak egymással kompatibilisek, annyira eltérő volt az utasításkészletük és a szintaxisuk. Hiába készítettünk tehát egy jól működő BASIC programot például Commodore 64-en, ez a program szinte biztosan nem volt futtatható egy Videoton TV Computeren átalakítás nélkül.³

A programok hordozhatósága azt jelenti, hogy egy adott számítógépen megírt program egy másik típusú (hardverű, operációs rendszerű) számítógépen is futtatható. Mivel a különböző számítógépek processzora nagyon különböző utasításkészlettel rendelkezik és a gépeken a memóriacímek kiosztása is teljesen eltérő egymástól, a gépi kód szintjén a programok gyakorlatilag nem hordozhatók. A magas szintű nyelvek esetében a hordozhatóság megoldható lehetne, ha a fordítóprogramokat készítő szoftvercégek 100%-ig követnék a hivatkozási nyelveket. Ez viszont még sohasem sikerült senkinek.⁴

Éppen a jegyzetünkben tárgyalt Java programozási nyelv volt az első, amelyik a hordozhatóságot egy újfajta módszerrel közelítette meg. A nyelvet a Sun kezdte fejleszteni az 1990-es évek elejétől, és a fejlesztés egyik célja az volt, hogy legyen egy programozási nyelv, amelyen az interneten futó programokat lehet elkészíteni. Az internet köztudottan egy nyílt hálózat, ami azt jelenti, hogy bármilyen hardverrel rendelkező számítógép csatlakozhat hozzá, ha megfelel bizonyos követelményeknek. Ahhoz, hogy az internethez csatlakozó számítógépek mindegyikén futtatható legyen egyazon program, nyilvánvalóan hordozható programozási nyelvre van szükség.

A Java megalkotói a problémát úgy oldották meg, hogy a Java fordítóprogramok nem közvetlenül futtatható gépi kódot állítanak elő, hanem egy köztes

³ A BASIC nyelvet már sokszor, sokan leírták, pedig a nyelv a mai napig létezik és meglehetősen sok programozó használja is. Azonban azt is meg kell jegyeznünk, hogy a ma használható BASIC-ek sem arra a BASIC-re nem hasonlítanak, amelyet Kemény János és munkatársai 1964-ben specifikáltak, sem pedig arra, amelyet a '80-as években mindenki elsajátított alacsonyabb-magasabb szinten, akinek volt otthon egy Commodore vagy egyéb számítógépe.

⁴ Habár sem a HTML, sem a CSS nem programozási nyelvek, a weboldalak készítése kicsit hasonlít a programozáshoz. Aki már megpróbált weboldalt készíteni, az tudja, hogy szinte képtelenség olyan oldal létrehozása, amely minden létező böngészőben minden számítógépen, minden operációs rendszer alatt ugyanúgy néz ki és helyesen működik. Ez nagymértékben következik abból is, hogy a böngészőprogramok gyártói rendszerint nem követik 100%-ban sem a HTML, sem a CSS szabványokat.

kódot, ún. bájt-kódot. A bájt-kódot ezután a különböző hardverű és operációs rendszert futtató számítógépeken működő virtuális gépek (futtató rendszerek) interpretálják. A Java virtuális gép neve JVM (Java Virtual Machine).

Ez a módszer annyira bevált, hogy más nyelvek esetében is átvették. Közismert a Flash alkalmazás, amely egy önálló programozási nyelvet (az ActionScript nevű nyelvet) is tartalmazza. Egy Flashben készült programot a böngészőprogram nem tud lejátszani, csak akkor, ha a böngésző alá telepítetünk egy FlashPlayer nevű alkalmazást, ami egy olyan virtuális gép, amely a Flash-állományokat és ActionScript nyelvű programokat tudja futtatni.

- ☐ Java nyelvű forráskódokból készülhet azonnal futtatható gépi (más szóval natív) kód is. A Java alkalmazásokat azonban – általánosan – a mai napig a fent leírtaknak megfelelően szoktuk futtatni.



Az egyik, nagyon sokak által ismert Java nyelven írt alkalmazás az elektronikus adóbevalláshoz használható ABEV Java című program.

Ahhoz tehát, hogy Javában készült programokat futtassunk a számítógépen, a gépünkön futnia kell egy Java virtuális gépnek. Ezeket a programokat ingyen letölthetjük a www.java.com címről.

- ☐ A Java nyelvet a Sun Microsystem alkotta meg. A céget 2009-ben felvásárolta az Oracle.
- ☐ Azok körében, akik nem foglalkoztak még Java programozással, sajnálatosan széles körben terjedt el az a tévhit, hogy a Java és a JavaScript rokonok. Szeretnénk ezt a tévhitet eloszlatni, mert két, egymástól nagyon különböző technológiáról van szó. A Java egy magas szintű, hordozható, objektumorientált programozási nyelv, de egyben egy platform is. A JavaScript pedig egy olyan scriptnyelv, amelyet a böngészőprogramok tudnak értelmezni (interpretálni), és segítségével a weboldalaink tehetők interaktívvá, illetve azok tartalmát tudjuk megváltoztatni a böngészőbe való letöltés után is.

2.2.5 Alkalmazásfejlesztés Javában

Ahhoz, hogy Java nyelven írassunk programot, csak egy egyszerű szövegszerkesztőre (pl. Jegyzettömb), illetve a Java fordítóprogramra van szükségünk, amely a forráskódot bájt-kódra fordítja le. A lefordított bájt-kódot ezután a JVM le tudja futtatni, ez akár parancssorból is kezdeményezhető művelet. Ezzel a módszerrel azonban meglehetősen körülményesen lehet programot fejleszteni, bár a programozók között sokan esküsznek máig is erre a technikára.

A modern programozási nyelvekhez rendszerint rendelkezésre áll egy integrált fejlesztői környezet (IDE, Integrated Development Environment), amely magában foglal egy kifejezetten a nyelvhez kialakított szövegszerkesztőt, a fordítóprogramot, hibakeresőt (debuggert) és számos egyéb, a fejlesztést segítő eszközt. Ilyen integrált fejlesztői környezet a Javához is elérhető, a legismertebbek a NetBeans és az Eclipse.

Ebben a jegyzetben a NetBeans IDE telepítését ismertetjük.

Természetesen mindegy, hogy ezek közül választjuk ki valamelyiket, vagy más IDE-t használunk (esetleg semmilyen), hiszen a legfőbb feladatunk „csak” az, hogy szintaktikailag és szemantikailag is helyes Java-forráskódokat készítsünk, amelyeket azután a compiler működőképes bájtkóddá tud fordítani.

A lefordított bájtkódot a JRE interpretálja, vagyis futtatja le az adott hardverkörnyezetben. (JRE = Java Runtime Environment, azaz Java futtató környezet, ami virtuális gépet [JVM], illetve a futtatáshoz szükséges legfontosabb osztályokat – lásd később – tartalmazza).

A JRE minden felhasználó számára fontos, aki Java nyelven írott programot, pontosabban abból fordított bájtkódot szeretne a gépén futtatni. Integrált fejlesztői környezetre pedig csak azoknak van szükségük, akik Java nyelven szeretnének programozni.

A Java nyelv esetében mind a futtató környezet, mind pedig az alább ismertetendő integrált fejlesztői környezet ingyenesen áll a felhasználók és a programozók rendelkezésére.

2.2.6 A NetBeans IDE telepítése és használata

A NetBeans integrált fejlesztői környezet a www.netbeans.com címről tölthető le. Ennek a jegyzetnek az írásakor (2012 nyara) a 7.2-es verzió volt a legfrissebb. A webcím begépelése után az oldalon a böngészőablak felső részén azonnal látható a közvetlen letöltésre mutató link.



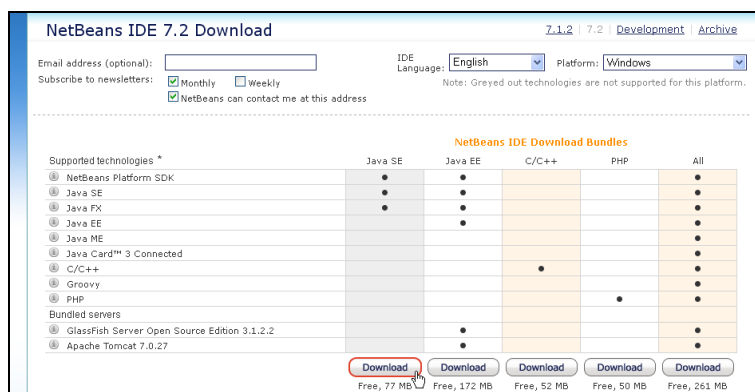
4. ábra: A NetBeans azonnali letöltésére mutató link

A gombra kattintva egy olyan oldalra jutunk, ahol kiválasztható a letölten-dő fejlesztői környezet nyelve (2012 nyarán a magyar nyelv nem szerepelt a választható nyelvek listájában). Természetesen ki kell választani az operációs rendszert, amelyben dolgozunk, illetve feliratkozhatunk hírlevelekre. Opcionáli-san megadhatjuk az e-mail címünket is.

Ez alatt egy táblázatot látunk, amelyben kiválaszthatjuk azt a fejlesztői kör-nyezetet, amely a legmegfelelőbb a számunkra.

- ☐ Több ok miatt is jogos a választás lehetősége. Egyrészt a NetBeans nemcsak a Java nyelvű fejlesztést támogatja, mivel vannak eszközei a PHP és C++ nyelvű programfejlesztés támogatásához is. Másrészt a Java nyelv is többféle szinten érhető el. A Java SE (Standard Edition) a nyelv olyan változatát jelenti, amellyel szerveren és munkaállomásokon futtatható programokat is készíthetünk. Ez a Java alapváltozata.
- ☐ A Java EE (Enterprise Edition) kifejezetten szerveralkalmazások fejlesztésére van hangolva. Támogatja a hibátűrő elosztott alkalmazások fejlesztését is. A Java SE-nél több programkönyvtárat tartalmaz, ezáltal nagy szoftverprojektekből való fejlesztéshez használható.
- ☐ A Java ME (Micro Edition) a nyelv egy olyan változatát takarja, amelyet a mobiltelefonokon és egyéb eszközökön (PDA-k, kábeltévé-csatornák vételéhez alkalmas set-top boxokhoz, nyomtatókhoz stb.) futtatható alkalmazások fejlesztésének támogatására alkottak meg.
- ☐ A nyelv különböző változatairól részletesen lehet olvasni a www.oracle.com/technetwork/java/javase/overview/index.html címen.

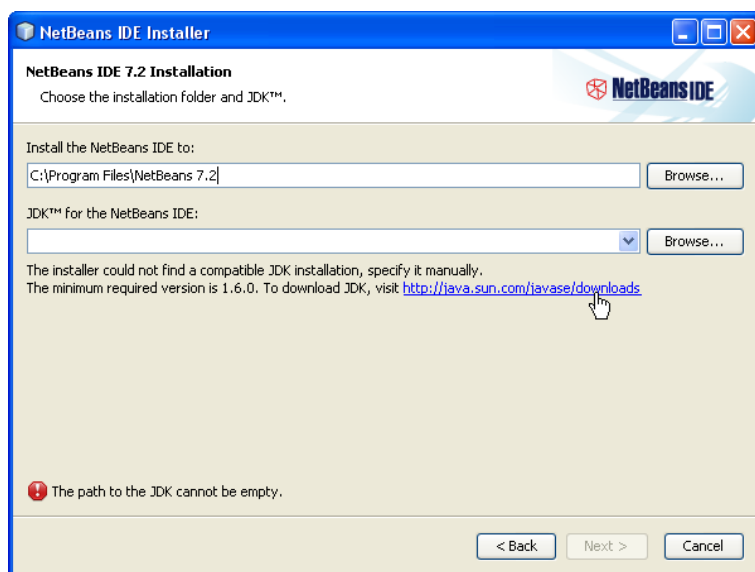
A Java nyelvvel most ismerkedők számára biztosan megfelelő lesz a Java SE változat letöltése, így célszerű az első oszlop alatti Download gomb választása.



5. ábra: NetBeans 7.2 letöltése a Java SE változathoz

A Download gombra kattintást követően újabb oldalra jutunk, ahol néhány másodperc után automatikusan meg kell jelennie a telepítő fájl letöltését engedélyező párbeszédablaknak. Töltsük le a számítógépünk merevlemezére a felkínált alkalmazást, majd indítsuk is el a telepítést!

A folyamat elején a telepítő megvizsgálja, hogy a számítógépen telepítve van-e már a JDK (Java Development Kit) minimálisan szükséges változata, ez ugyanis szükséges előfeltétel a NetBeans telepítéséhez. Ha nincs telepítve a JDK, akkor először ezt kell telepítenünk.



6. ábra: Először a JDK-t kell telepítenünk

A fenti ábrán látható link a Java weboldalára navigál bennünket, ahonnan célszerű a Java nyelv legújabb változatához tartozó JDK-t letöltenünk. Itt csak el kell fogadnunk a felhasználási feltételeket, és ki kell választanunk a kívánt operációs rendszerhez tartozó változatot.

A telepítés teljesen automatikus, és magában foglalja a JRE (Java Runtime Environment, vagyis a futtató környezet) telepítését is, ha az esetleg nem lenne a gépünkre telepítve.

- ☐ Emlékeztetőül: ha eddig nem írtunk, csak használtunk (futtattunk) Java-programokat a gépünkön, akkor elegendő, de egyben szükséges is volt a JRE telepítése a gépünkre. A JRE magában foglalja azt a virtuális gépet (JVM), amely a lefordított Java tárgykódok futtatását teszi lehetővé a számítógépünkön futó operációs rendszer alatt. Ha korábban nem is

használtunk Java-programokat a gépünkön, akkor most a JDK telepítésekor a JRE is telepítődik a gépre.

A JDK telepítése alatt a NetBeans telepítőjéből ki kell lépünk, majd a JDK sikeres telepítése után újraindíthatjuk azt.

Ha a JDK telepítése sikeres volt, akkor a NetBeans (újraindítás után) automatikusan megkeresi azt a könyvtárat a gépünkön, ahol a JDK telepítve van. Ha minden jól megy, már csak 1-2 perc, és 1-2 kattintás a Next gombon választ el bennünket a NetBeans sikeres telepítésétől.

2.3 ÖSSZEFOGLALÁS, KÉRDÉSEK

2.3.1 Összefoglalás

Ebben a leckében áttekintettük a magas szintű programozási nyelvek típusait, jellemzőit és kialakulásuk történetét. Kijelenthetjük, hogy mindenféle programozási feladat megoldható bármilyen programozási nyelvvel, de ez a kijelentés nagyon általános. Rendszerint a probléma jellege meghatározza azt is, hogy milyen típusú programozási nyelvvel érdemes hozzálátni a megoldáshoz.

A Java programozási nyelv mellett sok meggyőző érv szól. A Java egy nagyon sokak által ismert és használt magas szintű, tisztán objektumorientált programozási nyelv, amelyet nagyon sok eszközön való programozáshoz használhatunk fel hatékonyan. Segítségével nemcsak számítógépeken (munkaállomásokon), de szervereken, vagy éppen mobil eszközökön futó alkalmazásokat is fejleszthetünk.

A Java-programok hordozhatók, ami azt jelenti, hogy a forráskód átírása nélkül egy program lefuttatható különböző operációs rendszert futtató, teljesen különböző hardverrel rendelkező eszközökön is. A Java esetében ezt úgy valósították meg, hogy a forráskódból a fordítóprogram nem közvetlenül futtatható (natív) kódot állít elő, hanem egy köztes kódot. A köztes kódot a különböző eszközökön (különböző operációs rendszerek alatt) telepített JRE (Java Runtime Environment, Java futtató környezet) futtatja le. A JRE magában foglalja a JVM-et (Java Virtual Machine), amely a tárgykódot tudja interpretálni. A JRE emellett tartalmazza azokat a programkönyvtárakat is, amelyek a futtatáshoz az adott környezetben szükségesek.

A Java nyelvvel most ismerkedők számára talán a legközérhetőbb, de mindenki számára nagyon fontos érv, hogy a Java nyelv ingyenesen elérhető. Ez azt jelenti, hogy mind a fordítóprogram, mind a futtató környezet, mind pedig többféle integrált fejlesztői környezet ingyen és jogtiszta letölthető az inter-

netről. Ezen felül a Java közösség tagjai példaprogramok, tutoriálok és egyéb segítség millióit bocsátja rendelkezésünkre, rendszerint szintén teljesen ingyen.

2.3.2 Öellenőrző kérdések

1. Melyek a magas szintű programozási nyelvek általános jellemzői?
2. Mi a különbség gépi kód és assembly között?
3. Mi a különbség a fordítás és az értelmezés között?
4. Hogyan csoportosíthatók a magas szintű programozási nyelvek?
5. Mit jelent a hordozhatóság kifejezés?

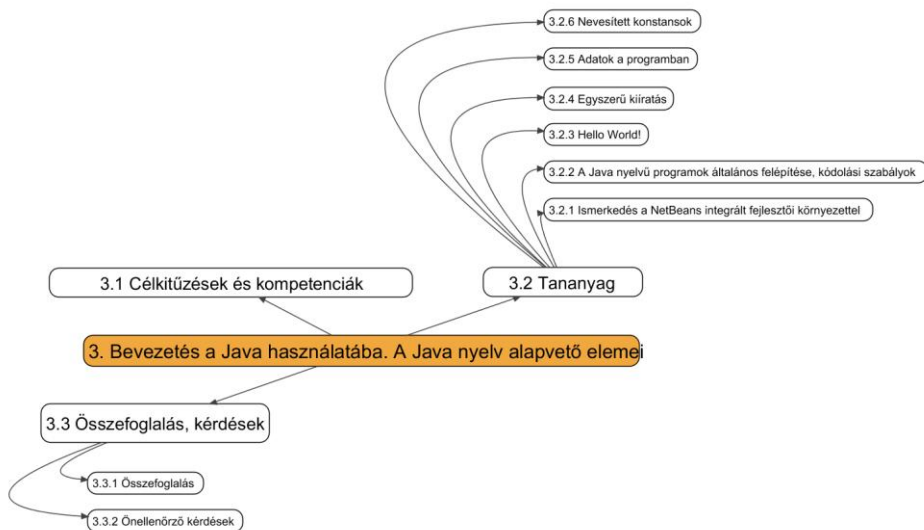
3. LECKE: BEVEZETÉS A JAVA HASZNÁLATÁBA. A JAVA NYELV ALAPVETŐ ELEMEI

3.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

A leckében elkészítjük az első Java nyelvű programot, ezáltal megismerjük a Java nyelv legalapvetőbb jellemzőit. A lecke elsajátítása alapvetően fontos, mert a legtöbb tankönyv, jegyzet, webes tutoriál és egyéb segédeszközök rendszerint nem teljes forráskódokat közölnek, hanem csak kódrészleteket. Ahhoz, hogy ezeket fel tudjuk használni a saját programok fejlesztésében illetve az ismeretszerzésben, ismernünk kell a Java nyelvű programok általános felépítését.

A leckéhez kapcsolódó kompetenciák: rendszerező képesség, logikai képesség, absztrakt gondolkodás, lényegfelismerés.

3.2 TANANYAG



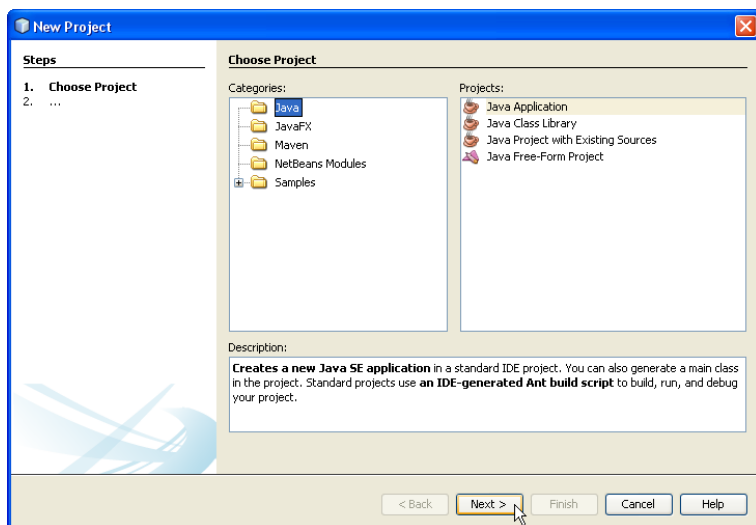
3.2.1 Ismerkedés a NetBeans integrált fejlesztői környezettel

A következőkben feltételezzük, hogy az olvasó telepítette a számítógépére a JDK-t, illetve a NetBeans 7.2 integrált fejlesztői környezetet. Természetesen Java-programok nemcsak ezzel a fejlesztői környezettel készíthetők. A Java-kódok fejlesztői környezettől függetlenek, és más integrált fejlesztői környezetek működése, használata is sokban hasonlíthat a NetBeanshez, így a közölt képernyőképek és információk más eszközök használata mellett is hasznosak lehetnek.

Indítsuk el a NetBeans 7.2 programot. A megjelenő nyitóképernyőn található információkra egyelőre nincs szükségünk, a lap bal felső sarkában látható Start Page fülön található x-szel be is zárhatjuk. A NetBeans támogatja, hogy egyszerre több projektben is dolgozzunk, és egyszerre több képernyő (ablak) is meg lehet nyitva a programban. Ezek közül a lapok tetején található „fülekkel” választhatunk.

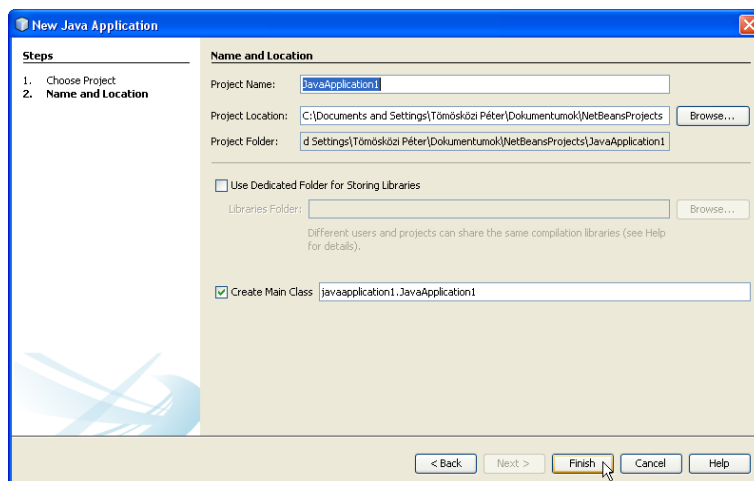
Válasszuk a File menü New Project menüpontját.

A megjelenő párbeszédablakban kategóriák között, azon belül pedig projekttypusok között lehet választani. Hagyjuk bejelölve az alapértelmezett Java kategóriát, azon belül pedig a Java Application opciót, majd kattintsunk a Next gombra!



7. ábra: Új projekt létrehozása NetBeansben

A Next gombra kattintva nevet kell adnunk a projektnek. Ennek az első alkalmazásunk esetében nincs túl nagy jelentősége, hagyjunk itt is mindent az alapbeállításoknak megfelelően.



8. ábra: Első projektünk neve maradhat az alapértelmezett név

A Finish gombra kattintva elkészül első alkalmazásunk „csontváza”. A program ablaka, egy függőleges vonallal elválasztva, két fő részre oszlik: jobb oldalon látható a program forráskódja, bal oldalon pedig azok az eszközök, amelyek a navigációt, illetve az eligazodást segítik a program szövegében. Ezeknek az eszközöknek nagy projektek esetében nagy szerep juthat, első programunk esetében csak kisebb.

A bal oldali ablakrész felső részében az eddig készített projektjeink tállózására szolgáló eszköz található. Ebben a projektek komponenseit tudjuk tállózni. Az előző leckében szó esett arról, hogy a program forráskódja, illetve az abból fordított tárgykód önmagában rendszerint kevés ahhoz, hogy a program futtatható legyen. A futtatáshoz szükséges lehet további fájlok használata is. Egy nagy projekt több forrásfájlból is felépülhet, és sok, korábban már elkészített állományt is felhasználhatnak a futtatható kód előállításához.

A Projects fül alatt a jelenlegi és már korábban fejlesztett projektjeink listája böngészhető, illetve a projektek futtatásához szükséges programkönyvtárak elemei.

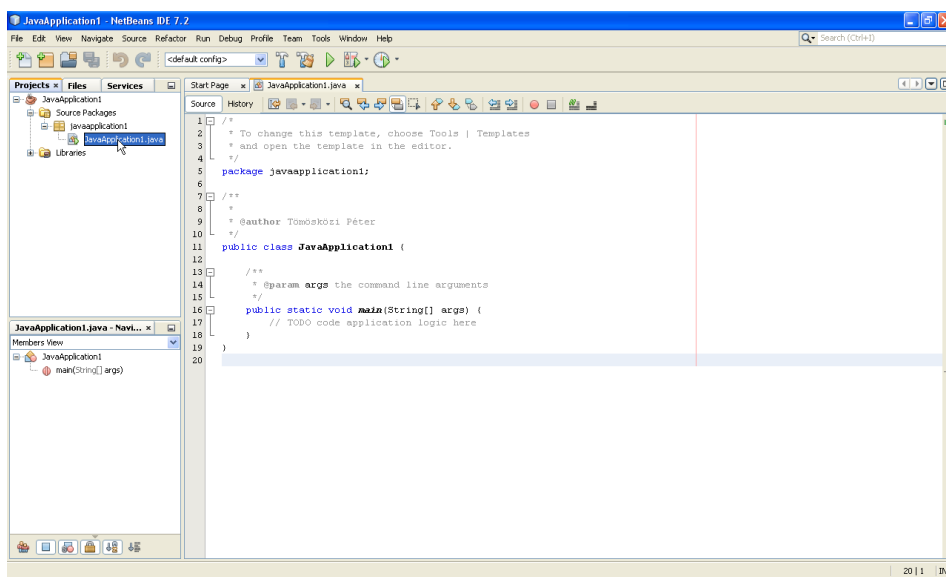
A Files fül alatt a projekt fejlesztése során létrehozott állományokat érhetjük el. Első projektünk jelenleg egy viszonylag szerény forráskód, és a Javához készített igen komoly erőforrások felhasználásával készül. A projekt létrehozásakor nem egyszerűen egy szövegfájl készült, hanem egy könyvtárstruktúra,

amely alapértelmezés szerint a Dokumentumok mappánkban található NetBeansProjects mappában kapott helyet. Ennek elemeit (is) tallózzhatjuk, ha a bal oldali ablakrész tetején található Files fülön kattintottunk.

A Services fül alatt a jelenleg elérhető szolgáltatások (adatbázisok, webes szolgáltatások stb.) tallózzhatók.

A bal oldali ablakrész alsó részén a fent kiválasztott elemről kaphatunk további információkat (Navigator). Valójában nemcsak információk kaphatók itt, hanem a projekt komponensein belüli gyors navigációt teszi lehetővé ez az eszköz. Ha fent kiválasztjuk az első programunk forráskódját (pl. Projects > Source Packages ><projektnév>><projektnév>.java), akkor lent a Navigatorban a projektünkben jelenleg definiált osztályok, illetve azok mezőinek és metódusainak azonosítói lesznek láthatóak. Dupla kattintással a metódusok és a mezők létrehozásának (deklarációjának) sorára tudunk ugrani a forráskódban.

Tekintsük most magát a forráskódot.



9. ábra: Első programunk

Első ránézésre látható, hogy a NetBeans színekkel segíti a kód olvasását. A szürke színű szövegrészek nem a fordítóprogramnak, hanem a programozónak szólnak: ezek ún. kommentek, vagy megjegyzések. Figyeljük meg, hogy háromféle módon lehet kommentet elhelyezni a program szövegében.:

Ha egysoros megjegyzést akarunk elhelyezni, azt a `//` karaktersorozattal jelezhetjük. A `//` utáni rész a sor végéig megjegyzés, ezt a fordítóprogram a fordításkor figyelmen kívül hagyja.

Ha több soros megjegyzést szeretnénk a forráskódba írni, akkor használjuk a `/*` és a `*/` karaktersorozatokat. A `/*` a megjegyzés elejét, a `*/` pedig a megjegyzés végét jelöli. A két karaktersorozat között bármilyen hosszú megjegyzés elhelyezhető, amely tartalmazhat sortöréseket is.

Speciális bevezető karaktersorozat a `/**`, amely az ún. dokumentációs megjegyzéseket vezeti be. A Java-projektekhez gyorsan készíthető dokumentáció a `javadoc` című programmal. (A NetBeansben ez egyébként beépített szolgáltatás.) A dokumentációs megjegyzések a `javadoc`-kal készített dokumentációban is meg fognak jelenni.

A kézzel írott szavak a Java kulcsszavai, más szóval lefoglalt szavai. A program készítése során az általunk definiált osztályokat, objektumokat, változókat, metódusokat stb. ún. azonosítókkal nevezhetjük el. A kulcsszavakat a nyelv fenntartja magának, így kulcsszó nem lehet azonosító.

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

Félkövér betű jelöli az osztályok nevét, félkövér-dőlt pedig a metódusok nevét.

3.2.2 A Java nyelvű programok általános felépítése, kódolási szabályok

Egy Java nyelvű program egymással kölcsönhatásban álló objektumokból áll. Mielőtt egy objektumot használatba vennénk, azt létre kell hoznunk. Ezt a műveletet nevezzük példányosításnak, és a `new` kulcsszó segítségével végezhetjük el (erről természetesen a későbbiekben sokkal bővebben is szó esik majd).

```
1 Osztálynév objektumnév = new Osztálynév();
```

A nyelvvel ismerkedők számára fontos tudnivaló, hogy a Java ún. case-sensitive nyelv. Ez azt jelenti, hogy a fordítóprogram határozottan megkülönbözteti a kis- és nagybetűket. A fent felsorolt kulcsszavak mindegyikét kisbetűvel kell írunk. Ha a karaktereik közül csak egyet is nagybetűvel írunk, akkor azt a fordítóprogram már nem ismeri fel kulcsszóként. Minden általunk létrehozott azonosítót is pontosan ugyanúgy kell írunk a program szövegében, ahogyan azt a létrehozásakor tettük.

 **Az azonosítók neve olyan karaktersorozat, amely betűvel kezdődik és betűvel vagy számjeggyel folytatódhat. A betűk nemcsak latin betűk lehetnek, a Java bármilyen Unicode betű karaktert megenged az azonosítók nevében is, tehát például a magyar ábécé ékezetes magánhangzóit is. A nyelv betűnek tekinti még a \$ és az _ (aláhúzás) jeleket is, de a \$ jelet saját céljaira is felhasználja.**

Az azonosítók elnevezésére a Sun vállalat tett néhány ajánlást, amelyet nem kötelező követnünk, de a Java-közösség tagjai rendszerint követik ezeket.

- Az osztályok nevét kezdjük nagybetűvel, a belső betűk kisbetűk, de ha az osztály neve több szóból áll, a szókezdő belső betűk szintén nagybetűk. (Szóköz nem lehet azonosítóiban.) Az osztályok nevei rendszerint főnevek. Pl. KonvexSokszog, DerekszoguHaromszog, Sikidom.
- Az interfészek (ld. 6–8. fejezet) nevei az osztályok neveihez hasonlóak. Az interfésznevek rendszerint melléknevek. Pl. Futtathato, Kiszamithato.
- A metódusok nevei kisbetűvel kezdődnek, a több szóból álló metódusnevek belső, szókezdő betűi nagybetűk. A metódusnevek gyakran ige-főnév párosok. Pl. kiirAdat, bekerMezo.
- Változók: a metódusokhoz hasonlóan kisbetűvel kezdődnek, a belső szókezdő betűk nagybetűk.
- Konstansok: a konstansok olyan változók, amelyeknek csak egyszer adhatunk értéket (ld. később). A konstansok neveit csupa nagybetűvel írjuk, a több szóból álló konstansnevekben a szóközt _ (aláhúzás) jelöli.

Bár a Java nem korlátozza az azonosítók maximális hosszát, célszerű nem túlságosan hosszú azonosítókat használni. A programozók (a magyarok is) gyakran definiálnak angol nyelvű azonosítókat, ami eleinte idegennek tűnik, de hamar megszokható.

Egy osztály mezői tárolják az osztály adatait, a metódusok pedig leírják az osztályban végezhető műveleteket. Az osztályok között van egy kitüntetett szerepű osztály, amelynek neve ugyanaz a karaktersorozat kell hogy legyen,

mint a fájlnev, amelyben tároljuk. A mi programunk forráskódja a `JavaApplicvation1.java` állományban van tárolva, így a kitüntetett szerepű osztály nevének `JavaApplication1`-nek kell lennie, ahogyan ez a fenti képernyőképen látszik is.

Ebben a kitüntetett szerepű osztályban van definiálva a `main` nevű metódus. Amikor a fordítóprogram lefordítja a forráskódot tárgykódra és azt a JRE elkezd lefuttatni, akkor a program futása itt, a `main` metóduson kezdődik el. Úgy is fogalmazhatunk, hogy a fájl nevével megegyező osztályban definiált `main` metódus a program belépési pontja. Ahhoz, hogy a Java programunk futtatható legyen, szükséges és elégséges feltétel, hogy legyen egy, a fájl nevével azonos nevű osztály definiálva, és abban adjunk meg egy publikus és statikus `main` nevű metódust. Ha egy metódus (vagy mező) publikus, akkor az az osztályon kívülről, azaz más osztályokból is elérhető.

Ahogyan korábban leírtuk, ahhoz, hogy egy objektumot használhassunk, azt először létre kell hoznunk, vagyis példányosítanunk kell egy osztályból. Példányosítani viszont csak metóduson belül lehet. Ha ez alól a szabály alól nem lenne kivétel, akkor nem lehetne futtatható programot írni Javában, hiszen nem tudnánk példányosítani azt az osztályt, amelyik a `main` metódust tartalmazza, hiszen az a program belépési pontja. Ha egy metódus vagy mező statikus, akkor anélkül is lehet használni, hogy példányosítanánk azt az osztályt, amelyben le van írva. Szakszerűbben ezt úgy fogalmazhatjuk meg, hogy a statikus metódusok és mezők az osztályhoz kötődnek, és nem az osztályból létrehozott objektumpéldányhoz.

Az első program megírásához ezekre az információkra még nincs is feltétlenül szükségünk. A 6–8. fejezetben részletesen olvashatunk a Java OOP-s eszközeiről. A lényeg:

- legyen a forráskódban egy olyan publikus osztály definiálva, amelynek neve megegyezik a forráskód fájlnevével
- ebben legyen definiálva egy publikus, statikus, `main` nevű metódus.

Ha ez a két feltétel teljesül, akkor a programunk futtatható lesz.

A forráskód legelején egy csomagdefiníció látható:

```
2 package javaapplication1;
```

A csomagokkal később részletesen foglalkozunk, egyelőre annyit róluk, hogy a csomagokban osztályokat tudunk definiálni. Az egy csomagban definiált osztályok logikailag jobban összetartoznak, mint a különböző csomagokban tárolt osztályok.

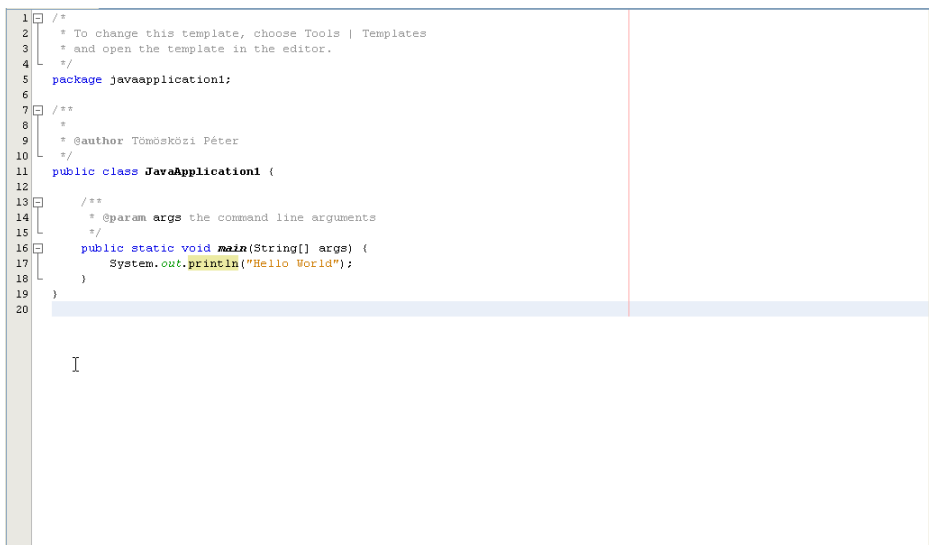
3.2.3 Hello World!

Készítsük el végre az első, futtatható programot, amely csinál is valamit! A `JavaApplication1` forráskódjában a `main` metódus törzsébe írjuk be a következő sort:

```
3 System.out.println("Hello World!");
```

A `main` metódus törzse a név után elhelyezett, kapcsos zárójelek közötti szövegrész. Mint az korábban olvasható volt, a `//` jelek mögött található, a sor végéig tartó karakterek csak megjegyzések a program szövegében, azokat a fordítóprogram figyelmen kívül hagyja. Ezt a sort akár ki is törölhetjük, a program futását nem befolyásolja. A teljes `main` metódus tehát a következőképpen néz ki:

```
4 public static void main(String[] args) {  
5     System.out.println("Hello World!");  
6 }
```

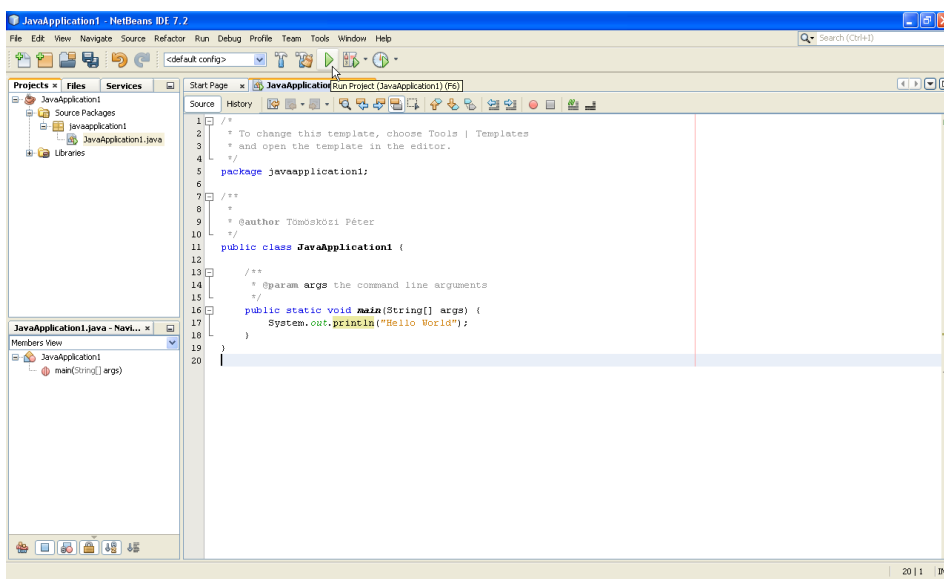


10. ábra: Hello World!

Ahogy az a példában látható, a metódus törzsét behúzással írtuk (a `System` szó beljebb kezdődik, mint a fölötté látható `public`). A fordítóprogram számára ennek sincs jelentősége, a fordítás elején a fordító ugyanis minden, számára szükségtelen szövegrészt (szóközök, sortörések, kommentek) kiszűri a forráskódból, és az így „kitisztított” kódot kezdi el elemezni.

Nagyon fontos, hogy betűre (kis-nagy betűre) pontosan ugyanúgy gépeljünk be mindent, ahogyan az a példában látható! Annak nincs jelentősége, hogy a NetBeans kiszínezi a forráskódot, illetve bizonyos szavakat dőlt, vagy éppen félkövér betűvel szed. Ez csak a kód olvashatóságát javítja, illetve a kódban való navigálást segíti. Annak sincs jelentősége, hogy a `System` szó előtt hány szóközt, vagy éppen tabulátort helyeztünk el.

Fordítsuk le és futtassuk az első programot. A legegyszerűbben ezt úgy tehetjük meg, ha a NetBeans felső sorában, a menü alatti eszköztáron rákattintunk a zöld „Play” gombra.



11. ábra: Fordítás, futtatás

Ha minden jól ment, a forráskód ablakrésze alatt megjelenik a futás eredményét mutató ablakrész, benne a Hello World! szöveggel.



12. ábra: A Hello World program futásának eredménye

Kétségtelen, hogy ez eddig nem túl látványos, de elkészült az első, futtatható Java nyelvű programunk.

Ha nem a fentiekben leírt dolog történt, akkor valószínűleg hibaüzenet jelent meg a képernyőn. Ellenőrizzük le, hogy mindent pontosan úgy csináltunk-e, ahogyan az le volt írva. Ha nem, javítsuk a hibát és próbáljuk újra futtatni a programot!

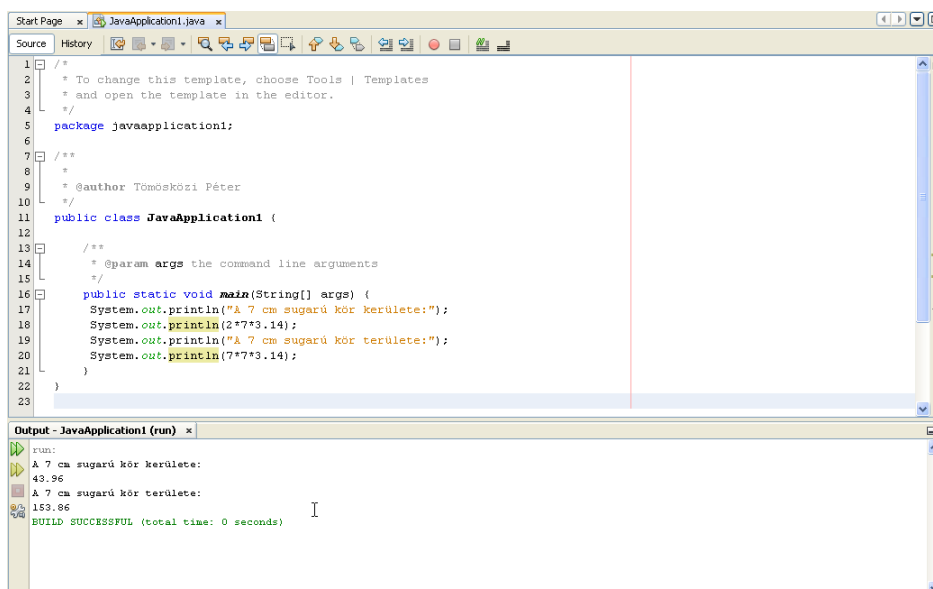
3.2.4 Egyszerű kiíratás

Az első példaprogramból kitalálható, hogy a képernyőre (konzolra) íratáshoz a `System.out.println` használható. Zárójelek között kell megadnunk, hogy mi az, amit szeretnénk a képernyőre íratni. Ez lehet szöveg, de lehet például egy számítás eredménye, vagy akár a kettő együtt is:

```
7 System.out.println("A 7 cm sugarú kör kerü-  
te:");  
8 System.out.println(2*7*3.14);  
9 System.out.println("A 7 cm sugarú kör terü-  
le:");  
10 System.out.println(7*7*3.14);
```

Írjuk be ezt a négy sort a Hello World szöveg kiíratása helyére (vagy az alá), és futtassuk a programot! Vegyük észre, hogy a forráskódba a `System.out.println` utáni zárójelben idézőjelbe tett szöveg betűről betűre pontosan ugyanúgy jelenik meg futáskor a képernyőn, mint ahogyan a forráskódba írtuk. Az idézőjelek közé írott karakterek egy karakterláncot (sztringet) alkotnak, az idézőjel azt jelenti, hogy ezt pontosan így kell kiírni a képernyőre. A fenti példa második és negyedik sorában nincs idézőjel a zárójelen belül. Ilyenkor a Java nem karakterről karakterre ugyanígy ír ki a képernyőre, hanem elvégzi a leírt műveletet ($2*7*3.14$, illetve $7*7*3.14$), és a számítás eredményét írja ki. (A csillag itt a szorzást jelöli. A műveleti jelekről – operátorokról – a következő leckében részletesebben írunk.)

A futás eredménye – ha mindent pontosan írtunk – az alábbi lesz:



13. ábra: Szöveg és számított érték kiírása

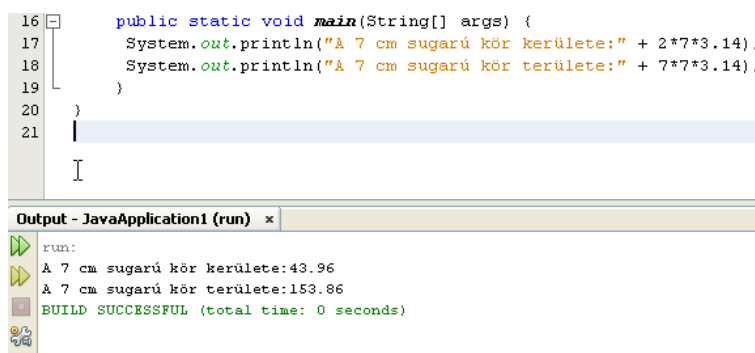
Figyeljük meg, hogy a kiírás négy sort eredményezett. Ennek nem az az oka, hogy a forráskódban is négy sorba írtuk a négy kiírató utasítást. Vegyük észre, hogy minden kiírató utasítás végén egy pontosvessző áll. A pontosvessző az utasításlezáró jel. Egy utasítás ott ér véget, ahol pontosvesszőt írunk. Egy sorba több utasítás is írható (természetesen pontosvesszővel egyenként lezárva), és a későbbiekben látni fogjuk, hogy egy utasítás több sorba is tördelhető.

A fenti példában tehát akár egy sorba is írhattuk volna a négy kiírató utasítást, a futás eredménye ettől függetlenül ugyanaz maradt volna. Ennek az az oka, hogy a `System.out.println` nemcsak kiírja a zárójelben megadott kiírandókat, de a kiírás végén egy soremelést is tesz. Ez olyan, mint amikor egy szöveg gépelésekor megnyomjuk az Entert.

Ha szeretnénk a kerületet és a területet a szöveggel egy sorba kiíratni, két megoldás is lehetséges. Az egyik, hogy csak két kiíró utasítást használunk, és ezekben a szöveget és a számított értéket is kiíratjuk, vagy marad a négy kiíró utasítás, de ebből kettőt úgy módosítunk, hogy ne végezzenek soremelést.

Lássuk az első megoldást:

```
11 System.out.println("A 7 cm sugarú kör kerülete:"  
    + 2*7*3.14);  
12 System.out.println("A 7 cm sugarú kör területe:"  
    + 7*7*3.14);
```



```

16 public static void main(String[] args) {
17     System.out.println("A 7 cm sugarú kör kerülete:" + 2*7*3.14);
18     System.out.println("A 7 cm sugarú kör területe:" + 7*7*3.14);
19 }
20 }
21

```

Output - JavaApplication1 (run) x

```

run:
A 7 cm sugarú kör kerülete:43.96
A 7 cm sugarú kör területe:153.86
BUILD SUCCESSFUL (total time: 0 seconds)

```

14. ábra: Szöveg és számított érték kiírása egyszerre

Az eredetileg négy utasítást összevontuk két utasítássá. A kiírásban az idézőjelek közötti szöveg betűről betűre megjelenik a képernyőn, a záró idézőjelet követő + jel pedig azt jelenti a fordítóprogram számára, hogy a mögötte található műveletsorozat eredményét szöveggé kell alakítania, és az idézőjelek közötti szövegrészhez kell fűznie. A $2*7*3.14$ művelet eredménye 43.96, ami egy szám típusú adat. A fenti utasításban szereplő + jel ezt a számadatot átalakítja a „43.96” szöveggé (az érték marad, csak az adattípus változik), majd ezt a szöveget hozzáfűzve az idézőjelek közötti szöveghez végrehajtható a kiírás.

- ☐ A következő leckében részletesen olvashatunk az aritmetikai operátorokról, vagyis a matematikai műveleteket megvalósító műveleti jelekről. A + jel nemcsak a szövegösszefűzés jele, hanem a matematikai összeadás is. Gondoljuk végig, hogy mennyire különböző műveleteket valósít meg a + jel: egészen másféle gépi kód áll a sztringek összefűzése és a számok összeadása mögött. Ha még azt is tudjuk, hogy mennyire különböző módon van ábrázolva a memóriában egy egész szám és egy valós szám, akkor azt is beláthatjuk, hogy még a számok összeadása sem mindig ugyanazt a gépi kódot jelenti. Máshogyan kell összeadni két egész, két valós, illetve egy egész és egy valós számot. A + jel értelmezése tehát attól függ, hogy milyen környezetben írjuk le. Ennek a jelenségnek a neve az operátor-túlterhelés.

A másik módszer az, ha olyan kiíró utasítást használunk, amely nem végez soremelést. Ilyen utasítás a `System.out.print`.

```

13 System.out.print("A 7 cm sugarú kör kerülete:");
14 System.out.println(2*7*3.14);
15 System.out.print("A 7 cm sugarú kör területe:");
16 System.out.println(7*7*3.14);

```

```

16 public static void main(String[] args) {
17     System.out.print("A 7 cm sugarú kör kerülete:");
18     System.out.println(2*7*3.14);
19     System.out.print("A 7 cm sugarú kör területe:");
20     System.out.println(7*7*3.14);
21 }
22 }
23

```

Output - JavaApplication1 (run) x

```

run:
A 7 cm sugarú kör kerülete:43.96
A 7 cm sugarú kör területe:153.86
BUILD SUCCESSFUL (total time: 0 seconds)

```

15. ábra: Kiíratás soremelés nélkül

Ebben a példában az első és a harmadik utasítás nem végez soremelést, így a számított érték a kiírt szöveg mögé kerül, és csak az érték után következik új sor.

Escape-szekvenciák

A képernyőre íratáskor esetenként speciális karakterek kiírására is szükség lehet. Ezeknek a karaktereknek speciális jelei vannak, ezeket nevezzük escape-szekvenciáknak. Minden escape-szekvencia a \ (backslash) karakterrel kezdődik. A Java az alábbi escape-szekvenciákat ismeri:

1. Escape-szekvenciák

Kód	Leírás
\n	újsor (soremelés)
\t	Tabulátor
\b	backspace (utolsó karakter törlése)
\r	kocsivissza, soremelés nélkül
\f	Lapdobás
\\	\ karakter (backslash)
\'	' karakter, aposztróf
\"	„ karakter, idézőjel
\ooo	karakter oktális számrendszerben 0-377
\uhhhh	Unicode karakter hexadecimálisan (0-0xffff)

3.2.5 Adatok a programban

A programokban előforduló adatok alapvetően két fő csoportba sorolhatók. Konstansnak nevezzük azokat az adatokat, amelyek értéke a program során nem változik. Ilyen adat például a „Hello World!” szöveg, illetve a 7 vagy a 3.14 számok is.

Változóknak nevezzük azokat az adatokat, amelyek értéke a program futása során megváltozhat.

A változók olyan programozási eszközök, amelyeknek négy tulajdonsága van:

- név
- attribútumok
- tárbeli cím
- érték.

A változók neve mindig egy-egy azonosító, a változóra a nevével hivatkozhatunk a program szövegében. A változók nevei nem lehetnek a Java kulcsszavai, és a változóneveknek egyedinek kell lennie, vagyis azonos metóduson belül nem lehet két változó neve ugyanaz.

A Java egy ún. típusos nyelv, ami azt jelenti, hogy minden adatnak (a konstansoknak és a változóknak is) adattípusa van. Az adattípus a változók attribútumai közül az egyik. Az adattípus háromféle dolgot határoz meg:

- a felvehető értékek köre (szám, szöveg, logikai stb.)
- az adott típusú adaton értelmezett műveletek köre (számok esetében pl. aritmetikai műveletek, szövegek esetében összefűzés stb.)
- az adott adat memóriabeli ábrázolása, azaz a tárbeli reprezentáció.

☐ Mint tudjuk, a memóriában minden adat kettes számrendszerbeli számjegyek formájában, azaz bináris módon van tárolva. Az adattípus azt is meghatározza, hogy milyen algoritmussal kell az adott típusú adat értékét bináris jelsorozattá alakítani. Számok esetében szá mábrázolásról, szöveges adatok esetében karakterkódolásról beszélünk. A különböző szá mábrázolási és karakterkódolási algoritmusok bemutatása nem tartozik tankönyvünk témakörébe.

A változók attribútumai közé tartozik még többek között az, hogy az adott változó meddig rendelkezik címkomponenssel, vagyis mi a változó élettartama, illetve látható-e a változó az adott osztályon kívülről vagy sem. Ezekről a kérdésekről a jegyzet későbbi részében lehet olvasni.

A változók értékét a memória tárolja, vagyis minden változóhoz hozzárendel a programunk egy-egy memóriaterületet. Amikor a változó értéket kap, akkor ez az érték bekerül az adott memóriaterületre. Amikor fel akarjuk használni a változó aktuális értékét, akkor a program a változóhoz rendelt memóriaterületről olvassa azt ki. A Javában a címhozzárendelés automatikus, a programozó nem tudja közvetlenül befolyásolni azt, hogy a változóhoz rendelt memóriacím hol legyen a memóriában.

A negyedik tulajdonság a változó értéke. A változó értéke a program futása során megváltozhat, de csak az első értékadástól létezik. Ahhoz, hogy egy változónak értéket adhassunk, azt először létre kell hoznunk, definiálnunk, vagy más szóval deklarálnunk kell. A deklaráció általános alakja:

```
17 típus változónév;
```

A Javában használt elemi adattípusok a következők:

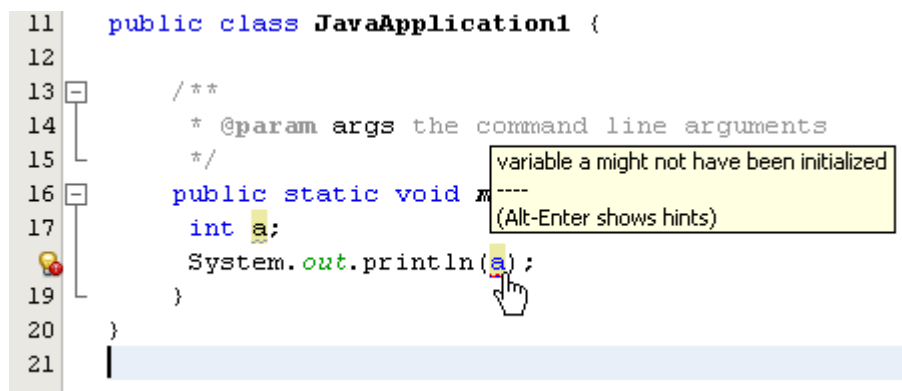
2. A Java elemi adattípusai

Típus	Leírás
byte	8 bites előjeles egész
short	16 bites előjeles egész
int	32 bites előjeles egész
long	64 bites előjeles egész
float	32 bites egyszeres lebegőpontosságú (IEEE 754 szabvány)
double	64 bites kétszeres lebegőpontosságú (IEEE 754 szabvány)
char	16 bites Unicode-karakter
boolean	logikai érték (<code>true</code> / <code>false</code> , <i>azaz igaz hamis</i>)

Példák a változók deklarálására:

```
18 int a;  
19 double ketPi;  
20 char betu;
```

A deklaráció pillanatában megtörténik a címhozzárendelés, viszont ilyenkor a változó értéke még határozatlan. Ha megpróbáljuk felhasználni egy határozatlan értékű változó értékét, hibaüzenetet kapunk.



16. ábra: Határozatlan értékű változó használata

A NetBeans IDE használatának számtalan előnye közül az egyik az, hogy nagyon sok hibát már a kód írásakor felismer. Ha parancssori fordítót használtunk volna, akkor ez a hiba csak fordításkor derült volna ki. Igaz, ez még mindig jobb eset, mintha csak a futás során történt volna meg, mert akkor az a program futásának megállását jelentette volna.

A változóknak az értékadás utasítással adhatunk értéket. Ennek jele az egyenlőségjel (=). Általános alakja:

```
21 változónév = érték;
```

Az érték nemcsak konstans lehet, hanem akár egy másik változó, illetve ún. kifejezés is.

A deklaráció és a (kezdő)értékadás össze is vonható egy utasítással:

```
22 int a = 12;
```

A Java ismeri és megengedi az ún. többszörös értékadást:

```
23 int x = y = z = 10;
```

Az értékadásról a 4.2.3. fejezetben olvashatunk részletesen.

Egy változó lehet karakterlánc (sztring) típusú is, a sztring viszont nem elemi típus, hanem objektum. Ettől függetlenül a deklaráció történhet az elemi adattípusokéhoz hasonló módon:

```
24 String hétNapja = „csütörtök”;
```

Fontos, hogy megértsük a karakter (char) és a sztring (String) típus közötti különbséget. Egy char típusú változó értéke egyetlen (vagyis pontosan egy darab) karakter lehet. (Az escape-szekvenciák egy-egy karaktert jelölnek, tehát

egy char típusú változó értéke lehet egy escape-szekvenciával megadott speciális karakter is.)

```
25 char betű = 'a';  
26 char újsor = '\\n';
```

A String típusú változók mindig karakterláncot tárolnak. Még akkor is, ha csak egyetlen karakter hosszúságúak. Az ilyen sztringek egy karakter hosszúságú karakterláncok. Egy String típusú változó értéke lehet a nulla darab karakterből álló üres sztring is.

```
27 String név = „László”;  
28 String egyBetű = „s”;  
29 String üres = „”;
```

Figyeljük meg, hogy a karakteres típusú változók értékét aposztrófok közé, míg a String típusú változók értékét idézőjelek közé kell írni.

A sztringekről részletesen olvashatunk az 5. leckében.

3.2.6 Nevesített konstansok

A forráskód olvashatóbbá tétele miatt a sokszor előforduló konstansokat nevesített konstansokkal helyettesítjük. Például olyan programot készítünk, amelyben egy iskolai osztály adataival végzünk számításokat. Ilyenkor az osztály létszáma (mint konstans érték) gyakran előfordul a forráskódban. Mi a teendő, ha az eddig 32 fős osztályból két tanuló másik iskolába megy tanulni? A program szövegében meg kell keresnünk a 32 (mint addigi osztálylétszám) összes előfordulását, és le kell cserélnünk 30-ra. Bár a szövegszerkesztőkben van keresés és csere funkció, az nem minden esetben oldaná meg a problémát, mert nincs arra garancia, hogy a 32 érték ne forduljon elő más környezetben is. Mi pedig csak azokat a 32-ket akarjuk 30-ra javítani, amelyek ennek az osztálynak a létszámát jelentik. Például az osztályfőnök életkorát nem, ha ő történetesen éppen 32 éves.

Ilyenkor az a módszer javasolható, hogy hozzunk létre egy ún. nevesített konstanst. A Javában ezek olyan speciális változók, amelyek csak egyszer kaphatnak értéket, ezt később már nem lehet megváltoztatni. Az ilyen speciális változók deklarációja elé ki kell írunk a `final` kulcsszót:

```
30 final int OSZTALYLETSZAM = 32;
```

A program szövegében mindenhová az `OSZTALYLETSZAM` nevet írjuk, ahová egyébként az eredeti 32 került volna. Például:

```
31 double osztalyatlag =  
32 (double) erdemjegyekOsszege / OSZTALYLETSZAM;
```

Így, ha változik az osztály létszáma, akkor az új létszámot *csak egyetlen helyen* megváltoztatnunk a program forráskódjában.

Ahogy a fenti példában is látszik, a nevesített konstansokat szokás csupa nagybetűvel írni. Ez nem kötelező, de általánosan elfogadott jelölésmód. (Lásd még: 3.2.2 fejezet.)

3.3 ÖSSZEFOGLALÁS, KÉRDÉSEK

3.3.1 Összefoglalás

Ebben a leckében elkészítettük első Java nyelvű programunkat. Ehhez a NetBeans 7.2 integrált fejlesztői környezetet használtuk, amely számos előnnyel bír a „hagyományos” programozási módszerhez képest. A hagyományos módszer alatt azt értjük, hogy a forráskódot egy tetszőleges szövegeditorban (pl. Jegyzetömb) készítjük el, majd a fordítást parancssorból végezzük. A létrejövő tárgykódot azután szintén parancssorból futtatjuk le a JVM segítségével.

A NetBeans 7.2 tartalmaz egy olyan szövegszerkesztőt, amely kifejezetten a Java nyelvre van hangolva. Ez azt jelenti, hogy a syntax highlight technika segítségével áttekinthetővé, könnyen olvashatóvá teszi a kódot, továbbá számos hibát már a kód írásakor felismer, és azonnal jelez is. A kód fordítása és a lefordított program menüből valósítható meg. Ezen kívül számos, ebben a leckében be nem mutatott szolgáltatással segíti a programozót, ráadásul teljesen ingyenesen.

A leckében megismertük a Java nyelvű programok általános felépítését, megismerkedtünk a képernyőre írás utasításaival, illetve áttekintettük a Java nyelvű programokban használható elemi adattípusokat, valamint a változók használatának alapjait.

3.3.2 Önellenőrző kérdések

1. Hogyan lehet megjegyzést elhelyezni egy Java nyelvű programban?
2. Mi az a kulcsszó?
3. Mi az az azonosító? Milyen formai követelmények vonatkoznak az azonosítókra Javában?
4. Milyen utasítással lehet a képernyőre íratni?
5. Mire valók az Escape-szekvenciák?
6. Mi az a változó?
7. Milyen elemi adattípusok használhatók Javában?
8. Mi az a nevesített konstans?

4. LECKE: VEZÉRLÉSI SZERKEZETEK ÉS UTASÍTÁSOK. KIVÉTELKEZELÉS JAVÁBAN

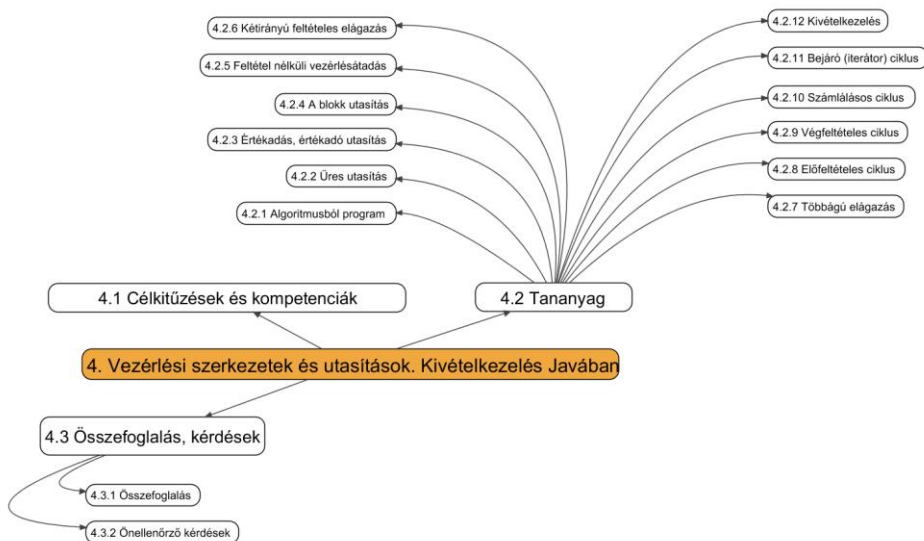
4.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

Ebben a fejezetben a Java nyelv vezérlési szerkezeteit, alapvető utasításait tekintjük át. Utasítások formájában írjuk le a programok elemi lépéseit, míg a vezérlési szerkezetek segítségével az utasítások végrehajtásának sorrendjét lehet befolyásolni.

Az alapvető utasítások, illetve a vezérlési szerkezet eszközeinek bemutatása után áttekintjük a kivételkezelés eszközeit is a Java nyelvben.

A lecke feldolgozásához szükséges kompetenciák: lényegfelismerés, absztrakt gondolkodás, kreativitás, motiválhatóság.

4.2 TANANYAG



4.2.1 Algoritusból program

A programozók a gyakorlati életben felmerülő problémákra keresik a megoldást számítógépes programok készítésével. Ahhoz, hogy egy problémát meg

lehessen oldani, egyértelműen definiálnunk kell a megoldás lépéseit, vagyis azt a módszert, amelynek segítségével elérhetjük a megoldást, a célállapotot. Egy probléma megoldásának módját, módszerét algoritmusnak nevezzük. Az algoritmus meghatározása után a programozók az adott programozási nyelv – esetünkben ez a Java – eszközrendszerét felhasználva készítik el a számítógépes programot.⁵

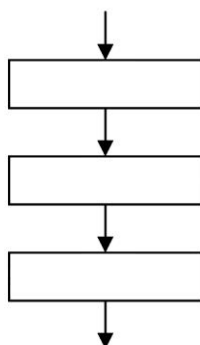
Egy algoritmus esetében a következőket várjuk el:

- Álljon egyértelműen meghatározott lépésekből.
- Minden lépésnek végrehajthatónak kell lennie, azaz az algoritmusnak lépésenként végrehajthatónak kell lennie.
- Legyen véges.
- A lépéseket pontosan kell leírunk, hiszen a számítógép csak az általunk megadott lépések pontos végrehajtására képes.
- Az algoritmus a kezdőállapotból juttat el a célállapotba. Mind a kezdőállapotnak, mind a célállapotnak pontosan definiálnunk kell lennie. Ehhez le kell írunk a probléma bemenő adatait, és azt, hogy ezekből milyen kimenő adatokat szeretnénk előállítani az algoritmus végrehajtásával.

Az 1966-ban két olasz matematikus, Corrado Böhm (1923–) és Giuseppe Jacopini által publikált cikk mutatta be azokat a kutatási eredményeket, amelyeket azóta az algoritmusok strukturált leírásának egyik alaptételeként használunk. A tételt Böhm–Jacopini-tételnek nevezik: bármely algoritmust leírhatjuk három alapstruktúra segítségével: *szekvenciával, iterációval és szelekcióval*.

A Neumann-elvű számítógépek processzorai úgy működnek, hogy a memóriában található utasításokat a memóriabeli sorrendjüknek megfelelő sorrendben hajtják végre.

⁵ Meg kell jegyeznünk, hogy ez a módszer csak az ún. imperatív nyelvek esetében igaz. Vannak olyan magas szintű programozási nyelvek is, amelyek használatakor a programozó nem a megoldás algoritmusát használja. Az ilyen, ún. deklaratív nyelvek esetében a megoldáskeresést maga a programozási nyelv eszközrendszere biztosítja.

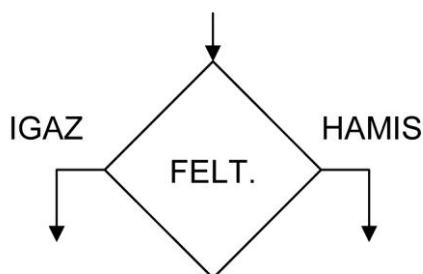


17. ábra: Szekvencia

Ez a szekvenciális vezérlés. Ezt a végrehajtási módot többféleképpen is befolyásolhatjuk:

- Feltétel nélküli vezérlésátadással
- Feltételes vezérlésátadással
- Alprogramhívással (eljárás- és függvényhívással)
- Visszatéréssel az alprogramokból.

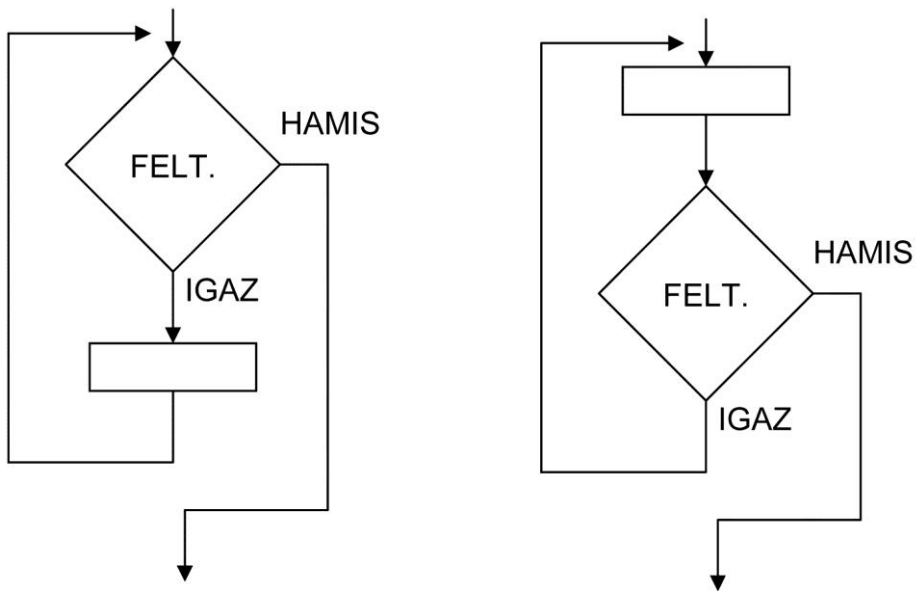
A szelekció olyan struktúra, amelynek során egy feltételtől függően két vagy több végrehajtható lépés közül választunk ki és hajtunk végre egyet. A programozási nyelvek ezeket a vezérlési szerkezeteket feltételes elágazásnak nevezik. A legtöbb programozási nyelvnek – így a Javának is – van eszköze a kétirányú feltételes elágazásra, illetve a többirányú feltételes elágazásra, amelyet esetszétválasztásnak is neveznek.



18. ábra: Kétirányú feltételes elágazás

Az iteráció olyan alapstruktúra, amelynek során egy vagy több lépést többször megismételve hajtunk végre. Az ismétlések száma függhet egy feltételtől,

de lehet előírt lépésszámú is. Az iterációt a programozási nyelvekben ún. ciklus-szervező utasításokkal valósítjuk meg. Az előbbi típusú ciklusokat ún. feltételes ciklusoknak, míg az utóbbi kategóriába tartozó ciklusokat előre megadott lépésszámú ciklusoknak nevezzük.



19. ábra: Előfeltételes és végfeltételes ciklus

Ebben a fejezetben az alábbi utasításokat, illetve vezérlési szerkezeteket tekintjük át:

- Üres utasítás
- Értékadás, értékadó utasítás
- Blokk utasítás
- Feltétel nélküli vezérlésátadás
- Kétirányú feltételes elágazás
- Többirányú, többágú elágazás
- Előfeltételes ciklus
- Végfeltételes ciklus
- Számlálósos ciklus
- Bejáró (iterátor) ciklus
- Kivételkezelés

4.2.2 Üres utasítás

Az üres utasítás az az utasítás, amely nem csinál semmit. Gépi kódban az üres utasításnak – bármennyire is meglepő – komoly szerepe is lehet, mert bár valóban nem csinál semmit, a processzornak ezt az utasítást is végre kell hajtania, ami időbe telik. Gépi kódú programok írásánál az üres utasítást gyakran szokták használni a program futásának bizonyos ideig tartó felfüggesztésére, vagyis várakozás programozására.

Magas szintű programozási nyelvekben az üres utasításnak nincs túl nagy szerepe, de a nyelvek gyakran megengedik. A Java is ismeri az üres utasítást, mint programozási eszközt. Az üres utasítás jele Javában a ; (pontosvessző).

```
33 ;
```

Szekvenciában, vagyis lépésről lépésre végrehajtandó utasítássorozatban szinte sohasem használunk üres utasítást, mert semmilyen előnye sincs. Viszont egészen gyakran találkozunk vele olyan ciklusokban, amelyekben a ciklusmagot a fejben vagy a ciklus végfeltételében adjuk meg.

4.2.3 Értékadás, értékadó utasítás

Az előző fejezetben szó esett a változók szerepéről. Az értékadás az a művelet, amelynek során a jobb oldalon található értéket a program bemásolja a bal oldalon álló változóba. Az értékadó utasítás a Java nyelvben az = (egyenlő) jel.

```
34 változó = érték;
```

Az egyenlőségjel bal oldalán annak a változónak kell állnia, amelynek értéket akarunk adni. Egy változó csak akkor kaphat értéket, ha azt előzetesen már deklaráltuk. A változók deklarálásáról az előző fejezetben már olvashattunk. Az értékadás során az egyenlőségjel bal oldalán álló változónak és a jobb oldalon álló értéknek az értékadás szempontjából kompatibilisnek kell lennie. Ez leggyakrabban úgy valósul meg, hogy a bal oldalon álló változó és a jobb oldalon álló érték típusa azonos. Ha a két típus nem azonos, akkor legalább egymásba konvertálhatónak kell lenniük.

Tekintsük a következő eseteket:

```
35 int a = 5;  
36 // mindkét oldal egész, az értékadás helyes  
37  
38 double b = 12;  
39 // a double bővebb halmazt jelöl,  
40 // mert a jobb oldalon egy egész áll,
```

```
41 // az értékadás helyes
42
43 int c = 13.5;
44 // a bal oldal típusa szűkebb halmazt jelöl,
45 // mert a jobb oldalon valós szám áll,
46 // ez az értékadás helytelen
```

A jobb oldalon álló érték lehet literál (egy konkrét érték, konstans), vagy kifejezés. A kifejezés operandusokból és operátorokból (műveleti jelekből) állnak, és az értéküket a program futás közben számolja ki. Értékadásnál először mindig a jobb oldalon álló kifejezés értéke kerül kiszámításra, majd a kiszámított érték bekerül a bal oldalon álló változóba.

```
47 int a;
48 a = 27;
49 int b = 23; // deklaráció és értékadás egyben
50 int c = (a + b) * 5; // a c értéke 250 lesz
```

A Javában engedélyezett a többszörös értékadás is. Az

```
51 int x, y, z;
52 x = y = z = 71;
```

utasítás eredményeként az *x*, *y* és *z* változó is felveszi a 71-es értéket.

- Az értékadó utasítás maga is egy kifejezés. Az értékadás-kifejezés *eredménye* az az érték, amely értékként bekerül a bal oldalon álló változóba. Az *a = b = 15* többszörös értékadás egyenértékű ezzel a kifejezéssel: *a = (b = 15)*. A jobb oldalon álló értékadás-kifejezés (*b = 15*) értéke 15, ez az értéke kerül bele az a változóba is.

Aritmetikai operátorok

Numerikus típusú (egész, valós) kifejezésekben esetében használhatók a matematikai műveleteket megvalósító aritmetikai operátorok. A Java a következő aritmetikai operátorokat ismeri:

- + összeadás
- kivonás
- * szorzás
- / osztás
- % az osztás maradéka (modulus)

Az összeadás, a kivonás és a szorzás nem igényel különösebb magyarázatot. Az osztás művelete annyiban speciális, hogy egész operandusokon végrehajtva mindig egészet eredményez.

```
53 int a = 17 / 6;
```

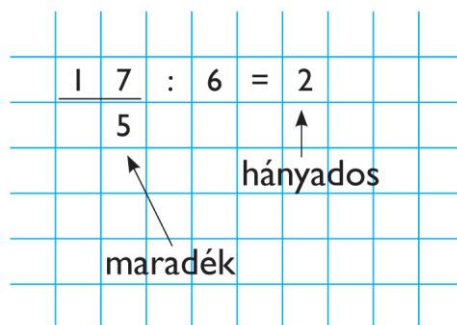
A fenti művelet matematikai értelemben kivezet az egész számok halmazából: $17 / 6 = 2,8333$. Mivel a kapott érték nem egész, így a bal oldalon álló egész típusú változó nem is kaphatná meg értékként. A Java (a C, C++, C# stb. nyelvekhez hasonlóan) az ilyen hibák elkerülése végett az egészek között végrehajtott osztást minden esetben egészosztásként értelmezi. A fenti értékadás tehát szintaktikailag helyes, és eredményeként az *a* változó értéke 2 lesz.

- Az a változóba kerülő 2 érték nem kerekítés eredményeként állt elő. Mivel a 17 és a 6 is egész szám, a Java egészosztást végzett a két szám között, vagyis az eredményként előálló 2 érték azt jelenti, hogy 17-ben a 6 2 egészszer „van meg”. (A maradék 5, bár ezt ez a művelet nem tudja.)

A % művelet az osztás maradékát számítja ki.

```
54 int a = 17 % 6;
```

A fenti értékadás eredményeként az a változó értéke 5 lesz, amely a 17 (egész)osztva 6-tal művelet eredményeként előálló maradék.



20. ábra: Az osztás, ahogyan az iskolában tanultuk

Hogyan kaphatjuk meg két egész szám hányadosát pontosan? Két egész szám hányadosa racionális szám, amely lehet véges tizedes tört, vagy végtelen, ismétlődő tizedes tört. Utóbbi esetben természetesen nem fogunk pontos értéket kapni, hiszen végtelen számok tárolására a véges memória nem alkalmas. Az eredmény adattípusától függően azonban nagy pontosságú eredményt kaphatunk.

Ha két egész szám hányadosaként pontos értéket (általában tizedes törtet) szeretnénk kapni, akkor ehhez többféle eszköz is a rendelkezésünkre áll.

Ha valamelyik operandust nem egészként, hanem valósaként (pl. double vagy float típusúra) deklarálunk, akkor az osztás eredményeként ilyen típusú

értéket fogunk kapni. Ez az ún. bővítő konverzió eredménye, mivel a valós típusok bővebb halmazt írnak le, mint az egész típusok.

```
55 int összeg = 1000;  
56 double db = 16; // a db értéke valójában 16.0  
    lesz  
57 double átlag = összeg / db;
```

Van olyan eset, amikor nem jó megoldás a bővebb típusú deklaráció, mert az osztó változónak is muszáj valamiért egész típusúnak lennie. Ilyenkor van módunk arra, hogy a műveletet úgy hajtsuk végre, hogy a két egész szám hányadosának kiszámításához mi magunk adjuk meg a kívánt eredménytípust. Ezt a módszert nevezik típuskényszerítésnek is. Ehhez valamelyik (vagy mindkét) operandus elé zárójelben ki kell írunk a kívánt típus nevét.

```
58 int összeg = 1000;  
59 int db = 16;  
60 double átlag = (double) összeg / db;
```

A kifejezés kiértékelése során az összeg változó értéke változatlan marad, ugyanakkor a kiértékelés idejére (és csak arra) az értéke nem egész, hanem valós típusú lesz. Valószínűleg egészzel osztva már valós számot fogunk kapni.

Aritmetikai operátorok tömörebb formában

Gyakran előfordul, hogy a bal oldalon álló változó a jobb oldali kifejezésben is előfordul, vagyis egy változó új értékének kiszámításához felhasználjuk a jelenlegi értékét is.

Az

```
61 x = x + 5;
```

értékkadás megnöveli az x változó aktuális értékét 5-tel. A fenti értékkadás szintaktikailag helyes, nem okoz problémát, hogy az x változó a bal és a jobb oldalon is szerepel. Mivel mindig először a jobb oldali kifejezés kiértékelése történik meg, az x aktuális értékével ki lehet számolni az x + 5 kifejezés értékét. (Természetesen csak akkor, ha az x változó az adott környezetben használható változó, azaz deklarálva van, és van aktuális értéke. Ehhez kapcsolódik a változók hatóköre és élettartama témakör is, amelyről a 7.2. leckékben részletesen olvashatunk.)

A fentihez hasonló értékkadások tömörebb formában is írhatók a Java nyelvű programokban. Az x = x + 5 értékkadást átírhatjuk így is:

```
62 x += 5;
```

Bővebb írásmód	Tömörebb írásmód
$x = x + 5$	$x += 5$
$x = x - 5$	$x -= 5$
$x = x * 5$	$x *= 5$
$x = x / 5$	$x /= 5$
$x = x \% 5$	$x \% = 5$

3. aritmetikai operátorok átírása

Inkrementálás és dekrementálás

A programokban a fenti példák közül az a leggyakoribb eset, amikor egy változó aktuális értékét 1-gyel kell növelni vagy csökkenteni. Az 1-gyel való növelést inkrementálásnak, az 1-gyel való csökkentést dekrementálásnak nevezzük. Ezekre a műveletekre – gyakori használatuk miatt – a Java saját operátort is kínál, a ++ (plusz-plusz) és a -- (mínusz-mínusz) operátorokat. A két operátor önálló utasításként, de kifejezések belsejében is használható, és hatékonyságukat tovább növeli, hogy prefix és posztfix formában is írhatjuk őket. A prefix alak esetében az operátor a változó neve elé, míg posztfix alak esetében a változó neve mögé kerül. A működésüket tekintve azonban sokkal többről van szó egyszerű formai kérdésnél.

```
63 int a = 5;
64 a++; // az a értéke 6 lett
```

```
65 int b = 10;
66 ++b; // a b értéke 11 lett
```

```
67 int c = 23;
68 c--; // c értéke 22 lett
```

```
69 int d = 35;
70 --d; // d értéke 34 lett
```

A fenti példákban azt láthattuk, hogyan működnek az inkrementáló és dekrementáló operátorok önálló utasításként. Érdekesebb viszont, ha az inkrementálás vagy dekrementálás egy kifejezés belsejében történik:

```
71 int a = 10;
72 int b = ++a * 2; // a = 11, b = 22 lesz
```

```
73 int c = 10;
74 int d = c++ * 2; // c = 11, d = 20 lesz
```

A fenti két értékadás-sorozatban mindkét esetben az eredeti *a* és *c* változók értéke eggyel növekszik az inkrementáló utasítás hatására. Az érdekesség a *b* és *d* változók esetében figyelhető meg.

Figyeljük meg, hogy az `int b = ++a * 2;` kifejezésben a `++` operátor prefix alakban, vagyis az *a* változó neve előtt szerepel. Ennek az a jelentősége, hogy ilyenkor a jobb oldalon álló teljes kifejezést úgy értékeli ki a Java, hogy először megnöveli a változó értékét eggyel, majd a már növelt értéket felhasználva számítja ki a kifejezés értékét.

A második esetben, ahol a `++` operátor posztfix alakban (a változó neve mögött) szerepel, a kifejezés értékének kiszámításához a változó aktuális értékét használja fel a Java, majd a kifejezés értékének kiszámítása után megnöveli a változó értékét.

Habár első ránézésre ez a lehetőség egy feleslegesen bonyolult dolognak tűnhet, a későbbiekben nagyon sokszor ki tudjuk majd használni az előnyeit.

Természetesen a prefix és posztfix alakok közötti különbség a dekrementálás esetén is hasonlóan működik. Inkrementáló és dekrementáló utasítás minden olyan helyre írható a programban, ahová egy változó nevét írhatjuk.

Sztringoperátorok

Karakterláncokkal kapcsolatosan egyetlen operátort említhetünk ez a `+` (plusz). Sztringek esetében a `+` természetesen nem összeadást jelent, hanem a karakterláncok összefűzését.

```
75 String név = vezetéknév + " " + keresztnév
```

Összehasonlító operátorok

Elágazások és ciklusok feltételeinek megadásakor nagyon gyakran van szükségünk arra, hogy egy változó, vagy egy kifejezés értékét összehasonlítsuk egy másik változó vagy kifejezés értékével. Egy összehasonlítás eredménye egy logikai érték, ami lehet igaz (`true`) vagy hamis (`false`). Mivel a Java nyelvben létezik önálló logikai adattípus, ezért összehasonlítási művelet eredményét akár értékül is adhatjuk egy logikai változónak.

Ez alapján a

```
76 boolean nagyobb;  
77 if (a > b)  
78     nagyobb = true;  
79 else  
80     nagyobb = false;
```

kód írható sokkal tömörebben, egyetlen értékadásként:

```
81 boolean nagyobb = a > b;
```

A Javában az alábbi összehasonlító operátorok használhatók:

- > nagyobb
- < kisebb
- >= nagyobb vagy egyenlő
- <= kisebb vagy egyenlő
- = egyenlő
- != nem egyenlő

Figyeljük meg, hogy az összehasonlítás operátora a két egyenlőségjel. Az egymagában álló egyenlőségjel az értékadás művelete.

Feltételes kifejezés

A Javában lehetőségünk van arra, hogy egy változó egy feltételtől függően kapjon értéket. Ehhez a feltételes kifejezés használható, amely használatával kiválthatók azok az értékadáspárok, amelyeket egyébként egy feltételes elágazás igaz és hamis ágába írnánk.

Például szeretnénk a *min* változó értékéül adni az *a* és *b* változó közül a kisebbiknek az értékét. Feltételes elágazással ez a következőképpen történik:

```
82 if (a < b)
83   min = a;
84 else
85   min = b;
```

A feltételes kifejezéssel ezt egyetlen sorba tudjuk írni:

```
86 min = (a < b) ? a : b
```

Látható, hogy a feltételes kifejezés háromoperandusú:

```
87 (feltétel) ? op1 : op2
```

Ha a feltétel igaz, akkor az értékadás bal oldalán álló változó az op1 értékét kapja értékül, ha hamis, akkor az op2-ét. Az op1 és op2 tetszőleges kifejezés lehet, de fontos, hogy az értékadás bal oldalán álló változó és az op1, op2 típusai kompatibilisek legyenek.

Logikai operátorok

Szintén ciklusok, elágazások feltételében, vagy akár logikai változók értékadásában szerepelhetnek logikai operátorok. Ezek a klasszikus matematikai

logika műveleteinek megfeleltetései a Java nyelvben. Három operátor használható logikai kifejezésekben: ! (NEM), && (ÉS), || (VAGY). A ! (NEM művelet, tagadás) operátor egyoperandusú. Hatására a logikai kifejezés értéke negálódik (igazból hamis, hamisból igaz érték lesz). Az && (ÉS művelet, konjunkció) operátor kétoperandusú. Az && operátorral összekapcsolt kifejezés értéke akkor igaz, ha mindkét operandus értéke igaz volt. A || (VAGY művelet, diszjunkció) két operandus között akkor szolgáltat igaz értéket, ha legalább az egyik operandus igaz volt.

```
88 int a = 5;
89 int b = 6;
90 boolean c = ((a > b) && (b > 2)); // c hamis
    lesz
91 boolean d = ((a > b) || (b > 2)); // d igaz lesz
```

4.2.4 A blokk utasítás

A későbbiekben látni fogjuk, hogy a kétirányú feltételes elágazások (if) igaz és hamis ágába, illetve a ciklusok belsejébe (ciklusmag) a Java nyelvben csak egy-egy utasítás írható. Ha ezekre a helyekre a program szövegében több utasítást szeretnénk írni, akkor ezeket az utasításokat blokkba kell foglalni. A blokk utasítás jele a kapcsos zárójelpár: { ... }.

```
92 if (feltétel)
93     egy utasítás;
94 else
95     egy utasítás;
96 if (feltétel)
97 {
98     több utasítás;
99 }
100 else
101 {
102     több utasítás;
103 }
```

A blokk utasításon belül bárhová írható deklaráció és végrehajtható utasítás is, akár összekeverve. A blokkon belüli deklarációra annyi megkötés van, hogy minden változót az előtt kell deklarálni, hogy értéket adnánk neki, illetve a blokkon belül deklarált változó hatóköre a blokk végéig tart. Ez azt jelenti, hogy a blokkot lezáró kapcsos zárójel után a blokkon belül deklarált változók számára lefoglalt memóriaterület felszabadul, és a változó innentől fogva már nem használható.

Blokk utasítás mindenhová írható, ahová végrehajtható utasítás vagy deklaráció írható.

```
104 int a = 10;
105 {
106     int b = 5;
107     int c = a * b; // c értéke 50 lesz
108 }
109 int d = a + b; // hiba: a b változó nem létezik
```

4.2.5 Feltétel nélküli vezérlésátadás

A legelső magas szintű programozási nyelvekben (FORTRAN, COBOL) már létezett a GOTO utasítás a feltétel nélküli vezérlésátadásra. A GOTO utasítás után egy címkét kellett megadni, hatására a program végrehajtása a címkével jelölt ponttól folytatódott. A GOTO utasítás a mikroszámítógépek BASIC nyelvének is oszlopos tagja volt, ezekben a BASIC-ekben szinte nem lehetett programot írni GOTO nélkül.

A GOTO utasítás használata széles körben vitatott a mai napig. Az általános nézet mára az, hogy a használata egyáltalán nem javasolt, mivel előnye szinte nincs, hátránya viszont rengeteg. Ezért a legtöbb mai programozási nyelven nem indokolt a használata, és GOTO nélkül is kiváló programokat lehet írni.

A Java nyelv ennél is tovább ment. A `goto` ugyan kulcsszó a Java nyelvnek, tehát nem deklarálhatunk `goto` nevű változót, nem hozhatunk létre `goto` nevű objektumot, viszont az utasítást nem implementálták. Ez azt jelenti, hogy létezik `goto` kulcsszó a Javában, de nem csinál semmit.

A feltétel nélküli vezérlésátadásnak ugyanakkor létezik két olyan utasítása, amelyiket a Java nyelv is ismeri, sőt, gyakran használjuk is őket. Ezek a `break` és a `continue` utasítások.

Mindkét utasítás használható önmagában, vagy címkével. A címke a Javában egy olyan azonosító, amely segítségével egy utasítást lehet azonosítani. A címke után pontosvesszőt kell írunk.

```
110 címke: utasítás;
```

A `break` utasítás (címke nélkül) a legbelső `switch`, `while`, `do` vagy `for` (lásd később) utasításból lép ki úgy, hogy a végrehajtás az adott `switch`, `while`, `do` vagy `for` utasítás után folytatódik. A `break` hatására tehát megszakad a `switch`, illetve `while`, `do` vagy `for` utasítás végrehajtása, és a következő utasításra lép a vezérlés.

A `break címké;` változat esetén a végrehajtás az adott címkéjű utasítás után folytatódik.

A `continue` utasítás csak `while`, `do` vagy `for` utasításon, vagyis ciklusutasításon belül (a ciklus magjában) szerepelhet.

A `continue` utasítás (címké nélkül) az adott ciklus magjának végére lép, és a következő iterációs lépést kezdi végrehajtani – ha a ciklusban maradás feltétele továbbra is igaz.

A `continue címké;` változat az adott címkéjű `while`, `do` vagy `for` ciklus következő iterációs lépését kezdi végrehajtani.

4.2.6 Kétirányú feltételes elágazás

A kétirányú feltételes elágazás utasítás a *ha ... akkor ... különben* műveletnek feleltethető meg. Az utasítás általános alakja Javában a következő:

```
111 if (feltétel)
112     utasítás;
113 else
114     utasítás;
```

Vegyük észre, hogy a fenti kódban nem szerepel az *akkor* szó megfelelője (*then*). Ennek az az oka, hogy a *then* kulcsszó csak az *if (feltétel)* után állhatna, ott viszont minden esetben kötelezően állnia is kellene. Vagyis mindahányszor leírjuk az *if (feltétel)* részt, azt kötelezően egy *then* kulcsszónak kellene követnie. Ha valamit mindig ki kellene írunk, és csak erre az egy dologra lehetne használni, akkor ezzel teljesen egyenértékű az a megoldás, ha sosem írjuk ki. Ez alapján a Javában (illetve a C-szerű nyelvekben: C, C++, C# stb.) nincs *then* kulcsszó definiálva, szemben más nyelvekkel (Pascal, BASIC stb.), ahol a használata kötelező.

Az *if* kulcsszó után kerek zárójelek között egy logikai értékű kifejezést kell megadnunk, vagyis egy olyan kifejezést, amelynek az értéke `true` (igaz) vagy `false` (hamis). A kerek zárójelek a szintaxis részét képezik, tehát nem hagyhatók el még a legtriviálisabb esetben sem, pl.

```
115 if (true)
```

Az *if* (feltétel) rész után meg kell adnunk egy utasítást, amely akkor kerül végrehajtásra, ha a megadott feltétel igaz volt. Ide egyetlen utasítás írható. Ha az elágazás igaz ágába több utasítást is szeretnénk írni, vagyis a feltétel igaz volta esetén több utasítást is végre kell hajtani, akkor alkalmazzuk a blokk utasítást: az igaz ág utasításait írjuk kapcsos zárójelek közé.

```
116 if (feltétel)
117 {
118     utasítás1;
119     utasítás2;
120     utasítás3;
121     ...
122 }
```

Az elágazás igaz ága után következhet egy *else* utasítás, amely nem kötelező. Ezzel tudjuk megadni az elágazás különben ágát. Ha nem adunk meg *else* ágat, és az elágazás feltétele hamis volt, akkor nem történik semmi, az elágazás egy üres utasítás lesz. (Ebben az esetben szoktak egyirányú elágazásról is beszélni, hiszen csak az egyik ágat definiáltuk, a másik ág üres.)

Ha megadunk *else* ágat, akkor abban is csak egy utasítást adhatunk meg. Ha többet szeretnénk megadni, alkalmazzuk a blokk utasítást.

4.2.7 Többágú elágazás

Tekintsük az alábbi példát! Egy változóban 1 és 5 közötti egész számként tárolunk egy érdemjegyet, amelyet szeretnénk betűvel kiírni a képernyőre. Kétirányú elágazással ez a következő formában történhetne:

```
123 if (jegy == 1)
124     System.out.println(„elégtelen”);
125 else
126     if (jegy == 2)
127         System.out.println(„elégséges”);
128     else
129         if (jegy == 3)
130             System.out.println(„közepes”);
131         else
132             if (jegy == 4)
133                 System.out.println(„jó”);
134             else
135                 System.out.println(„jeles”);
```

Ez hosszú és bonyolult megoldás. Az ilyen szerkezetek egyszerűbb leírásához használható a többirányú vagy többágú elágazás, amelyet a *switch* utasítással valósíthatunk meg. Általános alakja a következő:

```
136 switch (kifejezés)
137 {
138     case érték1: utasítások1; break;
139     case érték2: utasítások2; break;
140     case érték3: utasítások3; break;
```

```

141 ...
142 default: utasításokn;
143 }

```

A `switch` kulcsszó után kötelezően kerek zárójelek közé írott *kifejezés* csak egész típusúra konvertálható típusú lehet, illetve a Java SE 7 verzió megjelenése óta *String* típusú is lehet. Ezt kötelezően egy kapcsos zárójelek közé írott (blokk) szerkezet követi. Ebben az egyes ágakat (a lehetséges eseteket) a `case` kulcsszavak mögé írott konstansok jelölik. Egy ágban több utasítás is megadható, itt nem szükséges blokkokba szervezni a több utasításból álló utasítássorozatokat. Nem kötelező, de célszerű az egyes ágakat `break;` utasítással zárni, ugyanis ha ezt nem tesszük meg, akkor a végrehajtás az adott ágról továbbcsorog a következőkre.

```

144 switch (kifejezés)
145 {
146     case érték1:
147     case érték2: utasítás2; break;
148     case érték3: utasítás3; break;
149 }

```

Ebben a példában akár *érték1*, akár *érték2* a *kifejezés* értéke, mindenképpen az *utasítás2* fog végrehajtódni.

Ha egyetlen *case*-szel jelölt ág sem kerül végrehajtásra, mert egyetlen, a *case*-ek után írott értékkel sem egyenlő a *kifejezés*, de adtunk meg *default* ágot, akkor az az ág fog végrehajtódni. Ha nem adtunk meg *default* ágot, akkor ilyen esetben nem történik semmi, az elágazás üres utasítás lesz. (Vagyis ez nem hiba.)

Az ágak címkéinek (a *case* után álló értékeknek) egyedinek kell lenniük. Ha több ág is tartalmazza ugyanazt az értéket, akkor hibaüzenetet kapunk.

```

16 public static void main(String[] args) {
17     int jegy = 5;
18     switch (jegy)
19     {
20         case 1: System.out.println("elégtesen");
21         case 1: System.out.println("elégtesen"); break;
22         case 3: System.out.println("közepes"); break;
23     }
24 }
25

```

21. ábra: Az 1-es érték két ágban is szerepel

A fent leírt érdemjegy-átíró kód többágú elágazással így írható fel:

```
150 switch (jegy)
151 {
152     case 1: System.out.println(„elégtelen”);
153         break;
154     case 2: System.out.println(„elégséges”);
155         break;
156     case 3: System.out.println(„közepes”); break;
157     case 4: System.out.println(„jó”); break;
158     case 5: System.out.println(„jeles”); break;
159 }
```

(Ebben az átírásban feltételeztük, hogy a *jegy* változó értéke biztosan 1 és 5 közötti egész.)

4.2.8 Előfeltételes ciklus

Az iteráció vezérlési szerkezet gyakran használt változata az előfeltételes (előltesztelő) ciklus. Javában ennek az általános alakja a következő:

```
158 while (feltétel)
159     utasítás;
```

A feltételnek a kétirányú feltételes elágazáshoz hasonlóan logikai értékkel (*true* vagy *false*) kell rendelkeznie, és kötelezően kerek zárójelben kell állnia. A ciklus magja egy utasításból állhat. Ha több utasítást szeretnénk a ciklus magjaként végrehajtani, akkor alkalmazzuk – szokás szerint – a kapcsos zárójeleket: készítsünk blokkot a ciklus magjából!

A *while* kulcsszóhoz érve a Java kiértékeli a megadott feltételt. Ha az igaz, akkor végrehajtja a ciklus magját, majd visszatérve a fejhez, újból kiértékeli a feltételt. Ha az továbbra is igaz, akkor a ciklusmag ismét lefut, és ez a tesztelés-végrehajtás sorozat mindaddig ismétlődik, amíg a feltétel hamissá nem válik.

Szélsőséges esetben a feltétel sohasem válik hamissá, ekkor beszélünk végtelen ciklusról.

Ha az első kiértékeléskor a feltétel hamis, akkor a ciklus mag egyszer sem fut le. Ekkor üres ciklusról beszélhetünk. Az előfeltételes ciklus lehet üres ciklus, és válhat végtelenné és, ha nem gondoskodunk a cikluson belül arról, hogy a feltétel idővel hamissá váljon.

A *while* ciklusból ki tudunk lépni a *break*; vagy *break* címke; utasítással, illetve a következő iterációs lépésre ugorhatunk a *continue*; vagy *continue* címke; utasítással. (Lásd fentebb.)

4.2.9 Végfeltételes ciklus

A végfeltételes ciklusban – ahogyan a neve is mutatja – a ciklusban maradás feltétele a ciklus végén van elhelyezve. (A végfeltételes ciklust nevezik hátultesztelő ciklusnak is.)

A Javában az általános alakja a következő:

```
160 do  
161   utasítás  
162 while (feltétel);
```

Ahogyan előfeltételes ciklusnál, a *feltétel* itt is egy logikai értékű kifejezés. A kerek zárójelek itt is a szintaxis részét képezik, tehát nem hagyhatók el. Ha a ciklus magjában egynél több utasítást akarunk végrehajtani, akkor használjunk utasításblokkot.

A hátultesztelő ciklus működése: a ciklusba lépéskor a program végrehajtja az utasítást (a ciklus magját), majd megvizsgálja a ciklus feltételét. Ha az igaz, akkor a vezérlés ismét a fejre kerül, végrehajtásra kerül a ciklus magja, majd ismét sor kerül a feltétel vizsgálatára. Ez a folyamat mindaddig ismétlődik, amíg a feltétel hamissá nem válik.

A hátultesztelő ciklus is válhat végtelenné, ha a ciklusban nem gondoskodunk arról, hogy a ciklusfeltétel idővel hamissá váljon. Nagyon fontos különbség viszont az előfeltételes ciklussal szemben az, hogy a hátultesztelő ciklus sohasem lehet üres, hiszen a lefutása mindig azzal kezdődik, hogy a program végrehajtja a ciklusmagot. Ha az első teszteléskor a feltétel hamis is, a ciklusmag egyszer már biztosan lefutott.

A `do ... while` ciklus esetében a `while` ciklushoz hasonlóan használhatók a `break` és a `continue` utasítások.

4.2.10 Számlálós ciklus

Gyakori eset, hogy egy ciklus majdani lefutásának lépésszámát előre ismerjük. Például szeretnénk egy tömb összes elemén végrehajtani ugyanazokat a műveleteket. Mivel a tömb statikus adatszerkezet, ráadásul a Java-tömbök ismerik is a saját elemszámukat⁶, ha a feladatot ciklussal oldjuk meg, már a ciklus felírásakor pontosan tudhatjuk, hogy a ciklus magjának hányszor kell lefutnia.

⁶ A tömbök a különböző típusú tömbosztályokból példányosított objektumok. Minden tömb rendelkezik információval a saját elemszámáról, amelyet a `tömbnév.length` módon lehet elérni.

Tekintsük a következő példát: ismerjük egy sokfős társaság tagjainak magasságát (cm), szeretnénk megtudni, hogy mennyi az átlagos magasságuk. Tegyük fel, hogy a tíz személy magasságát egy tömbben tároljuk:

```
163 int magasság[] = new int[] {172, 180, ...,
    192};
164 int i = 0;
165 int összeg = 0;
166 while (i < magasság.length)
167 {
168     összeg = összeg + magasság[i];
169     i++;
170 }
171 double átlag = (double) összeg / magasság.length;
```

Az ehhez hasonló feladatok megoldására nagyon kényelmes eszköz az ún. számlálós ciklus, vagy ahogyan a Java nevezi, a `for` utasítás. A fenti programrészletet így írhatjuk át a `for` segítségével:

```
172 int magasság[] = new int[] {172, 180, ...,
    192};
173 int i;
174 int összeg = 0;
175 for (i=0; i < magasság.length; i++)
176 {
177     összeg = összeg + magasság[i];
178 }
179 double átlag = (double) összeg / magasság.length;
```

Egy `while` ciklus mindig átírható `for` ciklussá az alábbi módon:

```
180 ciklusváltozó inicializálása;
181 while (feltétel)
182 {
183     utasítás(ok);
184     inkrementálás;
185 }
```

```
186 for (inicializálás; feltétel; inkrementálás)
187 {
188     utasítás(ok);
189 }
```

A `for` ciklussal lényegében az előtesztelő (`while`) ciklusokat írhatjuk át tömörebbé, áttekinthetőbbé. A Java, illetve egyéb C-szerű nyelveken írott kódokra egyébként is jellemző a tömörség. A rövidebb kód gyakran áttekinthetőbb is, de – főleg a nyelvvel most ismerkedők számára – a túlságosan tömör kód nehezítheti a megértést.

A fenti kód még tömörebb módon felírható az alábbiak szerint:

```
190 int magasság[] = new int[] {172, 180, ..., 192};
191 int i, összeg = 0;
192 for (i = 0; i < magasság.length; összeg += magasság[i++])
193     ;
194 double átlag = (double) összeg / magasság.length;
```

Habár a példákban a `for` ciklusban használt ciklusválogató mindig egész volt, a `for` ciklusban valós értékekkel is végezhetünk számításokat.

- ☐ Gondoljuk át a következőt! Lehetett volna-e a fenti példában az *i* változót a *for* belsejében deklarálni? Igen, lehetett volna. Azonban ha ezt tettük volna, tehát a ciklust `for (int i = 0; ...)` módon írtuk volna fel, akkor az *i* változó a ciklus lokális változója lett volna. Ez azt jelenti, hogy csak a cikluson belül rendelkezett volna címkomponenssel, vagyis a ciklusból való kilépés után az *i* értéke elérhetetlen lett volna. Habár az *i* értékére a ciklus lefutása után a fenti példában már nem is lett volna szükségünk, vannak olyan esetek, amikor a cikluson belül deklarált változó miatt kapunk fordításkori hibát.

4.2.11 Bejáró (iterátor) ciklus

A tömbök minden elemére végrehajtandó, cikluson belüli műveletek megadására még egy eszközt kínál a Java. Ezt nevezzük bejáró vagy iterátor ciklusnak. Tekintsük az alábbi példát:

```
195 int a[] = new int[] {10, 12, 15, 20, 23, 32};
196 int összeg = 0;
197 for (int x : a)
198     összeg += x;
```

Ez a programrészlet létrehoz egy *a* nevű tömböt a fenti elemekkel, majd kiszámítja az elemek összegét az *összeg* nevű változóba. A

```
199 for (változó : tömb)
200     utasítás;
```

szerkezet egyenként értékül adja a *tömb* összes elemét a *változónak*, és minden elemre lefut a ciklus magja.

- ☐ Az iterátor nemcsak tömbökre, de ún. *kollektciókra* is használható.

4.2.12 Kivételkezelés

Robusztus programozásnak nevezzük azt a módszert, amikor a programozó a program készítése során megpróbál minden lehetséges hibalehetőségre felkészülni. A hibalehetőségek nagy része a felhasználók hibájának eredménye, sok esetben egy program helyes működését tekintve a nem elég körültekintő felhasználók a gyenge láncszemek. Előfordulnak olyan hibák is, amelyek nem a felhasználók ügyetlenségéből erednek, pl. egy használni kívánt erőforrás éppen nem elérhető, mert azt egy másik folyamat használja.

A robusztus programozás lényege az, hogy a programozó igyekszik minden lehetséges hibát előre felismerni, és a program készítése során törekszik arra, hogy a lehetséges hibák minél nagyobb részét kezelni tudja a program a futás során. Az ilyen programok „bolondbiztosak” lesznek, vagyis a felhasználó feyelmezetlensége vagy ügyetlensége nem, vagy csak nagyon ritkán vezet a program futásának megszakadásához.

A robusztus programozási módszer egyik hibája, hogy a lehetséges hibalehetőségek előzetes felismerése és orvoslása nagyon bonyolulttá is teheti a programot. A programozótól az ilyen programok készítése sokkal több energiát kíván, és ez elvonja a figyelmet a valódi feladat megoldásától.

A robusztus programozással szembeni módszer az, amikor a hibákat nem már a bekövetkezésük előtt akarjuk orvosolni azzal, hogy lehetőség szerint meg sem engedjük azok bekövetkezését, hanem a helyes futás során várt eredményekre koncentrálunk, és a hibákat csak akkor próbáljuk kezelni, amikor azok valóban be is következnek. Az ilyen programokat a programozók úgy írják meg, hogy elsősorban a valóban megoldandó problémára koncentrálnak, és feltételezik azt, hogy a program futása során nem fog semmilyen hiba bekövetkezni. Ez a feltételezés azért kaphat létjogosultságot, mert a programok rendszerint valóban hibák, váratlan események és egyéb problémák nélkül szoktak lefutni az esetek döntő többségében.

A futás során fellépő váratlan eseményeket nevezzük kivételnek. A kivételkezelés az a módszer, amelynek során a fellépő hibákat csak akkor kezeljük, amikor azok valóban be is következnek, nem pedig előzetesen próbálunk tenni ellenük. Amikor a futás során bekövetkezik egy kivétel, akkor a vezérlést egy kivételkezelő alprogram kapja meg, amelyben a programozó megoldást adhat a fellépő problémára. Ennek a programozási technikának az az előnye, hogy ilyenkor a programozó a valóban megoldandó problémára koncentrálhat, és nem kell időt fecseirelnie azoknak a hibáknak az előzetes elkerülésére, amelyeknek nagy többsége valószínűleg soha nem is fog bekövetkezni.

A Java programokban a kivételektől védendő részt a `try` kulcsszóval bevezetett blokkban kell elhelyezni. Ezt követően kell elhelyezni a kivételeket kezelő `catch` blokkot vagy blokkokat.

```
201 try
202 {
203   védett_programrész
204 }
205
206 catch (kivételtípus1)
207 {
208   kivétel1 kezelése
209 }
210
211 catch (kivételtípus2)
212 {
213   kivétel2 kezelése
214 }
215
216 ...
```

Amikor a védett részben kivétel keletkezik, azt a védett rész eldobja, és valamelyik `catch`-szerkezet kapja el.

A Javában a kivételek objektumok, és minden kivétel a `java.lang.Throwable` osztály leszármazottja. Ennek az osztálynak két közvetlen leszármazottja van, az `Error` és az `Exception` osztály. Az `Error` osztály leszármazottjai ún. nem ellenőrzött kivételek, amelyeket nem kötelező elkapni. (Ezek olyan hibákat definiálnak, amelyeket egyébként is hiába kapnánk el, a program nemigen tudna velük mit kezdeni. Ilyen kivétel például az `OutOfMemoryError` kivétel.) Az `Exception` osztály leszármazottai között található a `RuntimeException` osztály, amely szintén nem ellenőrzött kivételeket tartalmaz. Ezek nem végzetes hibák, de nem muszáj őket elkapni. Ilyen hiba például a nullával való osztás, amely a programban bárhol előfordulhat, ahol osztást hajtunk végre. Ez azt jelenti, hogy gyakorlatilag minden osztás műveletnél meg kellene adnunk, hogy ott keletkezhessen nullával való osztás kivétel, ami értelmetlen lenne.

Természetesen a nem ellenőrzött kivételeket is el lehet kapni, ha nem teszünk (nem kezeljük a kivételt), akkor a program futása megszakad (a program hibaüzenettel leáll).

Az `Exception` osztály többi leszármazottja ellenőrzött kivétel, ami azt jelenti, hogy ha egy metódus megírásakor jelezzük, hogy a metódus dobhat ilyen kivételt, akkor azt *el is kell* kapnia egy `catch`-blokknak.

4.3 ÖSSZEFOGLALÁS, KÉRDÉSEK

4.3.1 Összefoglalás

Ebben a leckében áttekintettük a Java alapvető vezérlési szerkezeit, illetve betekintést nyertünk a kivételkezelésbe. Ez a lecke alapvető fontosságú a további lecek megértéséhez, ezért ha úgy érzi, hogy nem értett meg mindent pontosan, akkor ne haladjon tovább, hanem térjen vissza arra a részre, amelyet nem értett meg.

4.3.2 Önellenőrző kérdések

1. Mire való az üres utasítás?
2. Milyen szabályok vonatkoznak az értékadás műveletre?
3. Milyen aritmetikai, összehasonlító és logikai operátorok léteznek a Java nyelvben?
4. Mire való ++ és a -- operátor? Mi a jelentősége annak, hogy prefix vagy posztfix alakot használunk-e?
5. Mire való a feltételes kifejezés?
6. Sorolja fel a feltétel nélküli vezérlésátadás utasításait. Hogyan működik a goto utasítás Javában?
7. Hogyan kell használni a kétirányú feltételes elágazás utasítását Javában?
8. Hogyan működik `switch` utasítás? Milyen speciális esetekkel találkozhatunk a `switch` használata során?
9. Milyen ciklusutasítások használhatók Javában?
10. Mire való a kivételkezelés?

5. LECKE: TÖMB, KARAKTERLÁNC ÉS ADATBEVITEL

5.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

Ebben a leckében két, nagyon fontos adatszerkezettel foglalkozunk: a tömbbel és a sztringgel. A Java mindkettőt objektumokkal valósítja meg, ez a lecke bemutatja ennek a módszernek az előnyeit. A leckében érintőlegesen szót ejtünk a Java által (objektumokon keresztül) támogatott további néhány adatszerkezetről is.

Ebben a leckében kapott helyet az adatbevitelről szóló rész is. A nyelvvel most ismerkedők számára a billentyűzetről (konzolról) érkező adatok fogadása a Java nyelvben meglehetősen bonyolultnak tűnhet. Való igaz, hogy más nyelvekben ennél kényelmesebb eszközök is a rendelkezésünkre állnak, de a Java adta módszer sem megérthetetlen, még a kezdők számára sem.

A lecke feldolgozásához szükséges kompetenciák: áttekintő képesség, rendszerező képesség, absztrakt gondolkodás, problémaelemzés és -feltárás.

5.2 TANANYAG

5.2.1 Az adatszerkezetekről dióhéjban

A Java által támogatott elemi adattípusokkal a 3. leckében ismerkedtünk meg. Elemi adatnak tekinthetünk minden olyan adatot, amelyet nem lehet, vagy legalább is nem célszerű további részekre bontani. Ilyen adat az ember életkora, magassága, neve stb., hiszen nincs értelme további részekre bontani azt az adatot, hogy valaki nő vagy férfi, illetve például 48 éves vagy 183 centiméter magas. (Elképzelhető olyan feladat, amikor éppen arra van szükségünk, hogy megvizsgáljuk egy felnőtt ember magasságának a 10-esek helyi értékén álló számjegyét, de ez nem túl gyakori probléma.)

Az összetett adatok ezzel szemben mindig elemi adatokból épülnek fel, azaz egy összetett adatot lehet (és általában érdemes vagy szükséges is) részeire bontani. Személyi adatainkat tekintve tipikusan összetett adat például a lakcímünk, amely általában az alábbi részekre bontható: irányítószám, helységnév, közterület neve, közterület típusa, házszám. Összetett adat lehet egy iskolai csoport névsora, vagy egy vasútállomás menetrendje stb.

Az összetett adatokat a magas szintű programozási nyelvek mindig adatszerkezetekkel valósítják meg. Az adatszerkezeteket alapvetően két féle szem-

pont alapján csoportosíthatjuk. Mindkét szempont szerint különböző típusú adatszerkezetekről beszélünk, minden adatszerkezet besorolható a két szempont alapján egy-egy kategóriába.

Adatszerkezetek fajtái a bennük tárolt elemek típusa szerint

Vizsgálunk kell, hogy az adott adatszerkezetet alkotó elemi adatok típusa megegyező-e, vagy az adatszerkezetben különböző típusú elemi adatok találhatók. Ha az adatszerkezetet alkotó elemi adatok azonos típusúak, akkor *homogén adatszerkezet*ről beszélünk, ha az adattípusok különböz(het)nek, akkor pedig *inhomogén* vagy *heterogén adatszerkezet*et használunk. Homogén adatszerkezetek pl. a *tömb*, *halmaz*, *sor*, *verem*, *lista*, *fa*, *háló*, *fájl*, heterogén adatszerkezet pedig a *rekord*. A *sztring* (karakterlánc) tekinthető elemi adatnak és adatszerkezetnek is, a Java adatszerkezetnek tekinti és objektumként kezeli. (Vannak olyan programozási nyelvek, amelyekben nem is létezik ez az adattípus, ilyen pl. a C nyelv.) Azokban a nyelvekben, amelyek támogatják a sztring típust, létezik eszköz a karakterlánc elemeinek (vagyis az egyes karaktereknek) az elérésére is. Ebből a szempontból a sztring értelmezhető egy karakterekből álló homogén adatszerkezetként (pl. tömb) is.

A rekord adatszerkezetben az egyes elemi adatokat *mezők*nek nevezzük. A mezők típusa eltérő lehet. A rekord adatszerkezet alapvető szerepet tölt be az ún. relációs adatbázis-kezelő rendszerekben.

Adatszerkezetek fajtái a bennük tárolt elemek száma szerint

Attól függően, hogy az adatszerkezet elemeinek száma változhat-e a program vagy az algoritmus végrehajtása során, beszélhetünk *statikus* és *dinamikus adatszerkezetek*ről. A statikus adatszerkezetek elemszáma a létrehozás (dekларáció) pillanatában eldől, a végrehajtás során ez a tulajdonság nem változtatható meg. Tipikusan statikus adatszerkezet a tömb, amelynek elemszámát a deklarációkor kell megadnunk. Programozási nyelvtől függően fordítási vagy futásidőben a rendszer lefoglalja a megfelelő méretű memóriaterületet (a memóriaterület az elemi adatok helyfoglalásától és az elemszámtól függ), ezért nem tudjuk megváltoztatni az elemszámot.⁷

⁷ A tömb a legismertebb és leggyakrabban használt homogén, statikus adatszerkezet, minden magas szintű programozási nyelv ismeri és támogatja. Számos nyelvben van lehetőségünk arra, hogy újradefiniáljuk a tömb méretét, vagyis megváltoztassuk az elemszámot futásidőben. Vannak olyan programozási nyelvek, amelyek tömbnek nevezett adatszerkezetében különböző típusú adatokat tárolhatnak. Ebben a jegyzetben nem célunk a különbségek teljes körű feltárása, sem pedig eldönteni, hogy valóban tömb-e az az adatszerkezet, amelyet egy adott nyelv annak nevez. A jegyzetben ezért mi ragaszkodni fogunk ahhoz a definícióhoz, hogy a tömb statikus, homogén adatszerkezet. Így is fogjuk használni.

Arra van lehetőségünk, hogy egy 100 eleműre deklarált tömbnek csak az első harminc elemét dolgozzuk fel, de tudnunk kell, hogy a maradék hetven elem számára is foglalt helyet a rendszer, és az elemszám így statikus.

Dinamikus adatszerkezeteknél a fordító vagy futtató rendszer nem foglal állandó méretű memóriaterületet. Az ilyen adatszerkezetek elemszáma futás-időben változhat. Ezeknél az adatszerkezeteknél gyakori az az eljárás, hogy létrehozunk egy üres (egyetlen elemet sem tartalmazó) adatszerkezetet, és futás-időben bővítjük vagy csökkentjük egy-egy elemmel. Dinamikus adatszerkezet például a lista, a sor, a verem, a különböző gráfok, illetve a fájl.

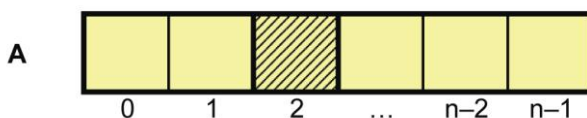
5.2.2 A tömb

A tömb statikus, homogén adatszerkezet, tehát minden elemének azonos típusúnak kell lennie, és az elemszám az adatszerkezet létrehozásakor (vagyis a deklaráció pillanatában) eldől. A tömb elemeire azok sorszámaival tudunk hivatkozni, az elemek sorszámát tömbindexnek nevezzük. A tömbök létrehozásakor az elemszámukat kell megmondanunk, a sorszámozás azonban mindig 0-tól kezdődik. Így ha például egy 10 elemű tömböt hozunk létre, akkor abban az elemek indexei 0 és 9 közöttiek lesznek. Egy tömb i -edik elemére `tömb-név[i]` módon hivatkozhatunk.

- ☐ Egy n elemű tömbnek sosincs n -edik indexű eleme. A legnagyobb index minden esetben $(n-1)$.

Amikor egy tömb elemeit egyetlen index is egyértelműen azonosítja, vagyis minden elemet egyetlen sorszámon keresztül el tudunk érni, egydimenziós tömbről beszélünk.

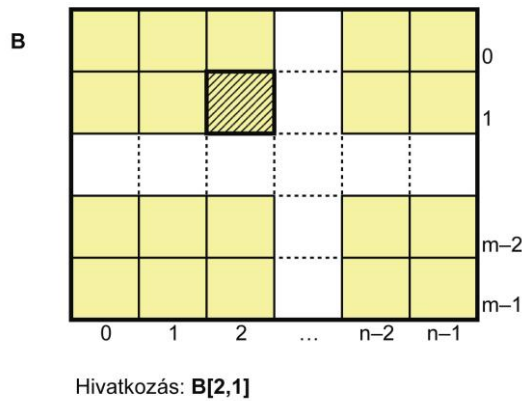
- ☐ Az egydimenziós tömböt hagyományosan szokás *vektornak* is hívni, ebben a jegyzetben azonban nem követjük ezt a hagyományt. A Javában ugyanis létezik egy `Vector` nevű adatszerkezetet, amely nem statikus: a létrehozás után is változhat az elemek száma újak beszúrásával, illetve meglévők törlésével. Ezért a félreértéseket elkerülendő az egydimenziós tömböt mindig egydimenziós tömbnek fogjuk nevezni.



Hivatkozás: **A[2]**

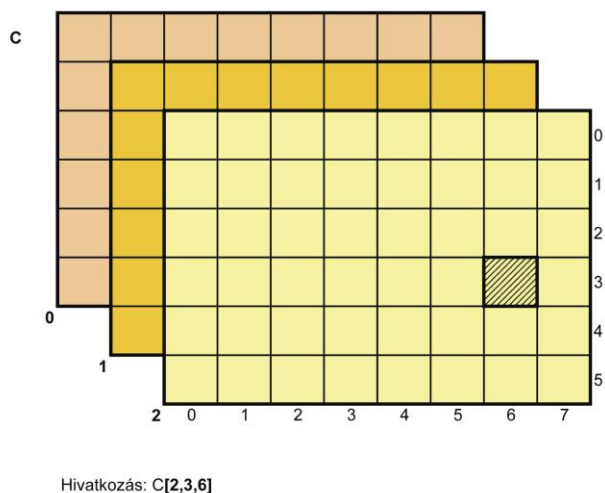
22. ábra: Egydimenziós tömb

Azokat a tömböket, amelyekben az egyes elemek eléréséhez két indexet kell megadnunk, kétdimenziós tömbnek (hagyományos elnevezéssel mátrixnak) nevezzük. A kétdimenziós tömböket úgy is elképzelhetjük, mint egy $n \times m$ méretű táblázatot.



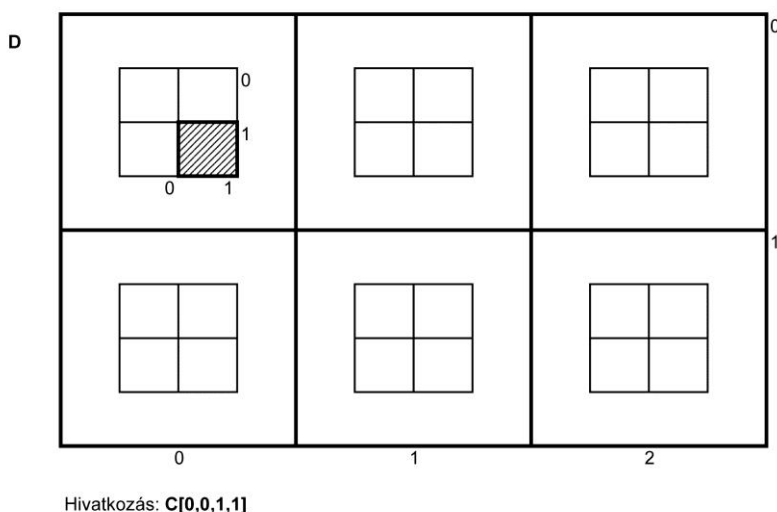
23. ábra: Kétdimenziós tömb

Amikor egy tömb elemeit egyenként három-három indexszel érhetjük el, háromdimenziós tömbről beszélünk. Egy háromdimenziós tömböt úgy képzelhetünk el, mint egy olyan struktúrát, amely lapokból áll, minden lapon azonos méretű $n \times m$ -es kétdimenziós tömbökkel.



24. ábra: Háromdimenziós tömb

Elképzelhetünk ennél magasabb dimenziószámú tömböket is. Egy négydimenziós tömböt például úgy, mint egy olyan $n \times m$ -es kétdimenziós tömböt, amelynek minden eleme egy-egy $u \times v$ méretű kétdimenziós tömb. Egy ötdimenziós tömb olyan konstrukció, amely lapokból áll, minden lapon azonos méretű négydimenziós tömbökkel, és így tovább.



25. ábra: Négydimenziós tömb

Két ok miatt nem szükséges, hogy belebonyolódjunk a dimenziókba. Egyrészt nagyon ritka az olyan probléma, amely nem oldható meg legfeljebb háromdimenziós tömbbel. Ha nem elegendő három dimenzió, lehetséges, hogy nem is tömböt kellene használnunk a megoldás során.

A másik, ennél sokkal triviálisabb ok az, hogy a Java nem ismeri a többdimenziós tömb fogalmát, azaz minden Java-tömb egydimenziós. Ez azonban nem jelent semmilyen megszorítást, ugyanis deklarálhatjuk a tömbök tömbjét, vagyis olyan tömböt, amelynek az elemei maguk is tömbök. Ez – kicsit leegyszerűsítve a dolgot – tulajdonképpen annyit jelent, hogy nem `B[2, 5]` módon hivatkozunk egy kétdimenziós tömb egyik elemére, hanem `B[2][5]` formában.

5.2.3 Tömbök létrehozása

A tömbök deklarálása hasonlít az elemi adattípusú változók deklarálására, mert itt is azonosítót használunk a tömb megnevezéséhez, és meg kell adnunk a tömb bázistípusát, azaz azt, hogy milyen típusú elemeket fog tárolni a tömb. Az

eltérés viszont az, hogy jelölnünk kell, hogy nem egyetlen adatot tároló változót deklarálunk, hanem több adatot tároló tömböt. Ezt szögletes zárójelekkel jelöljük. A Java két féle szintaxist is ismer tömbök deklarálására:

```
217 típus[] azonosító  
    illetve  
218 típus azonosító[]
```

Ezek teljesen egyenértékűek.

A deklaráció ezzel nem ért véget, mert meg kell adnunk a létrehozandó tömb elemszámát is. Ehhez használunk kell a `new` operátort.

- A Javában a tömbök is objektumok, ezért ugyanúgy példányosítjuk őket az osztályokból, mint a többi objektumot.

```
219 int[] a = new int[10];
```

Ezzel létrejön egy *a* nevű, tíz egész számot tárolni képes tömb. Az első elemére `a[0]` formában, míg a 10. elemére az `a[9]` módon hivatkozhatunk.

5.2.4 Értékadás tömbökben

Egy tömb elemeinek az alábbi módokon adhatunk értéket:

```
220 int[] a = new int[5];  
221 a[0] = 10;  
222 a[1] = 98;  
223 a[2] = -17;  
224 a[3] = 11;  
225 a[4] = 19;
```

Természetesen a tömbfeltöltés történhet cikluson belül is:

```
226 double[] b = new double[8];  
227 int i;  
228 for (i=0; i<8; ++i)  
229 {  
230     b[i] = i * 23 / 19.6;  
231 }
```

Ha a tömbelemek értéke nem számítható ki a fenti példa szerint a tömbindexekből (ez a gyakoribb eset), akkor az egyenkénti értékadásnál kényelemebb, ha felsoroljuk őket kapcsos zárójelek között.

```
232 int[] c = new int[] {7, 7, 23, 1, 92, -4, 6};
```

Figyeljük meg, hogy ebben a példában nem adtunk meg elemszámot. Ez egyrészt szükségtelen, hiszen felsoroltuk az összes elemet, másrészt a redundancia elkerülésével egy lehetséges hibát is elkerültünk. Ha a megadott elem-

szám nem egyezne meg felsorolt elemszámmal, vajon melyiket kellene figyelembe vennie a fordítóprogramnak?

Végezetül, az utolsó példa még ennél is egyszerűbben írható, mert amikor megadjuk egy tömb elemeinek kezdőértékét, akkor az értékadás jobb oldalán elegendő csak a felsorolást megadni:

```
233 char[] karakterek = {'z', 'm', 'g', 'F', '\t'};
```

Ezek a módszerek többdimenziós tömböknél is alkalmazhatóak:

```
234 int[][] skandinavLottoNyeroszamok = {  
235     {1, 7, 8, 9, 11, 20, 31},  
236     {2, 3, 5, 19, 24, 25, 26}  
237 };
```

- ❑ A skandináv lottó számainak húzásakor mindig két számsort húznak ki. Az elsőt géppel, a másodikat kézzel. A fenti példában definiált sorsoláskor a kézi sorsolás harmadik nyerőszámát a `skandinavLottoNyeroszamok[1][2]` formában érhetjük el.

Ahogy arra többször is utaltunk már, a tömbök a Java nyelvben objektumok. Ennek számos előnye van, az egyik – kezdők számára is kézzel fogható – sajátosság, hogy a tömbök ismerik a saját elemszámukat. Ezt a `tömb-név.length` formában érhetjük el.

```
238 int[] a = {19, 23, 41, 42, 56, 78, 98, 100};  
239 int i;  
240 for (i = 0; i < a.length; ++i)  
241 {  
242     System.out.println(a[i]);  
243 }
```

5.2.5 A tömb mint referenciatípus

A Java másként kezeli az egyszerű adattípusú változókat, mint az objektumokat. Amikor deklarálunk egy elemi adattípussal rendelkező változót, akkor az a létrehozás területén (a befogadó területen, például az őt deklaráló objektumnak az adatterületén) helyezkedik el, vagyis ezen a tárterületen fog megtörténni a címhozzárendelés.

Referenciatípusok esetén a létrehozott példány (objektum) által lefoglalt memóriaterület nem a létrehozás helyén fog elhelyezkedni, hanem egy külön tárterületet fog elfoglalni a memória egy másik területén. A befogadó területen csak egy hivatkozás, más szóval referencia lesz eltárolva. A referencia segítség-

gével ugyanazt az adatszerkezetet az adatok tényleges lemásolása nélkül több helyről is elérhetjük.

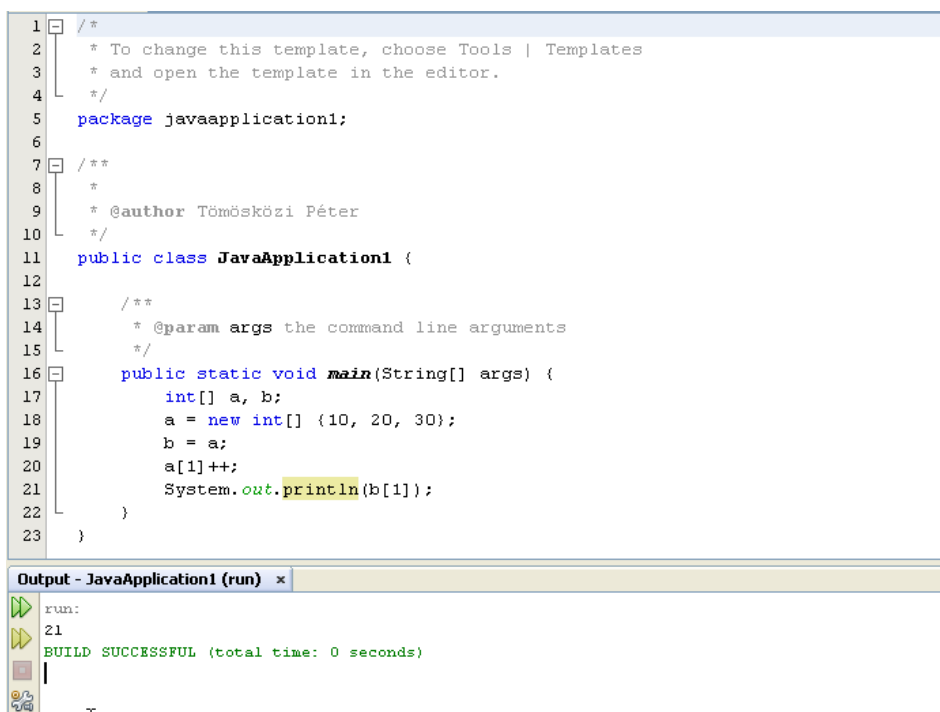
Tekintsük a következő kódrészletet:

```
244 public static void main(String[] args) {  
245     int[] a, b;  
246     a = new int[] {10, 20, 30};  
247     b = a;  
248     a[1]++;  
249     System.out.println(b[1]);  
250 }
```

A példában deklaráltunk két tömböt, *a*-t és *b*-t, mindkettőt egészek tárolására. Az *a* tömbnek értékeket adtunk, majd a *b* tömbnek *értékül adtuk* az *a* tömböt. Más programozási nyelvekben (pl. C++) itt valódi adatmozgatás történt volna: az adatok az *a* tömb által lefoglalt memóriaterületről átmásolódtak volna a *b* tömb által lefoglalt memóriaterületre. A Javában nem ez történt: a *b* tömb referenciaként megkapta az *a* tömb referenciáját, vagyis a *b* tömbhöz rendelt memóriaterület ugyanaz lett, mint ami az *a* tömbhöz volt rendelve. Magyarul: az értékadás után az *a* és a *b* tömb *ugyanarra* a memóriaterületre hivatkozik.

Ezután megnöveltük az *a* tömb második (1-es indexű) elemének értékét 1-gyel, de ezzel egyben a *b* tömb második (1-es indexű) elemének az értékét is megnöveltük, hiszen a két tömb valójában ugyanazt a memóriaterületet használja. Így, hiába az *a* tömb egyik elemének értékét változtattuk meg, a *b* tömb ugyanazon elemének kiírásakor is az így megnövelt értéket kapjuk.

A fenti kód futtatásának eredménye 21 lesz.



```
1  /*
2  * To change this template, choose Tools | Templates
3  * and open the template in the editor.
4  */
5  package javaapplication1;
6
7  /**
8   *
9   * @author Tömösközi Péter
10  */
11  public class JavaApplication1 {
12
13      /**
14       * @param args the command line arguments
15       */
16      public static void main(String[] args) {
17          int[] a, b;
18          a = new int[] {10, 20, 30};
19          b = a;
20          a[1]++;
21          System.out.println(b[1]);
22      }
23  }
```

Output - JavaApplication1 (run) x

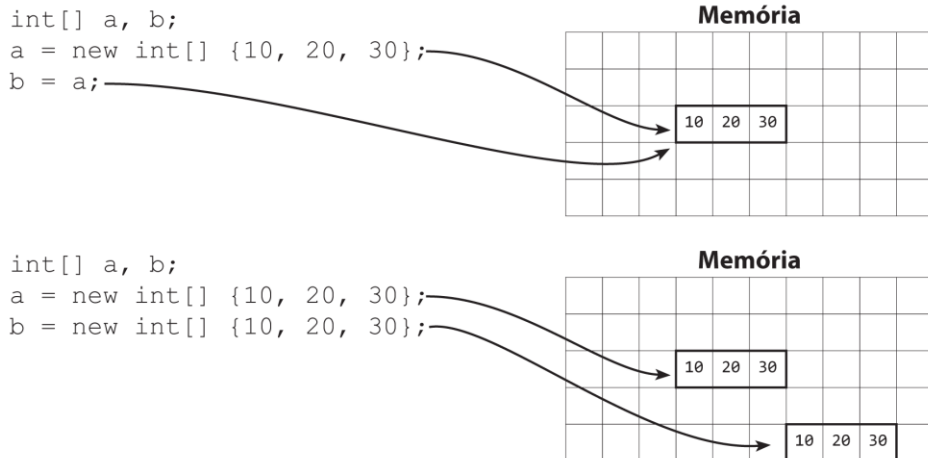
```
run:
21
BUILD SUCCESSFUL (total time: 0 seconds)
```

26. ábra: Az a tömböt manipuláltuk, de ezzel a b is változott, hiszen a két tömb azonos memóriaterületet használ

Két tömböt tekinthetünk egyenlőnek, ha ugyanazokat az elemeket tartalmazzák.

Ha egy tömb értékül kap egy másik tömböt, akkor az értékadás bal oldalán álló tömb referenciaként megkapja a jobb oldalon álló tömb referenciáját. Ebben az esetben a két tömb valójában azonos adatterületet kap, vagyis ők nem csupán ugyanazokat az elemeket fogják tartalmazni, hanem a két tömb valójában egy és ugyanaz lesz, csak más azonosítókkal férünk hozzá ugyanahhoz a tárterülethez. Ilyenkor azt mondjuk, hogy a két tömb *azonos*.

Ha a két tömb úgy kap azonos értékeket, hogy nem az egyiket adjuk értékül a másiknak, akkor a két tömb *egyenlő* lesz, de nem *azonos*.



27. ábra: Tömbök azonossága és egyenlősége

Ha két tömb azonos, akkor őket az `==` operátorral összehasonlítva `true` (igaz) értéket kapunk.

```

251 public static void main(String[] args) {
252     int[] a, b;
253     a = new int[] {10, 20, 30};
254     b = a;
255     if (a == b)
256     {
257         System.out.println("Azonosak");
258     }
259     else
260     {
261         System.out.println("Nem azonosak.");
262     }
263 }

```

Ennek a kódrészletnek a futtatása az „Azonosak” kimenetet produkálja.

Ha a két tömb nem azonos, viszont szeretnénk az elemeik egyezőségét vizsgálni, akkor az `Array` nevű statikus osztály `equals` metódusát kell használnunk.

```
264 public static void main(String[] args) {  
265     int[] a, b;  
266     a = new int[] {10, 20, 30};  
267     b = new int[] {10, 20, 30};  
268     if (Arrays.equals(a, b))  
269     {  
270         System.out.println(„Egyenlők.”);  
271     }  
272     else  
273     {  
274         System.out.println(„Nem egyenlők.”);  
275     }  
276 }
```

Ez a kódrészlet futáskor az „Egyenlők” eredményt fogja kiírni. Ha az *a* és *b* tömböt az `==` operátorral hasonlítanánk össze, akkor viszont negatív eredményt kapnánk: ez a két tömb nem azonos.

5.2.6 Karakter és String

Az elemi adattípusok között már bemutattuk a `char` típust. Egy ilyen típusú változó értéke egyetlen karakter lehet. A szöveges adatok kódolásához a számítógépek a kezdetektől fogva kódtáblázatokat használnak. Ez azt jelenti, hogy a szöveges adatokat alkotó karakterekhez egy-egy számot rendelünk, amelyet egy kódtáblázat tartalmaz. Tulajdonképpen tehát az egyetlen karakter tárolására szolgáló adattípus a tárolt karakter kódját tárolja.

Az egyik legelterjedtebb ilyen kódtábla az ASCII (American Standard Code for Information Interchange) volt, amely nevével ellentétben sajnos nem volt mindenki által elfogadott szabvány. Ugyan minden számítógép használt valamilyen ASCII-implementációt, de ezek nem voltak egymással kompatibilisek, ami lehetetlenné tette a tényleges információcserét különböző típusú számítógépek között. Az ASCII másik hibája, hogy elsősorban angol nyelvű szövegek kódolására volt alkalmas. Az ASCII eredetileg 7 bites kódolás volt, ez 128 különböző karakter kódolását tette lehetővé. Később 8 bitesre egészítették ki a szabványt, ezzel a kódolható karakterek szám a duplájára, 256-ra nőtt. Ez a 256-os korlát elegendő volt az angol ábécé 26 kis- és nagybetűjének, az írásjeleknek, a számjegyeknek a kódolásához, tehát azoknak a jeleknek a tárolásához, amelyek angol nyelvű szövegekben fordulnak elő. Az ASCII ezen kívül tartalmazott még számos grafikai jelet, amelynek segítségével pl. táblázatok szegélyét lehetett megrajzolni.

Más, latin betűs írást használó nyelveken írott szövegek általában nem voltak 100%-ig kódolhatók ASCII-vel, mert ez a kódrendszer nem tartalmazott minden ékezetes vagy egyéb, segédjellel ellátott latin betűt. Magyar nyelvű szövegek kódolásához vagy ékezetek nélkül, vagy a magyar ékezetes betűkre legjobban emlékeztető („kalapos”, „hullámos”) ékezetes jelek bevonásával nyílt mód.

Az ASCII-nek később számos kiterjesztése jelent meg, ezeket az ISO (International Standard Organization) szabványosította. Az ISO 8859-es szabványcsaládban olyan karakterkódolási szabványok kaptak helyet, amelyek alapvetően az ASCII-re épültek, de az egyik kódtáblákba beépítették más, latin betűs ábécék jeleit. Az ISO-8859-1-es szabvány a nyugat-európai nyelvek speciális jeleit tartalmazza, az ISO-8859-2 a kelet- és közép-európai nyelvek ábécéinek speciális jeleit.

Az ISO-8859-ről bővebben olvashatunk pl. az alábbi címen: http://hu.wikipedia.org/wiki/ISO_8859-n

A leírt problémák megoldására született meg az Unicode szabvány, illetve a szabványhoz tartozó karakterkódolások. Az Unicode szabvány 16 bites karakterkódolást definiál, ez azt jelenti, hogy összesen 65536 elemű kódtáblát használhatunk a segítségével. Ekkora méretű kódtábla lehetővé teszi gyakorlatilag az összes ma létező és használt írás jeleit.

A Java nyelv az Unicode-kódolást használ, vagyis egy `char` típusú adat egy tetszőleges Unicode jel lehet.

Egy `char` típusú változó egyetlen karaktert kaphat értékül, a karakterliterálokat aposztrófok között kell megadni. A 3.2.4. fejezetben szó esett az Escape-szekvenciákról. Ezek segítségével egyetlen karaktert definiálunk, így ha egy `char` típusú változónak ilyen értéket adunk, az aposztrófok között két jelet adunk meg. Pl.:

```
277 char újsor = '\n';
```

A `char` típus egy egészjellegű típus. Ez azt jelenti, hogy egy `char` típusú érték egészszé konvertálható. Ilyenkor az egészszé konvertált érték az eredeti karakter kódja lesz.

```

1  /*
2   * To change this template, choose Tools | Templates
3   * and open the template in the editor.
4   */
5   package javaapplication1;
6
7   /**
8    *
9    * @author Tömösközi Péter
10   */
11   public class JavaApplication1 {
12
13       /**
14        * @param args the command line arguments
15        */
16       public static void main(String[] args) {
17           int a;
18           char b = 'A';
19           a = b;
20           System.out.println(a);
21       }
22   }

```

Output - JavaApplication1 (run) x

```

run:
65
BUILD SUCCESSFUL (total time: 2 seconds)

```

28. ábra: A `char` típusú adat `int` típusúvá konvertálható

A Javában a karakterláncok (sztringek) ábrázolásához a `String` osztály használható. A sztringek tehát a `String` osztályból példányosított objektumok.

```
278 String név1 = new String("László");
```

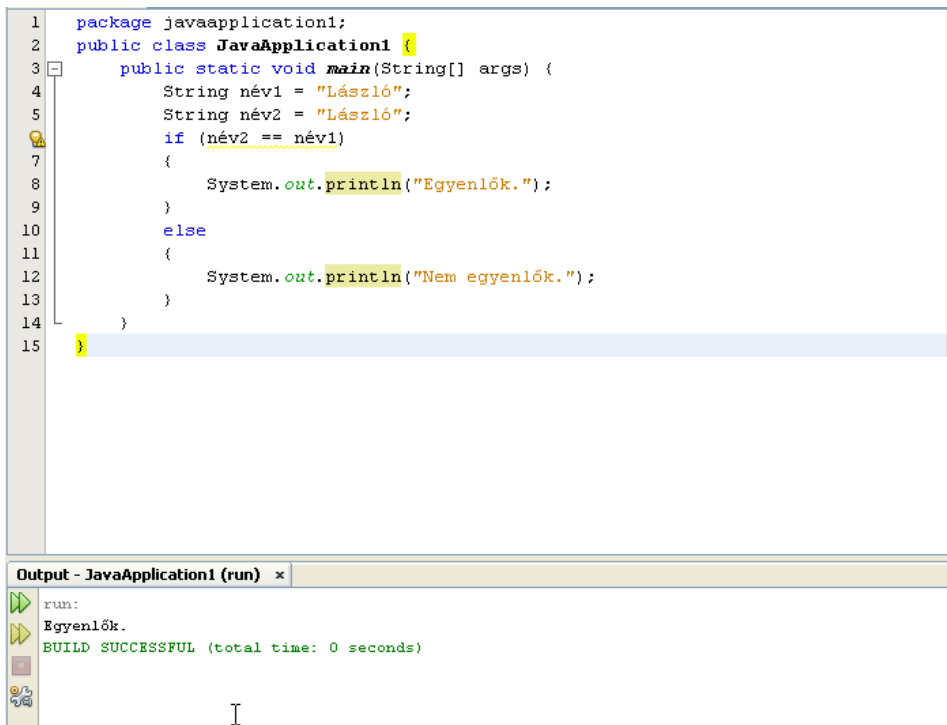
Ugyanakkor a nyelv több eszközzel is támogatja a sztringek egyszerűbb kezelését. Idézőjelek közé írt sztringliterálok úgy is értékül adhatók egy `String` típusú változónak, ha a példányosítást nem írjuk ki explicit módon a `new` operátor használatával.

```
279 String név2 = "Tamás";
```

A sztringek tárolását a Java karaktertömbökkel valósítja meg. Így egy sztring létrehozható egy karaktertömbből is.

```
280 char[] betuk = {'H', 'é', 't', 'f', 'ő'};
281 String nap = new String(betuk); // „Hétfő”
```

A nyelv a könnyebb kezelés érdekében lehetővé teszi, hogy két sztring egyenlőségét megvizsgáljuk az `==` operátorral.



```
1 package javaapplication1;
2 public class JavaApplication1 {
3     public static void main(String[] args) {
4         String név1 = "László";
5         String név2 = "László";
6         if (név2 == név1)
7         {
8             System.out.println("Egyenlők.");
9         }
10        else
11        {
12            System.out.println("Nem egyenlők.");
13        }
14    }
15 }
```

Output - JavaApplication1 (run) x

run:
Egyenlők.
BUILD SUCCESSFUL (total time: 0 seconds)

29. ábra: Sztringek egyenlősége

Két sztringliterál csak akkor egyenlő, ha kis- és nagybetűk szerint tartalmazzák ugyanazokat a karaktereket, ugyanabban a sorrendben. A sztringek összehasonlításakor a Java a sztringeket tároló karaktertömbök azonos indexű pozícióin álló karaktereit hasonlítja össze. Mivel az azonos betűk kis- és nagybetűs változatának kódja különbözik (pl. A – 65, a – 97), a csak kis- és nagybetűkben eltérő sztringek sem lesznek egyenlők. Hozzá kell tennünk, hogy ez az állítás csak akkor igaz, ha az összehasonlítást az == operátorral végezzük. A String osztály használatával van módunk arra, hogy két sztring egyezőségét megvizsgáljuk a kis- és nagybetűk figyelmen kívül hagyása mellett. (Erről később, a String osztály metódusai között írunk részletesen.)

```
1 package javaapplication1;
2 public class JavaApplication1 {
3     public static void main(String[] args) {
4         String név1 = "Nemecsek Ernő";
5         String név2 = "nemecsek ernő";
6         if (név2 == név1)
7         {
8             System.out.println("Egyenlők.");
9         }
10        else
11        {
12            System.out.println("Nem egyenlők.");
13        }
14    }
15 }
```

Output - JavaApplication1 (run) x

run:
Nem egyenlők.
BUILD SUCCESSFUL (total time: 0 seconds)

30. ábra: A kis- és nagybetűk különböznek sztringek összehasonlításakor

Sztringliterálok között értelmezett a + operátor, de természetesen nem összegképzésre. A + operátor sztringek között a konkatenálást, azaz az összefűzést jelenti.

```
1 package javaapplication1;
2 public class JavaApplication1 {
3     public static void main(String[] args) {
4         String családnév = "Kovács";
5         String keresztnév = "Mihály";
6         String név = családnév + " " + keresztnév;
7
8         System.out.print("Teljes név: ");
9         System.out.println(név);
10    }
11 }
```

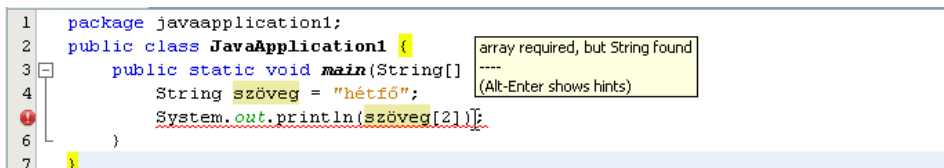
Output - JavaApplication1 (run) x

run:
Teljes név: Kovács Mihály
BUILD SUCCESSFUL (total time: 0 seconds)

31. ábra: Sztringek összefűzése a + operátorral

A String osztályból létrehozott objektumok konstansok, azaz amikor egy sztringen módosítás jellegű műveletet hajtunk végre, nem az eredeti sztring változik, hanem mindig új sztring jön létre.

Habár a sztring karaktereit tömb segítségével tárolja a Java, a karakterek mégsem érhetők el úgy, ahogyan a tömböknél leírtuk azt.



```

1 package javaapplication1;
2 public class JavaApplication1 {
3     public static void main(String[]
4         String szöveg = "hétfő";
5         System.out.println(szöveg[2]);
6     }
7 }

```

32. ábra: A sztring közvetlenül nem kezelhető tömbként

A megoldáshoz tekintsük át a String osztály néhány hasznos metódusát!

5.2.7 A String osztály fontosabb metódusai

Az alábbi felsorolás nem teljes. A metódusok megértéséhez, illetve használatukhoz szükséges lehet a 6–8. fejezet áttekintése is. Vegyük észre, hogy a felsorolt metódusok között több is polimorf, azaz ugyanaz a metódusnév más-más műveletet eredményez attól függően, hogy milyen a metódushívásnál megadott aktuális paraméterlista.

`toCharArray()` – Stringből karaktertömböt hoz létre.

```

282 String név = „László”;
283 char[] betűk = név.toCharArray();
284 // eredmény: a betűk tömb tartalma az alábbi
    lesz:
285 // {'L', 'á', 's', 'z', 'l', 'ó'};

```

`valueOf(adat)` – Szöveget hoz létre a megadott elemi típusú adatból. Az adat típusa `boolean`, `char`, `int`, `long`, `float` és `double` lehet.

```

286 boolean logikai = (5 > 3);
287 int szám = 23;
288 double kétPI = 6.28;
289 String strLogikai = String.valueOf(logikai); //
    „true”
290 String strSzám = String.valueOf(szám); // „23”
291 String strKétPI = String.valueOf(kétPI); //
    „6.28”

```

`charAt(pozíció)` – Megadja az adott pozíción található karaktert. A sztring első karakterének pozíciója – a tömbindexekhez hasonlóan – a 0.

```
292 String hónap = „május”;  
293 char betű = hónap.charAt(2); // a betű értéke  
    'j' lesz
```

`length()` – A szöveg hossza (a karaktereinek darabszáma).

`substring(eleje, vége)` – Rész-sztringet hoz létre az `eleje` indexű karaktertől a `vége-1` pozíción álló karakterig.

`substring(kezdőpozíció)` – Részszöveget hoz létre a kezdőpozíciótól az eredeti sztring utolsó karakteréig.

`trim()` – Levágja a sztring elejéről és a végéről a szóközöket, tabulátorokat ('\\t') és újsor-karaktereket ('\\n').

`compareTo(szöveg)` – Összehasonlítja a hívó sztringet a paraméterként megadott sztringgel. A visszaadott érték 0, ha a két sztring teljesen megegyezik, negatív, ha az alapsztring kisebb, pozitív, ha a paraméterként megadott sztring a kisebb.

```
1 package javaapplication1;  
2 public class JavaApplication1 {  
3     public static void main(String[] args) {  
4         String egyik = "körző";  
5         String másik = "kötény";  
6         System.out.println(egyik.compareTo(másik));  
7     }  
8 }
```

33. ábra: A `compareTo` metódus használata

A visszaadott (vagyis a képernyőre írt) érték a fenti példában `-2` lesz, mivel a „körző” szó a kisebbik (előrébb van a betűrendben). (A visszaadott érték abszolút értéke egy viszonylag bonyolult eljárással számítható ki. Az abszolút érték függ attól, hogy a szavak azonos hosszúságúak-e vagy sem, illetve hogy mi a karakterkódja az első eltérést mutató pozíciókon található karaktereknek.)

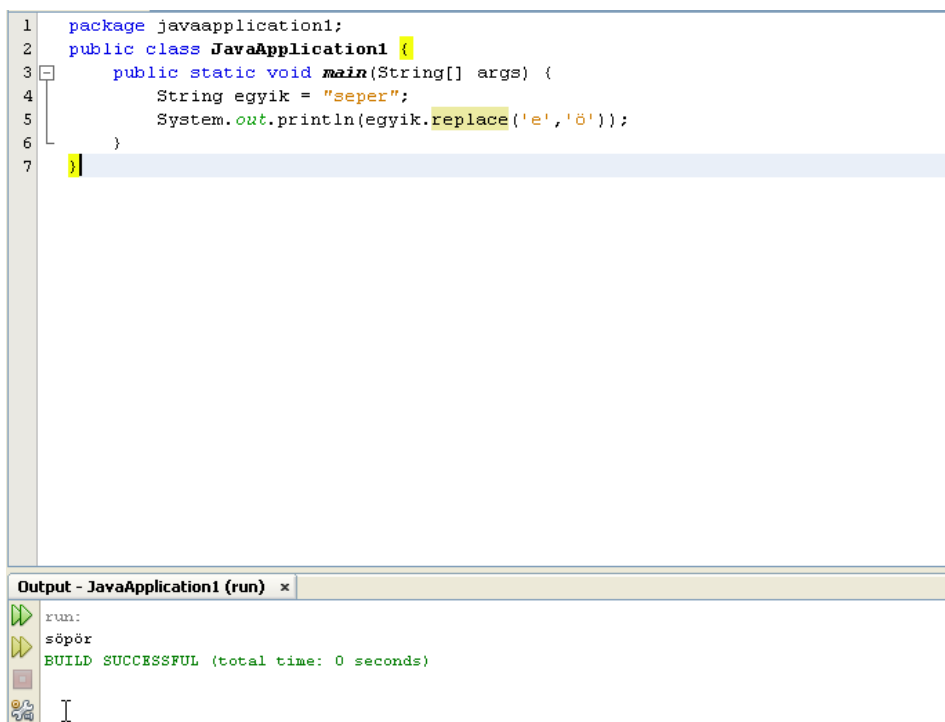
`equals(szöveg)` – Összehasonlítja a hívó sztringet a paraméterként átadott sztringgel. A visszaadott érték `true`, ha a sztringek megegyeznek, és `false`, ha különböznek. Ez a metódus különbséget tesz a kis- és nagybetűk között.

`equalsIgnoreCase(szöveg)` – Ugyanúgy működik, mint az előző metódus, de nem tesz különbséget a kis- és nagybetűk között.

`toLowerCase()` – A hívó sztring betűit kisbetűssé konvertálja.

`toUpperCase()` – A hívó sztring betűit nagybetűssé konvertálja.

`replace(régi, új)` – A hívó sztringben a régi karakter összes előfordulását lecseréli az új karakterre.



```

1 package javaapplication1;
2 public class JavaApplication1 {
3     public static void main(String[] args) {
4         String egyik = "seper";
5         System.out.println(egyik.replace('e', 'ö'));
6     }
7 }

```

Output - JavaApplication1 (run) x

```

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

```

34. ábra: A `replace()` metódus működése

5.2.8 Adatbevitel

A képernyőre írás a legelső utasítás volt, amelyet megismertünk a jegyzet elején. Más programozási nyelvekben a billentyűzetről való adatbekérés hasonlóképp egyszerű, a Javában azonban nem ez a helyzet. Éppen ezért az eddigi példákban kerültük is ezt a kérdést.

Ha olyan programot akarunk készíteni, amely a felhasználótól kér be adatokat a futtatás során, akkor három lehetőségünk van. Az első, ha ezeket az adatokat a felhasználó már a program futtatásakor, mint parancssori paramétereket ad meg.⁸

⁸ A NetBeans vagy más integrált fejlesztői környezet használata nélkül a felhasználó az operációs rendszer parancssorát kényyszerülne használni a megírt Java program fordításához és futtatásához. Ilyenkor a forráskódot a parancssorba írt `javac` paranccsal lehet fordítani, a lefordított

Parancssori paraméterek (argumentumok)

Vessünk egy pillantást ismét a Java programok szerkezetére. Eddig nem esett szó róla, hogy a `main` metódus paramétere egy sztringeket tartalmazó tömb. A Java nyelvben ez a tömb szolgál a parancssorból történő futtatás során megadott argumentumok kezelésére.

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

Természetesen nem kötelező éppen az `args` nevet adnunk ennek a tömbnek. Amikor a lefordított program futtatását kezdeményezzük, megadhatunk olyan bemenő paramétereket, amelyek magának a programnak lesznek a bemenő paraméterei, vagy argumentumai.

Tekintsük az alábbi példát! Az a feladatunk, hogy a futtatáskor bemenő paraméterként megadott vezetéknévből és keresztnévből állítsuk össze valaki teljes nevét.

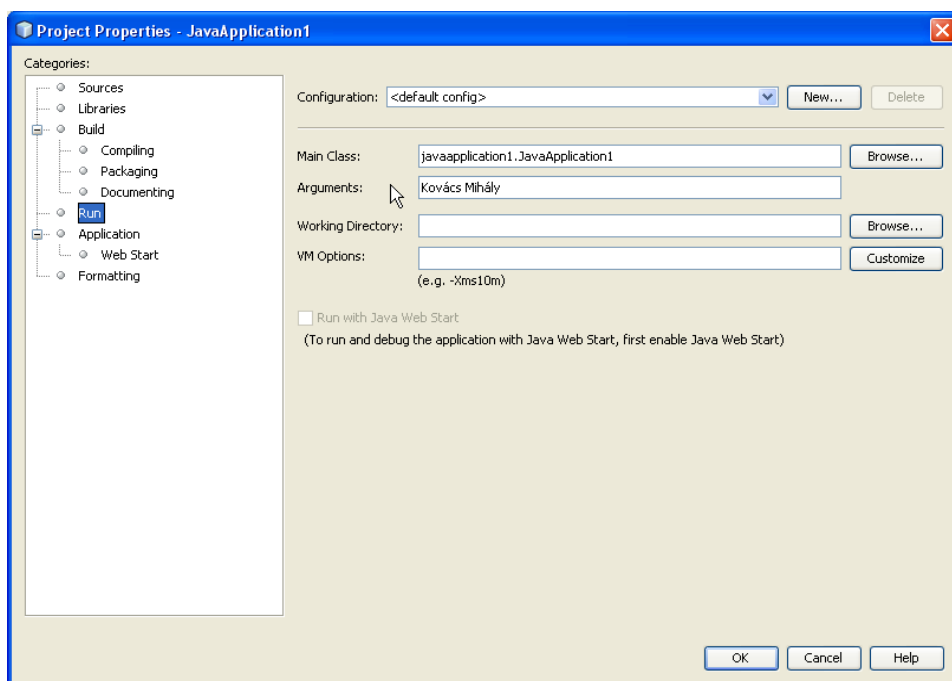
```
1 package javaapplication1;
2 public class JavaApplication1 {
3     public static void main(String[] args) {
4         String név = args[0] + " " + args[1];
5         System.out.println("A teljes név: " + név);
6     }
7 }
```

35. ábra: Felhasználjuk a program bemenő paramétereit

Ha a futtatást parancssorból végeznénk, akkor a bemenő paramétereket a `java <programnév> paraméter0 paraméter1 ...` módon adhatnánk meg, ahol a `<programnév>` helyére a lefordított bájtkódot tartalmazó fájl nevét kellene írunk.

A NetBeans használata mellett ezt a Run menü Set project configuration menüpontjának Customize... parancsával tudjuk beállítani.

bájtkód interpretálása pedig a `java` paranccsal történhet. Ez utóbbi után kell odaírni a lefordított bájtkódot tartalmazó fájl nevét, e mögé pedig szóközzel elválasztva az esetleges argumentumokat.



36. ábra: A bemenő paramétereket szóközzel elválasztva az Arguments mezőnél adjuk meg

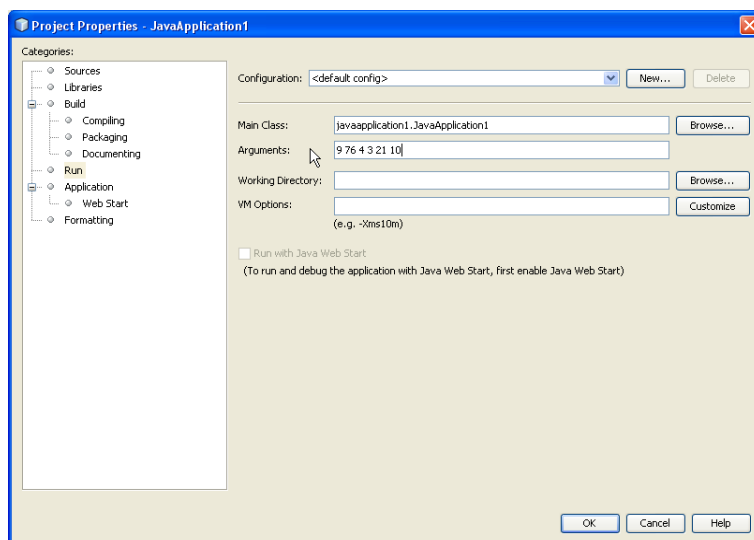
Ha most lefuttatjuk a programot, akkor a futás – ami nem lesz túl látványos – eredményeként megjelenik a paraméterek formájában megadott teljes név. (A végeredmény ugyanaz lesz, mint amit beírtunk az Arguments mezőbe, de az args tömb külön elemként fogja tárolni a vezetéknévét és a keresztnévét.)

Nézzünk egy látványosabb példát: készítsünk összeadó gépet. A program feladata az lesz, hogy a bemenő paraméterek formájában megadott egész számokat összeadja, és kiírja a számok összegét. A programnak működnie kell, akárhány számot is ad meg a felhasználó bemenő paraméterként.

Az első feladat az lesz, hogy meghatározzuk a megadott paraméterek darabszámát, majd egy összeg nevű segédváltozóban összeadjuk ezeket. Az összeg kezdőértékét 0-ra állítjuk, majd a megadott paramétereket egyenként hozzáadjuk ehhez a változóhoz, így képezve a részösszegeket. Amikor az utolsó paramétert is hozzáadtuk az előző paraméterek részösszegeihez, akkor a megkapott végösszeget kiírjuk a képernyőre.

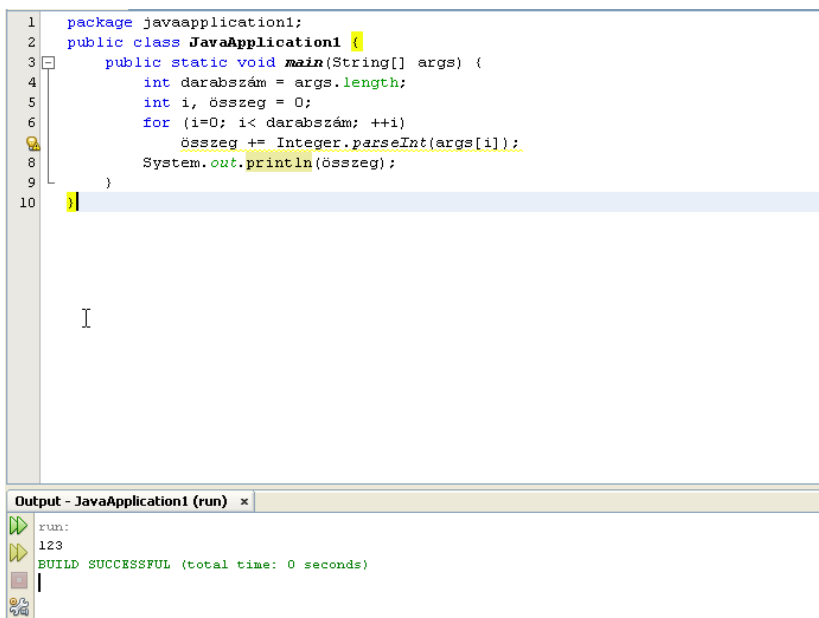
Arra ügyelnünk kell, hogy a bemenő paramétereket tároló tömb String típusú, sztringeket viszont nem lehet összeadni. Ezért minden tömbelemet egészszé kell alakítanunk (konvertálnunk), és így az összegzés már elvégezhető lesz. Ehhez az Integer osztály `parseInt()` metódusát fogjuk használni.

Legyenek a bemenő paraméterek az alábbi számok: 9, 76, 4, 3, 21 és 10.



37. ábra: Adjuk meg a bemenő paramétereket szóközzel elválasztva

Lássuk a programot és a futás eredményét:



38. ábra: 9, 76, 4, 3, 21 és 10 összege 123.

Adatbevitel billentyűzetről

A parancssori argumentumok használata nem mindig megfelelő módszer, hiszen az adatbevitelnek általában interaktív módon kell működnie: a felhasználó a futás során adja meg a bemenő adatokat. A Javában erre nincs olyan egyszerű konzolművelet, mint a `System.out.print()`, ami az adatkíráshoz használható. Helyette úgy kell kezelnünk a billentyűzetet, mintha az egy szövegfájl volna, és a fájl végét az Enter jelezné. Ehhez egy ún. csatornát kell létrehozunk. Ezt hozzárendeljük a standard inputhoz (a konzol bemeneti eszközéhez, azaz a billentyűzethez), és puffertelt adatbeolvasást hajtunk végre. A megoldáshoz szükséges a `java.io` csomag importálása, ezért ezt jeleznünk kell a fordító számára a forráskód elején! (Csomagokról a 6. fejezetben esik szó bővebben.)

Problémát az okozhat, ha nem olyan típusú adatot ad meg a felhasználó, amilyen típusúnak azt a program fel akarja dolgozni. (Valójában ez már a futtatókor megadott argumentumok esetében is problémát okozhatott volna, de csak most térünk ki rá részletesebben.)

Ha a program például egy egész értéket vár, de a felhasználó olyan adatot ír be, ami nem konvertálható egészszé, az egy kivételt fog generálni. Ha ezt nem kezeljük, akkor a program futása hibával leáll. A fájlkezelés esetében, így a billentyűzetről való adatbeolvasáshoz is rendszerint kivételkezelést kell alkalmaznunk. (A kivételkezelésről a 4. leckében olvashattunk részletesen.)

Az alábbi kódrészlet egy egész számot kér be a billentyűzetről, majd kiírja a kétszeresét.

```

1 package javaapplication1;
2 import java.io.*;
3 public class JavaApplication1 {
4     public static void main(String[] args) {
5         System.out.print("Írjon be egy egész számot: ");
6         BufferedReader reader = new BufferedReader(new InputStreamReader (System.in));
7         try {
8             String str = reader.readLine();
9             int szám = Integer.parseInt(str);
10            System.out.println("Ennek a számnak a kétszerese: " + szám * 2);
11        }
12        catch (IOException hiba) { }
13    }
14 }

```

Output - JavaApplication1 (run) x

```

FUTÓ:
Írjon be egy egész számot: 62
Ennek a számnak a kétszerese: 124
BUILD SUCCESSFUL (total time: 5 seconds)

```

39. ábra: Adatbekérés konzolról

Űrlap készítése adatbevitelhez

Az adatbevitel feladatának megoldásához a legjobb megoldás, ha grafikus felülettel rendelkező programot készítünk, és űrlapot hozunk létre a felhasználó által megadandó adatok fogadására. Ezt a megoldást a 9–10. fejezetben mutatjuk be.

5.3 ÖSSZEFOGLALÁS, KÉRDÉSEK

5.3.1 Összefoglalás

Ebben a leckében a tömb és sztring adatszerkezetekkel ismerkedtünk meg. A Java mindkettőt osztályokkal (és így objektumokkal) valósítja meg. Az osztályokról és az objektumokról a következő fejezetekben szólnunk részletesen, ezért előfordulhat, hogy az ebben a leckében szereplő információk egy része csak a következő fejezetek feldolgozása után lesz teljesen világos az Olvasó számára.

5.3.2 Önellenőrző kérdések

1. Hogyan csoportosíthatók az adatszerkezetek?
2. Hogyan kell tömböt definiálni Javában? Legfeljebb hány dimenziós tömböket definiálhatunk Javában?
3. Hogyan tudunk értéket adni tömbelemeknek?
4. Mit jelent az, hogy a tömb egy referenciatípus?
5. Mi a különbség karaktertömbök és sztringek között?
6. Hogyan tudunk összehasonlítani két sztringet lexikografikusan?
7. Milyen lehetőségek állnak rendelkezésünkre adatbevitel megvalósításához?

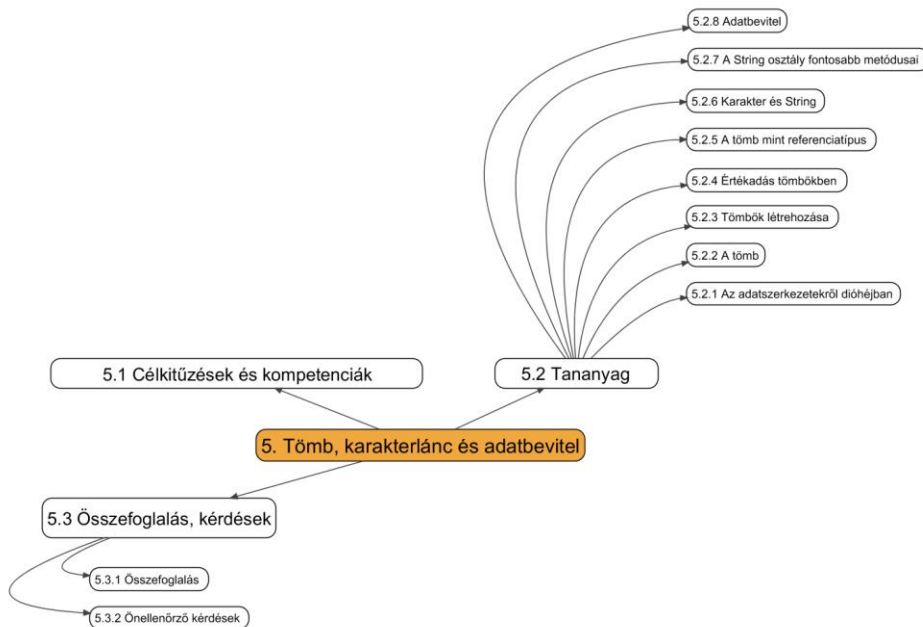
6. LECKE: OBJEKTUMORIENTÁLT PROGRAMOZÁS I. – OSZTÁLYOK ÉS OBJEKTUMOK. JAVA OSZTÁLY- KÖNYVTÁR, CSOMAGOK

6.1 CÉLKITÚZÉSEK ÉS KOMPETENCIÁK

Az első öt leckében számos ponton szóba kerültek az objektumorientált programozás alapvető eszközei, illetve a Java OOP-s jellemzői, sajátosságai. Ennek a leckének az a célja, hogy feltárja azokat az összefüggéseket, amelyek nélkülözhetetlenek a Java programozási nyelv használatához, és ehhez felhasználjuk az előző leckékben megszerzett tudást.

A lecke feldolgozásához szükséges kompetenciák: logikai képesség, induktív gondolkodás képessége, deduktív gondolkodás képessége, absztrakt (elméleti) gondolkodás, áttekintő képesség, rendszerező képesség és figyelemösszpontosítás.

6.2 TANANYAG



6.2.1 Osztályok, objektumok, konstrukció

Ahogy arról a bevezető leckében szó esett, az objektumorientált programozás (a továbbiakban OOP) egyik alapgondolata, hogy foglaljuk egységbe az adatokat, illetve azokat a műveleteket, amelyeket az adatokon szeretnénk értelmezni. Oldjunk meg egy egyszerű számítási feladatot, amelyben bizonyos adatok ismeretében más adatokat számítunk ki.

Matematikai értelemben minden ilyen művelet (számítás) egy függvény, amely az ismert adatok birtokában meghatározza a kiszámított értéket.

Legyen a feladatunk az, hogy egy kör sugarának ismeretében számítsuk ki annak területét és kerületét. Ehhez – matematikai értelemben – két függvényre (köznapibb kifejezéssel két funkcióra) lesz szükségünk, de mindkét funkció ugyanarra a dologra vonatkozik, az r sugarú körre.

OOP esetén a feladatban megjelenő dolgot, fogalmat osztályként (class) kell definiálnunk. Az osztályban kell leírunk az adott dologra (esetünkben a körre) vonatkozó funkciókat, melyeket az OOP terminológia metódusoknak nevez.

Minden kör esetében ugyanúgy kell kiszámítanunk a kerületet és a területet, a különböző körök csak a sugaruk nagyságában térnek el egymástól. Bár ez sem feltétlenül igaz, hiszen léteznek azonos sugarú körök is.

Ahhoz, hogy egy kör „meg tudja mondani” a saját kerületének és területének a nagyságát, ismernie kell a saját sugarának nagyságát. A kör sugara a kör tulajdonsága (property). Körök esetében a sugár az egyetlen fontos tulajdonság, más síkidomok (pl. háromszög, téglalap, paralelogramma, deltoid vagy éppen ellipszis) esetében több tulajdonság is szükséges lehet ahhoz, hogy ki tudják számítani a saját kerületüket és területüket.

Készítsünk egy *Kör* nevű osztályt, amely a sugár ismeretében meghatározza az ő saját kerületét és területét! Nagyon fontos, hogy tisztán lássuk: az osztály nem egy konkrét dolog, hanem csak egy „tervrajz”, ami alapján majd létrehozhatunk olyan köröket, amelyek ismerik (meg tudják mondani másoknak) a saját területük és kerületük nagyságát. Egy olyan keret, ami alapján tetszőleges számú konkrét példányt, objektumot tudjunk létrehozni. Az osztályban csak azt írjuk le, hogy milyen tulajdonságai vannak egy körnek (egy körnek csak egy tulajdonsága van: a sugarának a nagysága), illetve hogy a sugár ismeretében hogyan kell kiszámítani a kerületet és a területet. Az osztályban nincs konkrét információ a sugár nagyságáról, ilyen konkrét információval csak az osztály alapján létrehozott objektumpéldányok fognak rendelkezni. Úgy is felfoghatjuk ezt, hogy minden kör objektum adattípusa a *Kör* osztály. (A jelölésmóddal követjük a 3.2.2. fejezetben leírt konvenciókat: az osztályok nevét nagy kezdőbetűvel írjuk, az objektumok, változók stb. neveit pedig kis kezdőbetűvel.)

Gondoljuk végig, hogy melyek azok az adatok, amelyeket megmondhat magáról egy kör objektum, és melyek azok, amelyeket nem feltétlenül kell a kívüllág számára nyilvánosságra hozni. A terület és a kerület mindenképpen nyilvános (publikus) kell hogy legyen, hiszen az az eredeti feladatunk, hogy a sugár ismeretében minden kör meg tudja mondani magáról ezt a két információt. A sugár (vagyis a kör tulajdonsága) már nem biztos, hogy ilyen információ. A publikusság ugyanis annak a veszélyével is járhat, hogy a kívüllágból valaki nemcsak látja ezt az információt, de esetleg meg is változtathatja. Biztos, hogy nem szerencsés dolog az, ha egy eredetileg 5 sugarú kör nagyságát valaki a kívüllágból egyszer csak meg tudja változtatni. Ezzel beavatkozik a kör belügyeibe, és ez gyakran bajt okoz.

A példa egy kicsit esetlen, hiszen milyen bajt is okozhatnánk azzal, hogy kívülállóként beleszólunk egy kör belügyeibe: ti. hogy mekkora is legyen az ő sugara. Az OOP másik fontos alapgondolata, hogy legyen lehetőségünk elrejteni a kívüllág szeme elől olyan dolgokat, amelyek nem tartoznak a kívüllágba. Ha egy nagyméretű projektet egyszerre sok programozó együttes munkával old meg,

akkor egymás között úgy osztják fel a feladatokat, hogy tisztázzák, milyen részekből fog állni a kész program, és az egyes részek hogyan kommunikálnak egymással. Minden részfeladat szolgáltat mások számára adatokat, melyeket megint más folyamatoktól kapott adatokból számít ki. Az fontos, hogy a kiszámított adatok minden körülmények között helyesek legyenek, és minden adathoz minden illetékes folyamat időben hozzá is juthasson, az viszont már nem tartozik a külvilág számára, hogy ezeket az adatokat hogyan számítja ki az adott részfolyamat. Lehet, hogy egy tömböt definiál, lehet, hogy valamilyen más adatszerkezetet használ a megoldáshoz. A cél az, hogy a kapott, bejövő adatokból helyes kimenetet állítson elő, de ehhez a külvilágnak nem kell (nem is szabad) tudnia, hogy milyen módszereket, eljárásokat, segédváltozókat stb. használ a folyamat. Azért nem szabad ezeket nyilvánosságra hozni, mert esetünkben a nyilvánosság azt is jelenti, hogy a külvilágból bárki módosíthatja is egy ilyen tömb vagy segédváltozó értékeit. Az OOP-ban ami publikus, azt mindenki láthatja és mindenki módosíthatja is.

Térjünk vissza az eredeti feladathoz. Definiáltuk már a Kör osztályt, de ahhoz, hogy legyenek köreink, azokat létre is kell hozni. Amikor az elvont tervrajz (osztály) alapján létrehozuk a konkrét példányt (objektum), példányosításról vagy konstrukcióról beszélünk.

A konstrukció során a konkrét egyedet gyakran bizonyos értékek alapján készítjük el. Az objektum létrehozásához egy speciális metódust, az ún. konstruktort használunk. A konstruktor az a metódus, amely a példányosítás során létrehozza a bizonyos tulajdonságokkal rendelkező objektumot.

Lássuk mindezt Java nyelven!

```
295 public class Kör {  
296     private double r;  
297     public Kör(double r_) { r = r_; }  
298     public double kerület() { return 2*r*3.14; }  
299     public double terület() { return r*r*3.14; }  
300 }
```

A fenti forráskód első sorában megadjuk a létrehozandó osztály nevét. Ezt követően deklarálunk egy privát (`private`) változót `r` névvel, ez lesz az osztályunk egyetlen tulajdonsága. Azt nem szeretnénk, ha ezt a változót a külvilágból is el lehetne érni, és a külvilág (egy másik osztály) is meg tudná változtatni, a sugár nagysága minden kör saját magánügye.

A következő sorban definiáljuk az osztály konstruktorát. Ez – ahogyan írtuk – egy speciális metódus, ez hozza létre az objektumokat az osztályból. A gyakorlatban ez úgy történik, hogy amikor objektumot hozunk létre, akkor automatikusan lefut az osztályban a konstruktor, és végrehajtja a létrehozandó objek-

tum alapbeállításait. A konstruktor neve mindig megegyezik az osztály nevével, és mögötte zárójelben felsoroljuk azokat az értékeket, amelyekkel az objektumot létre kívánjuk hozni.

Esetünkben ez egy `double` típusú, `r_` nevű paraméter. Ahogyan látható, a név nagyon hasonlít az osztály egyetlen tulajdonságának nevére, az `r`-re. Eleinte valószínűleg gondot okozhat a kód olvasásakor ez a hasonlóság, mégis célszerű követni ezt a jelölésmódot. Az `r_` értékét a külvilág szolgáltatja, ez lesz a kör objektumaink esetében a bejövő adat. A *Kör* osztály konstruktora most csak annyit tesz, hogy ezt a kívülről kapott értéket bemásolja az `r_` nevű változóba, ami az osztály privát tulajdonsága, vagyis a külvilág szeme elől el van rejtve. Valószínűleg most még mindenki számára titokzatos dolog ez: vajon mi szükség van erre? A válasz most annyi, hogy a külvilágtól származó adatot az osztály elrejt egy a külvilág számára láthatatlan tulajdonságba, és a műveleteket ezen végezzük majd el. A *Kör* osztály esetében valószínűleg nem járna végzetes következményekkel, ha nem tennénk ezt, mégis próbáljuk megszokni ezt a technikát: az osztályon belül ne végezzünk műveleteket olyan adatokkal, amelyeket a külvilág is módosíthat.

A következő két sorban definiáljuk a *Kör* osztály két metódusát. Ezek kiszámítják a sugár ismeretében az adott kör kerületét és területét, de ehhez a „titkos” változó értékét használják fel. A kiszámított értéket természetesen nyilvánosságra kell hozniuk, ezért ők publikus előtagot (`public`) kaptak. A `public` kulcsszó és a metódusnevek között szerepel a `double` típusnév, ez azt jelenti, hogy a két metódus egy-egy `double` típusú értéket ad vissza a külvilág számára. Ahogyan korábban leírtuk, a metódusok lényegében függvények, amelyek bizonyos bemenő adatok ismeretében egy új adatot számítanak ki. A kiszámított adatot nevezzük a függvény (metódus) visszatérési értékének, ennek típusát kell megadni a metódus neve előtt.

Figyeljük meg, hogy az osztály konstruktora neve előtt nem szerepel semmilyen típusnév:

```
301 public Kör(double r_) {...}
```

A konstruktor nem ad vissza semmilyen visszatérési értéket, mert a konstruktor futásának eredménye maga az objektum, ami létrejön a példányosítás során.

Más metódusok esetében is előfordulhat, hogy az adott metódus nem szolgáltat semmilyen visszatérési értéket. Csak a konstruktorok esetében nem kell semmilyen típusnevet leírunk a metódus neve elé, minden más metódus neve előtt jeleznünk kell a visszaadott (visszaadandó) adattípust. Ha egy metódus nem szolgáltat semmilyen visszatérési értéket (és az a metódus nem maga

a konstruktor), akkor a `void` kulcsszót kell használnunk. A `void` azt jelenti, hogy a metódusnak nincs visszatérési értéke.

A *Kör* osztály két metódusa, a *kerület* és a *terület* nevű metódusok egy-egy `double` típusú adatot szolgáltatnak a külvilág számára. A metódusok törzsében (a kapcsos zárójelek között) kiszámítjuk és visszaadjuk a visszaadandó adatokat. A `return` kulcsszó mögé azt az adatot kell írni, amelyet a metódus visszaad. Ez – ahogyan a példában is látszik – lehet egy kiszámítandó kifejezés is.

6.2.2 Még egyszer a main metódusról

Elkészült az osztály, de ez önmagában még nem használható. Az elvont definíció (osztály) alapján konkrét egyed(ek)et (objektum) kell készítenünk, illetve el kell készítenünk a főprogramot, fő metódust (`main`) is, hiszen ez a program belépési pontja, innen kezdődik a futás.

Hozzunk létre egy új projektet a NetBeans segítségével. A projekt címe legyen az, hogy *Kör*. A projekt típusa a szokásos Java Application legyen! A NetBeans automatikusan elkészíti ennek a projektnek is a vázát, benne a *Kör* nevű osztállyal és a `main` metódus fejével. A kommenteket kitörölhetjük a forráskódból. Gépeljük be a teljes program szövegét!

```

1 package kör;
2 public class Kör {
3     private double r;
4     public Kör(double r_) { r = r_; }
5     public double kerület() { return 2*r*3.14; }
6     public double terület() { return r*r*3.14; }
7
8     public static void main(String[] args) {
9         Kör kiskör = new Kör(10);
10        Kör nagykör = new Kör(800);
11        System.out.println("A kiskör sugara 10 egység.");
12        System.out.println("Kerülete:" + kiskör.kerület());
13        System.out.println("Területe:" + kiskör.terület());
14        System.out.println("A nagykör sugara 800 egység.");
15        System.out.println("Kerülete:" + nagykör.kerület());
16        System.out.println("Területe:" + nagykör.terület());
17    }
18 }
19

```

Output - Kör (run) x

```

run:
A kiskör sugara 10 egység.
Kerülete: 62.800000000000004
Területe: 314.0
A nagykör sugara 800 egység.
Kerülete: 5024.0
Területe: 2009600.0
BUILD SUCCESSFUL (total time: 0 seconds)

```

40. ábra: Első objektumorientált programunk

A kód eleje már ismerős: a *Kör* osztály privát tulajdonsága, konstruktora és két metódusa ugyanaz, mint a lecke előző részében. A fő metódus (`main`) neve és paramétere is a már jól megszokott, de egy pillanatra álljunk meg itt. A `main` metódus publikus. Muszáj annak lennie, hogy a kívülág (jelen esetben ez most maga az operációs rendszer, ami alatt futtatjuk a programot) láthassa, használhassa. Ha a `main` privát lenne, akkor nem lehetne elindítani a programot, hiszen a program belépési pontja a `main` metódus, ott kezdődik el a program futtatása. A `main` metódusnak tehát mindenképpen publikusnak kell lennie.

A `static` kulcsszóról eddig nem esett szó. Amikor egy tulajdonságot vagy metódust `static` kulcsszóval jelölünk meg, azzal azt jelezzük a Java felé, hogy ez a tulajdonság nem az objektumokhoz fog kötődni, hanem az osztályhoz. A *Kör* osztály alapján létrehozott kör objektumok (*kiskör*, *nagykör*) sugara eltérő volt. A körök sugara az objektumokhoz kötődő tulajdonság. Az objektumokat úgy hoztuk létre, hogy megmondtuk a konstruktornak, hogy mekkora legyen a létrehozandó *Kör* típusú objektum sugara, és ezt a konstruktor bemásolta a létrehozott kör privát, sugár tulajdonságába. A köröknek tehát csak onnantól létezett sugár (*r*) tulajdonságuk, hogy létrehoztuk magukat a köröket.

Más szavakkal úgy is fogalmazhatunk, hogy a tulajdonság csak a példányosítás elvégzése után vált használhatóvá. A `main` metódus esetében ez végzetes lenne, hiszen a `main` a program belépési pontja, onnan indul a program futása. Ha ezt a metódust csak egy objektumon keresztül lehetne elérni, vajon mi tudná ezt az objektumot létrehozni? A probléma tehát az, hogy ha a `main` egy objektumhoz kötődő metódus lenne, akkor nem lehetne futtatni, hiszen a `main` a program futása során a legelső dolog, ami lefut. Hogy ne kelljen példányosítani a `main` metódust tartalmazó osztályt ahhoz, hogy a `main` futtatható legyen, a `static` kulcsszóval jelezzük, hogy a `main` metódus az osztályhoz kapcsolódik és nem egy, az osztályból létrehozott objektumhoz.

A `static` kulcsszó a `main` metódus esetében speciális szereppel bír, de nemcsak itt használható. Minden esetben, amikor azt szeretnénk, hogy egy tulajdonság az osztályhoz kötődjön és nem a belőle létrehozott objektumhoz, statikussá kell tennünk. Nagyon gyakran statikus az a tulajdonság az osztályon belül, amely számon tartja, hogy hány példány készült már az adott osztály alapján. Az osztály konstruktora növeli eggyel ezt a statikus tulajdonságot, így minden esetben, amikor létrehozunk egy új példányt, ez a szám eggyel nő. Így az osztályon keresztül minden objektum le tudja azt ellenőrizni, hogy a program futása során hány objektum jött már létre ebből az osztályból.

A `void` kulcsszóról már esett szó, ezt most csak annyival egészítjük ki, hogy a Javában a `main` metódusnak kötelező `void`-nak lennie.

6.2.3 Objektum létrehozása

A main metódus első két sorában létrehozunk egy *kiskör* és egy *nagykör* nevű objektumot a *Kör* osztály alapján. A példányosításhoz a *new* operátort kell használni. Egy objektum létrehozása tehát a következő szintaxis alapján történik:

```
302 Osztály objektumnév = new Osztály(paraméterek);
```

A *new* kulcsszó után az osztály nevét írjuk, de mint tudjuk, az megegyezik az osztály konstruktorának nevével, valójában itt tehát a konstruktorra való hivatkozás történik. Ha a programozó nem készít konstruktort az osztályhoz, akkor a Java létrehoz egyet automatikusan, de ez természetesen nem jelenik meg a forráskódban. Konstruktornak tehát minden osztályban lennie kell, de ha a programozó nem készít ilyet, akkor az létrehoz egy – lényegében majdnem üres – konstruktort.

6.2.4 Objektumok törlése

Amikor egy objektumra már nincs szükség a továbbiakban, akkor mindenképpen érdemes felszabadítani az általa lefoglalt területet a memóriában, hiszen a memória az egyik legdrágább erőforrás (nem feltétlenül anyagi értelemben), és nagyon szűkösen áll rendelkezésünkre. Más OOP nyelvekben az objektumok megszüntetésére létezik egy másik, speciális metódus, amelyet destruktornak neveznek. Nagyon nehéz kérdés azt megállapítani, hogy mikor jelenthetjük ki az biztosan, hogy egy objektumra már nincs szükségünk a továbbiakban, éppen ezért a destruktor használatának mindig van némi kockázata: mi van akkor, ha egy olyan objektum által lefoglalt memóriaterületet szabadítunk fel, amelyikre egy másik objektum a későbbiekben még hivatkozni akar.

A Javában ez a kérdés nem kérdés, a Java ugyanis nem ismeri a destruktor fogalmát, az allokált memóriaterület felszabadítása automatikus. Az ún. garbage collection (szemétgyűjtögetés) lényege az, hogy a Java-értelmező tartalmaz egy belső memóriaellenőrző eljárást, amely az értelmezővel párhuzamosan fut, és időről időre felszabadítja azokat a memóriaterületeket, amelyekre már semmi sem hivatkozik.

6.2.5 Csomagok

Programunk alkotóelemeit (pl. az osztályokat) nagyobb egységekbe, csomagokba (*package*) szervezhetjük. A Java nyelv környezetét jelentő, előre elkészített elemek is a funkcióik alapján csomagokba vannak csoportosítva.

A csomagok hierarchikus szerkezetűek, ami azt jelenti, hogy egy csomagon belül más csomagok is elhelyezkedhetnek. A Java a csomagokon belüli alcsomagok, illetve elemek megadására a `.` (pont) jelölést alkalmazza. A `java.awt.Label` jelölés a *java* csomagon belüli *awt* alcsomag *Label* osztályára hivatkozik. A konvenciók szerint a csomagok elnevezését kisbetűvel kezdjük.

A Java szabványos környezetét jelentő alapcsomagokat a *java* csomagon belül helyezték el. A legfontosabb csomagok a funkcióik alapján a következők:

4. Néhány fontos Java-csomag

Csomagnév	Funkció
<code>java.applet</code>	Appletek készítése és a böngészővel történő kommunikáció
<code>java.awt</code>	Felhasználói felület készítéséhez, képek megjelenítéséhez és grafika rajzolásához
<code>java.io</code>	Állományok kezelése
<code>java.lang</code>	A Java nyelvhez szükséges alapvető osztályok
<code>java.math</code>	Tetszőleges pontosságú egész- és decimális aritmetika
<code>java.net</code>	Hálózati kommunikáció
<code>java.sql</code>	Adatbázisok kezelése
<code>java.text</code>	Szöveg-, dátum-, és számformák kezelése
<code>java.util</code>	Segédosztályok

A *javax* csomagban alapvető jelentőségű kiterjesztéseket (Java extension) helyeznek el.

Bármely osztály forrásszövegében hivatkozhatunk bármely csomag bármely, korábban már elkészített osztályára, ha a pontjelöléssel tartalmazás szerint felsoroljuk a legfelső szintű csomagot, az alcsomagokat és végül az osztály elnevezését. Például `Java.applet.Applet`.

Azonban nagyon fárasztó lenne, ha a programban ugyanazon csomag egy vagy több osztályára többször is hivatkoznánk. Az `import` utasítással közvetlenül, a csomagok megadása nélkül is elérhetővé tehetjük az osztályokat.

Az `import` utasításnak alapvetően két fajtája van. A következő sor egyetlen osztály, a `java.applet` csomag `Applet` osztályának közvetlen elérését teszi lehetővé:

```
303 import java.applet.Applet;
```

Ha egy csomagból (pl. a `java.awt` csomagból) több osztályt (`Label`, `TextField` stb.) is fel szeretnénk használni, akkor célszerű a másik formát alkalmazni:

```
304 import java.awt.*;
```

Ebben az esetben a csomag összes osztályára közvetlenül, a csomag megadása nélkül is hivatkozhatunk.

Az `import` utasítás mindössze egy rövidítési lehetőséget jelent, így egyrészt egyetlen `import` utasítás nélkül is írhatunk programot, másrészt a pontjelöléssel az osztály csomagját az egyértelműség miatt akkor is megadhatjuk, ha több importált csomagban ugyanolyan nevű osztályunk van.

A Java nyelvhez szorosan kapcsolódó `java.lang` csomag különleges, mivel a nyelv az abban elhelyezkedő osztályokat közvetlenül, külön `import` nélkül is elérhetővé teszi.

Nagyobb, több osztály készítését igénylő programjainkat nekünk is célszerű csomagokba szervezni. A Java forrásprogram első utasításaként a `package` utasítással megadhatjuk az osztály csomagját, a `package` kulcsszó után a csomag nevét írjuk.

Fontos, hogy egy osztálynak soha nem kell importálnia a saját csomagját, illetve az abban elhelyezkedő többi osztályt (ahogy a `java.lang` esetén) közvetlenül is elérheti.

Egy Java forrásállomány a következő alapszerkezetű lehet: legelső utasításként egyetlen, az osztály csomagját meghatározó `package` utasítás állhat. A forrásszövegben ennek és a `java.lang` csomagnak az osztályaira közvetlenül is hivatkozhatunk.

Ezután `import` utasításokat adhatunk meg, melyekkel a közvetlenül hivatkozni kívánt osztályokat vagy azok csomagjait soroljuk fel. A forrásállomány ezután egy vagy több osztály (vagy interfész, lásd később) definícióját tartalmazza, azonban csak egyetlen osztály lehet publikus, és a nevének meg kell egyeznie a forrásállomány nevével.

6.3 ÖSSZEFOGLALÁS, KÉRDÉSEK

6.3.1 Összefoglalás

Ebben a leckében áttekintettük és kiegészítettük az OOP-val kapcsolatos legfontosabb tudnivalókat. A programozás önmagában is összetett, absztrakt tudomány, az OOP megértése pedig különösen nehéz feladat lehet annak, aki most találkozik vele először. Mégis, ennek a fejezetnek a megértése alapvető fontosságú, mind az előző fejezetek szempontjából, mind pedig a most következők megértéséhez.

6.3.2 Önellenőrző kérdések

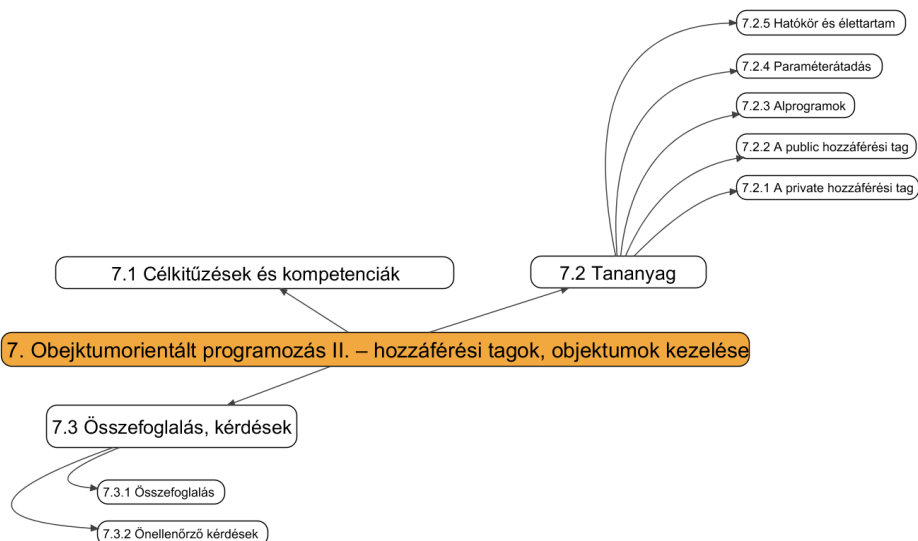
1. Melyek az OOP legfontosabb alapelvei?
2. Mi a különbség osztály és objektum között?
3. Mire való a konstruktor? Hogyan kell definiálnunk egy osztály konstruktorát?
4. Mire való a `void` kulcsszó?
5. Mire való a `static` kulcsszó?
6. Hogyan lehet létrehozni egy objektumot egy osztályból?
7. Hogyan kell felszabadítani egy már szükségtelen objektum által lefoglalt memóriaterületet?
8. Mire való a csomagok?
9. Mire való az `import` utasítás?

7. LECKE: OBEJKTUMORIENTÁLT PROGRAMOZÁS II. – HOZZÁFÉRÉSI TAGOK, OBJEKTUMOK KEZELÉSE

7.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

Folytatjuk az előző fejezetben megkezdett témakör tárgyalását. Az OOP egyik alapelvéről, az adatrejtésről az előző leckében már szoltunk. A `public` és a `private` tagokat is használtuk már korábbi példaprogramokban, ennek a leckének az elején ezekről az ún. hozzáférési tagokról ejtünk néhány szót. A lecke második felében az alprogramok típusait vizsgáljuk meg közelebbről, szó esik a változók hatóköréről és élettartamáról, valamint áttekintjük a paraméterátadás tulajdonságait. A lecke feldolgozásához szükséges kompetenciák: logikai képesség, induktív gondolkodás képessége, deduktív gondolkodás képessége, absztrakt (elméleti) gondolkodás, áttekintő képesség, rendszerező képesség és figyelem-összpontosítás.

7.2 TANANYAG



7.2.1 A private hozzáférési tag

Tekintsük az alábbi példaprogramot:

```
305 class Pénz {
306     private int euro, cent;
307     public Pénz(int euro_, int cent_) {
308         euro = euro_;
309         cent = cent_;
310     }
311     private int centÉrték() {
312         return 100 * euro + cent;
313     }
314 }
315 class Átváltó {
316     public static void main (String[] args) {
317         Pénz pénz = new Pénz(5,20);
318         System.out.println("Érték (Euró.cent): „ +
319             pénz.euro + „.” + pénz.cent );
320         System.out.println("Érték (cent):” +
321             pénz.centÉrték());
322     }
```

Ha bemásoljuk a kódot a NetBeans IDE-be (létre kell hozni egy új projektet *Átváltó* néven, majd a fenti forráskódot be kell gépelni), a program három hibát fog jelezni, és a fordítás el sem kezdhető.

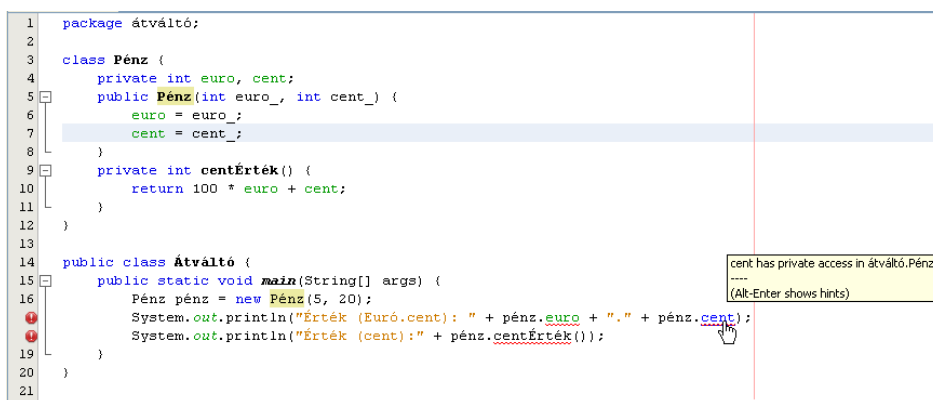
A fenti példaprogramban két osztályt definiáltunk, a *Pénz* és az *Átváltó* nevű osztályokat. Az *Átváltó* osztályban deklaráltunk két `private` előtagú változót, vagyis ezek a tulajdonságok az osztály rejtett tulajdonságai lesznek. Az osztály konstruktora átveszi a paraméterként megadott Euró és cent értékeket, majd ezeket bemásolja a két rejtett tulajdonságba. A *centÉrték()* metódus, mely szintén ebben az osztályban van deklarálva, átszámítja az Euróban és centben megadott értéket centekbe.

Az *Átváltó* osztály tartalmazza a `main` metódust. Ebben először létrehozunk egy pénz nevű objektumot a *Pénz* osztályból, a konstruktornak az 5 Euró 20 cent értékeket adjuk át. (Ezeket a *Pénz* osztályban a konstruktor bele is másolja az *euro* és a *cent* privát tulajdonságokba.)

Itt következik a hiba. A `main` metódus következő két sorában megpróbáljuk kiolvasni a *pénz* objektum két privát tulajdonságának értékét, de a Java ezt

nem engedi. Az *euro* és a *cent* adattagok *private* hozzáférésűek, azaz csakis abban az osztályban lehet őket elérni (látni), amelyben definiáltuk őket. Küldő osztály nem férhet hozzájuk.

Ugyanezért nem férhet hozzá az *Átváltó* osztályban leírt *main* metódus a *Pénz* osztályban leírt *centÉrték()* metódushoz.



```

1 package átváltó;
2
3 class Pénz {
4     private int euro, cent;
5     public Pénz(int euro_, int cent_) {
6         euro = euro_;
7         cent = cent_;
8     }
9     private int centÉrték() {
10         return 100 * euro + cent;
11     }
12 }
13
14 public class Átváltó {
15     public static void main(String[] args) {
16         Pénz pénz = new Pénz(5, 20);
17         System.out.println("Érték (Euró.cent): " + pénz.euro + "." + pénz.cent);
18         System.out.println("Érték (cent): " + pénz.centÉrték());
19     }
20 }
21

```

41. ábra: A *Pénz* osztály elrejtja a külvilág elől az *euro* és *cent* adattagokat, illetve a *centÉrték()* metódust

7.2.2 A public hozzáférési tag

Első ránézésre kínálja magát a megoldás: változtassuk meg a *Pénz* osztály *private* hozzáférésű elemeit (tulajdonságait és metódusát) *public* hozzáférésűvé.

Valóban, első ránézésre ez jó megoldásnak tűnik, a NetBeans sem jelez már hibát, le is futtathatjuk tehát a programot. Vegyük észre, hogy a *main* metódust is kiegészítettük még egy sorral. Miután létrehoztuk a *pénz* nevű objektumot 5 Euró 20 cent értékkel, a következő sorban megváltoztatjuk az *euro* adattag értékét –1000-re. A *public* hozzáférési tag megengedi, hogy egy osztály tulajdonságát egy másik osztályból megváltoztassuk. Nagyon elővigyázatlan lenne az a bank, amelyik ilyen szoftvert használna!

```

1  package átváltó;
2
3  class Pénz {
4      public int euro, cent;
5      public Pénz(int euro_, int cent_) {
6          euro = euro_;
7          cent = cent_;
8      }
9      public int centÉrték() {
10         return 100 * euro + cent;
11     }
12 }
13
14 public class Átváltó {
15     public static void main(String[] args) {
16         Pénz pénz = new Pénz(5, 20);
17         pénz.euro = -1000;
18         System.out.println("Érték (Euró.cent): " + pénz.euro + "." + pénz.cent);
19         System.out.println("Érték (cent):" + pénz.centÉrték());
20     }
21 }
22

```

Output - Átváltó (run) x

```

run:
Érték (Euró.cent): -1000.20
Érték (cent):-99980
BUILD SUCCESSFUL (total time: 0 seconds)

```

42. ábra: A külvilág (egy külső osztály) módosította egy tulajdonság értékét

A `public` taggal ellátott tulajdonságok és metódusok tehát nyilvánosak, a `private` tagúak rejtettek. Létezik egy ún. félnyilvános láthatóság is: ez az alapértelmezett, ha nem írunk mást helyette. A félnyilvános láthatóság az azonos csomagban (`package`) definiált osztályok számára is enged hozzáférést. A Javában a láthatóság osztályszintű, vagyis azonos osztályba tartozó objektumok elérik egymás rejtett tagjait is.

A negyedik láthatósági típus a `protected`. Erről a következő fejezetben beszélünk részletesen, mivel ennek megértéséhez először az öröklés, öröklődés fogalmát kell majd tisztáznunk.

7.2.3 Alprogramok

Minden magas szintű (imperatív) programozási nyelv lehetőséget biztosít arra, hogy egy programból a többször végrehajtandó, ismétlődő részeket kiemeljük, és névvel ellátva alprogramot hozunk létre belőle. Gondoljunk a következő példára: ismerjük 30 ember magasságát és testsúlyát, szeretnénk kiszámolni az átlagos magasságot és az átlagos testsúlyt. Ehhez kétszer kellene

ugyanazt az eljárást megírnunk, és a két programrészlet között lényegében csak annyi lenne a különbség, hogy mások lennének a bemenő adataik.

A többször előforduló részt elegendő csak egyszer leírni (definiálni), majd ehhez egy azonosítót (nevet) rendelve a későbbiekben sokszor tudunk rá hivatkozni. Amikor egy alprogramra a nevével hivatkozunk, az alprogram hívásáról beszélünk.

Az előző példánál maradva készíthetnénk egy átlagszámító alprogramot, amelynek paraméterként megadhatjuk a bemenő adatokat (ezek első esetben a magasságok, második esetben a testsúlyok lehetnek). Az alprogram kiszámítja a kapott adatok átlagát, majd visszatérési értéként visszaadja ezt a számított értéket a hívó programrésznek. Egy alprogram paramétereit az alprogram argumentumainak is nevezzük.

- ❑ Emlékezzünk vissza az adata bevételénél olvasottakra. Egy Java-program parancssorból indítva argumentumokat is kaphat, ezek fognak bekerülni a `main` metódus paramétereként megadott `String` típusú tömbbe. A Java-program indítása felfogható a `main` nevű alprogram (ami a főprogram) hívásának.

Az átlagszámító alprogram a bejövő adatokból egy kimenő adatot (visszatérési értéket) számít ki. Az ilyen alprogramokat nevezzük függvényeknek.

```
323 public static double átlag(int[] adatok) {
324     int összeg = 0;
325     int i;
326     for (i = 0; i < adatok.length; ++i) {
327         összeg += adatok[i];
328     }
329     return összeg / (double) adatok.length;
330 }
```

Az alprogram (függvény) hívása:

```
331 public static void main(String[] args) {
332     int[] magasságok = {167, 172, 158, 192, 166};
333     int[] testsúlyok = {62, 69, 47, 78, 65};
334     double m = átlag(magasságok); // 1. hívás
335     double s = átlag(testsúlyok); // 2. hívás
336     System.out.println("Átlagmagasság: „ + m);
337     System.out.println("Átlagos testsúly: „ + s);
338 }
```

```

1 package átlag;
2
3 public class Átlag {
4     public static double átlag(int[] adatok) {
5         int összeg = 0;
6         int i;
7         for (i = 0; i < adatok.length; ++i) {
8             összeg += adatok[i];
9         }
10        return összeg / (double) adatok.length;
11    }
12
13    public static void main(String[] args) {
14        int[] magasságok = {167, 172, 158, 192, 166};
15        int[] testsúlyok = {62, 69, 47, 78, 65};
16        double m = átlag(magasságok); // 1. hívás
17        double s = átlag(testsúlyok); // 2. hívás
18        System.out.println("Átlagmagasság: " + m);
19        System.out.println("Átlagos testsúly: " + s);
20    }
21 }
22

```

Output - Átlag (run) x

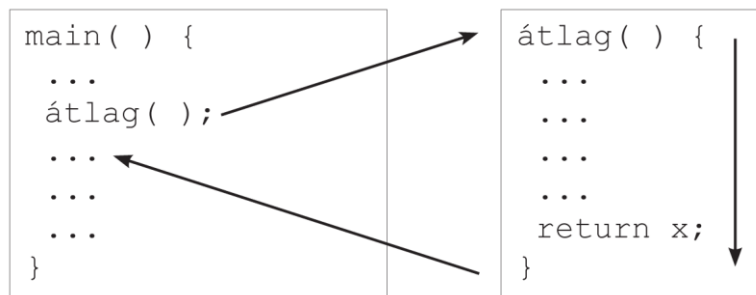
```

run:
Átlagmagasság: 171.0
Átlagos testsúly: 64.2
BUILD SUCCESSFUL (total time: 1 second)

```

43. ábra: A program és hívása

A főprogram kétszer számított ki átlagokat, ehhez kétszer hívtuk meg az *átlag* nevű alprogramot, amely jelen esetben egy függvény, hiszen van visszatérési értéke.



44. ábra: Hívás és visszatérés

A függvényünk visszatérési értéke egy *double* érték, ezt a függvénydeklaráció elején jelezzük:

```
339 double átlag(...) {...}
```

A kiszámított visszatérési értéket az alprogram törzsének a végén a `return` kulcsszó után adjuk meg.

```
340 return összeg / double (adatok.length);
```

Az `összeg` változó a paraméterként megkapott adatok összegét tartalmazza, az `adatok.length` pedig az átadott adatok számát mondja meg. A `double` kulcsszó azért kell az `adatok.length` elé, mert mint tudjuk, Javában ha egész számot osztunk egész számmal, akkor a kapott eredmény is egész lesz. (A `/` operátor egészek között az egészosztást jelenti.)

A `static` módosító most azért szükséges a függvénydeklaráció elején, mert a `main` metódus is statikus, és statikus környezetből a Java nem engedi nem statikus metódusok (alprogramok) hívását. Másrészt látható, hogy a fenti példában úgy futtattuk le az *átlag* metódust, hogy nem készítettünk objektumpéldányt az *Átlag* osztályból. A fenti példában hiányzik az osztály konstruktorának megadása is. Ezzel a példaprogrammal most nem az OOP eszközeit akartuk bemutatni, hanem azt, hogy hogyan lehet a kódot áttekinthetőbbé tenni azzal, ha az ismétlődő részeket kiemeljük alprogramként. Ugyanakkor fontos tisztán látni azt is, hogy az osztályok metódusai szintén alprogramok. De míg az alprogram, mint eszköz már a strukturált programozásban is megjelenik (jóval az OOP megjelenése előtt), az osztályok és objektumok csak az OOP megjelenésével váltak a programozók számára elérhetővé. A fenti példában lényegében OOP környezetben készült egy nem OOP program.

Térjünk vissza az alprogramokhoz. Az *átlag* nevű alprogram egy függvény, hiszen van visszatérési értéke. Vannak olyan alprogramok is, amelyek nem adnak visszatérési értéket. A Javában minden alprogram elé ki kell írunk, hogy milyen visszatérési értéket szolgáltat. Csak az osztályok konstruktorainak neve elé nem írunk semmilyen információt, mivel a konstruktorok által szolgáltatott „eredmény” maga a létrejövő objektum. Minden más metódus elé ki kell írunk, milyen típusú visszatérési értéket adnak. Ha egy alprogram nem szolgáltat semmilyen visszatérési értéket, akkor ezt a `void` kulcsszóval jelöljük. A `void` jelenti azt, hogy az ilyen fajta alprogram nem ad visszatérési értéket. Azokat az alprogramokat, amelyek nem adnak visszatérési értéket, eljárásnak is szoktuk nevezni. Más programozási nyelvekben (pl. Pascal) külön kulcsszó létezik a függvény és az eljárás definiálására (ezek a `function` és `procedure` kulcsszavak). Javában nincs külön kulcsszó a függvények és az eljárások definiálására, eljárás az a függvény, amelynek `void` a visszatérési értéke.

7.2.4 Paraméterátadás

Amikor definiálunk egy alprogramot – akár függvényt, akár eljárást –, az alprogram neve mellé kerek zárójelek között megadjuk az alprogram bemenő paramétereinek típusát és nevét. Tekintsük a következő példát:

```
341 public static int minimum(int a, b) {  
342     return a < b ? a : b;  
343 }
```

Ennek a most deklarált *minimum* nevű függvénynek két egész szám, *a* és *b* a bemenő paraméterei. A függvény *a*-t adja vissza eredményül, ha két szám közül *a* a kisebbik, különben *b*-t. (A *b* értéke lesz a visszaadott érték, ha *b* a kisebb vagy a két szám egyenlő.)

Azt mondjuk, hogy a *minimum* függvénynek a formális paraméterlistájában két érték szerepel, az egész típusú *a* és *b*.

Híváskor a függvény neve mögött szintén kerek zárójelek között kell megadnunk azokat a konkrét értékeket, amelyekre szeretnénk meghívni az alprogramot.

```
344 {  
345 ...  
346 int k = minimum(9, 5);  
347 System.out.println("9 és 5 közül a kisebb: „ +  
    k);  
348 ...  
349 }
```

A híváskor megadott konkrét értékeket nevezzük aktuális paraméterlistának. A fenti példában ez a 9 és az 5:

```
350 k = minimum(9, 5);
```

Az ún. paraméterátadás során a formális paraméterlista elemei sorban megkapják az aktuális paraméterlista értékeit. Ez azt jelenti, hogy a fenti példában híváskor az a 9-es érték bekerül a *minimum* alprogramban deklarált a változóba, az 5-ös érték pedig a *b* változóba.

Az *a* és *b* változó a *minimum* függvény lokális változói lesznek. Ez azt jelenti, hogy úgy viselkednek, mintha ezeket szokásos változóként deklaráltuk volna az alprogram törzsében. Amint az alprogram futása befejeződik, az *a* és *b* változók által lefoglalt memóriaterület felszabadul, tehát az *a* és *b* változók gyakorlatilag megszűnnek.

Ha visszatekintünk az átlag nevű alprogramra, ott azt láthatjuk, hogy híváskor nem konkrét (konstans) értékeket adtunk át, hanem egy tömböt. Ilyenkor azt történt, hogy az átadott tömb(változó) értékei bemásolásra kerültek az átlag függvény fejében megadott formális paraméterlistában szereplő tömbváltozóba. Az első híváskor a *magasságok* nevű tömb adatai másolódtak be az *adatok* nevű tömbbe, a második híváskor pedig a *testsúlyok* nevű tömb adatai másolódtak be az *adatok* nevű tömbbe. (A példában mindkét tömbnek 5-5 eleme van, de ennek nincs jelentősége, a tömbök különböző elemszámúak is lehetnek volna.)

A Javában ún. érték szerinti paraméterátadás történik. Amikor az aktuális paraméterlistában egy változó neve szerepel, ennek az értéke bemásolódik a formális paraméterlistában megadott megfelelő változóba. Így elkerülhető az, hogy a hívott alprogram módosítsa a hívó programrészben deklarált változó értékét.

```
1 package paraméterátadáspróba;
2
3 public class ParaméterátadásPróba {
4
5     public static void alprogram(int a) {
6         a++;
7         System.out.println("Az a változó értéke az alprogramban: " + a);
8     }
9
10    public static void main(String[] args) {
11        int a = 10;
12        alprogram(a);
13        System.out.println("Az a változó értéke a főprogramban: " + a);
14    }
15 }
16
```

Output - ParaméterátadásPróba (run) x

run:
Az a változó értéke az alprogramban: 11
Az a változó értéke a főprogramban: 10
BUILD SUCCESSFUL (total time: 2 seconds)

45. ábra: Érték szerinti paraméterátadás

Figyeljük meg, hogy a főprogramban (`main`) deklarált változó neve megegyezik az alprogram formális paraméterlistájában szereplő változó nevével: mindkét változó neve *a*. A főprogramban a változó megkapja a 10-es kezdőértéket, majd meghívjuk az alprogramot. Az alprogram átveszi ezt az értéket, megnöveli eggyel, majd az így kiszámított értéket kiírja a képernyőre. (Az alprogram `void` típusú, vagyis nem szerepel benne a `return` kulcsszó, mivel nincs visszatérési értéke.)

Miután az alprogram futása véget ért, a vezérlést visszakapja a főprogram, melyben a végrehajtás az alprogram hívása utáni utasításon folytatódik. A főprogram itt kiírja az *a* változó értékét. Vegyük észre, hogy hiába módosítottuk az alprogramban az *a* változó értékét, az a főprogramban létrehozott *a* változóra nincs hatással, ugyanis ez a két változó két teljesen különböző változó, csak azonos a nevük. A magyarázat az, hogy az alprogramokban létrehozott lokális változók csak az alprogramok területén rendelkeznek címkomponenssel. A főprogram *a* nevű változója és az alprogram *a* nevű változója külön-külön memóriacímmel rendelkezik. Amikor az alprogram futása véget ér, a benne deklarált lokális változók címkomponense megszűnik: a futtató rendszer felszabadítja a lefoglalt memóriaterületeket.

A fenti példa végrehajtása semmilyen gondot nem okoz a fordítóprogramnak, illetve a futtató rendszernek, de a programozót nagyon össze tudja zavarni. Kerüljük azt a szokást, hogy ugyanolyan nevű változókat használunk a hívó és hívott programrészekben, vagy jegyezzük meg jól, hogy két azonos nevű változó nem jelenti feltétlenül ugyanazt a tárbeli területet, amely a változó(k) értékét tárolja.

7.2.5 Hatókör és élettartam

Egy név hatóköre az a része a programnak, ahonnan a jelölt programelem (változó, alprogram stb.) az adott néven elérhető. Egy név feloldása annak megkeresését jelenti, hogy a név melyik programelemet azonosítja a program adott pontján. A következő leckében látni fogjuk, hogy lehetőségünk van arra, hogy azonos nevű alprogramokat (metódusokat) definiáljunk. A metódusok neve azonos lesz, de a paraméterlistájuk különböző. A Java megengedi a nevek túlterhelését, vagyis azt, hogy egy név többféle dolgot jelentsen. Ahhoz azonban, hogy az azonos nevű dolgokat meg lehessen különböztetni, valamiben el kell térniük, hogy a név feloldható legyen. Erre tehát a következő leckében visszatérünk.

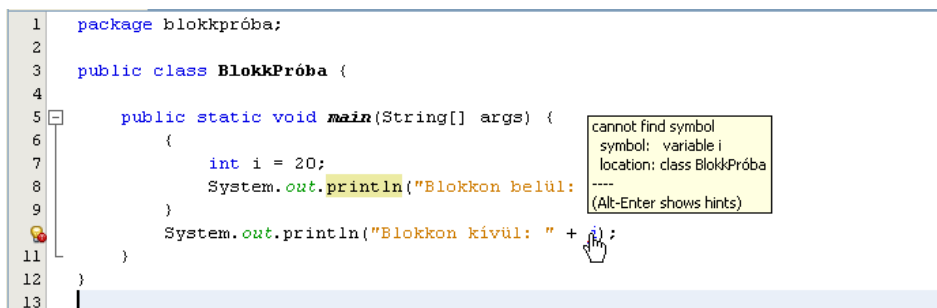
A programozási nyelvekben a változók (tárterületek) egyik alapvető tulajdonsága az élettartam. A változó élettartama a program végrehajtási idejének az a része, ameddig a tárterületet arra használjuk, hogy az adott objektum értékét hordozza.

Láttuk, hogy az alprogramokban létrehozott lokális változók élettartama az alprogram végéig tart. Amint az alprogram futása befejeződik, a lokális változók által lefoglalt memóriaterület felszabadul, és a változók értéke többé már nem lesz elérhető. Ez azonban nemcsak az alprogramokra érvényes. Tekintsük az alábbi példát:

```
351 public static void main(String[] args) {  
352     ...  
353     {  
354         int i = 20;  
355         System.out.println("Blokkon belül: " + i);  
356     }  
357     System.out.println("Blokkon kívül: " + i);  
358 }
```

Ha megpróbáljuk futtatni ezt a példát, hibaüzenetet kapunk (a NetBeans lefordítani sem engedi). A hibát az okozza, hogy a main metódus belsejében deklaráltunk egy *i* nevű változót, de az a blokkon kívül nem elérhető. Ennek az a magyarázata, hogy a változók hatóköre az a blokk, amelyben deklaráltuk őket, élettartamuk a blokk végén megszűnik.

(A Javában a blokkot kapcsos zárójelek határolják, a blokkon belül tetszőleges sorrendben követhetik egymást deklarációk és végrehajtható utasítások. Ld. 4.2.4. fejezet.)



46. ábra: A változók hatóköre az a blokk, amelyben deklaráltuk őket

Blokk egy osztály vagy egy metódus törzse, de egy ciklus magja is. Kezdő programozók gyakran elkövetik azt a hibát, hogy egy ciklusváltozóra olyan környezetben próbálnak hivatkozni, amelyben az (már) nem is látszik, pontosabban nem is létezik.

```
359 for (int i = 0; i < 100; ++i) {  
360     ...  
361 }  
362 int j = i + 1;
```

Ez a kódrészlet hibás, mivel az *i* változó lokális a ciklusra. Habár nem a ciklusmag blokkján belül deklaráltuk, a hatóköre mégis ez a blokk lesz. Amint a ciklusmag utoljára lefut, az *i* változó memóriaterülete felszabadul, a cikluson kívül ezért már nem lesz elérhető az utoljára felvett érték. (A programrészlet nem futásidejű hibát generál, hanem fordítási idejűt, mert az ilyen hibákat a fordítóprogram ki tudja szűrni.)

7.3 ÖSSZEFOGLALÁS, KÉRDÉSEK

7.3.1 Összefoglalás

Ebben a leckében az alprogramok tulajdonságaival ismerkedtünk meg behatóbban. Habár az alprogramok nemcsak OOP környezetben használható eszközök, mégis nagyon fontos a velük kapcsolatos tudnivalók ismerete, mivel az osztályok metódusai is alprogramok, így minden, ami az alprogramokra igaz nem OOP nyelvekben, igaz OOP nyelvekben is. A leckében áttekintettük az alprogramokkal kapcsolatosan a paraméterátadás jellemzőit is. A Java az érték szerinti paraméterátadást teszi lehetővé, más programozási nyelvekben ez nem feltétlenül van így. Mivel azonban a Java ezt a megoldást vallja, más paraméterátadási módokat be sem mutattunk.

A leckében szó esett a hatókör és az élettartam kérdéseiről is. A korábbi leckékben megvizsgáltuk az osztályok jellemzőinek és metódusainak hozzáférési módjait. Nagyon fontos, hogy különbséget tudjunk tenni a láthatóság, a hatókör és az élettartam kifejezések között. Ezek nagyon hasonló dolgot jelentenek, mégis alapvető különbség van közöttük.

7.3.2 Önellenőrző kérdések

1. Milyen hozzáférésű lehet egy osztályban definiált tulajdonság vagy metódus?
2. Milyen veszélyekkel jár a publikus hozzáférés engedélyezése?
3. Milyen típusú alprogramok léteznek?
4. Mi a különbség formális és aktuális paraméterlista között?
5. Mit jelent az érték szerinti paraméterátadás?
6. Határozza meg a hatókör és az élettartam kifejezések jelentését!

8. LECKE: OBJEKTUMORIENTÁLT PROGRAMOZÁS III. – ÖRÖKLÉS, POLIMORFIZMUS, ABSZTRAKT OSZTÁLYOK

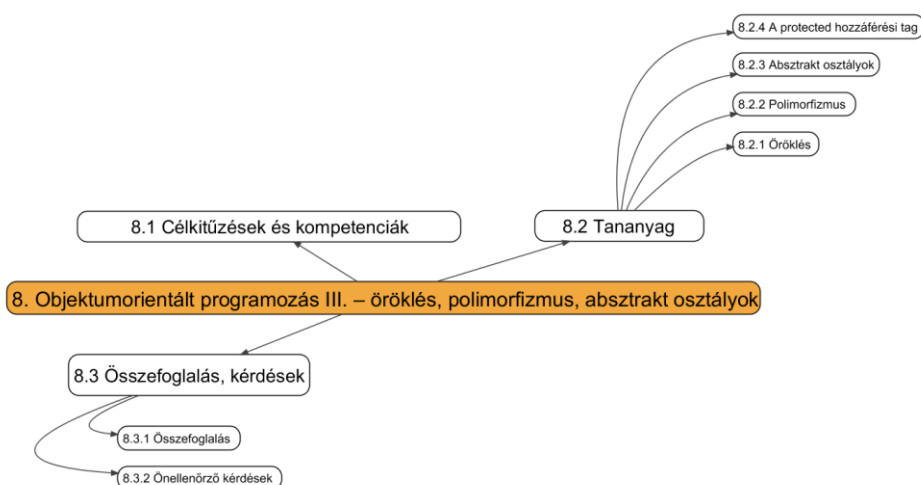
8.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

Ebben a rövid leckében megismerkedünk az OOP két nagyon fontos alap-pillérével, az örökléssel és a polimorfizmussal, más szóval sokoldalúsággal. A lecke megértéséhez szükséges az előző lecek teljes feldolgozása, hiszen itt olyan ismeretekről esik szó, amelyeket csak az OOP alapjainak biztos ismerete birtokában lehet megérteni.

Ez a lecke a legrövidebb, de ebben írjuk le az OOP talán legfontosabb definícióit. A definíciók konkrét alkalmazását a következő lecekben mutatjuk be részletesen, példákon keresztül.

A lecke feldolgozásához szükséges kompetenciák: logikai képesség, induktív, deduktív és absztrakt (elméleti) gondolkodás, áttekintő- és rendszerezőképeség, figyelem-összpontosítás.

8.2 TANANYAG



8.2.1 Öröklés

Az OOP nyelvek nagy előnye, hogy az újabb osztályok létrehozásakor az adatokhoz hozzákapcsolhatjuk az azokat kezelő metódusokat, ezt nevezzük egységbe zárásnak (encapsulation). Az OOP azonban azt is lehetővé teszi, hogy egy osztályt úgy definiáljunk, hogy az alapján újabb, speciális változatokat hozzassunk létre.

Az osztály specializálása alatt azt értjük, hogy felvehetünk újabb attribútumokat (tulajdonságokat), újabb műveleteket (metódusokat), illetve a már korábban definiált műveleteket átdefiniálhatjuk. A pontosított alosztály átveszi, örökli az alaposztály vagy őszosztály jellegzetességeit, így azokat a pontosított alosztályban már nem kell újra definiálnunk. A metódusok átdefiniálásával az alaposztály működését a specializált osztályokban megváltoztathatjuk.

Az öröklés lehetővé teszi a helyettesíthetőséget. Ez azt jelenti, hogy egy alosztályból létrehozott objektum bárhol felhasználható, ahol az őszosztály objektumára hivatkozunk. Így egy bizonyos változatra bizonyos helyeken általános módon is hivatkozhatunk, például egy metódust megadhatunk egy általános típus alapján, amelyet ezután a megfelelő speciális típussal is fel tudunk használni.

Amikor egy osztályt úgy akarunk létrehozni, hogy ahhoz felhasználunk egy őszosztályt, az osztály deklarációjában az osztálynév után az `extends` kulcsszóval hozzákapcsoljuk a definícióhoz az őszosztály nevét:

```
363 public class alosztály extends őszosztály { ... }
```

Az így létrehozott alosztály az őszosztály leszármazottja, ezért azt mondjuk, hogy a létrehozott alosztályt az őszosztályból származtattuk.

A Java az egyszeres öröklést vallja, ami azt jelenti, hogy minden származtatott osztálynak pontosan egy őszosztálya lehet. Az osztályhierarchia így egy fa gráffal szemléltethető.

8.2.2 Polimorfizmus

Amikor egy új osztályt úgy hozunk létre, hogy azt származtatjuk valamilyen őszosztályból, akkor az új osztályban az őszosztályban már szereplő metódusokat módosíthatjuk. Adott osztályúnak tekintett objektum azonos szignatúrájú metódusához más, a származtatott osztálynak megfelelő művelet sor is tartozhat. Ezt a jelenséget nevezzük polimorfizmusnak.

8.2.3 Absztrakt osztályok

Absztrakt osztályoknak nevezzük azokat az osztályokat, amelyek csupán egy általános keretet határoznak meg, de konkrét esetben (példányosításkor) csak valamely speciális alosztály jelenhet meg. Így absztrakt osztályú objektumot közvetlenül nem készíthetünk, csak valamilyen konkrét (nem absztrakt) származtatott osztályon keresztül.

```
364 public abstract class osztálynév { ... }
```

Absztrakt lehet egy metódus is, ha azt akarjuk kijelenteni, hogy az adott metódus csak valamilyen konkrét, származtatott osztályban legyen elérhető.

```
365 public abstarct típus metódusnév();
```

Figyeljük meg, hogy az absztrakt metódusoknak nincs törzse, az utasítás-blokk helyett egy pontosvesszőt adunk meg.

8.2.4 A protected hozzáférési tag

Az előző leckékben megismerkedtünk a `private` és a `public` láthatósági taggal. Ha nem jelöljük az adott elem láthatóságát, akkor alapértelmezés szerint ún. félnyilvános láthatóság jön létre.

Létezik egy negyedik láthatósági forma is, amelyet a `protected` kulcsszóval vezethetünk be. A `protected` (védett) jellegzetességek csak egy adott osztályon, illetve a belőle származtatott alosztályokban lesznek láthatók.

8.3 ÖSSZEFOGLALÁS, KÉRDÉSEK

8.3.1 Összefoglalás

Ezekkel a nagyon fontos definíciókkal befejeztük az OOP alappilléreinek tárgyalását. Az elmélet után lássuk a gyakorlatot is, a következő leckékben számos példát fogunk látni az ősosztályokból való származtatásra (öröklésre), illetve a polimorfizmusra.

8.3.2 Önellenőrző kérdések

1. Mit jelent az öröklés az OOP paradigmában?
2. Mi öröklődik egy ősosztály definíciójából a származtatott osztályban?
3. Mit jelent a polimorfizmus kifejezés?
4. Mik azok az absztrakt osztályok?

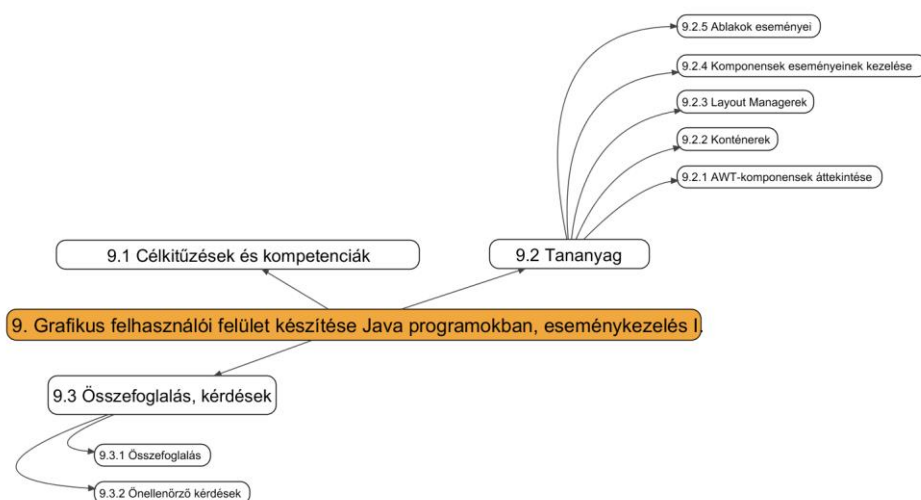
9. LECKE: GRAFIKUS FELHASZNÁLÓI FELÜLET KÉSZÍTÉSE JAVA PROGRAMOKBAN, ESEMÉNYKEZELÉS I.

9.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

Ebben a leckében elkezdzünk megismerkedni a Java grafikus felhasználói felületek (GUI) fejlesztéséhez kínált lehetőségeivel. A Java nyelv már kezdettől fogva tartalmazott egy osztálykönyvtárat GUI-k készítéséhez. Ezt később AWT-nek (Abstract Window Toolkit) nevezték el. Az AWT segítségével olyan grafikus ablakokat lehetett létrehozni, amelyek a szokásos grafikus felületeken megtalálható vezérléseket tartalmazhatták, pl. nyomógombok, szövegbeviteli mezők, görgetősávok stb. Az AWT jellegzetessége az volt, hogy a létrehozott GUI-k megjelenése attól függött, hogy milyen operációs rendszer alatt futtattuk a kész programot. Nem sokkal később ezt az osztálykönyvtárat továbbfejlesztve megjelent a Swing osztálykönyvtár, amelynek segítségével már olyan grafikus felületeket lehetett létrehozni, amelyek operációs rendszertől függetlenül ugyanolyan megjelenést biztosítottak. Ennek a leckének a célja, hogy bevezetést nyújtson a GUI-k készítéséhez.

A lecke feldolgozásához szükséges kompetenciák: logikai képesség, induktív gondolkodás képessége, deduktív gondolkodás képessége, absztrakt (elméleti) gondolkodás, áttekintő képesség, rendszerező képesség és figyelemösszpontosítás.

9.2 TANANYAG



9.2.1 AWT-komponensek áttekintése

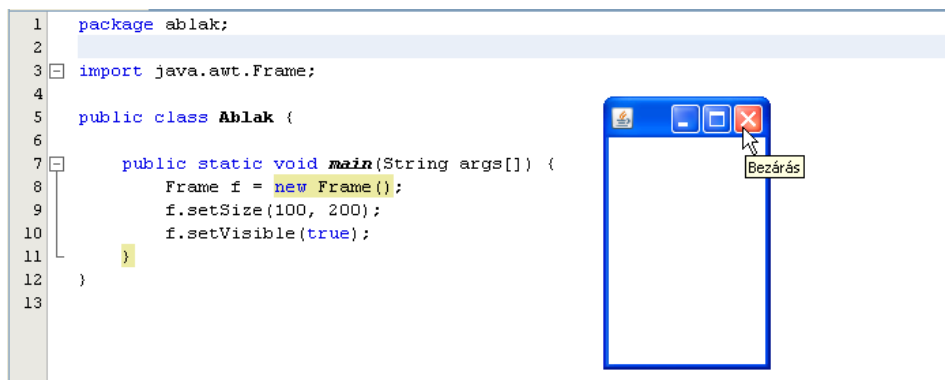
Az AWT egy mai szemmel kissé kezdetleges ablakozó rendszer, amely minden platformon azonos futtatási lehetőségeket biztosít, de az adott platformra jellemző megjelenéssel. Az AWT használatához importálni kell a `java.awt` csomagot, illetve annak alcsomagjait. Készítsük el az első AWT-s alkalmazást, amely megjelenít egy ablakot a képernyőn.

```
366 import java.awt.Frame;
367 public class Ablak {
368     public static void main(String args[]){
369         Frame f = new Frame();
370         f.setSize(100,200);
371         f.setVisible(true);
372     }
373 }
```

Ha futtatjuk a példaprogramot, megjelenik a képernyő bal felső sarkában egy 100 pixel széles, 200 pixel magas ablak. Ez úgy viselkedik, mint a Windows-ablakok⁹ általában: át lehet méretezni, mozgatni lehet, méretét minimalizálhat-

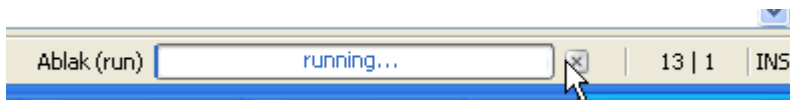
⁹ Jegyzetünk készítésekor Windows operációs rendszert használtunk.

juk, vagy teljes képernyősre állíthatjuk, de túl sok mindent még nem tud ez az ablak. Például nem lehet bezárni.



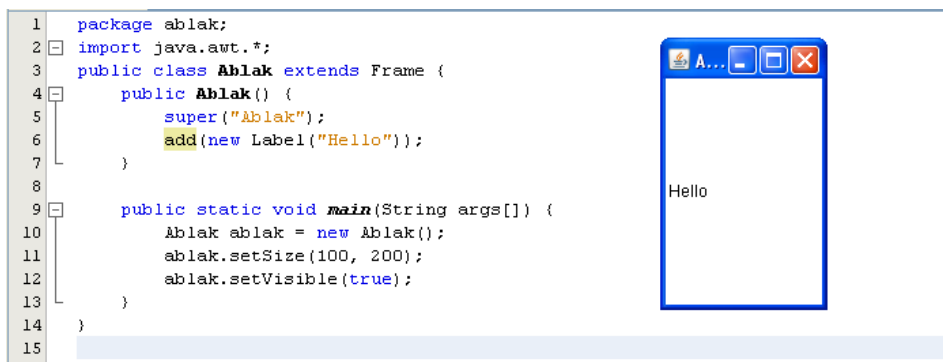
47. ábra: Az első Frame-ünk még nem zárható be

Ha NetBeans alatt dolgozunk, úgy lehet gyorsan bezárni ezt a most futó programot, ha a NetBeans ablakának alsó részén, a státuszsor jobb oldali felén rákattintunk a projekt neve melletti folyamatjelzőtől jobbra található kicsi bezárás gombra.



48. ábra: Futó projekt bezárása NetBeansben

Erre a grafikus felületre komponenseket (címkék, nyomógombok, listák, legördülő listák, választónégyzetek, rádiógombok, szövegbeviteli mezők stb.) helyezhetők el. A komponenseket úgy próbálhatjuk ki, ha az eddigi program (a fenti példában *Ablak* néven hoztuk létre a NetBeans projektet) konstruktorába helyezzük el a komponensek létrehozását.



49. ábra: Label komponens

A példa 6. sorában látható az a sor, amely létrehoz az ablak közepén egy címkét (label). Próbáljuk ki a következő komponenseket: nyomógomb, választónégyzet, szövegbeviteli mező, többsoros szövegmező, legördülő listamező, többszörös választást engedélyező listamező, kombinált listamező. Ehhez a következőket kell a konstruktorba írunk a fenti példa 6. sora helyett:

Nyomógomb

```
374 add(new Button("Hello"));
```

Ennek hatására az ablakot teljesen kitöltő nyomógomb jelenik meg a frame-ben. Ezen egyelőre hiába kattintunk, nem történik semmi. Ugyanez lesz a helyzet a többi, ezután következő komponenssel is, de csak egyelőre. Később természetesen megvizsgáljuk, hogy hogyan lehet a komponenseken bekövetkező eseményeket kezelni.

Választónégyzet

```
375 add(new Checkbox("Hello"));
```

Ez egy ki- bepipálható választónégyzetet, és mellette a Hello feliratot jeleníti meg.

Szövegbeviteli mező

```
376 add(new TextField("Hello"));
```

Egy a frame-et teljesen kitöltő szövegbeviteli mezőt kapunk, amelyben megjelenik a Hello felirat, de ezt kitörölhetjük, átírhatjuk. A szövegmező látszó-

lag ugyan kitölti a teljes frame-et, de valójában csak egysoros szöveg írható bele. Próbáljunk meg egy hosszabb szöveget begépelni.

Többsoros szövegmező (textarea)

```
377 add(new TextArea („Hello”));
```

Ez a komponens egy olyan szövegmezőt hoz létre, amely többsoros szövegeket is képes fogadni (nyomhatunk Entert a gépelés közben), és a szövegmező igény szerint át is méretezhető a jobb alsó sarkában található méretezővel. Ilyenkor természetesen maga a frame is átméreteződik. A textarea komponenshez görgetősávok is tartoznak, amelyek kezdetben inaktívak, de ha szükséges, aktívvá és görgethetővé válnak. Próbáljuk ki úgy ezt a komponenst, hogy begépelünk egy-két szót, azt kijelöljük az egérrel, majd a vágólap segítségével sokszor bemásoljuk. Így gyorsan teli tudjuk írni és ki tudjuk próbálni minden szolgáltatását.

Listamező

Ide már több sort kell írunk az eredeti kódban szereplő 6. sor helyére.

```
378 List list = new List();
379 list.add(„Hétfő”);
380 list.add(„Kedd”);
381 list.add(„Szerda”);
382 list.add(„Csütörtök”);
383 list.add(„Péntek”);
384 list.add(„Szombat”);
385 list.add(„Vasárnap”);
386 add(list);
```

Egy listamező jött létre, amelyben kattintással vagy a kurzormozgató nyilakkal tudunk egy elemet kiválasztani.

Listamező többszörös választással

Módosítsuk a listamezőnél látható kódrészlet első sorát a következőre:

```
387 List list = new List(7, true);
```

A konstruktornak átadott paraméterek közül a 7-es érték a listamező látható elemeinek számát jelöli (egyelőre ez nem működik, később majd ezt a problémát is megoldjuk), a második paraméter pedig egy logikai érték, amely azt

határozza meg, hogy engedélyezett-e a listamezőben a többszörös kiválasztás vagy sem. Ha az érték `true`, akkor a többszörös kiválasztás engedélyezett. Ezt úgy ellenőrizhetjük, ha az első elem kiválasztása után a továbbiakat a Shift vagy a Ctrl billentyű nyomva tartása mellett jelöljük ki egy-egy kattintással.

Legördülő listamező

Módosítsuk a listamezőnél és a többszörös kiválasztás engedélyező listamezőnél megadott kódot a következőre:

```
388 Choice choice = new Choice();
389 choice.add(„Hétfő”);
390 choice.add(„Kedd”);
391 choice.add(„Szerda”);
392 choice.add(„Csütörtök”);
393 choice.add(„Péntek”);
394 choice.add(„Szombat”);
395 choice.add(„Vasárnap”);
396 add(choice);
```

Egy legördülő listamező fog megjelenni a frame közepén. Ehhez nem tudjuk az előbbi trükköt felhasználni: egy legördülő listamező nem tehető többszörös kiválasztást engedélyező komponenssé. (A listamező ebben a tekintetben a választónégyzetekhez hasonlít, míg a kombinált listamező a rádiógombcsoportokhoz.)

9.2.2 Konténerek

A konténerek olyan komponensek, amelyek más komponenseket tartalmazhatnak, akár további konténereket is. Így ezeket egymásba lehet ágyazni. Arra való, hogy az összetartozó képernyőelemeket (komponenseket, űrlapmező-csoportokat) összefogják. A frame is egy konténer.

Egy speciális konténer a panel, amely szintén komponens, de nem látható.

```
397 Panel panel = new Panel();
398 panel.add(new Button(„alma”));
399 panel.add(new Button(„körte”));
400 panel.add(new Button(„szilva”));
401 panel.add(new Button(„barack”));
402 add(panel);
```

Ha a frame továbbra is 100×200 pixel méretű, akkor a panelen a négy gomb kettesével egymás alatt fog elhelyezkedni. Próbáljuk meg átméretezni az ablakot. A gombok automatikusan áthelyeződnek, ha a frame elég szélessé válik ahhoz, hogy már egymás mellett is elférjenek.

A komponensek az eddigi példákban automatikusan kerültek a helyükre, és a méretüket sem mondtuk meg. Amikor esztétikus űrlapot akarunk készíteni, akkor a komponensek elhelyezése nagyon sok számolást igénylő feladat, pláne, ha az űrlapnál engedélyezzük az átméretezést is. Ezt a feladatot könnyíti meg a Layout Manager.

9.2.3 Layout Managerek

A Layout Manager abban segíti a programozót, hogy nem kell a komponensek elhelyezésének részleteivel foglalkoznia, de mégis megmondhatja, hogy az egyes elemek mekkorák legyenek és hová kerüljenek. Többféle előre definiált Layout Manager is létezik, ezek közül néhány:

- BorderLayout
- FlowLayout
- GridLayout
- CardLayout

Saját Layout Managert is készíthetünk, ha kellően nagy gyakorlattal rendelkezünk már.

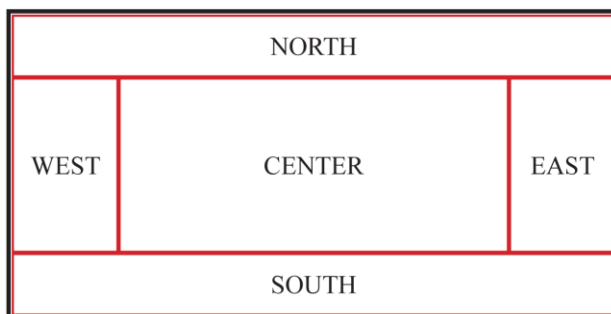
FlowLayout

A panel komponensnél ez az alapértelmezett. A komponenseket folyamatosan kell rajta elhelyezni. Alapértelmezés szerint a komponensek a FlowLayoutban középre igazítva jelennek meg, és a rendelkezésre álló szélesség tükrében a lehető legtöbb komponens egymás mellé fog kerülni. Ha átméretezzük a frame-et (és ezzel rajta a panelt, vagyis megváltoztatjuk a panel komponensei számára rendelkezésre álló képernyőszélességet), akkor a komponensek pozíciója változhat.

Frame-ek esetében nem a FlowLayout az alapértelmezett.

BorderLayout

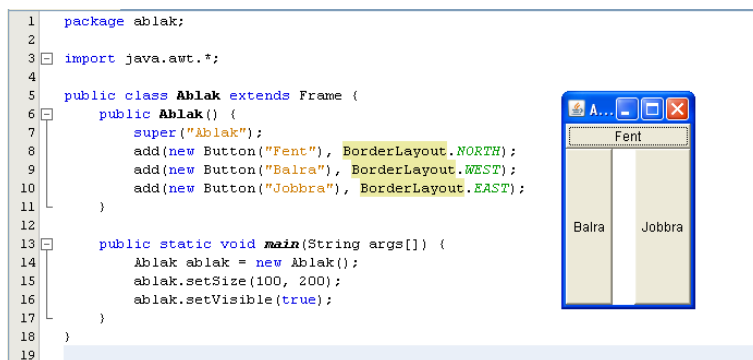
Ez a Layout Manager a konténer (pl. panel) széleihez igazítja a tartalmat. Az elhelyezés megadásánál az égtájak angol neveit használhatjuk (NORTH – észak, EAST – kelet, SOUTH – dél, WEST – nyugat), illetve a konténer közepét a CENTER jelöli.



50. ábra: A BorderLayout égtájai

Az ilyen Layout Manager használata mellett a konténeren legfeljebb öt komponens használható fel, kevesebb lehet, de több nem.

A Layout Manager úgy helyezi el és méretezi a komponenseket, hogy azok észak – dél – nyugat – kelet – közép sorrendben a rendelkezésre álló területet a lehető legjobban használják ki. Az alábbi példában három gombot helyezünk el: egyet északon, egyet nyugaton, egyet pedig keleten.



51. ábra: Border Layout három elemmel

Mivel délen nincs semmi, a nyugati és a keleti gomb magassága elfoglalja a déli területet is. A gombok szélességét a Java automatikusan határozta meg, a középső terület most üresen maradt. Próbáljuk meg átméretezni ez a frame-et: a gombok mérete is változik, az elrendezés viszont nem.

A Frame komponensnél a Border Layout az alapértelmezett, ha más konténerhez akarjuk ezt beállítani, akkor azt a `setLayout` metódussal tehetjük meg.

```
403 Panel panel = new Panel();  
404 panel.setLayout(new BorderLayout());  
405 panel.add(new Button("Bal"),  
    BorderLayout.WEST);  
406 ...
```

GridLayout

A GridLayout Manager használatával azonos méretű komponenseket helyezhetünk el a konténerben. A GridLayout konstruktora két paramétert vár. Ha az első paraméter pozitív szám, a második értéke bármi lehet, figyelmen kívül marad. Ha az első paraméter nulla, akkor a másodiknak pozitívnak kell lennie.

Ha az első paraméter pozitív, azzal a kialakítandó sorok számát határozzuk meg, ha az első nulla, a második pozitív, akkor a kialakítandó oszlopok számát határozzuk meg. Ha megadtuk vagy a sorok vagy az oszlopok számát, a rács méretét a komponensek száma határozza meg.

Például a

```
407 setLayout(new GridLayout(3, 0));
```

elrendező objektum az adott konténerben három sorban fogja elhelyezni a komponenseket.

9.2.4 Komponensek eseményeinek kezelése

Eddig hiába hoztunk létre űrlapokat különféle komponensekkel, a komponensek nem csináltak semmit. Hiába kattintottunk például egy nyomógombra, nem történt semmi. A következőkben megnézzük, hogyan lehet funkcionalitást rendelni a felületelemekhez. Beállítjuk, hogy mi történjen, amikor a felhasználó valamilyen műveletet hajt végre egy komponensen, vagyis amikor valamilyen esemény bekövetkezik az adott elemen.

Azt mondhatjuk, hogy egy GUI-val rendelkező program eseményorientált alkalmazásként működik. Az előző fejezettel bezárólag készített programjaink úgy működtek, hogy az értelmezőprogram egymás után végrehajtotta a program utasításait, amikor az utasítások elfogytak, az értelmezőprogram (illetve az egész futtató rendszer) befejezte a működését.

Az eseményorientált szemlélet lényege, hogy a program elhelyezi a képernyőn a megfelelő komponenseket, majd vár, hogy bekövetkezzen valamilyen esemény. Ha bekövetkezett egy olyan esemény, amelyhez a programozó az

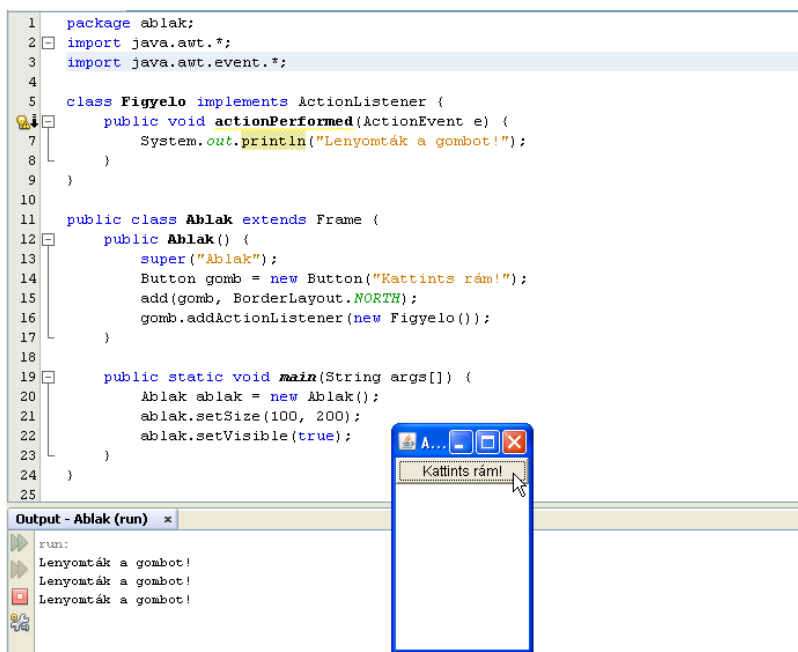
adott képernyőobjektummal kapcsolatosan tevékenységet állított be, akkor végrehajtásra kerül ez a tevékenység, majd a rendszer ismét várakozó állapotba kerül: várja a következő esemény bekövetkezését.

Ha egy komponens nem tud mit kezdeni azzal az eseménnyel, amelyet végrehajtottak rajta, akkor továbbadja ezt az eseményt az őt tartalmazó komponensnek, és így tovább.

Az eseménymodell három részből áll:

- az esemény forrása: ez az a komponens, amelyen bekövetkezett az esemény (pl. egy nyomógomb)
- az esemény: ez az, ami a komponenssel történt (pl. kattintás)
- figyelő: ez az, ami reagálni tud az eseményre, ez tartalmazza az esemény bekövetkezésekor végrehajtandó kódot.

Az események kezeléséhez importálnunk kell a *java.awt.event* csomagot. Készítsünk egy olyan programot, amely a konzolra kiírja a „Lenyomták a gombot!” üzenetet, ha egy frame-ben elhelyezett gombra rákattint a felhasználó.



52. ábra: Eseménykezelő készítése

9.2.5 Ablakok eseményei

Térjünk vissza most ahhoz a problémához, hogy nem lehet bezárni az ablakot a szokásos módon.

Az ablakokon bekövetkező eseményekhez használható a `WindowListener`, amely a következő eseményeket kezeli:

- `windowActivated(WindowEvent e)`
- `windowDeactivated(WindowEvent e)`
- `windowOpened(WindowEvent e)`
- `windowClosed(WindowEvent e)`
- `windowClosing(WindowEvent e)`
- `windowIconified(WindowEvent e)`
- `windowDeiconified(WindowEvent e)`

Ha a `WindowListener`-t akarjuk használni, mind a hét eseményhez meg kell valósítanunk egy-egy metódust, pedig lehet, hogy mi csak egyhez szeretnénk, pl. éppen a bezáráshoz. A többi hat metódus törzsét ebben az esetben üresen hagynánk.

Ehelyett célszerűbb a `WindowAdapter` osztályt használni, amely a `WindowListener` mind a 7 metódusát üres törzsszel valósítja meg. Származtassuk ebből az eseménykezelőnket, írjuk meg a szükséges metódusokat, a többi pedig (üresen) öröklődik.

A megoldást az olvasóra bízuk. Az eddigiek alapján ez a feladat már önállóan is megoldható.

9.3 ÖSSZEFOGLALÁS, KÉRDÉSEK

9.3.1 Összefoglalás

Ebben a leckében a Java grafikus lehetőségei közül az AWT alapjait mutatuk be. Létrehoztunk grafikus felülettel rendelkező ablakokat, ezeken komponenseket, illetve a komponenseket elrendező konténereket helyeztünk el. Áttekintettük a konténerek elrendezésének automatizált lehetőségeit a `LayoutManager`-eken keresztül, végül pedig megvizsgáltuk a komponenseken bekövetkező események kezelésének lehetőségeit.

9.3.2 Önellenőrző kérdések

1. Mit jelent az AWT rövidítés?
2. Mi a különbség az AWT és a Swing között?
3. Milyen Java csomagot vagy csomagokat kell importálnunk, ha az AWT-t akarjuk használni?
4. Milyen komponenseket használhatunk az AWT-vel?
5. Mire valók a konténerek?
6. Milyen konténerek léteznek?
7. Hogyan kezelhetjük a komponenseken bekövetkező eseményeket?

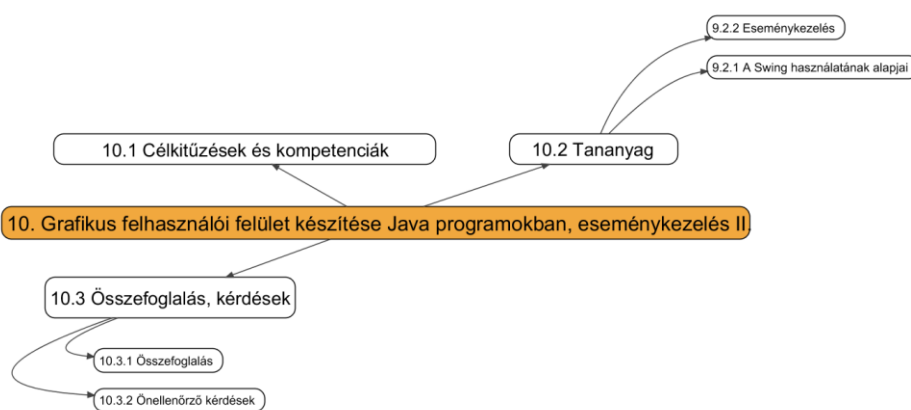
10. LECKE: GRAFIKUS FELHASZNÁLÓI FELÜLET KÉSZÍTÉSE JAVA PROGRAMOKBAN, ESEMÉNYKEZELÉS II.

10.1 CÉLKITŰZÉSEK ÉS KOMPETENCIÁK

Ebben a leckében folytatjuk az előző leckében elkezdett ismerkedést a Java grafikus lehetőségeivel. Ehhez a Java Swing eszközt hívjuk segítségül. A Swing használata alapvetően hasonlít az AWT használatához, mert itt is konténereket, paneleket, komponenseket tudunk használni, és ezek eseményeit kell kezel-nünk, amikor grafikus felületű alkalmazást készítünk. A különbség leginkább a megjelenésben lesz tetten érhető.

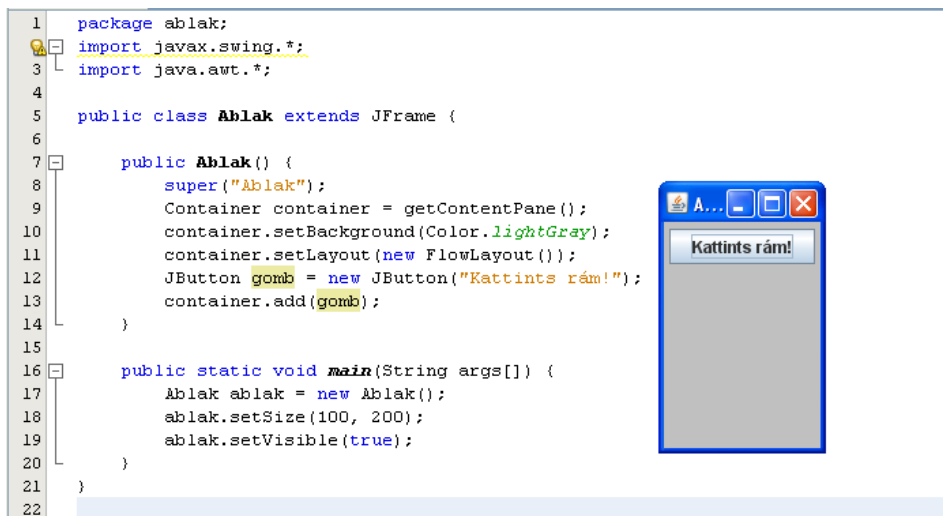
A lecke feldolgozásához szükséges kompetenciák: logikai képesség, induk-tív, deduktív és absztrakt (elméleti) gondolkodás, áttekintő- és rendszerező-képesség, figyelem-összpontosítás.

10.2 TANANYAG



10.2.1 A Swing használatának alapjai

Tekintsük az alábbi kódrészletet:



53. ábra: Frame létrehozása Swing használatával

A program elején az `import` utasítások közé felvettük a `swing` csomag összes alcsomagját importáló utasítást.

Az `Ablak` osztályt most nem a `Frame`, hanem a `JFrame` osztályból származtattuk. Figyeljük meg, hogy a `J` betű más osztályok neve elé is oda fog kerülni a következőkben, a fenti példában a `Button` osztály helyett írtunk `JButton`-t.

A most létrehozott frame annyiban különbözik az előző leckében létrehozottól, hogy meghatároztuk az ablak belső részének háttérszínét is, illetve a megjelenő gomb esztétikusabb, mint az AWT-beli párja. Ennél többről is szó van: ahogyan az előző leckében leírtuk, a komponensek nemcsak esztétikusabbak, de platformfüggetlenné is váltak. Ez azt jelenti, hogy ezt a kódot Linux vagy más operációs rendszer alatt kipróbálva ugyanilyen megjelenésű komponenseket látnánk.

Tekintsük át most vázlatosan a Swing komponenseit.

Szövegmező

Módosítsuk a fenti kódrészletben az `Ablak` osztály konstruktorának utolsó két sorát: `gomb` helyett hozzunk létre egy szövegbeviteli mezőt.

```
408 JTextField txtNév = new JTextField(10);  
409 container.add(txtNév);
```

A JTextField osztály konstruktorának átadott paraméter a szövegmező szélességét definiálja. Ez a szám az egyszerre látható karakterek számát jelenti. (Mivel rendszerint minden betűtípusban a kis *m* betű a legszélesebb, jó közelítéssel mondhatjuk azt, hogy a megadott szám a szövegmezőbe írható *m* betűk számát jelenti. Ha a szövegmezőbe csupa *i* betűt írunk, abból jóval több elfér.)

Jelszómező

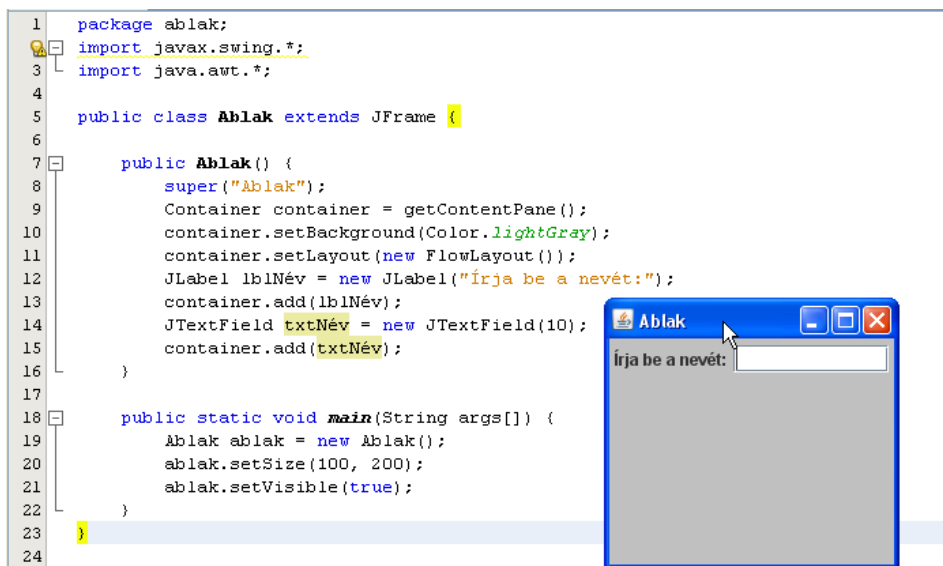
Módosítsuk a fenti példában a konstruktor utolsó sorait az alábbival:

```
410 JPasswordField pwd = new JPasswordField(10);  
411 pwd.setEchoChar('*');  
412 container.add(pwd);
```

A kapott szövegbeviteli mező most jelszómezővé válik, a begépelt karakterek helyett csillagok jelennek meg.

Címke

A komponensek, főként a szövegbeviteli mezők elé minden űrlapon ki szoktuk írni, hogy milyen adatot kell oda beírnia a felhasználónak. Ezeket a címkéket a JLabel osztályból példányosított objektumokkal írhatjuk ki. Az alábbi példa egy olyan űrlapot definiál, amelyben a felhasználó a nevét adhatja meg. Mivel a frame mérete alapértelmezés szerint túl keskeny ahhoz, hogy a címke és a szövegmező egymás mellé elférjenek, a FlowLayout Manager miatt ezek indításkor egymás alatt jelennek meg. Ha átméretezzük az ablakot, a komponensek egymás mellé kerülnek (a FlowLayout miatt a frame közepére igazítva).



54. ábra: JLabel és JTextField

Választónégyzetek és rádiógombok

Választónégyzetek létrehozásához a *JCheckBox* osztályból kell példányokat létrehoznunk, rádiógombokhoz pedig a *JRadioButton* osztály példányai alkalmasak.

Mindkettő osztály konstruktorának két paramétert kell átadnunk, pl.:

```

413 JCheckBox v1 = new JCheckBox("alma", true);
414 JCheckBox v2 = new JCheckBox("körte", false);

```

Az első paraméter a választónégyzet vagy rádiógomb címkéjét határozza meg, a második pedig azt, hogy az elem ki van-e jelölve, vagy sem. Egy választónégyzet-csoportban akármennyi választónégyzet ki lehet jelölve, egy rádiógombcsoportban viszont csak egy vagy (alapértelmezetten) egy sem.

Csoportokat a *ButtonGroup* osztályból létrehozott példányokkal tudunk definiálni. Ezekre a szokásos *add* metódussal tudunk újabb és újabb elemeket elhelyezni.

Listamező

A Swing használata mellett az AWT-hez képest némileg módosul a listamezők definiálása. Természetesen fel kell sorolnunk itt is a megjelenítendő eleme-

ket, de a többszörös kiválasztást, illetve az egyszerre megjelenítendő elemek számát másként kell meghatározunk, mint az AWT-ben.

```
415 String[] gy = {"alma", "szőlő", "som", "ba-  
nán"};  
416 JList lstGyümölcsök = new JList(gy);  
417 lstGyümölcsök.setVisibleRowCount(3);  
418 lstGyümölcsök.setSelectionMode(  
419     ListSelectionModel.SINGLE_SELECTION);  
420 JScrollPane sp = new  
421     JScrollPane(lstGyümölcsök);  
421 container.add(sp);
```

A példában létrehozunk egy sztringtömböt, amelyben a megjelenítendő elemeket soroljuk fel. Ezt átadjuk a *JList* osztály konstruktorának, ezzel létrehozuk a listamező objektumot. Az objektum tulajdonságai között beállítjuk azt, hogy egyszerre csak három elem legyen látható a négy közül, illetve hogy csak egy elem legyen kiválasztható a listamezőben. (Ez a *ListSelectionModel.MULTIPLE_INTERVAL_SELECTION* konstans használatával alakítható át többszörös kiválasztást engedélyező listamezővé.)

Végül, mivel a *JList* osztályból példányosított objektumok automatikusan nem kapnak görgetősávot, létrehozunk egy görgetősávval ellátott panelt, amelynek példányosításkor átadjuk tartalomként a listamezőt. Az utolsó utasítással pedig elhelyezzük a konténerben a most létrehozott görgetősávos komponenst.

10.2.2 Eseménykezelés

Készítsünk egy olyan alkalmazást, amelyen három gombot helyezünk el. A három gomb segítségével a felhasználó meg tudja változtatni az alkalmazás futtatóablakának háttérszínét. A forráskód elemzését az olvasóra bízuk.

```

1  package ablak;
2  import javax.swing.*;
3  import java.awt.*;
4  import java.awt.event.*;
5
6  public class Ablak extends JFrame {
7      public Ablak() {
8          super("Ablak");
9          Container container = getContentPane();
10         container.add(new Háttér());
11     }
12
13     public class Háttér extends JPanel implements ActionListener {
14         private JButton gombPiros, gombKék, gombZöld;
15         public Háttér() {
16             gombPiros = new JButton("piros");
17             gombKék = new JButton("kék");
18             gombZöld = new JButton("zöld");
19             gombPiros.addActionListener(this);
20             gombKék.addActionListener(this);
21             gombZöld.addActionListener(this);
22             add(gombPiros);
23             add(gombZöld);
24             add(gombKék);
25         }
26
27         public void actionPerformed(ActionEvent e) {
28             Object komponens = e.getSource();
29             Color szín = Color.white;
30             if (komponens == gombPiros)
31                 szín = Color.red;
32             else if (komponens == gombZöld)
33                 szín = Color.green;
34             else if (komponens == gombKék)
35                 szín = Color.blue;
36             setBackground(szín);
37             repaint();
38         }
39     }
40
41     public static void main(String args[]) {
42         Ablak ablak = new Ablak();
43         ablak.setSize(100, 200);
44         ablak.setVisible(true);
45     }
46 }
47

```

55. ábra: A háttérszín-változtató ablak programkódja

10.3 ÖSSZEFOGLALÁS, KÉRDÉSEK

10.3.1 Összefoglalás

Ebben a leckében megvizsgáltuk a Java Swing használatának legalapvetőbb jellemzőit. A Swing lényegében ugyanazt a célt szolgálja, mint az AWT, de a létrehozható GUI sokkal esztétikusabb, és minden operációs rendszer alatt azonos megjelenést biztosít. A GUI-val foglalkozó két lecke csak nagyon röviden

mutatja be a lehetőségeket, azonban bízunk benne, hogy az itt leírtak alapján az olvasó önállóan is képes lesz a további lehetőségek feltérképezésére. Ehhez az interneten nagyon bőséges forrás található.

10.3.2 Önellenőrző kérdések

1. Mi az alapvető különbség az AWT és a Swing között?
2. Milyen csomagokra van szükség a Swing használatakor?
3. Szükségesek-e az AWT csomagjai, ha Swing segítségével készítünk grafikus felhasználói felületet?
4. Mi a különbség az AWT és a Swing listamezői között?

11. LECKE: APPLETEK ÉS FEJLESZTÉS MOBIL ESZKÖZÖKRE

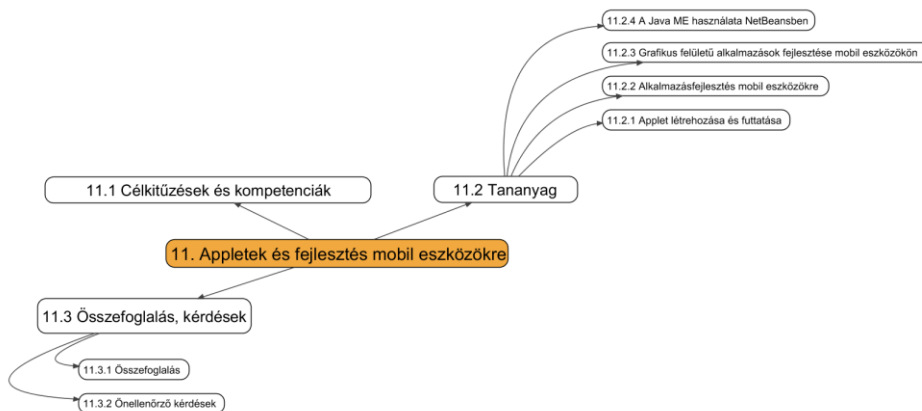
11.1 CÉLKITÚZÉSEK ÉS KOMPETENCIÁK

Jegyzetünk befejező leckéjében a Java Appletek használatát mutatjuk be. Az Appletek olyan programok, amelyeket böngészőprogramokban tudunk futtatni. Bármilyen Java-program elkészíthető Appletként is, így AWT-t vagy Swinget használó programok is.

Ahhoz, hogy egy Appletet böngészőben tudjunk megtekinteni, ismernünk kell az Appletek elhelyezéséhez szükséges HTML-eszközöket (tageket) is. Ez azonban csak minimális HTML-ismeretet igényel.

A lecke második felében betekintést nyerünk a mobilkészülökökön történő alkalmazásfejlesztésbe.

11.2 TANANYAG



11.2.1 Applet létrehozása és futtatása

Hozunk létre a szokásos módon egy új projektet a NetBeansben. Az Appletek Java-forrásfájljai némiképp eltérnek a hagyományos Java-programokétól, mert ezekben nem kell létrehoznunk `main` metódust. Ennek okán nem is tudjuk őket úgy futtatni, ahogyan a szokásos alkalmazások esetében megszoktuk azt.

Tekintsük az alábbi példát:

```
1 import java.awt.*;
2 import java.applet.*;
3 public class AppletTeszt extends Applet {
4     public void paint(Graphics g) {
5         Color hatterszin = new Color(255, 200, 200);
6         Color betuszin = Color.black;
7         Font font = new Font("Courier", Font.BOLD, 50);
8         setBackground(hatterszin);
9         g.setColor(betuszin);
10        g.setFont(font);
11        g.drawString("HELLÓ!", 120, 140);
12        g.drawLine(120, 160, 290, 160);
13        g.drawLine(120, 162, 290, 162);
14    }
15 }
16
```

56. ábra: Az első Applet forráskódja

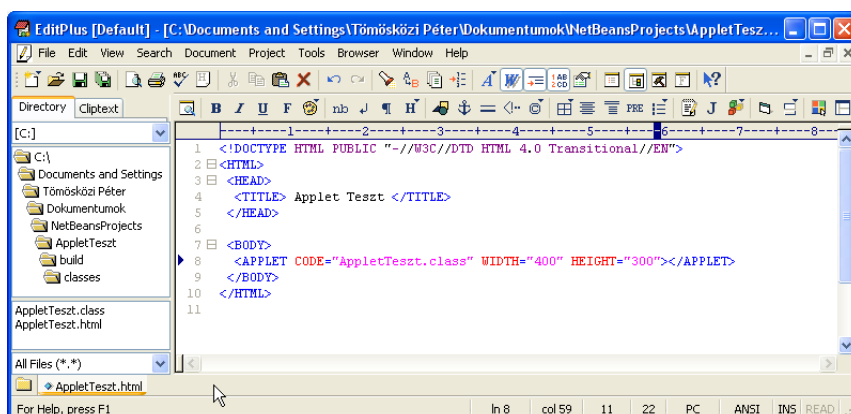
A program elején a szokásos dolgokat látjuk: mivel a program használja az AWT eszközeit, importálnunk kell az *awt* csomag alcsomagjait. Amikor Appletet készítünk, a fájlnevvvel megegyező osztályt mindig a *java.applet.Applet* osztályból kell származtatni. Mivel a program elején importáltuk a *java.applet* csomagot, így az osztálydeklarációban az *extends* kulcsszó után már elegendő csak az *Applet* osztálynevet kiírunk.

Ahogy az korábban írtuk, Appletek esetén nincs *main* metódus. Ebben a példában egy *paint* metódust írunk le, amely paraméterként megkapta a *java.awt.Graphics* osztályú *g* objektumot. (A *Graphics* statikus osztály, a *g*-t nem kell példányosítani.)

Ezután az AWT megszokott eszközeivel kiíratást végzünk a képernyőn.

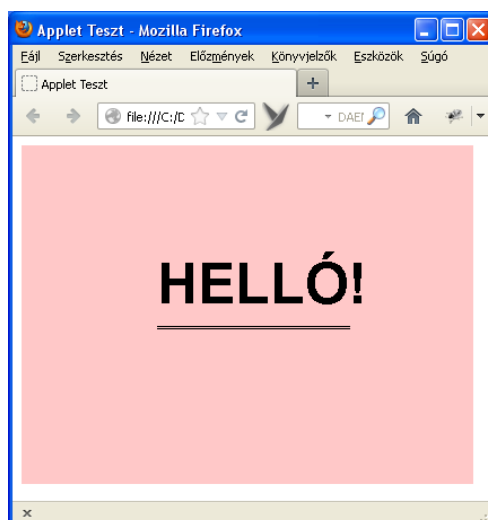
Az Applet nem futtatható közvetlenül, viszont a futtatáshoz le kell fordítanunk. Ehhez válasszuk a NetBeans Run menüjéből a *Compile File* menüpontot. A fordítás után elkészül a forrásfájl nevével megegyező tárgykód, amelynek kiterjesztése *.class* lesz. Ezt a fájlt (ha a NetBeansben nem változtattuk meg az alapértelmezett beállításokat) a Dokumentumok mappánk NetBeansProjects mappájában található, a projekt nevével egyező nevű mappán belüli *build/classes* mappában találjuk meg. A fenti példa alapján a fájlnev *AppletTeszt.class* lett.

Ugyanebben a mappában létre kell hoznunk egy HTML-fájlt. Ehhez bármilyen HTML-editort használhatunk, de akár Jegyzettömbbel is elkészíthető az alábbi, rövid HTML-kód:



57. ábra: Az Appletet megjelenítő HTML-forrás

Végezetül, ha elmentettük a HTML-fájlt (fontos, hogy ugyanabban a mappában legyen a HTML-állomány, mint a .class-fájl), nyissuk azt meg egy tetszőleges böngészőprogramban. Az alábbi képernyőkép Mozilla Firefoxban készült, de bármilyen böngésző megteszi.



58. ábra: Applet a böngészőben

11.2.2 Alkalmazásfejlesztés mobil eszközökre

A bevezető leckében megismertük a Java különböző implementációit, ezek közül a Sun (a Java nyelv eredeti fejlesztője) hivatalosan a Java ME (Mobile Edition) változatot javasolja alkalmazásfejlesztésre mobiltelefonokon. Nem

minden mobilgyártó cég ért ezzel egyet, de ebben a rövid betekintő fejezetben most a Java ME-t tárgyaljuk.

A Java ME kétféle specifikációban létezik mobil eszközökön: az egyik a Connected Device Configuration (CDC), a másik a Connected Limited Device Configuration (CLDC). Előbbi eredetileg nagyobb teljesítményű mobileszközökhöz készült, és ezért jobban hasonlít a Java SE változatához. A CLDC-t gyengébb képességű eszközökre tervezték, mégis ez most a népszerűbb nagyobb teljesítményű mobileszközökön is.

A CDC jellemzője, hogy létezik benne a Java SE-ből ismert AWT (Abstract Window Toolkit). Ennek az a hátránya is egyben, hogy ezt eredetileg nem mobil eszközökön futtatott grafikus felületek fejlesztéséhez alkották meg. A CDC lehetővé teszi a Swing használatát is, így a CDC szabványt követő mobil eszközökön valódi, asztali alkalmazásokhoz hasonló programok fejleszthetők.

A CLDC-hez külön felületet, ún. API-t (Application Programming Interface) fejlesztettek ki, ennek a neve MIDP (Mobile Information Device Profile). Ennek megalkotásakor a fő szempont az volt, hogy minél többféle eszközön való futtatást tegyen lehetővé. A MIDP lehetőségei mellett további programfejlesztési felületet is definiáltak a CLDC-eszközökhöz, azonban nem minden eszköz támogatja ezek közül mindet.

A MIDP-ben készült alkalmazásokat MIDleteknek nevezik. Egy MIDlet osztály létrehozásakor három absztrakt metódust kell implementálnunk, ezek neve *startApp*, *pauseApp* és *destroyApp*.

Amikor elindul egy MIDlet, a *startApp* metódus fut le. Ha az alkalmazás háttérbe kerül, lefut a *pauseApp* metódus, amely felszabadítja az összes felszabadítható erőforrást. Ezeket az erőforrásokat visszakapja a MIDlet, ha az alkalmazás a háttérből ismét aktív állapotba kerül: ilyenkor ismét lefut a *startApp*.

A *destroyApp* véglegesen felszabadítja az alkalmazás által lefoglalt erőforrásokat, és az alkalmazás futása szabályosan befejeződik.

11.2.3 Grafikus felületű alkalmazások fejlesztése mobil eszközökön

A MIDP-ben kétféle lehetőség van grafikus felületű alkalmazások fejlesztésére. A kétféle megközelítés együttesen is felhasználható az alkalmazásfejlesztésben, azzal a megkötéssel, hogy a kétféle megjelenítési mód nem keverhető.

A magasabb szintű megközelítés kifejezetten olyan grafikus komponenseket ír le, amelyeket mobil eszközökön használhatunk, míg az alacsonyabb szintű

megközelítés lehetővé teszi, hogy a programozó közvetlenül rajzoljon a mobil eszköz képernyőjére vagy kijelzőjére.

Utóbbi megoldáshoz a MIDP-ben leírt *Display* osztályból példányosított objektumra van szükségünk. A *Display* a kijelző fizikai megfeleltetése, és minden MIDletben létrejön egy objektumpéldány. A *Display*-ből létrehozott objektumban a *Displayable* osztályból léterhozott objektumok helyezhetők el, ennek az osztálynak két leszármazottja van: a *Screen* (ez a magasabb szintű API) és a *Canvas* (ez az alacsonyabb szintű) osztályok.

A *Screen* osztályai hasonlítanak az AWT és a Swing osztályaihoz, olyan komponensek vannak benne leírva, mint a *Form*, a *TextBox*, vagy a *List*. A komponensosztályok egyes objektumai az AWT-ben látott *add* metódussal szemben itt az *append* metódussal helyezhetők el az űrlapokon. A formokhoz is tartoznak *Layout* Managerek, így a formokon a komponensek elhelyezése az AWT-ben megismertekhez hasonlóan automatikusan történhet.

A *Canvas* osztály használata mellett ezek az elemek nem csatolhatók a származtatott osztályokhoz, viszont létezik a *Canvas* osztályban a *paint* metódus, amelynek segítségével a megjelenő képet mi magunk alkothatjuk meg.

11.2.4 A Java ME használata NetBeansben

A NetBeans weboldalan letölthető a NetBeans IDE olyan változata, amely kifejezetten a Java ME implementációhoz készült. Ha már telepítettük a NetBeans Java SE-hez készült változatát, a NetBeans Tools menüjében a Plugins menüpontban az Available Plugins szekcióban megtalálható a szükséges plugin (Java ME SDK Tools), amely egyszerűen telepíthető.

Ez a plugin tartalmaz egy Visual Mobile Designer című eszközt, amely segítségével hagyományos alkalmazások készíthetők, illetve egy *GameBuildert*, amely kifejezetten játékok készítésére való.

A VMD négy képernyőt kezel. A *Source* nézet az alkalmazás forráskódját mutatja. A *Screen* nézetben a *Screen* ősosztályú objektumok között tudunk válogatni. A harmadik nézet a *Flow*, amely a program vezérlésének struktúráját mutatja. A negyedik, az *Analyzer* nézet a program ellenőrzését teszi lehetővé.

A *GameBuilder* segítségével egy játékhoz szükséges erőforrásfájl készíthető el, és három elemet tartalmaz. A *Scene* elem a *LayerManager* objektumhoz tartozó elemeket tartalmazza. A *Layer* osztály elemei a képernyőn a kirajzolás sorrendjében jelennek meg, a magasabban elhelyezkedő rétegek elemei elta-
karják az alacsonyabban elhelyezkedő rétegek tartalmát.

A második elem a *Tile Layer* elemeket tartalmazza, ezek segítségével a *TiledLayer* osztály objektumait állíthatjuk elő. Ezeket a „csempéket” úgy képzelhetjük el, mint egy térkép alkotóelemeit. Tároljuk őket egy helyen, és a térképen már csak hivatkozni kell rájuk.

A GameBuilder harmadik eszköze a Sprite Editor. Ennek segítségével animált figurákat készíthetünk úgy, hogy elkészítjük az animációk fázisait.

A mobil alkalmazásfejlesztés annyira szerteágazó (és olyan érdekes) téma, hogy egy önálló jegyzet tárgya is lehetne. Mi csupán betekintést kívántunk nyújtani. Egy alapos áttekintés megtalálható például a NetBeans weboldalán: <http://netbeans.org/features/javame/>.

11.3 ÖSSZEFOGLALÁS, KÉRDÉSEK

11.3.1 Összefoglalás

Jegyzetünk záró leckéjében a Java nyelv speciális felhasználásának két eszközét ismertük meg. A lecke első részében a weboldalakon megjeleníthető Appletekről esett szó, a második részben pedig a mobiltelefonokon való alkalmazásfejlesztés legalapvetőbb kérdéseit tekintettük át. A jegyzet terjedelmi korlátai miatt mindkét rész meglehetősen rövid lett, de talán elegendő mindenki számára az elinduláshoz a további ismeretek megszerzése útján.

11.3.2 Önellenőrző kérdések

1. Mi az az Applet?
2. Milyen eltérések vannak egy Java-alkalmazás és egy Applet forráskódja között?
3. Milyen őssztály(ok)ból származtatjuk az Appleteket?
4. Hogyan lehet egy Applet futásának eredményét megnézni?
5. Milyen szabványok léteznek a Java mobil eszközökön való felhasználásához?
6. Mi az az API?
7. Hogyan támogatja a NetBeans a mobil alkalmazásfejlesztést?

12. ÖSSZEFOGLALÁS

12.1 TARTALMI ÖSSZEFOGLALÁS

A jegyzet célja a Java programozási nyelv alapvető eszközeinek bemutatása volt. Ahogyan azt a jegyzet bevezetőjében is leírtuk, jó programozóvá csak az válhat, aki programozik. A programozás olyan tudomány, amelyet csak rengeteg gyakorlás és tapasztalat során lehet megfelelőképpen elsajátítani. Ehhez nem elég elolvasni egy vagy több kötetnyi könyvet, hanem le kell ülni a billentyűzet mögé és programokat kell írni.

A jegyzet olyan hallgatók számára készült, akik eddig nem foglalkoztak programozással, és a programozást nem is tekintik tanulmányaik legfontosabb céljának. Ez a jegyzet ugyanis nem (elsősorban) programozó hallgatók számára készült, hanem olyan számára, akik szeretnének megismerkedni a programozás alapjaival, de elsősorban csak azért, hogy kiegészítsék egyéb informatikai ismereteiket a programozás tudományával.

Érhet bennünket az a vád, hogy a Java nem feltétlenül alkalmas kezdő programozási nyelvnek, ugyanakkor tudatában kell lennünk annak is, hogy esetleg egy régebbi, elavultabb programozási nyelvet használva kezdő nyelvként a hallgatók olyan programozási habitust vesznek fel, amely később akadályozza őket a modern nyelvek lehetőségeinek megismerésében. Ugyanakkor nem volt célunk a jegyzet elkészítése során a maximális részletességre törekedni. Elsősorban elméleti, programozáselméleti kérdések körében nem feltétlenül akartunk mindent teljes pontossággal leírni. A programozás és különösképpen az objektumorientált paradigma megértéséhez mindenkinek egyénileg kell felfedezni egy nyelv lehetőségeit. Ahogyan minden beszélt nyelv nyelvtana különbözik a másiktól, nincs két azonos jellemzővel bíró programozási nyelv sem. Az OOP-t is különböző szinten támogatják a különböző nyelvek. Mivel a jegyzetünk a Java nyelv bemutatását célozta meg, elsősorban a Java OOP-s lehetőségeire koncentráltunk. Más OOP nyelvek sajátosságait az alapok ismeretében már könnyebb lesz elsajátítani mindenki számára.

A programozónak nem elegendő egy nyelv utasításkészletét ismernie. Több nyelvet is ismernie kell, hogy ki tudja választani egy probléma megoldásához – illetve a megrendelő igényeihez – leginkább megfelelő eszközt. Szüksége van továbbá algoritmikus gondolkodásra, absztrakciós képességre, a kezdőknek pedig legfőképp türelemre és kitartásra. Kezdetben pedig mindenki kezdő.

A Java azért is előnyös választás, mert rendkívül népszerű (amellett ingyenesen hozzáférhető is). A világhálón milliószám találhatunk példákat és segéd-

leteket. Ezeket úgy tudjuk a legjobban felhasználni, ha megpróbáljuk őket megérteni is, nemcsak bemásolni a saját programjainkba. Talán minden egyéb mellett mások, jó és rossz példáiból lehet a legtöbb tudást megszerezni.

A jegyzetben a következő leckéket tekintettük át:

1. Bevezetés a magas szintű programozási nyelvek használatába
2. Bevezetés a Java programozási nyelv használatába
3. A Java nyelv alapvető elemei
4. Vezérlési szerkezetek és metódusok. Kivételkezelés
5. Tömb, karakterlánc és adatbevitel
6. Obejktumorientált programozás I. – osztályok és objektumok. Java osztálykönyvtár, csomagok
7. Obejktumorientált programozás II. – hozzáférési tagok, objektumok kezelése
8. Obejktumorientált programozás III. – öröklés, polimorfizmus, absztrakt osztályok
9. Grafikus felhasználói felület készítése Java programokban, eseménykezelés I.
10. Grafikus felhasználói felület készítése Java programokban, eseménykezelés II.
11. Java appletek. Fejlesztés mobil eszközre

Őszintén reméljük és kívánjuk, hogy a jegyzetünkkel hozzájárulunk minden kezdő programozó sikereihez.