Online tananyag

Interdiszciplináris tudományok

Sorozatszerkesztő: Dr. Balázs László

28.

Somlyai Pál, Kógelmann Gábor:

Bevezetés a Programozásba (Java nyelv)

DUNAÚJVÁROSI EGYETEM
UNIVERSITY OF DUNAÚJVÁROS

Tartalom

Kógelmann Gábor:
BEVEZETÉS A PROGRAMOZÁSBA
INF-501

1. fejezet

Bevezetés a programozásba

Programozási alapismeretek

Somlyai Pál

Dunaújvárosi Főiskola

2015

Tartalom

Bevezetés a programozásba – Programozási alapismeretek 1

1. Programozás fogalma és eredménye

A programozás egy szakemberek által végzett munkafolyamat, melynek eredménye az elkészült szoftver (~alkalmazás/számítógépes program).

Néhány példa a szoftverek főbb típusaira:

– Számítógépes- (Microsoft Word, Firefox)

– Mobil alkalmazás (Viber, Angry Birds)

– Operációs rendszer (Windows, Linux, OS X, Android, iOS)

– Weboldal
  – A komolyabb weboldalak mögött bonyolult logika áll, amelyik végrehajtja a felhasználó által kiadott parancsokat (pl.: Facebook frissítés, e-mail küldés, Instagram képfeltöltés)

– Beágyazott rendszer
  – Pl. Járművek fedélzeti számítógépe

Mivel a programozás eredménye egy szoftver, ezért szoftverfejlesztésnek is nevezzük[1]. A szakember, aki a programozás, vagy szoftverfejlesztés folyamatát elvégzi: a programozó, vagy szoftverfejlesztő.

1 Valójában a szoftverfejlesztés a programozáson kívül más feladatokat is magába foglal (pl.: tervezés).

2. A programozás célja

A programozás célja mindig egy, az igényeket kielégítő módon- és hibátlanul működő program előállítása. Az igényeket a program megrendelője fogalmazza meg a programozó(k) felé.

Az igények a következőekhez hasonlóak lehetnek:

– A program a megadott e-mail címre, küldje el a megadott tárggyal, a megadott üzenetet, miután a „Küldés” feliratú gombra kattintunk.

– Ellenőrizze a megadott adatok helyességét.

– A program nyújtson lehetőséget képek hozzáadására.

– Legyen világosszürke háttere, a betűszín legyen fekete, a díszítőelemek pedig élénkzöld színűek.

– Csak regisztrációt és bejelentkezést követően lehessen használni.

– Stb…

Az itt felsorolt igények megfelelhetnek egy üzenetküldő weboldal, vagy programmal támasztott igények részletének.

3. Programozás folyamata

A programozás kezdetén a programozó megkapja az igényeket tartalmazó specifikációt, melynek ismeretében hozzáláthat a tervezéshez. A tervezés során egyéb tervek mellett[2] elkészíti az algoritmust.

3.1 Algoritmus

Az algoritmus egymást követő utasítások (~műveletek) sorozata, amelyek együttesen egy adott problémát oldanak meg, azaz egy célt valósítanak meg. Vegyünk egy egyszerű példát: amely legyen egy négyműveletes számológép algoritmusa. Az algoritmusokat általában speciális algoritmizálási eszközökkel készítjük el (később), azonban most szöveges formában fogjuk megfogalmazni:

1. Kérd be az első számot, a műveleti jelet, illetve a második számot!

2. Ha a műveleti jel:

    a. +: a két szám összegét írd ki a képernyőre!

    b. -: a két szám különbségét írd ki a képernyőre!

    c. *: a két szám szorzatát írd ki a képernyőre!

    d. /: a két szám hányadosát írd ki a képernyőre!

    e. A fentiektől különböző: írd ki a képernyőre, hogy hibás művelet, majd lépj az 1. pontra!

Az algoritmus műveletei minden esetben adatokkal dolgoznak, amelyeket a megfelelő forrásból be kell olvasnunk.

2 Az egyéb tervekkel és tervezési módszerekkel későbbi tárgyaink során fogunk foglalkozni.

Az adat a forrása szerint lehet:

– A felhasználó által beírt-

– Egy fájlból beolvasott-

– Hálózaton keresztül fogadott adat

– Stb..

Az algoritmus elkészítése lényegében a cél eléréséhez szükséges lépések összegyűjtése, megfelelő sorrendbe rendezése, valamint a különböző feltételek teljesülése esetén végrehajtandó, feltételes lépések megfogalmazása.

A végrehajtandó lépések ismeretében a programozó a forráskód elkészítésével folytatja.

3.2 Forráskód

A forráskód a számítógép által végrehajtandó utasításokat tartalmazza szöveges formában és a megfelelő sorrendben.

Rendben. Írásban kell kommunikálni a számítógéppel. De melyik nyelve(ke)t ismeri? Magyarul, angolul, esetleg németül próbáljuk meg megértetni vele a feladatát? A helyzet sajnos nem ilyen egyszerű. A jelenlegi számítógépek nem rendelkeznek tudattal[3], nincs szókincsük, amelyet lehetne fejleszteni, emellett nem értik meg az általunk használt nyelvtani szerkezeteket.

Akkor melyik szavakat értik meg? Van valamilyen kifejezés-lista, amiben szereplő szavak jelentését képesek értelmezni? Igazából a valóság nagyon közel áll az előző kérdéshez. A számítógépek gépi kódú nyelveken képesek az utasításokat értelmezni.

3 A sci-fi művektől eltérően.

3.2.1 Gépi kód

A gépi kód a számítógépeknek szóló utasítások, valamint a műveletekhez szükséges adatok összessége, melyeket végrehajtva a számítógép a kívánt műveleteket végzi el. Például add össze a négyet és a tizenkettőt!

A gépi kód az egyetlen nyelv, amit a számítógép központi utasítás-feldolgozó egysége (processzor, CPU) végre tud hajtani. A gépi kódban az utasítások általában kettes- vagy tizenhatos számrendszeren alapuló számokkal kerülnek ábrázolásra. A gépi kódot fájlok tartalmazzák, melyeket a számítógép beolvas a memóriájába, majd végrehajtja a gépi kódú utasításokat.

Minden processzorcsaládnak[4] saját kifejezés-listája (utasításkészlet, azon utasítások listája, amelyeket a processzor el tud végezni) van, amelyet minden operációs rendszer picit másképp használ. Tehát ha van egy gépi kódunk, amit Windows rendszeren, Intel Pentium Core i3-as processzorral futtatunk, azt nem fogjuk tudni lefuttatni egy AMD Phenom II-es processzorral szerelt Linux alapú rendszeren.

Emellett az utasításkészletük viszonylag alacsony számú utasítást tartalmaz. Ezek az utasítások elemi problémák megoldására hivatottak (pl.: két egész szám összeadása), valamint a programozónak tökéletesen tisztában kell lennie a programozott processzor felépítésével és működésével, így egy viszonylag egyszerű feladatot is nehéz lehet megvalósítania. És akkor még csak egy processzorcsaládra készítettük el a programkódot! Folytathatjuk a többi processzor gépi kódjának megírásával, amiken futtathatóvá szeretnénk tenni az alkalmazásunkat. Képzeljük el, vajon hány processzoron szeretné tudni futtatni a Microsoft az Office programcsomagot? Valószínűleg mindegyiken, amelyik személyi számítógépekre lett tervezve.

A számítástechnika kezdetén használtak csak gépi kódot, amit később felváltottak a második generációs alacsony szintű programozási nyelvek.

3.2.2 Alacsony szintű programozási nyelvek

Ezeknél a nyelveknél a CPU nyelvétől való elvonatkoztatás[5] kismértékű (vagy egyáltalán nincs elvonatkoztatás), azaz ezek a nyelvek állnak legközelebb a gépi kódhoz.

Az „egyáltalán nincs elvonatkoztatás” azt jelenti, hogy gépi kódról beszélünk. Tehát a gépi kód az alacsony szintű programozási nyelvek közé tartozik, valamint azon belül az első generációs nyelvek közé sorolható (1GL, first-generation programming language).

4 A processzorcsalád megegyező utasításkészletű processzorokat tartalmazó csoport. Például egy Intel Pentium Core i5-ös és egy AMD FX processzor két teljesen eltérő processzorcsaládba tartozik. Emellett gyártónként és több processzorcsaládot különböztethetünk meg: tehát nem tartozik minden AMD processzor egy processzorcsaládba.

5 Az elvonatkoztatás során egy tárgy, dolog, jelen esetben pedig egy utasítás számunkra nem lényeges tulajdonságait/részleteit elhagyjuk és csak a számunkra lényeges részletekkel foglalkozunk tovább.

A kismértékű elvonatkoztatás például azt jelenti, hogy szöveges utasításokat[6] (karakterekből, azaz betűkből, számokból és speciális írásjelekből álló utasításokat) adhatunk ki a számokban ábrázolt gépi kódú utasítások helyett. Ezek az utasítások általában a processzor utasításainak megnevezéseinek rövidített változatai. Ezeket a nyelveket második generációs alacsony szintű nyelveknek nevezzük (2GL, second-generation programming language), melyeket a különböző assembly nyelvek alkotnak. Az olvashatóbb írásmód mellett ezek a nyelvek néhány alapvető memóriakezelési és egyéb feladat terhét is levehetik a vállunkról. Azonban ezeknek a nyelveknek a használata során is nagyon mély ismeretek szükségesek a használt processzortípusról.

Az alacsony szintű programozási nyelvek utasításkészletének utasításszáma azonban még mindig alacsony, így rengeteg egyszerű problémát nehézkes munkával kell megvalósítanunk. Egy karakterlánc (szöveg) kezelése, valamint fájlba írás, illetve olvasás megvalósítása nagy hozzáértést, körültekintést, valamint időt követel a programozótól. Ennek következtében kifejlődtek a magas szintű programozási nyelvek.

3.2.3 Magas szintű programozási nyelvek

A magas szintű programozási nyelvek elvontabbak, egyszerűbben használhatóak és segítségükkel több platformra (processzorra és/vagy operációs rendszerre) tudunk programot készíteni. Szokás ezeket a nyelveket harmadik generációs nyelvekként (3GL, third-generation programming language) említeni.

Ezen speciális programnyelvek kifejezés-készlete sokkal több elemet tartalmaz (bővebb a szókincsük), így egy bonyolultabb utasítást egyszerűbben ki tudunk adni a számítógépnek, feltéve hogy az említett utasításra van kifejezésünk az adott programnyelvben. Vegyük a következő példát: az autóvezetés-oktató utasítja a mellette, a volán mögött ülő tanulóját: „Induljunk!”. Sokkal egyszerűbb jeleznie a tanuló feladatát, mivel a tanuló (remélhetőleg) ismeri az elindulás folyamatát, így nem kell részletesen kiadni az utasításokat, mint például: „Nyomd be a kuplungot, és tartsd lent! Indítózz! Kapcsolj világítást! Egyenletesen nyomd le a gázpedált és közben egyenletesen engedd fel a kuplungot!”. Egy szó, a több mondattal szemben. És rengeteg apróságra ki sem tért az oktató, ami fontos lehet az indításkor: „Melyik a gáz- és a kuplung pedál? Hol van a gyújtáskapcsoló? Kösd be a biztonsági övet! Várd meg az előttünk elhaladó buszt!” Kis túlzással csak az „Induljunk!” utasítást kell kiadnunk egy magas szintű nyelv használata esetén, míg a többi utasítást kell felsorolnunk egy alacsony szintű nyelv esetén, amennyiben a programnyelvünk ismeri az „Induljunk!” utasítást.

6 Melyek általában rövidítések.

3.2.4 Fordítás

Az előző fejezetekből kiindulva megállapíthatjuk: hogy a forráskód tehát egy adott programnyelven készült szöveges utasításlista.

Egy pillanat! Korábban letisztáztuk, hogy csak a gépi kódú utasításokat képesek értelmezni a számítógépek. Akkor egy nem gépi kódú programnyelv utasításait hogyan képes végrehajtani? A válasz nagyon egyszerű. Mi történik a világban, ha két eltérő nyelvet ismerő személy szeretne kommunikálni? Egy tolmács segítségét kérik. Ugyan ez történik a mi esetünkben is. A programkódunk utasításait egy fordító (azaz tolmács), fordítja le gépi kóddá, amit már képes értelmezni a számítógépünk.

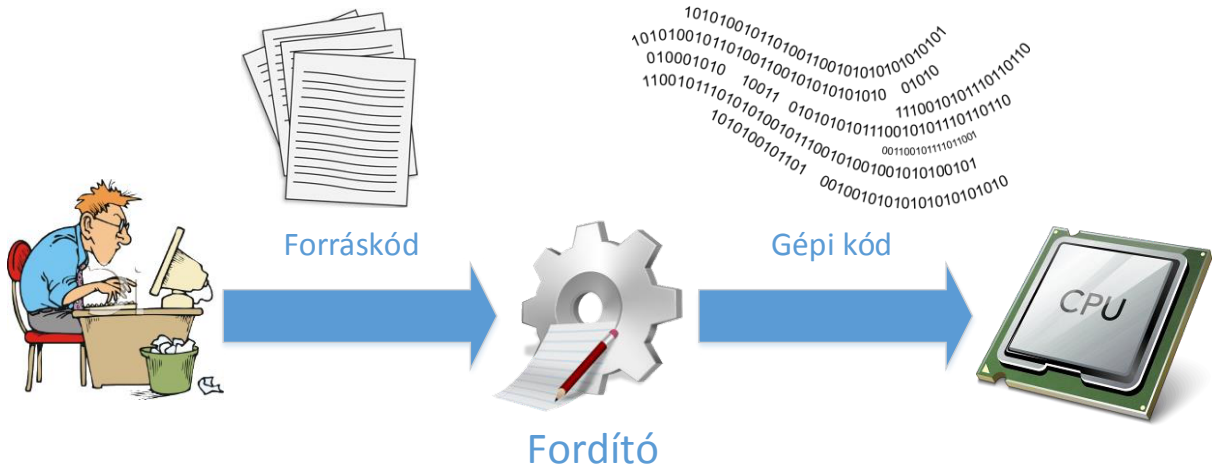
A fordítás (tolmácsolás) lépései a következőek:

1. A programkódunkat eltároljuk egy forrásfájlban (szöveges fájlban).

2. A fordítóprogramnak kiadjuk az utasítást, hogy tolmácsolja gépi kódra a forráskódunkat.
   Eredménye egy futtatható állomány (futtatható fájl).

3. Az előálló gépi kódot tartalmazó futtatható állományt (pl.: .exe kiterjesztésű fájl) már képes lefuttatni a számítógépünk.

1. ábra – Fordítás



Az első generációs alacsony szintű programnyelveknek nincs szükségük fordításra, hiszen ezeket nevezzük gépi kódnak. A második generációs alacsony szintű (assembly) programnyelveknek minimális fordításra van csak szükségük. Ezek fordítóját assemblernek nevezzük. Végeredményben, amikor fordítóról beszélünk, egy magas szintű programozási nyelv fordítóprogramjáról beszélünk.

Tehát a magas szintű programnyelveknél említett magas szintű elvonatkoztatás azt jelenti, hogy elhagyjuk a gépi kód készítéséhez szükséges többletismeretet és az utasításokat általános módon fogalmazzuk meg, amelyeket a fordító egészít ki többletinformációval és alakít át gépi kóddá. Tehát a magas szintű programnyelvek nem állnak az alacsony szintű programnyelvek felett, egyszerűen egy magas szintű programnyelv fordítója több többletinformációval ruházza fel a forráskódot, illetve jobban részletezi az utasításokat, mint egy alacsony szintű nyelv fordítója.

Egy fordítóprogram általában több platformot ismer, amelyek közül bármelyikre képes lefordítani a forráskódunkat. Így a legtöbb magas szintű programnyelv platformfüggetlen, azaz segítségükkel a legtöbb processzorra és operációs rendszerre lehetséges gépi kódot fordítani belőlük. A most említett programfuttatási folyamat az úgynevezett natív nyelvekre igaz.

3.2.5 Programozási nyelvek generációi

Tekintsük át még egyszer a programozási nyelvek generációit:

- Első generációs nyelvek
  - gépi kód
    - a kód számsorozatokból áll
  - nincs elvonatkoztatás
  - magas szintű hardware ismereteket szükségelnek
  - azonban a processzor minden funkcióját maximálisan ki tudják használni
  - leggyorsabb futású programok
  - lassú- és nehéz fejlesztés
  - manapság nem készítenek közvetlenül gépi kódot a programozók
    - fordítókat használnak

– Második generációs nyelvek
  – assembly nyelvek
  – szöveges kód
    – rövidítéseket használ
  – a gépi kódhoz ezek a nyelvek állnak legközelebb
    – alacsony szintű elvonatkozás a gépi kódtól
    – ugyancsak magas szintű hardware ismereteket szükségelnek
    – azonban a processzor minden funkcióját maximálisan ki tudják használni
  – leggyorsabb futású programok
  – lassabb- és nehezebb fejlesztés
  – speciális feladatoknál alkalmazzák őket

– Harmadik generációs nyelvek
  – magas szintű programnyelvek
    – magas szintű elvonatkoztatás a gépi kódtól
  – elméletben lassabban futó programok
    – azonban a mai mikroprocesszor architektúrák bonyolultsága miatta
      – a modern fordítóprogramok gyakran hatékonyabb gépi kódot hoznak létre, mint az alacsony szintű nyelveken írt gépi kód
  – gyorsabb- és egyszerűbb fejlesztés
    – egyszerűbb memória- és adatszerkezet kezelés
    – egyszerűbben használhatóak
    – fejlett hibakeresési eszközök

3.2.6 Programozási nyelvek típusai fordítás- és futtatás módja szerint

A programozási nyelveket rengeteg szempont szerint lehet csoportosítani, viszont számunkra jelenleg csak a fordítás- és futtatás módja szerinti csoportosítás érdekes.

A programozási nyelveket fordításuk szerint két csoportba soroljuk:

– fordított- és

– értelmezett nyelvek csoportjába.

A fordított nyelvek esetén a forráskódból egy alkalommal előállítjuk az általában gépi kódot (lásd: Menedzselt kód), majd minden futtatáskor ezt a gépi kódot futtatjuk.

Az értelmezett nyelvek esetében úgy tűnhet: a forráskódot futtatjuk. Végeredményben legtöbbször a programnyelv értelmezője (interpreter) hajtja végre közvetlenül a forráskódban (szkriptben) található utasításokat. Ritkábban a forráskódot a futtatás parancs kiadásakor fordítja le a szkriptnyelv értelmezője, majd futtatja az előállt köztes[7], vagy gépi kódot. Ez a folyamat minden futtatáskor megtörténik.

A fordított nyelveket ezen belül ismét két csoportba soroljuk:

– natív- és

– menedzselt kódú nyelvek csoportjába.

A natív kódú nyelvek fordítása során az előző fejezetben leírt folyamat kerül végrehajtásra, azaz a forráskódból gépi kód fordul, amit a processzor közvetlenül értelmezni képes. Természetesen egy alkalommal fordítjuk le a gépi kódot, majd minden alkalommal azt futtatjuk.

A menedzselt kódú nyelvek esetében a fordítás során egy úgynevezett köztes kód áll elő, amit a futtatókörnyezet futás közben fordít gépi kódra. Lényegében a fordított- és az értelmezett nyelvek keverékéről van szó. Általában a futtatókörnyezet előfordít, azaz lefordít egy adag gépi kódot, elkezdi futtatni, majd futás közben tovább fordít, majd futtat, és ezt ismétli, ameddig végre nem hajtja a teljes programot.

7 A köztes kód nem gépi kód. Egy futtatókörnyezetnek szóló köztes kód, amit a futtatókörnyezet hajt végre.

Azonban fontos megjegyezni, hogy a mai futás idejű fordítók (JIT, Just In Time) kellően optimalizált (azaz a lehető leggyorsabban futó kódot készítenek), így nem jelenti azt, hogy sokkal lassabb futásúak, mint natív kódú társaik.

Jó pár éve komoly vita folyik, hogy a natív, vagy a menedzselt kód a „legjobb". Véleményem szerint az adott programozási feladathoz kell kiválasztani a nyelvet, aminek egy nagy számításkapacitást igénylő feladat esetében inkább natívnak kell lennie, míg egy bonyolult hálózati feladatokat is ellátó, grafikus felülettel rendelkező, komoly üzleti logikát tartalmazó alkalmazás fejlesztésénél megfelelőbb lehet egy menedzselt nyelv.

Előnyök és hátrányok

Fordítás	Futtatás	Előny	Hátrány
Szkript nyelvek		Könnyen módosítható (nem igényel fordítást)	Általában lassabb, mint egy fordított nyelv
Fordított nyelvek	**Natív**	Gyorsabb futás	Nehezebb fejlesztés
	Menedzselt	A natív nyelvekhez képest lassabb futás, a szkript nyelvekhez képest gyorsabb.	Gyorsabb fejlesztés

4 Miért éri meg megtanulnom programozni?

A rövid és tapintatlan válaszom az, hogy különben nem fogod teljesíteni ezt a tantárgyat. De természetesen nem vagyok ennyire nyers, ezért felhozok pár érvet, hogy miért is jó programozónak lenni.

Szeretném felhívni a figyelmeteket, hogy az informatikai cégeknek rettentő nagy szüksége van jó programozókra, mind Magyarországon, mind külföldön[8]. Magyarországon a jegyzet készítésének pillanatában több ezer nyitott és meghirdetett programozói álláslehetőség létezik. A programozás, mint munkavégzés – általában – roppant kreatív tevékenység. Annak, aki szeret problémákat megoldani, kifejezetten jó hívatás lehet. A programozási hivatás iránti figyelmetek felkeltésének utolsó érve az általában kiemelkedően magas fizetés[9]. A javaslatom számotokra az, hogy tegyetek egy próbát a programozás elsajátítására vonatkozóan, ami lehetőleg ne csak egy-két hétig, vagy hónapig tartson, hanem szenteljetek rá pár félévet, hátha megtaláljátok a hivatásotokat[10], úgy ahogyan nekem sikerült korábban.

Azonban ha nem szeretnél programozó lenni, akkor is szükséged lehet a programozási ismeretekre, például a következő munkakörökben:

– Rendszerüzemeltető, rendszergazda
  – Ki kell, hogy ábrándítsalak, de a legtöbb rendszerüzemeltető/rendszergazda tetemes mennyiségű szkriptet készít munkája során, hogy ne kelljen minden konfigurációs feladatot, minden gépnél újra-és újra elvégeznie. A szkriptek megírása során pedig szükség van programozási ismeretekre.

– Grafikus felület fejlesztő, Web designer
  – Ők inkább a programok és weboldalak kinézetével foglalkoznak munkájuk során, de egy, vagy több programozóval kell együtt dolgozniuk, így nem árt tudniuk, hogy az általuk tervezett/elkészített felület mögötti programlogikát hozzá lehet-e illeszteni a megjelenítéshez.

– Tesztelő
  – A szoftverek nagy része borzasztó bonyolult. A fejlesztőik csak akkor jelenthetik ki, hogy minden helyes működik benne, ha le van ellenőrizve az összes funkció, a lehető legtöbb használati paraméterrel. Emiatt a nagy fejlesztőcégek külön tesztmérnököket alkalmaznak, melyeknek nagy részük automatizált teszteket készítenek, végeredményben programkódból tesztelnek programkódot.

Az itt felsorolt szakmákon kívül rengeteg másik (általában informatikai szakma) létezik, ahol előny esetleg elvárás a programozás ismerete, emiatt semmiképpen nem fog hátrányt jelenteni a munkakeresés során, ha rendelkezel programozási ismeretekkel.

8 http://pcworld.hu/eletmod/10-ezer-informatikus-kerestetik.html

9 http://gazdasagtv.hu/tech/2015-05-14/boven-atlag-felett-keresnek-a-programozok-es-hiany-is-van-beloluk
A cikkben található egy hwsw.hu portálon közölt cikk hivatkozása, amely a hír alapjául szolgál. Érdemes átböngészni az érdeklődőknek.

10 Ami többet jelent, mint a munkád, vagy szakmád.

Bevezetés a programozásba

Java nyelv alapjai

Somlyai Pál

Dunaújvárosi Főiskola

2015

Tartalom

1. Java dióhéjban

Minden programozási nyelv megtanulását a nyelv alapjainak megismerésével kell kezdenünk. Most sem teszünk másképpen, lássunk is hozzá.

1.1 Történet

Egy rövid történeti áttekintéssel kezdünk, hogy el tudjuk helyezni a Javát a programozás világában.

1. **1995. május 23:** A Sun Microsystems kiadja a Java programnyelv első hivatalos verzióját

2. 2**000. november:** Az amerikai College Board bejelenti, hogy a 2003-as évtől kezdve az informatikai felvételi vizsgák a Javán alapulnak.
   – A Collage Board egy több mint 5900 iskolát, főiskolát, egyetemet és más oktatási intézményt tömörítő szervezet az Egyesült Államokban.

3. **2004. június:** Egy szakmai hírportál bejelenti, hogy a Java programozókra vonatkozó igény 50 %-kal meghaladja a C++ programozókra vonatkozót.
   – A C++ egy magas szintű, natív kódú, fordított nyelv, amelyet tekintettek kiindulási pontnak a Java tervezésekor. Egy nagyon nagy múlttal rendelkező nyelv, amelyet még manapság is rengetegen használnak, főként, de nem kizárólag hardverközeli és számításigényes feladatokra.

4. **2010. január:** Az Oracle Corporation felvásárolja a Sun Microsystems-t, bevezetve ezzel a Java technológiát az Oracle termékeknél.

5. **2010. június:** Az eWeek „A 10 legjobb programozási nyelv, amely segít, hogy megőrizd a munkád” elnevezésű listáján a Java az első helyre kerül.

6. **2011. május:** A Java több mint 1,1 milliárd asztali számítógépen fut.

Ebből a rövid történeti áttekintésből látható, hogy a Java egy nagyon sikeres nyelv, amelyet manapság is rengeteg számítógépen használnak. Folytassuk a Java nyelv részletesebb megismerésével, hogy elsajátítsuk azt a tudást, aminek segítségével körülbelül 1,1 milliárd asztali számítógépre lehetünk képesek számítógépes alkalmazásokat készíteni.

1.2 Java programozási nyelv

A Java (ejds: Jáva, angol környezetben: Dzsávö) egy magas szintű, harmadik generációs, menedzselt kódot használó, fordított nyelv. Legfőbb ereje a platform-függetlenségben és a hihetetlen egyszerűen használható „beépített" megoldásokban (programkönyvtárak[1]) rejlik. A nyelv filozófiája a következő: „Write once, run anywhere." (azzaz „Egyszer megírod, mindenhol futni fog."). Az előző leckében körvonalazódott számunkra a menedzselt kód fordításának- és futtatásának menete. Pontosítsuk, hogyan történik mindez a Java esetében, hiszen ez az előbb említett filozófiájának az alapja.

A menedzselt kód fordításának és futtatásának általános menete a következő:

1. A forráskód fordítása a fordító segítségével, melynek eredménye a köztes kód.

2. A köztes kód fordítása „futásidőben" Just In Time módon, melynek eredménye a gépi kód.

3. A gépi kód futtatása a processzoron.

Ez az általános menet a Java esetében sem működik másképpen, viszont a köztes kód futtatására hivatott JIT fordító az összes nagyobb operációs rendszert és általános processzort támogatja (azaz képes számukra megfelelő gépi kódot készíteni a Java köztes kódból), így ténylegesen érvényesül az „Egyszer megírod, mindenhol futni fog." elv.

Itt az ideje az általános kifejezések Java megfelelőinek megadására.

1. ábra. Menedzselt kód Java megfelelők

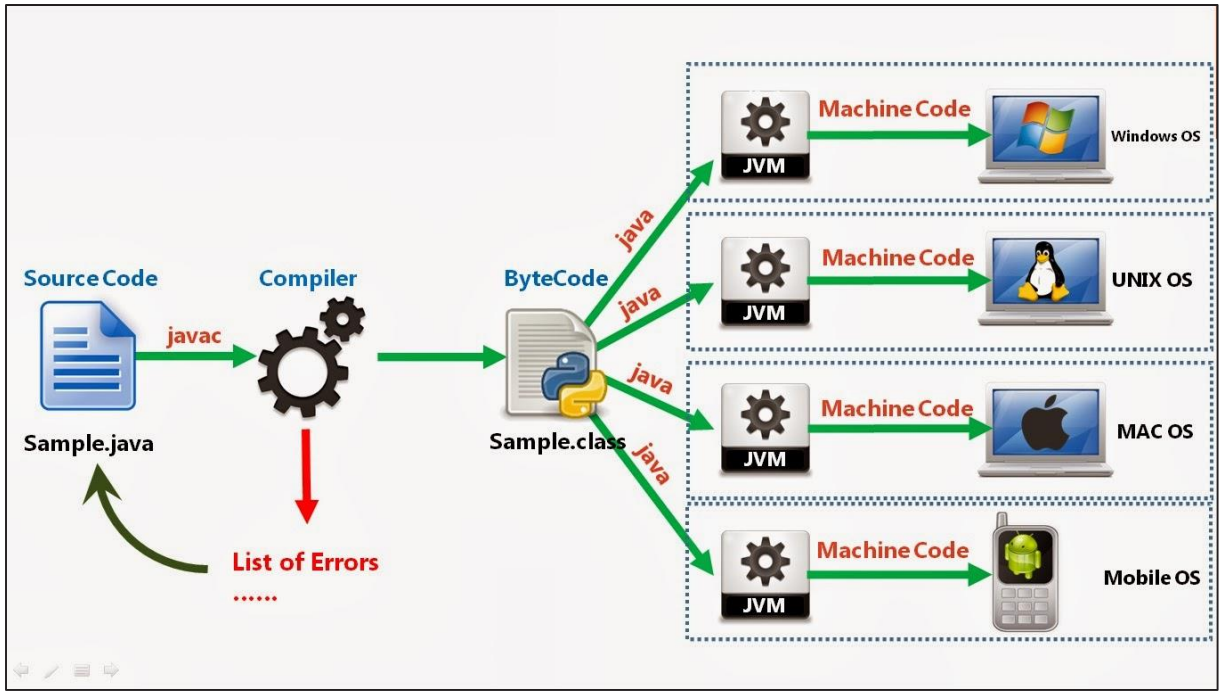
Általános	Java megfelelő
Köztes kód	(Java) bájtkód (bytecode)
Futtatókörnyezet	Java Futtató Környezet (Java Runtime Environment, JRE)
	Java Virtuális Gép (Java Virtual Machine, JVM)

1 Későbbi tárgy keretein belül fogjuk részletesen megismerni és használni őket, azonban pár elemet már ebben a tárgyban is fel fogunk használni belőlük.

A JRE tartalmazza a JVM-et, amelyik felelős a bájtkód gépi kódra fordításáért. Emellett a JRE tartalmaz még olyan eszközöket, amelyek segítségével tudjuk „elindítani” a Java programjainkat (bájtkódot), mivel ezek nem .exe kiterjesztésű futtatható állományok (.class, .jar), valamint egyéb eszközöket, amik az általánosan felmerülő feladatok megvalósításáért felelős „utasításokat” tartalmazzák (pl.: fájl beolvasás/írás, hálózati kommunikáció, alapvető matematikai műveletek, stb…).

Nézzük meg, hogyan lesz a Java forráskódból gépi kód (*2. ábra,* az angol kifejezések fordítását az ábra alatt találod), viszont először szükséges megadnom néhány további információt a Java programok fejlesztésével kapcsolatban. A Java forráskódok .java kiterjesztésű fájlok. A bájtkód .class kiterjesztésű fájl. A forráskód bájtkódra fordításának parancsa a javac (Java Compile, Java Fordítás) parancssori utasítás, a bájtkód futtatásáé pedig a java parancssori utasítás.

2. ábra. Java programok végrehajtása



Az ábrán szereplő angol szakkifejezések magyar megfelelői:

– Source Code: forráskód

– Compiler: Fordító

– List of Errors: Hibalista

– ByteCode: bájtkód

– Machine Code: gépi kód

Összegezve az ábrát látható, hogy a Java programok végrehajtásának lépései a következőek:

1. Forráskód elkészítése (.java kiterjesztésű fájlok).

2. Bájtkód fordítása a javac parancs kiadásával a fordító segítségével.

3. A bájtkód „futtatása" a java parancs segítségével, aminek eredményeként a Java Virtuális Gép lefordítja a megfelelő operációs rendszernek (és processzornak) szóló gépi kódot.
   – Egy adott rendszeren telepített JVM tisztában van azzal, hogy milyen rendszerre települt, így pontosan tudja, hogy milyen bájtkódot kell előállítani a processzornak és az operációs rendszernek.

Természetesen a Java programok futtatása csak akkor lehetséges, ha a JVM-et tartalmazó JRE telepítve van az adott számítógépen, viszont ha ez a feltétel teljesül, semmi akadálya nincs a bájtkód futtatásának. Természetesen az ábrán szereplő platformok mindegyikén elérhető a JRE.

A Java programok platformfüggetlenségét (hordozhatóságát) a bájtkód és leggyakrabban használt operációs rendszerekre elérhető Java Virtuális Gép együttese biztosítja. Szeretném idézni Barry Burd-öt, aki nagyon találóan részletezte a bájtkód előnyeit és működését.

„A Java segítségével foghatsz egy bájtkódot, amelyet egy windowsos gépen csináltál, átmásolhatod a bájtkódot bármilyen számítógépre[2], azután pedig futtathatod a bájtkódot minden nehézség nélkül. Ez az egyik ok a sok közül, hogy a Java olyan gyorsan annyira népszerű lett. Ezt a fantasztikus tulajdonságot, amely lehetővé teszi, hogy sokféle különböző számítógépen lehessen futtatni a kódot, hordozhatóságnak nevezzük.

Mi teszi a bájtkódot olyan univerzálissá? Ez egy fantasztikus sokoldalúság, amit a Java-bájtkód-programok élveznek, a Java virtuális gépből ered (…)

Képzeld el, hogy a Windows képviselője vagy az ENSZ Biztonsági Tanácsában. A Machintos képviselője ül a jobb oldaladon, a Linux képviselője pedig a baloldaon. … A Java kiváló képviselője beszél éppen a pódiumon, és éppen bájtkódban. Sem te, sem képviselőtársaid (Mac és Linux) nem értetek egy szót sem a Java-bájtkódból.

De mindegyikőtöknek van egy tolmácsa. A te tolmácsod bájtkódról Windowsra fordít, mialatt a Java-képviselő beszél. Egy másik tolmács bájtkódról Machintosra fordít. Egy harmadik pedig bájtkódról Linuxra.

Gondolj úgy a tolmácsodra, mint egy virtuális nagykövetre. A tolmács (…) a te nevedben értelmezi a bájtkódot. A tolmács azt teszi, amit te is tennél, ha a Java-bájtkód lenne az anyanyelved. A tolmács úgy tesz, mint ha ő lenne a Windows nagykövete, végigüli az unalmas bájtkódbeszédet, elraktároz minden szót, és így vagy úgy feldolgozza azokat.

Neked van egy tolmácsod – egy virtuális nagyköveted. Ugyanez van a Windowsnál is, a számítógép lefuttatja a saját bájtkódfordító szoftverét. Ez a szoftver a Java virtuális gép.”
(Barry Burd, 2012)

Vegyük figyelemre az előző ábrán *(2. ábra)* található operációs rendszereket. PC kategóriában (asztali gép, laptop) mindhárom nagy platform (Windows, MAC OS X, Linux) támogatott, azaz rendelkezik elérhető JVM-mel. Tablet és telefon kategóriában az Android rendszer összesítve körülbelül 50 % körüli részesedéssel[3] rendelkezik. Androidra nagyon nagy százalékban készítik az alkalmazásokat Java nyelven, hiszen ez az elsőszámú programnyelv a platformra. Emellett a hagyományos, nyomógombos, nem Wifi képes mobiltelefonok[4] kiemelkedő része volt képes/képes Java programok futtatására. Kiszolgálóoldalon pedig egyértelműen a Linux és Linux-alapú szerverek dominálnak (kb. 2/3-os részesedés), a maradék rész egyértelműen Windows szervereké[5]. Természetesen Linux és Windows szervereken is van elérhető JRE, így JVM is.

Összegezve: a Java szlogen – kis túlzással – megállja a helyét. Mostanra tisztában kell lennünk a Java programok készítésének alapjaival, tekintsük át, milyen eszközökre van szükségünk a Java programok készítéséhez.

2 Akár Mac-re, Linuxra, vagy Windows-ra.

3 A részesedés egy eszköztípuson belül, az adott rendszerrel/hardverrel/stb.. rendelkező eszközök arányát jelenti.

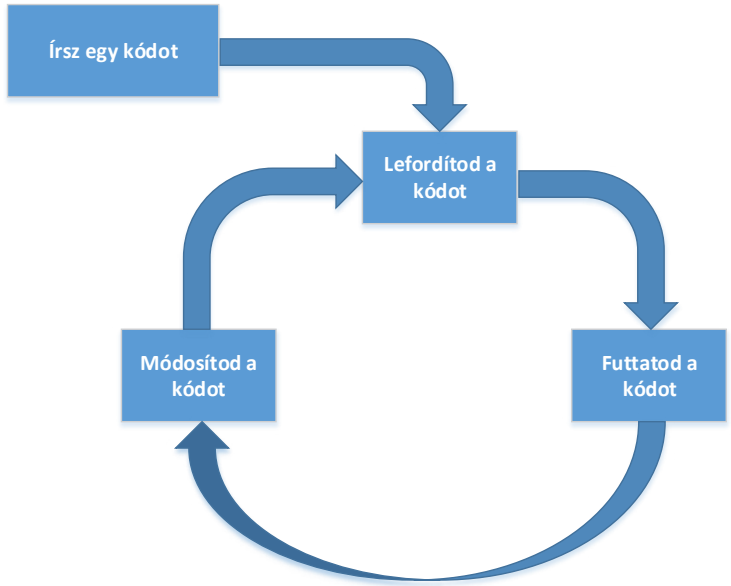
4 Melyekre sokszor „buta” telefonokként hivatkozik a szakma (hiszen az okostelefonok tulajdonságaival nem rendelkeznek).

5 https://en.wikipedia.org/wiki/Usage_share_of_operating_systems Wikipédia cikkeket nem igazán illik forrásként használni, de ennek az összesítő szócikknek a hivatkozáslistája igazán meggyőző, valamint páratlan összehasonlításokat tartalmaz minden eszközterületen (PC, mobile, server).

2. Java programamok fejlesztése

Bármilyen fordított programozási nyelvről is legyen szó, a fejlesztés folyamata megegyezik a következő ábrán *(3. ábra)* szemléltetett folyamattal.

3. ábra. Java programok fejlesztése



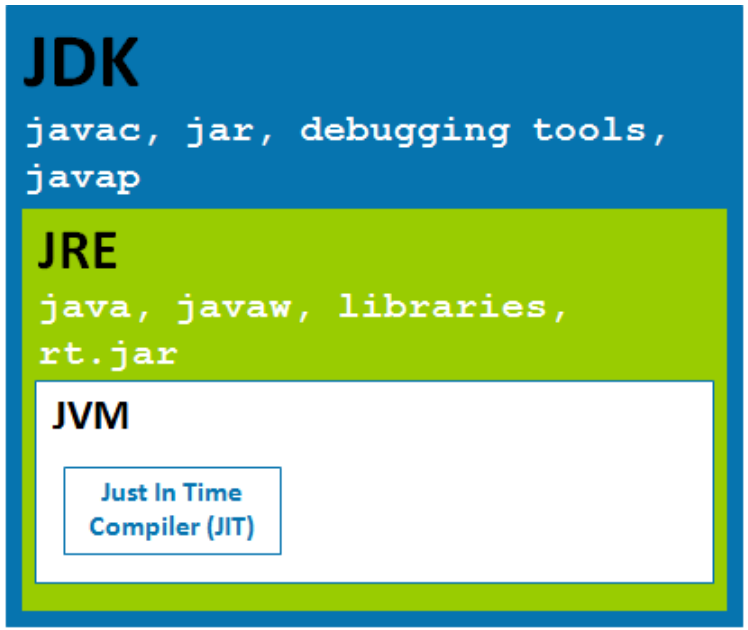
Első lépésként elkészítesz egy kódrészletet, ami a megoldandó feladat egy részét megvalósítja. Majd lefordítod. Ezt követően futtatod, hogy kipróbáld és megbizonyosodj a működésének helyességéről. Majd módosítod a kódot, vagy javítási céllal, vagy egy újabb részletet valósítasz meg benne. Majd lefordítod. Végül futtatod. Ezt a ciklikus műveletsort pedig addig ismétled, ameddig nem készülsz el az alkalmazással. Természetesen a Java esetén sem történik másképpen a fejlesztés.

2.1 Fejlesztési eszközök

Természetesen a fejlesztéshez szükségünk van a futtatókörnyezetre, a JRE-re, hiszen a lefordított bájtkódokat ki szeretnénk próbálni, hogy megbizonyosodjuk: minden utasítást helyesen adtunk ki. Ahogy már említettük, a JRE tartalmazza a java parancsot[6], aminek segítségével futtathatjuk a bájtkódokat, valamint tartalmazza az alapvető függvénykönyvtárakat (függvénykönyvtár: library), amikben találjuk meg a nyelv alapvető utasításainak nagy részét. Emellett része még a JVM, ami tartalmazza a JIT fordítót.

A fordítási és futtatási folyamatot szemléltető ábrát (*lásd: 2. ábra*) ismételten megtekintve láthatjuk, hogy szükségünk van a javac parancsra, aminek segítségével fordíthatjuk le a forráskódunkat bájtkóddá. A javac parancs a JDK (Java Fejlesztői Eszközkészlet: Java Development Kit) részét képezi *(lásd: 4. ábra)*.

4. ábra. JDK, JRE, JVM[7]



6 Ami Windows rendszeren egy java.exe futtatható állomány.

7 http://www.javabeat.net/what-is-the-difference-between-jrejvm-and-jdk/

Látható, hogy a JDK része a JRE, amelyik magába foglalja a JVM-et. Tehát nem szükséges minden J** programot feltelepítenünk a fejlesztéshez, elegendő csak a JDK csomag telepítése.

Beszéljünk az ábrán szereplő, eddig nem említett eszközökről:

– jar
  – ugyancsak egy parancs, aminek a segítségével a bájtkód fájljainkat egy .jar kiterjesztésű tömörített fájlba csomagolhatjuk.

– debugging tools
  – hibakeresési eszközök
  – későbbi leckékben fogjuk használni őket
– javap
  – egy újabb parancs, amivel egy java bájtkód fájl legfontosabb információit tudjuk kinyerni

– javaw
  – ennek a parancsnak a segítségével egy Javában írt grafikus alkalmazást tudunk elindítani, anélkül, hogy megjelenne egy parancsori ablak

– rt.jar
  – Run Time .jar
  – a Java futtatási környezetének alapvető bájtkódjait tartalmazó csomagolt .jar kiterjesztésű állomány

Hivatkozások

Barry Burd, P. (2012): Tantusz: Java. Budapest: Panem Könyvek.

Bevezetés a programozásba

Java szintaxis

Somlyai Pál

Dunaújvárosi Főiskola

2015

Tartalom

Bevezetés a programozásba – Java szintaxis 1

1. Java programok készítésének szabályai

Mint minden szellemi tevékenységnek, a programozásnak is pontos szabályrendszere van, amelyek egy része minden programnyelvnél azonos, más része programnyelvenként eltérő lehet. Mi nem fogjuk minden esetben megkülönböztetni az általános- és a Java specifikus szabályokat, ezeket a később, további programozási nyelvek elsajátítása során magatoktól el fogtok tudni különíteni.

Mint minden beszélt nyelvnek, így a programozási nyelveknek is vannak nyelvtani szabályai. Ezeket a szabályokat összefoglaló néven szintaktikának nevezzük.

1.1 Szintaktika/szintaxis

A szintaxis azon szabályok összessége, amelyek a programok kialakításának formai követelményeit hivatottak betartatni. Amikor a programozás során nyelvtani hibát vétünk, azt szintaktikai hibának nevezzük. Amikor az utasítások logikai felépítésében hibázunk, akkor szemantikai hibáról beszélünk.

A Java nyelv szintaxisa C alapú, azaz a C alapú nyelvek közé sorolhatjuk. Rengeteg szabályt tartalmaz a Java nyelv szabályrendszere, a Java nyelv specifikációja. Az SE változat (Standard Edition: Szabványos – itt inkább Alapvető – Kiadás) 8-as verziójának specifikációja a dokumentum készítésének pillanatában a következő címen érhető el:

https://docs.oracle.com/javase/specs/jls/se8/html/index.html

Itt megtalálhatsz minden szabályt a Java nyelvvel kapcsolatban, azonban javaslom, hogy inkább a tananyagban ismertetett sorrendben tanuld meg őket és csak kiegészítésként nézz utána a részletes specifikációban. Ismerkedjünk meg néhány alapvető szintaktikai szabállyal.

1.1.1 Alapvető szintaktikai szabályok

A Java nyelv kis- és nagybetűkre érzékeny (azaz case-sensitive). Ez azt jelenti, hogy amikor egy példa forráskódban ezt látod: System.out.println(), akkor nem írhatod be mondjuk ezt: sYsTem.OUT.pRinTLn().

Azokon a helyeken, ahol szóközt látsz, akár többet is elhelyezhetsz, azonban ez nem ajánlott, mert rontja az olvashatóságot.

Mint ahogy említettük, a Java C alapú szintaxisú. A C alapú szintaxis egyik alapeleme, hogy az utasításokat egymástól „;”-vel határoljuk. A soremelésnek (új sor kezdetének) nincs parancs-határoló hatása.

Az alapvető szabályok ismeretében itt az ideje, hogy megismerkedjünk a Java programok szerkezetével.

1.2 Java programok szerkezete

A Java egy objektumorientált nyelv. Az objektumorientáltságról részletesen a következő félévben fogunk tanulni, ezért most csak annyit árulok el: az objektumorientált programozásban egy fontos elem az osztály.

A Javában minden program egy osztály (de nem minden osztály jelent egy programot!). Minden osztálynak kötelezően adnunk kell egy nevet. Amennyiben szeretnénk készíteni egy Java programot, a következőt kell tennünk. Létrehozni egy fájlt, aminek neve megegyezik a később benne elkészített osztály nevével, valamint a kiterjesztése .java. Tehát ha a programunk nevének a „MyFirstJavaProgram”-ot (azaz AzÉnElsőJavaProgramom) választottuk, az egyszerű szövegfájl neve, amit létrehozunk, MyFirstJavaProgram.java legyen (a kis- és nagybetűk megkülönböztetettek).

A névválasztás miatt a forrásfájlban kötelezően a következő szerkezetnek kell szerepelnie.

```
public class MyFirstJavaProgram {

    public static void main(String[] args) {

    }

}
```

A public szó a class előtt jelenleg nem szükséges számunkra[1], csupán azért írtam be, mert a fejlesztőkörnyezetünk is alapértelmezetten elhelyezi. Ezt követően a class szó kötelező, az ő segítségével jelezzük, hogy az őt – szóköz után – követő, szóközzel nem megszakított karaktersor az osztályunk neve. A MyFirstJavaProgram-ot követő szóköz opcionális, csupán az olvashatóságot javítja.

A nyitó kapcsos zárójel jelzi, hogy itt kezdődik a MyFirstJavaProgram osztály tartalma. Minden nyitó kapcsos zárójelnek lennie kell egy záró kapcsos zárójel párjának, amely azt jelöli, hogy meddig tart az aktuális utasításblokk. Esetünkben a blokkzáró kapcsos zárójel a legalsó sorban található egyetlen karakter.

A Java fordító ellenőrzi a párok meglétét, és ha talál egy egyedülálló kapcsos zárójelet, hibát jelez. Képzeljük el, hogy a párokat a következőképpen keresi: elindul a forrásfájl elejétől. Ha talál egy nyitó kapcsos zárójelet, elindul a forrásfájl vége felől és megáll az első záró kapcsos zárójelnél, amit a nyitó párjaként érzékel. Majd tovább folytatja az első nyitó kapcsos zárójeltől lefelé, ha talál egy újabb nyitó kapcsos zárójelet, lentről tovább folytatva megkeresi a párját. Ezt egészen addig folytatja, ameddig megtalál minden kapcsos zárójel párt. Ha marad olyan kapcsos zárójel, aminek nincs meg a párja, hibát jelez, ellenkező esetben minden rendben van.

Ez azt jelenti, hogy minden kapcsos zárójelnek párban kell állnia, valamint a fentről lefelé haladva található első nyitó kapcsos zárójel párja, a lentről felfelé haladva szereplő első záró kapcsos zárójel. Fentről a második blokknyitó párja pedig a lentről a második blokkzáró.

Ha egy utasításblokkon belül található valami (akár egy másik utasításblokk), akkor az azt jelenti, hogy a belső blokk a külső része. A belső blokk tartalmát egy tabulátorral jobbra szokás eltolni.

Tehát a mi esetünkben ebből az következik, hogy az osztályunkon belül szerepel egy „public static void main(String[] args) { }”… bármi is legyen ez. Ez a fura szerzet a main (azaz fő) függvény/metódus[2]. Ez a függvény minden osztályunkban így fog kinézni, ám a felépítésének magyarázatát csak egy későbbi leckében fogom megadni, addig nincs szükségünk a pontos ismeretére. A magyarázat megadásáig elég annyit tudnunk róla, hogy ő a program belépési pontja: azaz az utasításblokkjai között szereplő első utasítás fog lefutni a program indítását követően.

Nagyon fontos szabály, hogy jelenleg csak a main függvényen belülre írhatunk utasításokat! Igazából nem a szerkezeti szabályokhoz tartozik, de fontosnak érzem most megemlíteni, hogy az utasítások végrehajtása a kiadásuknak megfelelő sorrendben fog megtörténni. Amint említettük, először a main függvényben szereplő első utasítás fog végrehajtódni, majd a második, a harmadik és így tovább.

1 De későbbi tárgyakban lényeges lesz.

2 Mi a kezdetben main függvényként fogunk rá hivatkozni.

Bevezetés a programozásba

Java nyelv szavai

Somlyai Pál

Dunaújvárosi Főiskola

2015

Tartalom

Bevezetés a programozásba – A Java nyelv szavai 1

1. A Java nyelv szavai

A beszélt nyelvekhez hasonlóan, a programozási nyelvek is tartalmaznak szavakat a nyelvtani szabályok összességén (szintaxison) kívül. A legtöbb programozási nyelv, így a C alapú nyelvek, valamint a Java is angol nyelvű beépített szavakat használ[1]. A Java szavait két nagy csoportba sorolhatjuk: kulcsszavak és azonosítók.

1.1 Kulcsszavak

A kulcsszavak olyan szavak, amelyet a Java tervezői lefoglaltak és speciális jelentéssel ruháztak fel. Az előző példából merítve a kulcsszavak a következőek: public, class, static, void. A class kulcsszó jelentését már ismerjük: az engem – egy szóköz után – követő, szóközökkel nem tagolt karaktersor legyen az osztály neve. A class kifejezést, szóköz után követő karaktersor egy azonosító.

A Java nyelv összes kulcsszava a következő:

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

1 Léteznek programozási nyelvek, amelyek más beszélt nyelvekből kölcsönzik a kulcsszavaikat. Emellett vannakkifejezetten „egzotikus” nyelvek is, a var’aq egy változata például a Klingon nyelven alapul(https://esolangs.org/wiki/Var'aq). Természetesen ezek a nyelvek inkább szórakoztató célzattal jöttek létre, mint üzleti alkalmazások fejlesztése céljául.

Nem kell minden kulcsszó jelentését most megismernünk, szépen fokozatosan fogom átadni számotokra őket, ahogy haladunk a leckék során. Emellett néhány kulcsszóval csak a következő félévek során fogunk megismerkedni.

1.1.1 Módosítók

A kulcsszavak egy részét módosítónak nevezzük. Egy adott Java nyelvi elem tulajdonságait módosítják. Például az osztályoknál használt public egy módosító. Tényleges hatását a következő félévben fogjuk tárgyalni. Emellett a main függvény static kulcsszava is egy módosító. Jelen tantárgyban minden függvényünket ilyen módosítóval fogjuk ellátni. A pontos hatását és az elhagyásának következményeit a következő félévben fogjuk részletesen megismerni.

1.2 Azonosítók

Az azonosítókat két csoportba soroljuk:

- A Java API-ban definiált azonosítók és
- A programozók által definiált azonosítók

A Java API (Application Programming Interface: Alkalmazás Programozási Felület) a Java tervezői által lefoglalt azonosítók összessége, amelyeknek megadásának (begépelésének) segítségével előre elkészített műveletek hajthatóak végre, illetve előre beállított adatok kérhetőek ki. A System.out.println() utasítás egy ilyen azonosító, amely egyértelműen azonosítja a soremeléses kiíró utasítást. Igazából a pontokkal határolt mindhárom karaktersor egy-egy azonosító. A System azonosítóval ellátott dolgon belüli, out azonosítóval ellátott dolog, println() utasításáról van szó, amely maga is egy azonosító (mármint a println()). Ezt csak érdekességként jegyeztem meg, csak a következő tantárgy keretein belül fogjuk őket részletesen megismerni.

Emellett a programozók is definiálhatnak saját azonosítókat, de még a programozást tanulók is, azaz ti is és én is. A Hello programunkban már mi is használtunk egy azonosítót, mégpedig a „Hello”-t. Az osztályunkat azonosítottuk ezzel a karaktersorral.

Megkötések a Java azonosítókkal kapcsolatban:

- Nincs megkötés a hosszukkal kapcsolatban

- Java betűk vagy Java számjegyek sorozata

    - o Java betű
        - kis- és nagybetűk (lehetnek ékezetes karakterek is, azonban nem ajánlom használatukat)
        - alulvonás: _
        - $ (használata nem javasolt, főleg automatikus kódgenerálásra van fenntartva)
    - o Java számjegy
        - arab számjegyek 0..9

- Az első karakternek Java betűnek kell lennie

- Nem egyezhet meg:

    - o A Java kulcsszavak egyikével sem
    - o „true”, „false” és „null” karaktersorokkal

- Nem létezhet két egyforma azonosító egy osztályon belül

    - o Ezen a szabályon még csiszolni fogunk a függvények használatakor, de most megállja a helyét.

Példák helyes azonosítókra:

- _anIdentifier
- AN_IDENTIFIER
- x12
- isLetterOrDigit

Ezen felül az azonosítók írásmódjára nincs szintaktikai szabály, csak egy megegyezés, aminek értelmében az osztályok nevét nagy kezdőbetűvel kezdjük és amennyiben több szóból áll, a szóközöket elhagyva, a kezdőbetűket nagybetűs formára alakítva, a többi betűt pedig kisbetűvel írjuk ki, lásd: MyFirstJavaProgram. Ezt az írásmódot általában camel case írásmódnak is nevezik, mert a teve (camel) púpjaira emlékeztet.

Azonban a Programozásban megkülönböztetünk két változatot:

- Amikor az első szó kezdőbetűje kisbetű: camel case
    - pl.: myFirstJavaProgram
- Amikor az első szó kezdőbetűje nagybet: Pascal case
    - pl.: MyFirstJavaProgram

Tehát valójában Pascal case írásmódról beszélünk.

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

Bevezetés a programozásba

Literálok és megjegyzések

Somlyai Pál

Dunaújvárosi Főiskola

2015

Tartalom

1. Literálok

A Javában nem csak kulcsszavakat és azonosítókat találhatunk, hanem értékeket is. Ezek az értékek lehetnek egész- és tört számok, logikai értékek, karakterek és szövegek, úgynevezett karakterláncok.

A literálok által hordozott értékeket ki tudjuk íratni a képernyőre, hozzá tudjuk rendelni őket változókhoz (következő lecke) és még rengeteg hasznos dologra használhatjuk őket, amely eseteket a tananyag során részletesen meg fogunk ismerni. Mivel eddig csak a képernyőre kiírást használtuk, ezért ezt a használati módot fogom példaként alkalmazni. A Java literálok legfontosabb adatait a következő táblázat *(1. táblázat)* tartalmazza. A 0 karakter a nulla számjegyet jelenti. A példa a képernyőn azt jelenti, hogy a megadott példa literál a System.out.println(); utasításban megadva, mit ír ki a képernyőre. A megadási formában az <érték> a megadandó érték behelyettesítésének helyét jelenti.

1. táblázat - Literálok

Literál csoport	Literál név	Megadási forma	Példa	Példa a képernyőn
Egész szám literál	Decimális szám	<érték>	45	45
	Bináris szám	0b<érték>	0b11	3
	Oktális szám	0<érték>	011	9
	Hexadecimális szám	0x<érték>	0x11	17
Tört szám literál	Float literál	<érték>f	45.16f	45.16
	Double literál	<érték>	678.519	678.519
	Logikai literál	<érték>	true	true
	Karakter literál	’<érték>’	’c’	c
	Karakterlánc literál	”<érték>”	”szöveg”	szöveg
	Null literál	null	null	Nem írható ki!

A táblázattal kapcsolatban szükségesnek érzem megjegyezni pár dolgot. A törtszámok esetén a tizedes vessző helyett – az angolszász írásmódnak megfelelően – tizedes pont kell használnunk. Ha a törtszám 1.0-nél kisebb, de 0.0-nál nagyobb törtszám, akkor a 0-t nem kell kírnod (például 0.45f helyett nyugodtan írhatsz .45f-et). A null literál használata a következő félévben fogunk kitérni, jelenleg nem fogjuk használni. A következő utasítás System.out.println(null); utasítás kiadását követően „a hivatkozás kétértelmű/félreérthető" hibaüzenetet kapunk. A megértéséhez csak a leckék legvégén lesz meg a kellő tudásunk, addig írjuk fel a bakancslistánkra.

2. Escape szekvenciák

Az escape szekvenciák segítségével tudjuk megváltoztatni a képernyőre írás módját. Valamint a literálok határoló-karaktereit tudjuk semlegesíteni velük. Emellett fájlba íráskor is használhatjuk őket, de most maradjunk a képernyőre írásnál és a literálok határoló-karaktereinek semlegesítésénél.

2. táblázat - A legfontosabb escape szekvenciák

Escape szekvencia	Hatás
\"	Szöveg literálban egy " jelölése
\'	Karakter literálban egy ' jelölése
\t	Tabulátor
\n	Soremelés: a következő karakter új sorba kerül

A tabulátor 8 karakteres „oszlopokra" osztja a konzol ablakot. A tabulátor segítségével táblázatos formára igazíthatjuk a kiírt szövegeket. Tehát ha az első oszlopban járunk, és kiírunk két karaktert, valamint elhelyezzük a tabulátor escape szekvenciáját, a következő karaktert a 9. pozícióba írhatjuk. Majd kiírunk tetszőleges mennyiségű (8-nál kevesebb) karaktert és még egy \t, akkor a harmadik oszlopba írhatjuk a következő karaktert. Ha egy oszlopba 8-nál több karaktert írunk, majd elhelyezu nk egy \t-t, akkor plusz egy oszloppal jobbra keru lu nk (mert az aktuális oszlopot tele írtuk).

3. Kiíró utasítások

A kiíró utasítások segítségével írhatunk ki szöveget a képernyőre. Az eddig használt kiíró utasítás a System.out.println() volt. Ez megjelenítette a kiírandó szöveget, majd elhelyezett egy sortörést (a következő utasítás már új sorba írt). Println, azaz PrintLine (sor kiírása). Egy másik kiíró utasítás a System.out.print() csak egy szöveget ír ki a képernyőre és nem helyez el sortörést, azaz a következő kiíró utasítás ugyan abban a sorban fogja folytatni a szöveg kiírását. Lényegében a println utasítás az a print utasítás, ami a kiírandó szöveg mögé elhelyez egy \n azaz sortörés escape szekvenciát.

4. Megjegyzések

Biztosan előfordult már veletek, hogy egy tankönyv értelmezése közben a könyv lapjaira postit cédulákat ragasztottatok, melyekre a megértést segítő megjegyzéseket írtatok, esetleg közvetlenül a tankönyv lapjaira írtátok lényeges hozzászólásotokat. A megjegyzések pontosan erre hivatottak a Javában.

A megjegyzések a szerzőnek (a kód készítőjének), esetleg a kódot később olvasóknak szólnak. Általában lényeges információkat tartalmaznak a kód működésével kapcsolatban, de előfordul, hogy csak a megértést próbálják segíteni számunkra. De hogyan írhatunk megjegyzéseket a forráskódba? Ha beírom a main függvény mellé, hogy „ez egy nagyon fontos elem, ő a program belépési pontja", a Netbeans rögtön hibát fog jelezni, mert nem fogja érteni az általunk leírt utasításokat. Bizony, a Java minden a forráskódban elhelyezett kifejezést és literált utasításként kezel. Akkor hogyan írhatok megjegyzéseket?

Három lehetőségünk van megjegyzések létrehozására:

- // egysoros megjegyzés
    - o csak a sorában található szavakat hagyja figyelmen kívül a fordító

- /* több soros megjegyzés */
    - o a /* és a */ által határolt rész teljes egészében figyelmen kívül lesz hagyva

- /** Dokumentációs megjegyzés */
    - o viselkedése teljesen megegyezik a többsoros megjegyzéssel
    - o általában osztályokat, függvényeket, változókat dokumentálnak a segítségével
        - ~leírják hogy mire használhatóak

```
 1. package literals;
 2. 
 3. /**
 4.  *
 5.  * @author somlyaip
 6.  */
 7. public class Literals {
 8. 
 9.     /**
10.      * @param args the command line arguments
11.      */
12.     public static void main(String[] args) {
13.         System.out.println(10);
14.         System.out.println(0x52);
15. 
16.         System.out.println(5.46f);
17.         System.out.println(5.46);
18. 
19.         System.out.println(true);
20.         System.out.println(false);
21. 
22.         System.out.println('c');
23.         System.out.println('\u0055');
24.         System.out.println('\u262F');
25. 
26.         System.out.println("String literal");
27.         System.out.println("This is a long line. " +
28.                 "This is another long line." );
29.         System.out.println("This is a long line. \nThis is another long line." );
30.         System.out.println("\"");
31.         System.out.println('\'');
32.         System.out.println("Number: " + 50);
33.     }
34. 
35. }
```

Bevezetés a programozásba
Gyakorló feladatok

1. Fejezet

1. Írja ki a képernyőre a saját nevét egy sorban!

2. Írja ki a képernyőre a saját nevét, úgy hogy minden karakter új sorba kerüljön:
Példa kimenet:

J
o
h

n
D
o
e

3. Írassa ki a következő szöveget a képernyőre: ”4 + 2 = 6”
– A megoldás során csak a következő literálokat használhatja fel:
  - o 4
  - o ” + ”
  - o 2
  - o ” = ”
  - o 6
– Csak egy kiíró utasítást adhat ki!

4. Készítsen programot, amely megjeleníti a képernyőn a következő táblázatot:
A kimenet:

```
Gyümölcs HUF/Kg
------------------------
Alma 221
Körte 450
Eper 980
Datolya 1200
```

2. fejezet

Bevezetés a programozásba

Adattípusok,változók, konstansok

Somlyai Pál

Dunaújvárosi Főiskola

2015

Tartalom

1. Adattípusok

A Javában használható adattípusok – pontosabban, amelyeket ebben a félévben fogunk használni – pontosan megegyeznek a literálok típusaival, a különbség csak az, hogy az egész szám literál több féle adattípusban is eltárolható (később: Változók). Amikor átadunk egy literált a System.out.println(); utasításnak, akkor az átadott literál mindig egy adott adattípusú érték. Az általunk használt adattípusok a primitív adattípusokból (lásd *1. táblázat*) és a String típusból áll. Az adattípusoknak általánosságban van egy tartománya, amelyben elhelyezkedő értékeket képesek felvenni.

1. táblázat – Primitív adattípusok

Típus	Mit tartalmazhat?	Alapér-telmezett érték	Méret	Tartomány
boolean	true vagy false	false	1 bit	Nincs tartomány, csak true vagy false lehet
char	Unicode karakter	\u0000	16 bit	\u0000 to \uFFFF
byte	Előjeles egész szám	0	8 bit	-128 - 127
short	Előjeles egész szám	0	16 bit	-32768 - 32767
int	Előjeles egész szám	0	32 bit	-2147483648 - 2147483647
long	Előjeles egész szám	0	64 bit	-9223372036854775808 - 9223372036854775807
float	Tört szám	0.0	32 bit	±1.4E-45 - ±3.4028235E+38
double	Tört szám	0.0	64 bit	±4.9E-324 - ±1.7976931348623157E+308

Fontos megjegyeznem, hogy néhány nyelvvel ellentétben, a Javában nincs előjel nélküli adattípus, pontosabban nem használható előjel nélküli módosító az adattípusoknál.

A String típus nem primitív, azaz egyszerű adattípus. Ő az első objektum típus, amit használni fogunk. Később részletesen kitérünk majd a String típus különleges viselkedésére, azonban most csak jegyezzünk meg annyit: hogy egy különleges adattípus.

De mire is jók az adattípusok azon túl, hogy a literálaink típusait és tartományát határozzák meg?
Változók adattípusaiként használhatjuk őket.

2. Változók

A változók lényegében azonosítóval (azaz névvel) megjelölt értékek. Képzeljük el őket úgy, mint ahogy a matematikai egyenletekben használtuk őket. Vegyük a következő példát:

```
x = 12
x = x + 2
y = x
```

Mennyi lesz az y változó értéke? Természetesen 14. A programozás során minden változó definíciójakor (bevezetésekor, esetleg létrehozásakor) meg kell határoznunk őket. Vezessük át őket a Java programozásba: lássuk el típusokkal a változókat és tegyünk ;-t a műveleteket követően:

```
int x = 12;
x = x + 2;
int y = x;
```

Egy változó „neve” az azonosítója. Mivel azonosító, ezért az azonosítókra vonatkozó szabályok érvényesek rájuk, emlékszünk:

– Számjegyek, illetve Unicode karakterek szerepelhetnek bennük, valamint az _ és $

– Mindenképpen karakterrel, illetve _, vagy $ jellel kell kezdődniük

– Nem létezhet két egyforma azonosító egy programon belül

A változók definícióját követően bármikor hivatkozhatunk rájuk a programunkban[1]. Természetesen a változók értékeit ki is írathatjuk. Ha lefuttatjuk a következő utasítást:

System.out.println(y);

a képernyőn megjelenik a 14. A változókhoz nem kell rögtön a definíciójuk idejében értéket rendelni (azaz inicializálni őket). Az inicializáció később is elvégezhető, például:

```
int x;      // Definíció
int y = 19; // Definíció és inicializáció
x = 25;     // Inicializáció
x = 59;     // Értékadás
```

Az értékadás egy olyan művelet, amely során az egyenlőségjel jobb oldalán álló értéket átadjuk az egyenlőségjel bal oldalán álló változónak. Az értékadás során az egyenlőségjel bal oldalán álló változó értéke felülírásra kerül. A jobb oldalon állhat literál és változó is. Az inicializáció egy speciális értékadás, az első értékadás.

Mi történik, ha egy változóhoz még nem rendeltünk értéket? Akkor mi az „értéke”? Mi történik, ha ilyenkor írjuk ki a változót a képernyőre? Ahogyan a jelen tartárgyban használjuk a változókat (lokális változók), nem kerül alapértelmezett érték beállításra. Ami azt jelenti, hogy ebben a félévben minden változónak kötelezően be kell állítanunk a kezdőértékét, mielőtt használni kezdjük (pl.: kiírjuk a képernyőre). Amikor a változókat osztályok adattagjaiként fogjuk használni (következő félév), definiálásuk során mindig az adattípusuknak megfelelő alapértelmezett értékre (lásd *2. táblázat*) kerülnek beállításra.

1 Később pontosítjuk ezt a megállapítást.

2. táblázat – Adattípusok alapértelmezett értékei

Adattípus	Alapértelmezett érték
boolean	false
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (vagy bármely objektum)	null

Az eddigi példákban nem igazán tudtam megmutatni a változók szerepének fontosságát. Gondoljunk vissza az általános-, vagy középiskolai matematika feladatokra. Biztosan sokszor előfordult veletek is, hogy egy bonyolultabb művelet részeredményeit le kellett írnotok a füzetetekbe, mert mondjuk egy másik művelet eredményével kellett elosztani. Ezért kiírtuk a füzetünkbe, kiszámoltuk a másik művelet eredményét, majd memóriába tettük az eredményt, visszaírtuk a füzetben „eltárolt” eredményt a számológépbe, majd elosztottuk a memóriában tárolt értékkel.

Sokkal egyszerűbb lett volna, ha a számológép memóriájában több értéket is el lehetett volna tárolni (az enyémben legalább is csak egyet lehetett), majd mondjuk az MA, MB, stb gombokkal hivatkozni az értékükre. Egy Java programban például a változók segítségével hasonló feladatokat probléma nélkül megoldhatunk.

2.1 Változó definiálása

Egy változó definiálása során helyet foglalunk számára a számítógép memóriájában. Pontosan akkora helyet, amekkora a változó adattípusának értéke (pl.: integer esetén 4 bájtot). Emellett megadjuk, hogy erre a lefoglalt területre szeretnénk a változó azonosítójával hivatkozni. Innentől kezdve ez a terület a miénk, más nem írhatja felül, csak mi[2]. Amikor hivatkozunk a változóra, pl. ki szeretnénk írni, akkor a számítógép kiolvassa a változó azonosítója által leírt memóriaterületről az értékét, majd átadja a print/println utasításnak. Amikor értékül adunk valamit a változónknak (felülírjuk), akkor pedig az azonosítója által lefoglalt területen felülírjuk a bájtokat az új értékkel.

2.2 Több változó egyidejű definiálása

Lehetőségünk van egyidejűleg definiálni több azonos típusú változót. Pl.:

```
int x, y;
```

Ennek eredményeként definiáltunk egy integer típusú x és egy integer típusú y változót. Emellett lehetőségünk van az egyidejű definíció során inicializálni is a változókat:

```
int x = 4, y = 12;
```

Azonban ne felejtsük el, hogy mindkét változót külön-külön kell inicializálni. A következő utasítás csak y értékét inicializálja, x kezdőértéke nem kerül beállításra:

```
int x, y = 12;
```

2 A Javában ez közel igaz. Későbbi tárgy keretein belül fogunk tanulni a puffer-túlcsordulásról, ennek veszélye a Javában minimális.

3. Konstansok

A konstansok lényegében ugyan arra szolgálnak, mint a változók. Az azonosítójuk segítségével az általuk eltárolt értékre hivatkozhatunk. Azonban a matematikai értelemben vett szabály vonatkozik rájuk. A konstans (azaz állandó) értékét a nevéből adódóan nem változtathatjuk meg. Pontosabban csak egyszer adhatunk neki értéket (inicializálhatjuk), majd később nem írhatjuk felül az értékét.

A konstansok definícióját nagyon hasonlóan tudjuk megadni, mint a változókét, mindösszesen egy final (azaz végleges) módosítószót kell elhelyeznünk a konstans adattípusa előtt.

```
final int HET_NAPJAINAK_SZAMA = 7;
```

Természetesen az inicializáció a definíciót követően is elvégezhető.

```
final int HET_NAPJAINAK_SZAMA;
HET_NAPJAINAK_SZAMA = 7;
```

Természetesen olyan értékek eltárolására-, majd később felhasználására használatosak, amelyek örökérvényűek.

4. Típuskonverzió

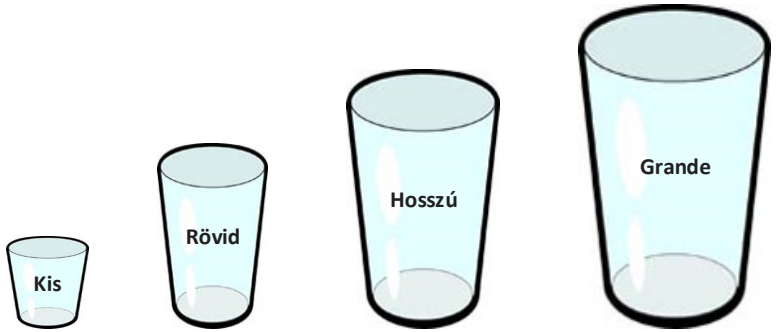
Előfordulhat, hogy egy adott típusú változó értékét egy másik (megfeleltethető[3]) típusú változóban szeretnék tovább tárolni. Ennek oka lehet az, hogy rájöttünk, az eddigi változótípusunk túl nagy, vagy túl kicsit tartományban képes értékeket eltárolni. A típuskonverzió legegyszerűbb magyarázatát az
Agyullám: Java könyvben találhatjuk meg.

„A Java-változókra gondolhatsz úgy, mintha csészék, bögrék vagy korsók lennének: kávéscsészék;
teásbögrék; söröskorsók (…)

3 Egész számot egész számban (pl.: int → short), tört számot törtben (float → double)

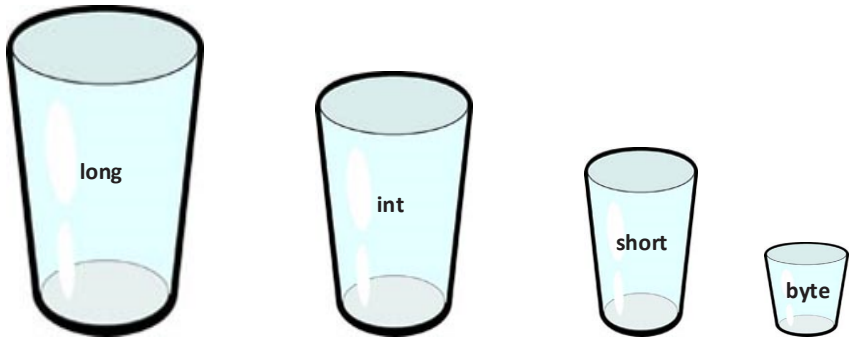
A változó csupán egy csésze. Egy tároló, ami tárol valamit.

A csészékhez hasonlóan a változóknak is van mérete és típusa. (…) Végig a csészehasonlatnál maradunk – bármilyen bárgyúnak is tűnik is ebben a pillanatban a párhuzam, sokat segít majd, amikor jobban belebonyolódunk a tárgyalásba (…) A primitívek olyanok, mint a kávézókban található csészék.

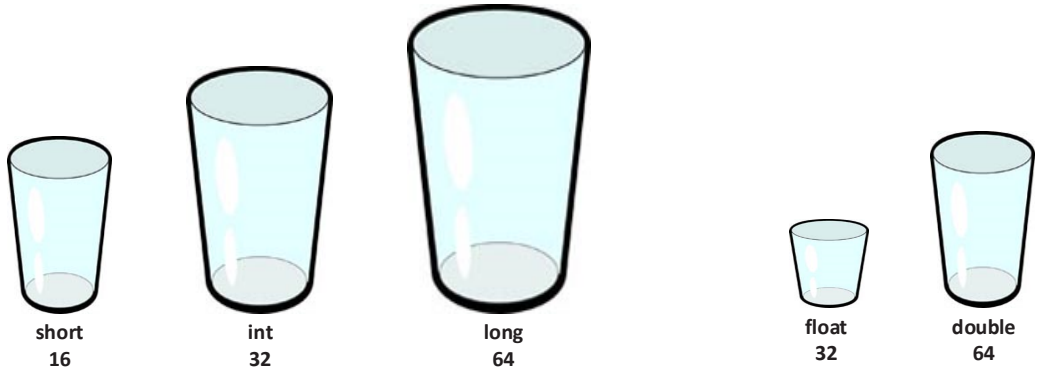


Ha jártál már Starbucks kávézóban, akkor tudod, miről beszélünk: azokról a különféle méretű csészékről, amelyek mindegyikéhez társul egy „név": „rövid", „hosszú", „Szeretnék egy „grande" mochát, csökkentett koffeintartalmút, extra tejszínhabbal." A csészéket esetleg ki is teszik a pultra, hogy segítsenek a rendelésben.

A Javában a primitívek (alaptípusok) különböző méretűek, és ezeknek a méreteknek is van neve. Amikor bevezetsz egy változót a Javában, meg kell adnod a konkrét típusát. Az itt található négy tároló a Java négy egész típusú alaptípusát ábrázolja:



Minden csésze egy értéket tárol, csak éppen Java-primitívek esetében nem azt mondod a fordítónak, hogy „Egy francia pörkölésű hosszút kérek”, hanem azt, hogy „Egy int változót legyen szíves, benne a 90-es számmal”. És van még egy aprócska különbség: a Javában nevet is kell adnod a csészéknek. A teljes rendelés tehát így hangzik: „Egy int változót legyen szíves, benne a 2486-os számmal, és a változó neve legyen height[4]”. Minden primitív változó rögzített számú bitet tárol (ez a csésze mérete). A Java hat primitív számtípusának a méretét[5] itt láthatod: ” (Kathy Sierra, Bert Bates, 2011)



Remélem mostanra jobban el bírtuk képzelni az alap adattípusok által eltárolható tartománykülönbségekből fakadó, értékeltárolási határokat. Ez nagyon bonyolultan hangzik, lássuk pontosan mire gondolok. Tudjuk, hogy az adattípusok mekkora értékeket tárolhatnak el (*1. táblázat)*. Látható, hogy egy byte típusú változóban -128 és 127 közötti értékeket tudunk eltárolni. Valamint a short típusban -32768 és 32767 közötti értékeket tárolhatunk. Képzeljük el, mi lesz a következő művelet eredménye:

```
byte b = 127;
short s = b;
```

A b változó a byte maximális méretét „hordozza”. Arra utasítjuk a fordítót, hogy helyezze el a b változó által hivatkozott értéket az s változóban, amely típusa sort, azaz -32768 és 32767 közötti értékeket képes eltárolni. Amennyiben az előző párhuzamnál maradunk, ez azt jelenti, hogy egy milliliter vizet (vagy kevertet, esetleg más italt) megpróbálunk áttölteni egy kettő és fél deciliter űrtartalmú pohárba. Természetesen sikerülni fog, és éppen csak egy kis folyadék fog megjelenni a short pohárban alján. Tehát a fenti utasítás probléma nélkül kiadható a Javában.

4 Magasság
5 A méretek bit mértékegységben kerültek megadásra.

Azonban képzeljük el a folyamat fordítottját:

```
short s = 32767;
byte b = s;
```

Most arra utasítottuk a fordítót, hogy egy 1 ml űrtartalmú pohárba töltsünk át 2,5 dl űrtartalmú, színültig telített pohár tartalmát. Természetesen nem fog sikerülni, hanem valami ilyesmi eredményre számíthatunk:

Eddig minden világos. Azonban mi történik, ha a következő kódrészletet próbáljuk lefuttatni:

```
short s = 5;
byte b = s;
```

Tudjuk, hogy az 5 érték bőven elfér a byte által eltárolható értéktartományban. Az előző párhuzammal élve: kevesebb, mint 1 ml víz van egy 2,5 dl-es pohárban, amit be szeretnénk tölteni egy maximum 1 ml űrtartalmú pohárkába. Elméletben nincs semmilyen akadálya, azonban a Java fordítót csak az érdekli, hogy valami nagyot próbálsz beletenni egy kicsi tárolóba. Tehát a fenti utasítást nem engedi lefuttatni. Azonban szerencsére a típuskényszerítés segítségével megoldhatjuk a problémát.

4.1 Implicit típuskonverzió

Mint ahogy a fenti példákból láttuk, a „szűkebb” típus adatvesztés nélkül konvertálódik „szélesebb” típusra. Az ilyen adatvesztés nélküli, automatikusan (azaz a programozó beavatkozása nélkül) végbemenő típuskonverziót nevezzük implicit típuskonverziónak. Lényegében ezek azok a konverziók, amikor adott típusú értékeket, típuskényszerítés nélkül (következő fejezet) adhatunk értékül más típusú változóknak. A következő esetekben történik implicit típuskonverzió.

3. táblázat – Implicit típuskonverziók

Típusok	Példa	Eredmény
Szűkebb tartományú egész számról, szélesebb tartományú egész számra	int i = 12; long l =i;	Egész szám
Szűkebb tartományú tört számról, szélesebb tartományú törtszámra	float f = 91.45f; double d = f;	Tört szám
Egész számról tört számra	int i = 561; double d = i;	Tört szám: egész része a konvertál érték, törtrésze 0. Jelen esetben: 561.0

Amennyiben olyan típuspárokra szeretnénk konverziót végrehajtani, amelyek implicit módon nem kerülnek átalakításra, típuskényszerítést kell alkalmaznunk.

4.2 Típuskényszerítés

A típuskényszerítés, vagy típusváltás során tudatjuk a Java fordítóval, hogy tisztában vagyunk vele, hogy ebben a nagy pohárban lévő víz belefér a kis pohárkánkba. A következőképpen tudjuk a típuskényszerítést elvégezni:

```
short s = 5;
byte b = (byte) s;
```

Tehát amennyiben az értékadás jobb oldalán lévő változó típusa nagyobb, mint a bal oldalon szereplőé,azonban tisztában vagyunk vele, hogy a jobb oldalon lévő érték „belefér” a bal oldali változóba, akkora jobb oldali változó értékének típusának átváltásával megoldhatjuk a problémát. Minden esetben aváltozó előtt, zárójelek között kell megadni a kívánt típust, amire szeretnénk váltani.

A típuskényszerítés minden esetben a programozó felelőssége, azaz ha helytelenül használja a típuskényszerítést, valótlan adatokat fog kapni.

```
short s = 500;
byte b = (byte) s;
```

Ebben az esetben a b értéke -12 lesz, pedig valószínűleg a programozó, aki kiadta, nem ezt várta.

4. táblázat– Lehetséges típuskényszerítések

Típusok	Példa	Eredmény
Szélesebb tartományú egész számról, szűkebb tartományú egész számra	long l = 12; int i = (int) long;	Egész szám
Szélesebb tartományú tört számról, szűkebb tartományú tört számra	double d = 15.67; float f = (float) d;	Tört szám
Tört számról, egész számra	double d = 45.67; int i = (int) d;	Tört szám: az átadott érték egész részével és 0 törtrésszel. Azaz ebben az esetben: 45.0
Megfelelő egész számról, karakterre	int i = 65; char c = (char) 65;	UNICODE kódolás szerinti karakterkódnak megfelelő karakter. Jelen esetben ’A’.

Karakterről, egész számra	char c = 'T'; int i = (int) c;	UNICODE kódolás szerinti karakternek megfelelő karakterkód. Jelen esetben ’84’.
Megfelelő tört számról, karakterre	float f = 65.0f; char c = (char) f;	UNICODE kódolás szerinti karakterkódnak (az egész értéknek) megfelelő karakter. Jelen esetben ’A’.
Karakterről, tört számra	char c = 'T'; double d = (double) c;	UNICODE kódolás szerinti karakternek megfelelő karakterkód (törtrésszel együtt). Jelen esetben ’84.0’.

Amennyiben olyan típuspárokon szeretnénk konverziót végrehajtani, amelyek sem implicit módon, sem pedig típuskonverzió segítségével nem alakíthatóak át, konverziós függvényeket kell alkalmaznunk.

4.3 Konverziós függvények

A konverziós függvények utasítások, amelyek segítségével egymástól nagyon távol álló adattípusokon is elvégezhetjük a típuskonverziót. Ezek a típuspárok általában a String és egy primitív adattípus.

Hogyan alakíthatunk át egy primitív típust String típussá?

```
int i = 45;
String s = String.valueOf(i);
```

Az előbbi utasítás bármelyik primitív adattípusnál működőképes.

Hogyan alakíthatunk át egy String típusú értéket primitív típusú értékké?

A primitív típus nevének teljes kiírásával, az első karakter nagybetűssé alakításával, majd .parse típus neve utasítással.

```
String s = "76.12";
double d = Double.parseDouble(s);
```

A függvénynevek a következőek:

- Byte.parseByte()
- Short.parseShort()
- Integer.parseInt()
- Long.parseLong()
- Float.parseFloat()
- Double.parseDouble()
- Boolean.parseBoolean()

A karakter típusnak nincs közvetlen típuskonverziós függvénye String típusból, hiszen a String több karaktert tartalmazhat, így valójában értelmetlen lenne.

Források

Kathy Sierra, Bert Bates. (2011). Agyhullám: Java. Budapest: Kiskapu Kft.

```java
package variablesconstants;

/**
 *
 * @author somlyaip
 */
public class Variables {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        byte byteVariable = 45;
        System.out.println(byteVariable);
        short shortValue = 5600;
        int integer = 5600413;
        long longValue = 41239875;

        float f = 345.4f;
        System.out.println(f);
        float f2 = (float) 341.5;
        float f3 = .45f;
        System.out.println(f3);
        double doubleValue = .4;

        char character = 'c';
        String text = "This is a short text."; // definíció és inicializáció
        System.out.println(text);

        int number; // definiálás
        number = 45; // inicializálás
        System.out.println(number);
        number = 456; // értékadás
        System.out.println(number);
    }

}
```

```
 1. package variablesconstants;
 2.
 3. /**
 4.  * {Description}
 5.  *
 6.  * @author somlyaip
 7.  */
 8. public  class Constants {
 9.     public   static void main(String[] args) {
10.         final int A_CONSTANTS = 45;
11.
12.         final int ANOTHER_CONSTANTS;
13.         System.out.println("--");
14.         ANOTHER_CONSTANTS = 12; // inicializáció
15.
16.         System.out.println(Math.PI);
17.     }
18. }
```

Bevezetés a programozásba

Utasítások és kifejezések

Somlyai Pál

Dunaújvárosi Főiskola

2015

Tartalom

Bevezetés a programozásba – Aritmetikai műveletek 1

1. Aritmetikai műveletek

Természetesen a Javában lehetőségünkben áll aritmetikai műveleteket végezni. A négy alapműveleten kívül lehetőségünkben áll a maradékos osztás (modulo) használata is. A Javában nagyon egyszerű a négy alapművelet használata, de mielőtt erre rátérnénk, ismételjük meg egy (kétoperandusú) művelet tagjait.

int x = 4 + 5; esetén x az eredmény, 4 az első operandus, + az operátor (műveleti jel), 5 a második operandus. Nekem ezt úgy sikerült megjegyeznem, hogy az operátor (az orvos) operál, azaz műveletet végez az operanduson (betegen), melynek lesz valamilyen eredménye. Ha nektek nem ennyire elvont a gondolkodásotok, a legegyszerűbb úgy megjegyezni, hogy az operandusokon végezzük a műveletet. Azért nevezzük őket kétoperandusú műveleteknek, mert két operandust használnak fel.

Az aritmetikai műveletek használata roppant egyszerű, hiszen a már jól ismert szimbólumokat kell használnunk operátorokként. Egyedül az eredmény eltárolására szolgáló változó adattípusának megválasztásánál szükséges nagyobb körültekintéssel eljárni. Tehát ha két egész számot adok-, szorzok össze, vagy vonok ki egymásból, az eredményt nyugodtan eltárolhatom egy egész számú adattípusban, mondjuk integerben. Egyedül az változó típusának értéktartományát kell figyelembe vennem.

Emellett ha tudjuk, hogy a művelet eredmény egész szám, nyugodtan használhatunk float, vagy double adattípust a tárolásukra, hiszen a törtrész ilyenkor 0 lesz, ami semmilyen adattorzulással sem jár.

Azonban ha a négy aritmetikai művelet legalább egy operandusa törtszám, az eredmény változó típusának törtet eltárolható típusnak kell lennie.

Egyedül egy fura jelenség van a Java aritmetikai műveletei között. Vegyük a következő példát:

```
int x = 5 / 2;
```

Érezzük, hogy valami nincs rendben, ugye? Két egész számot osztunk egymással, de az eredmény tört lesz. Miért engedi a Java, hogy egy egész szám típus tárolására hivatott int típusú változóban tároljuk el. Onnan, hogy nem lehet biztos abban, hogy az eredmény tört lesz-e? Melegszik, de még mindig nem forró. A Javában – mint minden C típusú nyelvben – ha két integer típusú operandussal végzel műveletet, az eredmény mindig integer lesz! Azaz nem lesz törtrész! Még akkor sem, ha egy tört számok tárolására tervezett típusú változót használsz!

```
double x = 5 / 2;
System.out.println(x);
```

Ne lepődj meg, a fenti példa 2.0-t ír majd ki a képernyőre. Most kezd elmenni a kedved az aritmetikai műveletektől, igazam van? Remélem nem. Nyugalom, ez egy könnyen kezelhető jelenség. Az osztás művelet is helyesen viselkedik, ha legalább az egyik operandus törtszám. int x = 5 / 2; double x = 5 / 2; System.out.println(x);

```
double x = 5 / 2.0;
System.out.println(x);
```

Az eredmény 2.5. Mindegy, hogy az osztó, vagy az osztandó, esetleg mindkettő operandus tört szám-e, az eredmény biztosan helyes lesz. Rendben, probléma letudva. Vagy mégsem? Mi történik, ha két egész típusú változót kell egymással elosztanom? Nagyon egyszerű. Típuskonverziót kell végrehajtani.

```
int a = 9;
int b = 4;
double x = (double)a / b;
System.out.println(x);
```

Mindegy hogy float, vagy double típusra, legalább is, ha a konvertálandó szám „elfér” a float típusban is. Amennyiben nem, akkor csak a double használata javasolt.

A moduláris-, maradékos-, vagy modulo osztás használata igazán egyszerű. Az operátora %, az eredménye pedig a két operandus osztásának maradéka.

```
int m = 9 % 5;
System.out.println(m);
```

Tehát az előbbi utasítások hatására 4 jelenik meg a kimeneten.
– 5 % 2 esetén 1,
– 13 % 5 esetén 3,
– 4 % 2 esetén pedig 0.

1.1 Rövid értékadások

A Javában amennyiben egy változón szeretnél aritmetikai műveletet végezni, azt egyszerűbb módon is el lehet végezni. Vegyük például a következőt: egy változónak értéket adunk, amit meg szeretnénk növelni kettővel, majd felülírni vele a változó értékét.

```
int i = 12;
i = i + 2;
```

Viszont ezt egyszerűbb módon is megtehetjük:

```
int i = 12;
i += 2;
```

A fenti két művelet eredménye megegyezik. Ezt a módszer mind a négy alapműveletnél és a maradékos osztásnál is használhatjuk (+=, -=, *=, /=, %=).

1.2 Inkrementáló és dekrementáló utasítások

Az inkrementálás egy művelet, melynek segítségével egy változó értékét egy értékkel megnöveljük.

```
i = i + 1;
i += 1;
```

Azonban még a második formánál is meg tudjuk egyszerűbben adni:

```
i++;
++i;
```

Mind a négy utasítás eredménye az, hogy az i változó értéke egy értékkel növekszik.

A dekrementálás az inkrementálás ellentéte, egy művelet, amelynek hatására egy változó értéke eggyel csökken.

i = i - 1;
i -= 1;
i--;
--i;

Rendben, ez mind szép és jó. Érthető, hogy az első alak az alapvető művelet végrehajtás, a második egy rövidített forma, amelyek nem csak az egyel csökkentésre/növelésre használhatóak, hanem más értékekkel is elvégezhetik ugyanezt. De miért van i++ és ++i, valamint i-- és --i? A válasz az, hogy ugyan az az eredményük, de nem pontosan ugyan úgy érik el. Azaz ugyan azt csinálják, de nem ugyan azon a módon. A következő kódrészlet minden kiíró sorának kimenete megjegyzésben szerepel mellette.

```
int i = 4;
System.out.println(i++); // 4
System.out.println(i);   // 5

System.out.println(++i); // 6
System.out.println(i);   // 6
```

Milyen következtetéseket vonhatunk le ebből?

– Az i++ utasítás először behelyettesíti i értékét a kiíró utasításba, majd csak később növeli meg az értékét.

– A ++i utasítás először megnöveli i értékét, majd csak utána helyettesíti be az értékét a kiíró utasításba.

Pontosan ugyan így futnak le a --i és i—utasítások, csak ott egy értékkel csökkentik i változó értékét.

2. Kifejezések

Most, hogy megismertük az operátorok fogalmát, valamint működését, folytassuk a kifejezésekkel, amelyek felépítéséhez használhatunk operátorokat is.

A kifejezés literálok, változók, operátorok és függvényhívások olyan sorozata (a nyelv szintaxisát figyelembe véve), amely egy értéket ad vissza. A függvényhívástól most tekintsünk el, csak annyit jegyeznék meg, hogy lényegében a függvényhívások eredménye is egy érték, ugyan úgy, mint ahogy a változókra való hivatkozás segítségével is egy értéket helyettesíthetünk be egy kifejezésbe. Tehát példa kifejezésre:

*a * b + 12*

Láthatjuk, a fenti kifejezés tartalmaz változókat (a, b), operátorokat (*, +) és egy literált (12). Valamint azzal is tisztában kell lennünk, hogy ennek a kifejezésnek (amire eddig műveletként hivatkoztam) lesz egy eredménye, amelyet eltárolhatok egy változóban, vagy kiírhatunk a képernyőre, stb. Pl.:

```
int eredmeny = a * b + 12;
Vagy,
System.out.println(a * b + 12);
```

Tehát végeredményben a kifejezés változókkal, literálokkal és függvényhívásokkal végzett műveletek eredménye, amelyet például kiírhatunk a képernyőre, vagy egy értékadásnál az egyenlőségjel jobb oldalán használva átadhatunk a bal oldalon álló változónak.

A kifejezések között léteznek összetett kifejezések is. A fenti példa tekinthető összetett kifejezésnek is, hiszen több adattal/értékkel dolgozik. Az ilyen összetett kifejezéseknél viszont a műveletek végrehajtása – hasonlóan a matematikához – nem balról jobbra történik meg. Létezik egy, úgynevezett precedencia sorrend. Ez a sorrend határozza meg a műveletek végrehajtási sorrendjét.

2.1 Precedencia sorrend

Minden operátornak van egy adott precedenciája. Amelyik operátoré magasabb, az általa képzett művelet fog előbb kiértékelésre kerülni. Mint a matematikában, a szorzás és osztás művelet hamarabb kerül elvégzésre, mint az összeadás és a kivonás. Tehát a következő két kifejezés nem egyezik meg egymással:

$$x + y * 2$$
$$(x + y) * 2$$

„Ha összetett kifejezést írunk, akkor kifejezetten figyelnünk kell a zárójelezésre és arra, hogy mely operátoroknak kell kiértékelődniük elsőként. Ennek a gyakorlása segíthet a forráskód jobb olvashatóságának és karbantarthatóságának elérésében. A következő táblázat a Java platformban használt operátorok precedencia-szintjeit mutatja be. Az operátorok a táblázatban precedencia-szint szerint vannak rendezve: legfelül a legnagyobb precedenciával rendelkező található. A magasabb precedenciával rendelkező operátorok előbb hajtódnak végre, mint az alacsonyabbal rendelkezőek. Az azonos szinten elhelyezkedő operátorok azonos precedenciával rendelkeznek. Ha azonos precedenciájú operátorok szerepelnek a kifejezésben, akkor szabályozni kell, hogy melyik értékelődjön ki elsőként. Minden bináris operátor, kivéve az értékadó operátorok balról jobbra hajtódnak végre. Az értékeadó operátorok jobbról-balra hajtódnak végre.” (Nagy, 2007)

postfix	expr++ expr--
unáris	++expr --expr +expr -expr ~ !
multiplikatív	* / %
additív	+ -
léptetés	<< >> >>>
relációs	< > <= >= instanceof
egyenlőség	== !=
bitenkénti és	&

bitenkénti kizáró vagy	^
bitenkénti vagy	|
logikai és	&&
logikai vagy	||
feltételes	? :
értékadás	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Ne ess kétségbe, ha nem ismersz minden operátort a táblázatban. A tananyag során több operátorral is meg fogunk ismerkedni, valamint a későbbi tárgyak során is bővülni fognak ismereteink.

Összegezve az előző bekezdéseket: figyelnünk kell az összetett kifejezések kiértékelési sorrendjére. Ameddig nem vagyunk biztosak a precedencia sorrendben, használjunk zárójelezést. Azonban ne essünk túlzásba a használatukkal és gyakoroljuk a sorrendet, mert a túlzott zárójelezés rontja a forráskód olvashatóságát!

Források

Nagy G. (2007): Java programozás.

```java
package aritmeticoperators;

/**
 *
 * @author somlyaip
 */
public class AritmeticOperators {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        int sum = 3 + 4;
        System.out.println("Sum: " + sum);

        float a = 3.56f;
        float b = 8.97f;
        float difference = a - b;
        System.out.printf("Differnece: %.2f\n", difference);

        int c = 9;
        int product = c * 12;
        System.out.println("Product: " + product);

        int d = 4;
        float quotient = c / (float) d;
        System.out.printf("Quotient: %.2f\n", quotient);
    }

}
```

```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package aritmeticoperators;

/**
 * {Description}
 *
 * @author somlyaip
 */
public class RectangleAreaPerimeter {

    public static void main(String[] args) {

        int a = 2;
        int b = 3;
        int area;
        int perimeter;

        area = a * b;
        perimeter = 2 * (a + b);

        System.out.println("Area: " + area);
        System.out.println("Perimeter: " + perimeter);
    }
}
```

Bevezetés a programozásba

Formázott kiírás

Somlyai Pál

Dunaújvárosi Főiskola

2015

Tartalom

Bevezetés a programozásba – Formázott kiírás 1

1. Formázott kiírás

Számtalanszor előfordulhat, hogy nem szeretnénk megjeleníteni egy double típusú változó összes tizedesjegyét, vagy esetleg balra, vagy jobbra kell zárnunk egy érték kiírását. Az eddig ismert konkatenálás segítségével ezeket nem igazán tudtuk megoldani. Azonban a Java erre is nyújt számunkra megoldást. A kiíró utasítások közül eddig megismertük a System.out.print() és a System.out.println() utasítást. A System.out.printf(), vagy System.out.format() utasítás segítségével tudunk formázottan kiírni. Ez a két utasítás teljesen ekvivalens (azaz felcserélhető), bármelyiket használjuk, ugyan az lesz a végeredmény.

Természetesen, egy szövegen más formázási utasításokat tudunk elvégezni, mint egy tört számon. A formázási utasításokat a Javában konverterek segítségével tudjuk megoldani.

1.1 Konverterek

A formázási parancs megadása során a Javában szükségünk van egy behelyettesítési mintára, valamint a behelyettesítendő értékekre. A konverterek segítségével írjuk le az behelyettesítési mintában, hogy melyik helyre, milyen típusú adatok legyenek behelyettesítve. Minden konverter %-el kezdődik a formátumban (behelyettesítési mintában).

1. táblázat – Konverterek

Konverter	Jelentés
%d	Decimális egész szám
%f	Tört szám
%c	Karakter
%s	String
%n	Platformfüggetlen sortörés karakter[1]. Használata javasolt a \n escape szekvenciával szemben.

1 Bármelyik operációs rendszeren is futtatjuk, a megfelelő sortörés escape szekvenciát fogja elhelyezni. A sortörések operációs rendszerenként eltérhetnek, bár főként a fájlok sortörései vannak megkülönböztetve. A \n sortörés a konzolablakban működik Windowson és Linux alapú rendszereken is. Mac-en az alapértelmező parancsfuttató környezettől (~cmd Windowson) függ a működésük.

Nézzük meg, mi az eredménye a következő formázott kiírásoknak:

```
int i = 591;
double d = 1671.4926;
char c = 't';
String s = "String";

System.out.printf("egész szám: %d %n", i);  // egész szám: 591
System.out.printf("tört: %f %n", d);         // tört: 1671.492600
System.out.printf("%c %n", c);               // t
System.out.printf("szöveg: %s");             // szöveg: String
System.out.printf("%s, %d, %c", s, i, c);    // String, 591, t
```

Látható, hogy egy formátumban, több konverter is elhelyezhető, így mintába több adat is beilleszthető. Ilyenkor a konverterek sorrendjének, és a bemeneti adatok sorrendjének is meg kell egyeznie (balról jobbra).

A konverterek bemenete a konverzió bemenő adata (alapanyaga), a kimenete a konverzió kimenő adata (eredménye).

Figyeljünk oda, a konverterek és a bemenő adatok típusainak egyezésére. Amennyiben a formátumban szereplő konverter típusa és a hozzá tartozó bemenő adat típusa nem egyezik meg: java.util.IllegalFormatConversionException típusú hibát fogunk kapni.

Eddig csak a mintába beillesztés módjával ismerkedtünk meg, de a konverterek beállításaival nem. A konverterek beállítására a flagek használhatóak.

1.2 Flagek

Az informatikában a flag valamilyen kiegészítő adatot, vagy módosítót jelent. A konverterek esetében a flagek a behelyettesítés módját tudják módosítani. Az alábbi táblázat tartalmazza a lehetséges flageket és a hozzájuk tartozó legfontosabb információkat. A táblázat kimenet oszlopában egy szóközt egy # jelöl. A flageket a konverterekben kell elhelyezni, a % és az adattípust jelölő betű közé.

2. táblázat – Flagek

Típus	Flag példa	Jelentése	Bemenet	Kimenet
Bármelyik	5	A bemenet utolsó karaktere legalább 5 karakter távolságban helyezkedjen el a kurzor jelenlegi pozíciójától. A kurzortól kezdődően 5 karaktert lefoglal, a bemenetet kiírja jobbra igazítva, a maradék helyeket szóközökkel tölti fel.	45	###45
Egész- vagy tört szám	06	A bemenet utolsó karaktere legalább 6 karakter távolságban helyezkedjen el a kurzor jelenlegi pozíciójától, valamint alkalmazzon sorkitöltő nullákat szóköz helyett.	519	000519
	+	Pozitív szám esetén is jelenítse meg az előjelet.	16	+16
	,	Ezres csoportok használata, a csoportokat , választja el egymástól.	454443111	454,443,111
	-5	A kurzortól kezdődően 5 karaktert lefoglal, a bemenet kiírja balra igazítva, a maradék helyeket szóközökkel tölti fel.	45	45###
Tört szám	.3	Három tizedesjegy megjelenítése.	45.678123	45.678
	6.1	Az első flag és azt utolsó előtti kombinációja.	12.456	##12.4

A táblázatban használt flageket bemutató példák a Javában a következőek:

```
System.out.printf("%5d", 45);
System.out.printf("%06d", 519);
System.out.printf("%+d", 16);
System.out.printf("%,d", 454443111);
System.out.printf("%-5d", 45);
System.out.printf("%.3f", 45.678123);
System.out.printf("%6.1f", 12.456);
```

```
 1. package formazottkiiras;
 2. 
 3. /**
 4.  *
 5.  * @author somlyaip
 6.  */
 7. public class FormazottKiiras {
 8. 
 9.     /**
10.      * @param args the command line arguments
11.      */
12.     public static void main(String[] args) {
13.         int i = 72;
14.         double d = 49.3745;
15.         char c = 'é';
16.         String s = "text";
17. 
18.         System.out.printf("%d\n", i);
19.         System.out.format("%f%n", d);
20. 
21.         System.out.printf("Az i értéke: %d%n", i);
22. 
23.         System.out.printf("c: %c, s: %s%n", c, s);
24. 
25.         System.out.printf("%5d%n", i);
26.         System.out.printf("%06d%n", i);
27. 
28.         System.out.printf("%+d%n", 45);
29.         System.out.printf("%+d%n", -45);
30. 
31.         System.out.printf("%,d%n", 454411313);
32. 
33.         System.out.printf("%-5d|%n", i);
34. 
35.         System.out.printf("%.3f%n", d);
36. 
37.         System.out.printf("%6.1f%n", 12.454567);
38.         System.out.printf("%12f%n", 12.454567);
39.     }
40. 
41. }
```

Bevezetés a programozásba
Gyakorló feladatok

2. FEJEZET

1. Készítsen programot, amely két változóban tárolt egész szám összegét megjeleníti a képernyőn:

– A kiírás adjon helyes eredményt akkor is, ha megváltoztatjuk a forráskódban a változók értékét!

Példa kimenet:

12 + 4 = 16

2. Készítsen programot, amely megjeleníti két változóban tárolt egész szám hányadosát két tizedesjegy pontossággal:

– A kiírás adjon helyes eredményt akkor is, ha megváltoztatjuk a forráskódban a változók értékét!

Példa kimenet:

5 / 2 = 2.50

3. Készítsen programot, amely két változóban tárolt tört szám szorzatát megjeleníti négy tizedesjegy pontossággal, az operandusokat két tizedes pontossággal:
– A kiírás adjon helyes eredményt akkor is, ha megváltoztatjuk a forráskódban a változók értékét!

Példa kimenet:

2.41 * 6.12 = 14.7492

4. Ismert egy három oldala az a, b és c változóban. Az oldalak törtszámokat is képesek legyenek eltárolni! Számítsa ki a téglatest felszínét és tárfogatát, majd jelenítse meg két tizedesjegy pontossággal!

5. Készítsen programot, amely megjeleníti a képernyőn a 459 értéket a következő módon:

– A kiíráshoz a System.out.printf(), vagy a System.out.format() utasítást használja

– A kiírást egyetlen kiíró utasítással valósítsa meg

A kimenet (egy #, egy szóközt jelöl):

###459

6. Készítsen programot, amely megjeleníti a képernyőn a 12.5659 értéket a következő módon:

– A kiíráshoz a System.out.printf(), vagy a System.out.format() utasítást használja

– A kiírást egyetlen kiíró utasítással valósítsa meg

A kimenet (egy #, egy szóközt jelöl):

12.57

7. Készítsen programot, amely megjeleníti a képernyőn a 32.12 értéket a következő módon:

– A kiíráshoz a System.out.printf(), vagy a System.out.format() utasítást használja

– A kiírást egyetlen kiíró utasítással valósítsa meg

A kimenet (egy #, egy szóközt jelöl):

32.120###

8. Készítsen programot, amely megjeleníti a képernyőn a 46, é, alma értéket a következő módon:

– A kiíráshoz a System.out.printf(), vagy a System.out.format() utasítást használja

– A kiírást egyetlen kiíró utasítással valósítsa meg

A kimenet (egy #, egy szóközt jelöl):

```
46##é
alma
```

9. Készítsen programot, amely megjeleníti a képernyőn a következő táblázatot:

A kimenet:

Gyümölcs	HUF/Kg
Alma	221
Körte	450
Eper	980
Datolya	1200

3. fejezet

Bevezetés a programozásba

Logikai érték, logikai műveletek

Somlyai Pál

Dunaújvárosi Főiskola

2015

Tartalom

Bevezetés a programozásba – Logikai érték, logikai műveletek 1

1. Logikai érték

A logikai értékek olyan adatok (~értékek), amelyek csak igazak, vagy hamisak lehetnek. Vegyük példának a Facebookra történő bejelentkezést. A weboldalon megadjuk az e-mail címünket és a hozzá tartozó jelszavunkat, majd rákattintunk a bejelentkezés gombra. A belépési adataink (e-mail, jelszó) helyességének logikai értéke vagy igaz, vagy hamis lehet, más lehetőségünk nincs. Igaz abban az esetben, ha mindkét belépési adatunkat megfelelően írtuk be, ellenkező esetben pedig hamis. Minden bejelentkezéshez kötött oldal így működik, ha helyesek a belépési adatok: beléptet, ha nem: akkor nem enged „belépni" az oldalra. Tehát a belépési adataink helyessége egy logikai érték.

Logikai értékeket akkor használhatunk, ha csak két kimenetele lehet egy vizsgálatnak. Lényegében a logikai értékek eldöntendő kérdések válaszai lehetnek (igen=igaz, nem=hamis). Az előző esetben az eldöntendő kérdés a következő volt: „A felhasználó helyesen adta meg a belépési adatait?". Erre csak az igen, vagy a nem válasszal válaszolhatunk. Nincs köztes út (pl.: az e-mail cím helyes, a jelszó helytelen), mivel az ilyen esetekben nem engedheti be a szoftver a felhasználót. Tehát a programozó számára minden „talán", esetleg „majdnem" esemény lényegtelen, neki azokkal nem kell törődnie.

A Javában lehetőségünk van logikai értékek eltárolására. A már ismertetett boolean típus segítségével. Tudjuk, hogy a boolean típus értéke true, vagy false lehet. Az informatikában különböző jelölésekkel találkozhatunk a logikai értékek leírására (lásd: 1. táblázat).

1. táblázat – Logikai értékek jelölése

Típus	Igaz	Hamis
Bináris (kettes számrendszerbeli)	1	0
Magyar rövidítés	I	H
Angol szó	true	false
Angol rövidítés	T	F

Tekintsünk meg az említetten felül néhány példát logikai értékekre vonatkozó eldöntendő kérdésekre:

– A 45 nagyobb-e mint a 10?

– Csütörtökön 15:00-tól van-e órám?

– A számítógépben van-e legalább 1 GB szabad memória?

– Van elég pénzem a jövő heti bulira?

Minden egyes kérdés esetében a válaszunk kizárólag igaz, vagy hamis lehet. Adjuk meg a választ az egyik kérdésre Java nyelven:

boolean *a45NagyobbMint10* = ***true;***

2. Logikai műveletek

Természetesen a logikai értékek felhasználása nem merül ki abban, hogy a programozó manuálisan beállítja (~"kézzel", azaz beírja a programkódba) a logikai változók értékeit. Említettük már az aritmetikai műveleteket, amelyek esetében számokon végzünk műveleteket (azaz az operandusok számok) és minden esetben az eredmény is szám lesz. A logikai műveletek esetében az operandusok „tetszőleges" típusúak lehetek, azonban az eredmény mindig egy logikai érték lesz. Az logikai műveletek első típusa a relációs műveletek.

2.1 Relációs műveletek

A relációs (összehasonlító) műveleteket két közel azonos típusú értéken végezzük. Az értékek literálok, változók, illetve konstansok[1] lehetnek, a műveletek pedig a matematikából ismert relációs műveletek:

– kisebb (<)

– nagyobb (>)

– kisebb, vagy egyenlő (<=)

– nagyobb, vagy egyenlő (>=)

Emellett használhatjuk még az egyenlő (==), illetve a nem egyenlő (!=) műveletet a közel azonos típusú értékeken. A „közel azonos" kifejezés arra utal, hogy ezek a műveletek a számok esetében elvégezhetőek két egész számon, két törtszámon, illetve egy egész és egy törtszámon. Emellett, mivel a karakterek tárolásukat tekintve számok, valamint mivel az int típuskényszerítés segítségével megkapható az Unicode karakterkódjuk (ami egy szám), így ők is hasonlíthatóak egy számhoz.

1 Valamint függvényhívás eredménye is, de velük csak később fogunk megismerkedni.

2 Valójában Stringek esetén más módon kellene hasonlítanunk őket, amit egy későbbi anyagrész során meg is fogunk tanulni.

A rajtuk kívül fennmaradó adattípust (String[2], boolean) már csak az egyenlő és nem egyenlő operátor segítségével hasonlíthatunk egymáshoz.

Példák relációs műveletekre:

```
int i = 12;
char c = 'a'; // az 'a' karakter Unicode kódja 97
System.out.println((int)c); // 97
boolean b1 = i < c;
System.out.println(b1); // true

float f = 12.34f;
boolean b2 = i >= f;
System.out.println(b2); // false

String s1 = "t";
String s2 = "t";
System.out.println(s1 == s2); // true
System.out.println(s1 != s2); // false
```

Számos alkalommal fordulhat elő, hogy nem elég egy tulajdonság vizsgálata (azaz egy eldöntendő kérdés feltevése), hanem egyszerre több tulajdonság teljesülését kell figyelembe vennünk. Az első bekezdésben említett példa során sincs másképpen. A felhasználó csak akkor tekinthető hitelesített felhasználónak, ha igazolta, hogy ismeri a fiókjához felhasználónév/e-mail cím és jelszó párost. Viszont ebben az esetben azt kell megvizsgálnunk, hogy a megadott e-mail cím egyezik-e egy eltárolt e-mail címmel, valamint a megadott jelszó egyezik-e a hozzá tartozó jelszóval. Több logikai változó, vagy logikai művelet eredményét felhasználva összetett logikai kifejezéseket készíthetu nk, amely megoldást nyújt a problémánkra.

2.2 Összetett logikai kifejezések

Összetett logikai kifejezések építéséhez a logikai alapműveleteket (ÉS, VAGY, NEGÁCIÓ) használhatjuk operátorként, operandusokként pedig logikai értékeket. Logikai alapműveletek:

– ÉS (AND):
  - akkor teljesül, ha mindkét operandus értéke IGAZ
  - operátor: &&

– VAGY (OR):
  - akkor teljesül, ha legalább az egyik operandus értéke IGAZ
  - operátor: ||

– NEGÁCIÓ (NOT, azaz NEM):
  - egy egyoperandusú logikai alapművelet
  - a megkapott operandus értékének ellenkező értékét adja vissza
  - operátor: !

A logikai alapműveleteket a következő táblázat (2. táblázat) tartalmazza. A táblázatban az Igaz értéket az „1”, a Hamis értéket pedig a „0” jelöli.

2. táblázat - Igazságtáblázat

a	b	a AND b	a OR b	NOT a	NOT b
1	**1**	1	1	0	0
1	**0**	0	1	0	1
0	**1**	0	1	1	0
0	**0**	0	0	1	1

Az előbb említett (bejelentkezési) példára az ÉS művelet használata a megoldás. Az „a” operandus értéke legyen az e-mail cím helyes kérdés válasza, a „b” operandusé pedig a jelszó helyes kérdés eredménye. Csak akkor engedhetjük be a felhasználót, ha mindkét operandus értéke igaz, így az ÉS műveletet kell használnunk.

A feladat megvalósítás Javában:

```
String emailEltarolt = "...";  // eltárolt érték
String jelszoEltarolt = "..."; // eltárolt érték

String emailMegadott = "john.doe@example.com";
String jelszoMegadott = "x13(_:1zL";

boolean helyesHitelesites =
      emailEltarolt == emailMegadott && jelszoEltarolt == jelszoMegadott;
```

Természetesen az értékadás sorában a sortörés nem kötelező, csak azért helyeztem el, hogy elférjek a sorban.

Az összetett logikai kifejezés első operandusa az emailEltarolt == emailMegadott művelet eredménye, a második pedig jelszoEltarolt == jelszoMegadott. Ha eltárolnánk az első operandusként megjelenő művelet eredményét egy „a” nevű logikai változóban (boolean a), valamint a második operandusként megjelenő művelet eredményét egy „b” nevű logikai változóban, akkor az igazságtábla alapján kiolvashatjuk, hogy a és b változó milyen értékeinél lenne igaz az összetett logikai kifejezésünk.

```java
package logicalvalues;

/**
 *
 * @author somlyaip
 */
public class LogicalValues {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        boolean trueValue = true;
        boolean falseValue = false;

        boolean notTrueValue = !trueValue;
        System.out.println("!trueValue? " + notTrueValue);

        boolean is2GreaterThan5 = 2 > 5;
        System.out.println("2 > 5? " + is2GreaterThan5);

        double x = 9.999;
        boolean isXLesserThan10 = x < 10;
        System.out.println("X < 10? " + isXLesserThan10);

        int a = 42;
        boolean isAEquals42 = a == 42;
        System.out.println("a == 42? " + isAEquals42);

        boolean isXIsNotEqualsA = x != a;
        System.out.println("x != a? " + isXIsNotEqualsA);
    }

}
```

Bevezetés a programozásba

Elágazások

Somlyai Pál

Dunaújvárosi Főiskola

2015

Tartalom

Bevezetés a programozásba - Elágazások 1

1. Elágazások

Számos alkalommal fordulhat elő, hogy egy programnak, egy bizonyos feltétel teljesüésének függvényében más-más műveleteket kell végrehajtania. Ha az előző bejelentkezési példánál maradunk, amennyiben helyesek a megadott hitelesítő adatok: be kell jelentkeztetni a felhasználót, amennyiben helytelenek: hibaüzenetet kell megjeleníteni a számára.

Az ilyen eseteket nevezzük a programozásban elágazásnak. A neve onnan ered, hogy a program végrehajtása, egy bizonyos feltétel teljesülése alapján elágazik. (lásd *1. ábra*). A bizonyos feltétel jelen esetben az adatok helyessége.

1. ábra – Bejelentkezési elágazás

Amennyiben a megadott hitelesítési adatok helyesek, a program a „Helyes adatok” úton halad tovább, ellenkező esetben pedig a „Helytelen adatok” úton folytatja a program végrehajtását. A helyes adatok út végén a bejelentkeztetés művelete szerepel, a helytelen adatok út végén pedig a hibaüzenet megjelenítése. A Java nyelv első elágazása az egyszerű if-else.

1.1 Egyszerű if-else elágazás

Az egyszerű if-else esetén egy adott logikai kifejezés értéke alapján választunk útvonalat az elágazásban.

Az if-else szerkezet lényegében 3 részből áll:

- if (ha)
  - a kiértékelendő logikai kifejezés
  - feltételnek nevezzük

- then (akkor)
  - mi történjen, ha a kiértékelt logikai kifejezés értéke true

- else (különben)
  - mi történjen, ha a kiértékelt logikai kifejezés értéke false

Az if-else vezérlési szerkezet felépítése a Javában:

```
if(feltétel) {
        utasítás1;
        utasítás2;
        ...
        utasításN;
} else {
        utasítás1;
        utasítás2;
        ...
        utasításN;
}
```

A zöld kapcsos zárójelekkel határolt utasításblokk a then (akkor) ág (azaz elágazási ág), a pirossal jelölt utasításblokk pedig az else ág. Amennyiben a feltétel kiértékelésének eredménye true: a then ágban folytatódik az utasítások végrehajtása, amennyiben false: az else ágban. Ebből következik, hogy egyszerre csak a then, vagy az else ág hajtódhat végre, azonban az egyik eset mindig be fog következni. Nem létezik olyan eset, hogy mindkettő végrehajtódhat, azonban olyan sem, hogy egyik sem kerül végrehajtásra.

2. ábra – Egyszerű if-else elágazás

Az if feltétel kiértékelésének eredménye az a „döntés”, amit az ábra alapján a sofőr (azaz a mi esetünkben a számítógép, amely a programot futtatja) meghoz. Mivel vagy balra, vagy jobbra haladhat tovább, egyenesen nem, ezért döntenie kell. Ha a „döntés” true: a then ágat választja, ha false: az else ágat.

Példa if-else vezérlési szerkezetre a Javában:

```
if(a > b) {
      System.out.println("Az \"a\" változó értéke nagyobb.");
} else {
      System.out.print("A \"b\" változó értéke nagyobb. ");
      System.out.println("Vagy egyenlőek.");
}

System.out.println("Vége.");
```

Az előbbi példa során, az elágazásunk feltétele az „a > b” logika művelet eredménye. Amennyiben az „a” változó értéke valóban nagyobb, mint a „b” változó értéke: a képernyőn „Az ”a” változó értéke nagyobb.” szöveg jelenik meg. Amennyiben a „b” változó értéke nagyobb, vagy egyenlő, mint az „a” változó értéke a program ennek megfelelő szöveget ír ki a képernyőre. Jegyezzük meg, hogy az else ág nem az if feltétel-el ellentétes esetben fut le, hanem minden más esetben, azaz amikor az if feltételben szereplő logikai kifejezés értéke hamis. Mivel az összehasonlítás arra vonatkozik, hogy a értéke nagyobb mint b, ezért a fennmaradó esetek:

– b nagyobb, mint a, vagy

– b és a egyenlőek.

Bármelyik eset is áll fenn az előző kettő közül, az else ág utasításai hajtódnak végre.

Emellett nagyon fontos megjegyezni azt, hogy az if és az else ág végrehajtása után az ágak újból egyesülnek, azaz bármelyik ág is fusson le az előző példánkban, a „Vége.” szöveg minden esetben meg fog jelenni a képernyőn.

1.1.1 If szerkezet

Lehetőségünk van csak az if utasítást használni, azaz az else ágat elhagyni. Ebben az esetben ha az if feltétel teljesül, végrehajtjuk a then ágat, ellenkező esetben „nem teszünk semmit", azaz az elágazás utáni utasítások végrehajtásával folytatjuk.

Az if szerkezet felépítése:

```
if(feltétel) {
        utasítás1;
        utasítás2;
        ...
        utasításN;
}
```

1.1.2 Egyetlen utasítást tartalmazó ágak

Amennyiben a then és/vagy else águnk csak egy utasítást tartalmaz, nem szükséges kapcsos zárójelek között elhelyeznünk őket:
Nyugodtan használjuk az előző szerkezetet, amennyiben csak egy-egy utasítást tartalmaznak az ágak, azonban legyünk körültekintőek a használatuk során.

Vegyük a következő példát:

```
if(feltétel)
        utasítás1;
else
        utasítás2;
```

Mi jelenik meg a képernyőn, ha lefuttatjuk ezt a programrészletet? Természetesen „Az "x" változó értéke kisebb." szöveg. Mi történne, ha x értékét 15-re változtatnám? Semmi sem jelenne meg a képernyőn? Dehogy nem. A képernyőn a „kisebb." szöveg jelenne meg. Ennek az az oka, hogy, hiába toltam egy tabulátorral beljebb a második utasítást az if feltétel alatt, az nem a then ághoz tartozik. Mivel nem használtam utasításblokkot (láthatjuk, hogy nincsenek kapcsos zárójelek), azért csak az első kiíró utasítás tartozik a then ághoz, a második pedig valójában az elágazást követő első utasítás.

1.2 Összetett if-else elágazás

Bizonyos esetekben előfordulhat, hogy két ág nem elég. Ahogy az előző példában is láthattuk, ha két változó közül meg szeretnénk állapítani, hogy melyik a nagyobb, illetve egyenlőek-e, nem tudunk minden helyzetet külön kezelni az egyszerű if-else szerkezettel. Az ilyen bonyolultabb helyzetek re kínál megoldást az összetett if-else szerkezet:

```
if(feltétel1) {
        utasítás1;
        ...
} else if(feltétel2) {
        utasítás1;
        ...
}
...
else if(feltételN) {
        utasítás1;
        ...
}
else {
        utasítás1;
        ...
}
```

Mint láthatjuk, több feltételt is megadhatunk, az elsőt csak az if kulcsszót követően, a többit pedig az else if kulcsszavakat követően. Ebben az esetben, ha az első feltétel nem teljesül, akkor a második kiértékelésével folytatja, stb. (Nincs áthágható korlátja a feltételek darabszámának.) Ha egyik feltétel sem teljesül, az else ág utasításait fogja végrehajtani. Itt is fontos megjegyezni, hogy egyszerre csak egy ág hajtódhat végre. Ha pl. a második feltétel kiértékelése igaz, az ő then ágában lévő utasítások lefutnak, majd az összetett if-else szerkezetet követő utasítások végrehajtása következik.

Az else ág itt is elhagyható, tehát nem kötelező olyan ágat írni, amelyben található utasítások akkor futnak le, ha egyik feltétel sem teljesült. Emellett itt is lehetséges elhagyni az utasításblokkokat, ha az ágban csak egy utasítás szerepel.

Példa az összetett if-else használatára:

```
if(a > b) {
      System.out.println("Az \"a\" változó értéke nagyobb.");
} else if(b > a){
      System.out.println("A \"b\" változó értéke nagyobb. ");
} else {
      System.out.println("Az \"a\" és \"b\" változó értéke egyenlő.");
}
```

1.3 Egymásba ágyazott if-else szerkezet

Az egyszerű- és az összetett if-else szerkezetre is igaz, hogy then, illetve else ágaikban elhelyezhetünk egyszerű-, vagy akár összetett if-else szerkezeteket. A lehetőségek tárháza szinte határtalan, tetszőleges módon kombinálhatjuk az eddig megismert if-else szerkezeteket:

```
if(feltétel1) {
        utasítás1;
        ...
        if(feltétel2) {
                utasítás1;
                ...
        } else {
                utasítás1;
                ...
        }

} else {
        utasítás1;
        ...
}
```

Ebben az esetben, ha az első feltétel teljesül, lefut az utasítás1 (bármilyen utasítás is legyen ☺), majd elérjük a beágyazott if-else szerkezetet, amely feltételének (feltétel2) kiértékelésének függvényében végrehajtjuk a then, vagy az else ágát, majd a külső if-else szerkezetet követő utasítások végrehajtásával folytatjuk.

Amint említettem, a lehetőségek tárháza szinte határtalan, azonban a túlzottan bonyolult összetett- és/vagy egymásba ágyazott if-else szerkezetek használata nem célra vezető, hiszen borzasztóan ronthatja az olvashatóságot (lásd *3. ábra*).

3. ábra – Túlgondolt If-else szerkezet

```
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```

pikabu.ru

A képen látható túlbonyolított if-else szerkezet olvashatóságán nem csak a kép alacsony felbontása ront ☺ (sokat kutattam utána, de nem találtam jobb minőségben). Ez egy weboldali regisztráció egy részét elvégző PHP kód (lényegében az adatellenőrzéseket tartalmazza, valamint ellenőrzi, hogy a felhasználó már regisztrált-e). Azonban csak remélni tudom, hogy a programkód is kizárólag szórakoztatásra készült, mert még ilyen alacsony felbontás mellett is több hibát vélek felfedezni benne, amit ez a finoman szólva is rosszul strukturált if-else szerkezet csak tovább fokoz.

Szerencsénkre sok bonyolult, összetett if-else szerkezetet ki tudjuk váltani a switch-case elágazás segítségével.

1.4 Switch-case

Míg az if-else elágazás alapértelmezetten kétirányú elágazás, amelyet a korábban említett praktikák használatával többirányú elágazásokká tudunk fejleszteni, addig a switch-case a kezdetektől többirányú elágazásnak lett tervezve.

A switch szó jelentése ebben az esetben kapcsol, kapcsoló, a case-é pedig eset. Jelen esetben a kapcsolót értsük úgy, mint egy vasúti vágányváltót (az egyik jelentése konkrétan ez is a switch szónak), a case szót pedig fogjuk fel vágányként. A váltó a felsorolt esetek alapján a megfelelő vágányra kapcsolja a végrehajtást, mintha a vonatszerelvény lenne a program végrehajtása.

Egy nem szószerinti értelmezés szerint a switch-case kiválasztja a felsorolt lehetőségek közül a megfelelőt, aminek átadja a program végrehajtását.

A switch-case elágazás szerkezete a következő:

```
switch(kifejezes) {
        case ertek1:
                utasitas1;
                ...
                break;
        case ertek2:
                utasitas1;
                ...
                break;
        ...
        case ertekN:
                utasitas1;
                ...
                break;
        default:
                utasitas1;
                ...
```

Működése során a felsorolt lehetőségek (esetek) közül kiválasztja a *switch* kulcsszó után megadott kifejezéssel megegyező értékű esetet, majd végrehajtja a hozzá tartozó ágban szereplő utasításokat. Ezt követően (a *break* miatt) kilép az elágazásból, majd az elágazást követő utasítások végreahjtásával folytatja. Fontos megjegyezni, hogy itt nincs szükség az egyes esetekhez tartozó ágakat utasításblokkokba szervezni, csupán a break utasítással szükséges zárni őket. Emellett a default ág feleltethető meg egy összetett *if-else* elágazás else ágának. Ez az ág fog lefutni, ha a felsorlot esetek közül egyik sem felel meg a kifejezés értékének. Tekintsük át a következő *switch-case* példát, amely egy négyműveletes számológépet valósít meg:

```
int operandus1 = 5;
int operandus2 = 12;
char operator = '-';

switch(operator) {
	case '+':
		System.out.printf("Eredmény: %d\n", operandus1 + operandus2);
		break;
	case '-':
		System.out.printf("Eredmény: %d\n", operandus1 - operandus2);
		break;
	case '*':
		System.out.printf("Eredmény: %d\n", operandus1 * operandus2);
		break;
	case '/':
		System.out.printf("Eredmény: %d\n", operandus1 / operandus2);
		break;
	default:
		System.out.println("Helytelen operátor: " + operator);
}
```

Látható, hogy a program futása függ az operator változó értékétől. Mivel ő szerepel a *switch* kulcsszót követően, ezért az ő értékével megegyező eset utasítása fog lefutni. Mivel az *operator* értéke „-”, ezért a második case ág fog lefutni, aminek eredményeként megjelenik a két operandus hányadosa (-7) a képernyőn. Abban az esetben, ha nem a megfelelő operátort adtuk volna értékül az operator változónak, a „Helytelen operátor:” szöveg és az *operátor* értéke jelenne meg a képernyőn. A példa elágazás egy ötirányú elágazás, hiszen az operátor értékének függvényében öt különböző irányban folytatódhat a program végrehajtása. A switch-case esetében sincs maximum korlátozása a case ágak darabszámának. A *default* ág pedig szabadon elhagyható.

Mint ahogy említettem, a swtich-case elágazás az összetett if-else elágazás egyszerűsített változata. Tehát amit meg lehet valósítani switch-case-el, azt le tudjuk programozni egy összetett if-else elágazással.

A példa switch-case if-else párja a következő:

```
int operandus1 = 5;
int operandus2 = 12;
char operator = '-';

if(operator == '+') {
     System.out.printf("Eredmény: %d\n", operandus1 + operandus2);
}
else if(operator == '-') {
     System.out.printf("Eredmény: %d\n", operandus1 - operandus2);
}
else if(operator == '*') {
     System.out.printf("Eredmény: %d\n", operandus1 * operandus2);
}
else if(operator == '/') {
     System.out.printf("Eredmény: %d\n", operandus1 / operandus2);
}
else {
     System.out.println("Helytelen operátor: " + operator);
}
```

2. Feltételes operátor

A feltételes operátor körülbelül félúton helyezkedik el az operátorok és az elágazások között. Mivel operátor, ezért valamilyen műveletet végez el, amelynek eredménye lesz. Mivel elágazás is, ezért valamilyen logikai értéktől fog függeni a művelet eredménye. Egy egyszerű if-else elágazáshoz hasonlóan kétirányú elágazásként működik.

A feltételes operátor szerkezete a következő:

feltetel ? ertekHaIgaz : ertekHaHamis

Tekintsünk meg egy példát:

```
int x = 5;
String szoveg = x % 2 == 0 ? "páros" : "páratlan";
```

A kódrészlet végrehajtását követően a szöveg változó értéke „páratlan” lesz. (x% 2: x kettővel történő osztásának maradéka, ami 1). Az előző példát továbbfejlesztve rögtön kiírhatjuk a feltételes operátor által visszaadott értéket:

```
int x = 5;
System.out.println("x " + (x % 2 == 0 ? "páros" : "páratlan"));
```

Itt azonban figyelnünk kell a zárójelezésre, mert a feltételes operátor egy összetett operátor, ezért ha egy Stringhez szeretnénk fűzni, vagy egy integerhez hozzáadni az eredményét, a teljes feltételes operátort zárójelbe kell helyeznünk. Az előző példa ekvivalens (~egyenértékű, azaz felcserélhető) a következő kódrészelttel:

```
int x = 5;
System.out.print("x ");
if(x % 2 == 0)
	System.out.println("páros");
else
	System.out.println("páratlan");
```

Látható, hogy az érdemi rész feltételes operátor használatával egy sor, míg egyszerű if-else használatával öt sor. A feltételes operátor igazi ereje a tömörsége, sok helyzetben kényelmesebb és célszerűbb az alkalmazása, mint az if-else elágazásnak.

3. Érvényességi kör

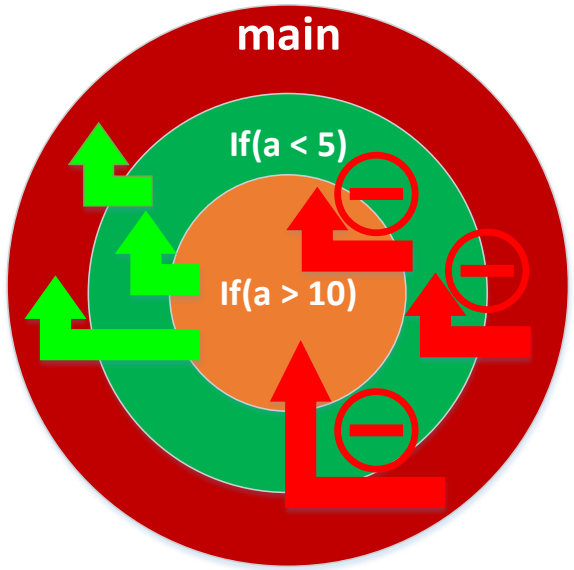
Egy változó érvényességi köre a programkód azon része, amelyen belül a változó jelentése ismert (~használható). Egy változóra nem tudunk hivatkozni az érvényességi körén kívül. Egy változó érvényességi körét általában az szabja meg, hogy hol definiáltuk. Eddig minden változót a main függvényen belül hoztunk létre, így azok elérhetőek voltak a main függvényen belül. Azonban mostantól, az elágazások ágaiban külön érvényességi köröket hozunk létre. Valamint egy egymásba ágyazott if-else elágazásban minden beágyazott elágazás, minden ága egy-egy külön érvényességi kör.

Lényegében az érvényességi körök határait az egymásba ágyazott utasításblokkok alkotják.

```
public static void main(String[] args) {
	int a = 4;
	if(a < 5) {
		int b = 5;
		if(a > 10) {
			int c = 12;
			System.out.println(c);
		} else {
			System.out.println(b);
		}
		System.out.println(a);
		System.out.println(c); // érvénytelen
	}
	else {
		System.out.println(b); // érvénytelen
	}
	System.out.println(b); // érvénytelen
	System.out.println(c); // érvénytelen
}
```

Mint ahogy az a kódrészleten és az ábrán látható, a main utasításblokkján belül helyezkedik el az if(a < 5) elágazás then utasításblokkja, amin belül található az if(a > 10) elágazás then utasításblokkja.

Az // *érvénytelen* komment azt jelenti, hogy az adott helyről nem érjük el a kiírandó változót.



A belső utasításblokkok beletartoznak egy külső utasításblokkban definiált változó érvényességi körének, azonban egy belső blokkban definiált változó érvényességi köréhez nem tartoznak hozzá a külső blokkok.

Ez röviden annyit tesz, hogy a külső utasításblokkok változóihoz hozzáférünk belülről, míg a belső utasításblokkok változóihoz nem férünk hozzá kívülről.

Emellett fontos megjegyeznem, hogy egy if-else elágazás then és else ága két külön érvényességi kör. Ha létrehozunk egy változót a then ágaban, azt az else ágból nem fogjuk tudni elérni.

```
1.  /*
2.   * To change this license header, choose License Headers in Project Properties.
3.   * To change this template file, choose Tools | Templates
4.   * and open the template in the editor.
5.   */
6.  package statements;
7.
8.  /**
9.   *
10.  * @author somlyaip
11.  */
12. public class SimpleIfElse {
13.
14.     /**
15.      * @param args the command line arguments
16.      */
17.     public static void main(String[] args) {
18.
19.         int x = 13;
20.         int y = 3;
21.
22.         if(x > y) {
23.             System.out.println("X is greater than Y");
24.         } else {
25.             System.out.println("X is lesser than or equal Y");
26.         }
27.
28.     }
29.
30. }
```

```
1.  /*
2.   * To change this license header, choose License Headers in Project Properties.
3.   * To change this template file, choose Tools | Templates
4.   * and open the template in the editor.
5.   */
6.  package statements;
7.
8.  /**
9.   * {Description}
10.  *
11.  * @author somlyaip
12.  */
13. public class ComplexIfElse {
14.
15.     public static void main(String[] args) {
16.
17.         int x = 0;
18.
19.         if(x > 0)
20.             System.out.println("The number is pozitive.");
21.         else if(x < 0)
22.             System.out.println("The number is negative.");
23.         else
24.             System.out.println("The number is zero.");
25.     }
26. }
```

```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package statements;

/**
 * {Description}
 *
 * @author somlyaip
 */
public class CalculatorIfElse {

    public static void main(String[] args) {

        int numberFirst = 5;
        int numberSecond = 8;
        char operator = '+';
        float result;

        if(operator == '+') {
            result = numberFirst + numberSecond;
            System.out.printf("%d %c %d = %.4f\n",
                    numberFirst, operator, numberSecond, result);
        } else if(operator == '-') {
            result = numberFirst - numberSecond;
            System.out.printf("%d %c %d = %.4f\n",
                    numberFirst, operator, numberSecond, result);
        } else if(operator == '*') {
            result = numberFirst * numberSecond;
            System.out.printf("%d %c %d = %.4f\n",
                    numberFirst, operator, numberSecond, result);
        } else if(operator == '/') {
            result = numberFirst / (float) numberSecond;
            System.out.printf("%d %c %d = %.4f\n",
                    numberFirst, operator, numberSecond, result);
        } else {
            System.out.println("Operator is not valid.");
        }
    }
}
```

```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package statements;

/**
 * {Description}
 *
 * @author somlyaip
 */
public class CalculatorSwitchCase {

    public static void main(String[] args) {

        int numberFirst = 5;
        int numberSecond = 8;
        char operator = '-';
        float result;

        switch(operator) {
            case '+':
                result = numberFirst + numberSecond;
                System.out.printf("%d %c %d = %.4f\n",
                    numberFirst, operator, numberSecond, result);
                break;
            case '-':
                result = numberFirst - numberSecond;
                System.out.printf("%d %c %d = %.4f\n",
                    numberFirst, operator, numberSecond, result);
                break;
            case '*':
                result = numberFirst * numberSecond;
                System.out.printf("%d %c %d = %.4f\n",
                    numberFirst, operator, numberSecond, result);
                break;
            case '/':
                result = numberFirst / (float) numberSecond;
                System.out.printf("%d %c %d = %.4f\n",
                    numberFirst, operator, numberSecond, result);
                break;
            default:
                System.out.println("Operator is not valid.");
                break;
        }
    }
}
```

```
 1. /*
 2.  * To change this license header, choose License Headers in Project Properties.
 3.  * To change this template file, choose Tools | Templates
 4.  * and open the template in the editor.
 5.  */
 6. package statements;
 7.
 8. /**
 9.  * {Description}
10.  *
11.  * @author somlyaip
12.  */
13. public class ConditionalOperator {
14.
15.     public static void main(String[] args) {
16.
17.         int x = 41;
18.         int y = 10;
19.         int max = x > y ? x : y;
20.         System.out.println("max: " + max);
21.
22.         int a = 6;
23.
24.         if(a % 2 == 0)
25.             System.out.println("'a' is an even number.");
26.         else
27.             System.out.println("'a' is an odd number.");
28.
29.         System.out.print("'a' is an ");
30.         if(a % 2 == 0)
31.             System.out.print("even");
32.         else
33.             System.out.print("odd");
34.         System.out.println(" number.");
35.
36.         System.out.println("'a' is an " +
37.                 (a % 2 == 0 ? "even" : "odd") + " number.");
38.     }
39. }
```

Bevezetés a programozásba
Ciklusok

Somlyai Pál

Dunaújvárosi Főiskola

2015

Tartalom

Bevezetés a programozásba – Ciklusok 1

1. Ciklusok

A ciklusok alkotják a vezérlési szerkezetek másik részét (az elágazások mellett). A ciklusok arra hivatottak, hogy adott utasításokat ismételjenek addig, ameddig a feltételu k teljesu l. A ciklus neve a ciklikus jelzőből ered, amely jelentése visszatérő, körforgásos.

1.1 While ciklus

Az első, és legegyszerűbb ciklusunk a while (amíg) ciklus. Ez a ciklus addig ismétli a ciklusmagban található utasítást/utasításokat, amíg a feltétele teljesül.

A while ciklus szerkezete a következő:

```
while(feltetel) {
        utasitas1;
        utasitas2;
        ...
        utasitasN;
}
```

A ciklusokkal kapcsolatos általános felépítés (amelyik természetesen a while ciklusra is igaz), a következő.

A ciklus két részből áll:

- ciklusfej:
    - o kulcsszavat tartalmazó sor
        - –jelen esetben az első sor, a nyitó kapcsos zárójel nélkül: while(feltétel)

– ciklusmag:
  - a ciklikusan ismételt utasítások
  - általában utasításblokk
    - azonban egy utasítás esetén a blokk elhagyható
    - az itt definiált változó érvényességi körének működése megegyezik az elágazásokéval

A while ciklus egy elöltesztelő ciklus, ami azt jelenti, hogy először kiértékeli a ciklus lefutásának feltételét, amennyiben az eredmény igaz: lefuttatja a ciklusmagot. Ezt követően ismét kiértékel, amennyiben az érték igaz: lefuttatja a ciklusmagot. Ezt a tevékenységet egészen addig ismétli, ameddig a feltétel kiértékelésének eredménye igaz.

Tekintsünk meg egy példát a while ciklus működésére. A következő ciklus addig emeli négyzetre x értékét, majd írja felül az eredménnyel az x változó értékét, ameddig az x változó értéke kisebb, mint 16.

```
int x = 2;
while(x < 16) {
      x = x * x;
      System.out.println(x);
}
```

A ciklus kezdésként megvizsgálja, hogy x értéke kisebb-e mint 16? Természetesen igen, ezért lefuttatja a ciklusmagot. Az első utasítás eredményeként az x változónak értéku l adunk 4-et, majd a második utasítás eredményeként kiírjuk x értékét a képernyőre. A következő cikluslefutás során ismét megvizsgáljuk az x változó értékét, amely kisebb, mint 16: így a feltétel teljesu l, azaz ismét le fogja futtatni a ciklusmagot. Amely során x értéke 16-ra vált, majd kiírja a képernyőre a 16-ot. A következő cikluslefutás során megvizsgálja, hogy x értéke (16), kisebb-e mint 16? Természetesen a feltétel nem teljesül, ezért kilépünk a ciklusból, így a ciklus nem fog többször lefutni.

Mivel a while ciklus elöltesztelő, ezért amennyiben x értékét mondjuk 17-el inicializáltuk volna, akkor a ciklusmag egy alkalommal sem futott volna le, mivel a ciklusmag utasításainak első lefutása előtt is ellenőrzi a feltétel teljesülését.

Azonban ha do-while ciklust használnánk, az előbb említett módon, akkor a ciklusmag egyszer lefutna.

1.2 Do-While ciklus

A do-while ciklus hátultesztelő ciklus. Ez azt jelenti, hogy először lefuttatja a ciklusmagot, majd csak ezt követően vizsgálja meg, hogy a feltétel teljesül-e. Amennyiben igen, újra futtatja a ciklusmagot, egészen addig, amíg a ciklus feltétele teljesül. Lényegében a do-while ciklus az a régimódi, vadnyugati rosszfiú, aki először lő (lefuttatja a ciklusmagot), majd csak ezt követően kérdez (megvizsgálja a feltételt). Tehát az előző bekezdés végén említett do-while ciklus egyszer mindenképpen lefut:

```
int x = 16;
do {
      x = x * x;
      System.out.println(x);
} while(x < 16);
```

Az előző kódrészlet kimeneteként megjelenik a 256 a képernyőn. A do-while ciklus ciklusfeje az utolsó sora, a záró kapcsos zárójel nélkül. Fontos megjegyezni, hogy itt a while kulcsszavat követő záró/csukó zárójelet követően kötelezően ;-t kell helyeznünk.

A do-while ciklust tipikusan ellenőrzött adatbeolvasás során használjuk, azonban természetesen előfordulhat olyan egyéb eset, amikor az ő használata célszerű.

1.3 For ciklus

A for ciklus egy elöltesztelő, irányított ciklus. Tipikusan olyan feladatokra használjuk, ahol pontosan ismerjük a ciklusmag lefutásának számát. A futása során egy ciklusváltozó értékét növeli, vagy csökkenti (általában) eggyel. A ciklusfeje összetettebb, mint a while ciklusé, viszont cserébe pontosabban irányítható.

A for ciklus szerkezete a következő:

```
for(utasitasInic; feltetel; utasitasLeptetes) {
      utasitas1;
      utasitas2;
      ...
      utasitasN;
}
```

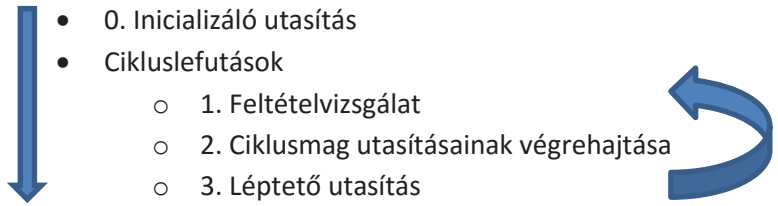
Tehát a for kulcsszavat követően a zárójelek között három utasítást/feltételt adhatunk meg:

1. Inicializációs utasítás. Itt inicializálhatjuk a ciklusváltozó értékét. Kizárólag a ciklus első lefutása előtt kerül végrehajtásra.

2. Feltétel. Ameddig ez a feltétel teljesül, addig fogja ismételten futtatni a ciklusmag utasításait. Minden cikluslefutás legelején, a ciklusmag lefutása előtt végrehajtódik. Ha a feltétel nem teljesül, nem hajtja végre a ciklusmagot és kilép.

3. Léptető utasítás. Ennek az utasításnak a segítségével léptetjük a ciklusváltozó értékét. Minden ciklusmag-lefutást követően végrehajtásra kerül.

Tehát a for ciklus lefutása a következő:



Tekintsük meg a következő for ciklust:

```
for(int i = 1; i < 4; i++) {
      System.out.print(i + ", ");
}
```

A ciklus lefuttatását követően a képernyőn a következő karakterek jelennek meg: „1, 2, 3, ”. Tekintsük meg, pontosan mi is történik lépésről-lépésre:

1. táblázat – Példa for ciklus lefutásának lépései

	Művelet	Ciklusleftás sorszáma	i értéke[1]
1.	Inicializálás. A ciklus számára definiál egy int típusú i változót, melyet inicializál 1 értékkel.	0.	1
2.	Feltételvizsgálat. A feltétel teljesül: a ciklusmag futtatható	1.	1
3.	Ciklusmag lefutása.	1.	1
4.	Léptető utasítás. Az i változó értékének implementálása.	1.	2
5.	Feltételvizsgálat. A feltétel teljesül: a ciklusmag futtatható	2.	2
6.	Ciklusmag lefutása.	2.	2
7.	Léptető utasítás. Az i változó értékének implementálása.	2.	3
8.	Feltételvizsgálat. A feltétel teljesül: a ciklusmag futtatható	3.	3
9.	Ciklusmag lefutása.	3.	3
10.	Léptető utasítás. Az i változó értékének implementálása.	3.	**4**
11.	Feltételvizsgálat. A feltétel nem teljesül: kilépés.	4.	**4**

1 A műveletet követően.

A for ciklus fejében csak a feltételvizsgálatot kötelező megadni, az inicializáló- és a léptető utasítás elhagyható:

```
// Inicializáció elhagyása a ciklusfejből
int i = 1;
for(; i < 5; i++) {
	// ...
}

// Léptetés elhagyása a ciklusfejből
for(int i = 1; i < 5;) {
	// ...
	i++;
}

// Inicializáció- és léptetés elhagyása a ciklusfejből
int i = 1;
for(; i < 5;) {
	// ...
	i++;
}
```

Természetesen az előbbi módokon csak alapos indokkal érdemes használni a for ciklust, hiszen az utolsó példa lényegében egy while ciklus, nyugodtan írhatnánk helyette:

```
int i = 1;
while(i < 5) {
	// ...
	i++;
}
```

Emellett fontos még megemlíteni pár érdekességet for ciklussal kapcsolatban. Az inicializáló blokkban egyszerre több változót is inicializálhatunk, természetesen ugyan azzal a típussal (pl.: int i = 1, j = 5;). Azonban általában egy ciklusváltozó elég számunkra, szóval ezt a lehetőséget is ritkán használjuk. Emellett a for ciklus ciklusváltozójának érvényességi köre kizárólag maga a ciklus, kívülről nem érhető el.

2. Vezérlésátadó utasítások

Előfordulhat, hogy egy bizonyos feltétel teljesülése esetén a ciklus működését módosítani szeretnénk. Ilyenkor a break és a continue utasítást használhatjuk, amelyek segítségével a vezérlést átadhatjuk a programunk egy speciális pontja számára.

2.1 Break

A break utasítás hatására kiugrunk a ciklusból, és az utasítások végrehajtását a ciklust követő utasítással folytatjuk.

Tekintsük meg a következő példát:

```
for(int i = 0; i < 5; i++) {
      System.out.print(i + ", ");
      if(i == 2) {
            break;
      }
}
```

A programrészlet kimenete a következő lesz: 0, 1, 2, . Ennek az az oka, hogy a for ciklus magja hiába futna le még két alkalommal, a break utasítás hatására kilépünk a ciklusból. A break utasítást akkor alkalmazzuk, ha valamiért azonnal ki kell lépnünk a ciklusból. A legtöbbször azonban – helytelenül – akkor alkalmazzuk, ha rossz ciklust használunk, pl: for-t while helyett. A break utasítás használható a switch-case elágazás esetén is.

2.2 Continue

A continue utasítás nem lép ki a ciklusból, csak a ciklusmagban utána lévő utasításokat ugorja át, és mint a nevéből is adódik, folytatja a ciklus végrehajtását a következő iterációval.

Tekintsük meg a következő példát:

```
for(int i = 0; i < 5; i++) {
      if(i % 2 == 0) {
            continue;
      }
      System.out.print(i + ", ");
}
```

A példa kódrészlet kimenete a következő: 1, 3, . Az oka pedig az, hogy amikor i értéke páros, akkor a kiíró utasítást átugorjuk és a következő iterációval folytatjuk. Olyan esetekben használjuk, amikor a ciklusmag egy részét bizonyos feltétel teljesülésekor nem szeretnénk végrehajtani.

2.3 Return

A return utasítás nem csak a ciklusok viselkedését képes módostani. Hatására kilépünk egy függvényből. Amennyiben a return utasítást a main függvényben adjuk ki, akkor kilépünk a programból. A következő példában nem kerül kiírásra a „*Nem fogok lefutni*” karaktersor:

```
public static void main(String[] args) {
      int i = 2;
      if(i == 2) {
            return;
      }
      System.out.println("Nem fogok lefutni.");
}
```

2.4 Címkézett break és continue

A goto ugróutasítás helyettesítésére a címkézett break és continue utasítás használható a ciklusok esetén. Használatuk általában egymásba ágyazott ciklusok esetében szükséges, és csak akkor, ha bonyolult kilépési feltételeket kell használnunk. Segítségükkel egy megcímkézett ciklusból is ki tudunk lépni, esetleg egy megcímkézett ciklus következő utasítására tudunk lépni. Mivel még nem használtunk többdimenziós tömböket, amikkel jól lehetne szemléltetni a működésüket, ezért egy egyszerűbb módon kell bemutatnom őket.

Vegyük a következő egymásba ágyazott ciklust:

```
kulso: for(int k = 0; k < 3; k++) {
      for(int b = 0; b < 3; b++) {
            System.out.printf("[%d, %d]\t", k, b);
      }
      System.out.println();
}
```

A *kulso:* azonosítóval ne foglalkozzunk. Ő egy címke, ameddig nem hivatkozunk rá, semmilyen hatása sincs. Az egymásba ágyazott ciklusok úgy működnek, hogy a külső ciklus minden egyes iterációjában (egyetlen ciklusmag lefuttatásában), lefut a teljes belső ciklus. A programrészlet kimenetében a []-ek közötti első szám a külső ciklus futóindexe, a második pedig a belső ciklusé. Az átláthatóság végett a belső ciklus lefutását követően elhelyezek egy sortörést. (Ha nem teljesen világos, irány Debugolni!)

A programrészlet futtatásának kimenete a következő:

```
[0, 0]      [0, 1]      [0, 2]
[1, 0]      [1, 1]      [1, 2]
[2, 0]      [2, 1]      [2, 2]
```

Látható, hogy a külső ciklus lefutott 3-szor, a belső pedig minden egyes külső lefutáskor ugyancsak 3-szor, azaz 9-szer.

Mi történik, ha lefuttatjuk a következő programrészletet?

```
kulso: for(int k = 0; k < 3; k++) {
      for(int b = 0; b < 3; b++) {
            System.out.printf("[%d, %d]\t", k, b);
            if(k == 1 && b == 1)
                  break kulso;
      }
      System.out.println();
}
```

A belső for ciklusban, ha a külső ciklus, és az első ciklus is a második iterációban jár, kilépünk mindkét ciklusból.

A kódrészlet kimenete a következő:

```
[0, 0]      [0, 1]      [0, 2]
[1, 0]      [1, 1]
```

Látható, hogy mindkét ciklusból kilépett a hatására. Ezt címke nélkül, csak a return utasítással tudjuk megtenni, azonban ott a teljes függvényből is kilépnénk.

Mi történik, ha lefuttatjuk a következő programrészletet?

```
kulso: for(int k = 0; k < 3; k++) {
      for(int b = 0; b < 3; b++) {
            System.out.printf("[%d, %d]\t", k, b);
            if(k == 1 && b == 1)
                  continue kulso;
      }
      System.out.println();
}
```

A címkézett continue utasítás hatására (amely címkéje a külső ciklusra mutat), amikor mindkét ciklus a második iterációjánál jár: kilép a belső ciklusból, átugorja a sortörés kiírását és a következő iterációjára ugrik a külső ciklusnak. A kódrészlet kimenete a következő:

```
[0, 0]      [0, 1]      [0, 2]
[1, 0]      [1, 1]      [2, 0]      [2, 1]      [2, 2]
```

Tehát a címkézett continue és break utasítás az egymásba ágyazott ciklusok során, bonyolult kilépési feltételeket igénylő feladatok esetén használatosak. Ne ijedjetek meg, ha most még nem teljesen világos a címkézett break és continue használata. Amikor eljutunk a többdimenziós tömbökig, akkor térjetek vissza rájuk és próbáljatok megoldani összetett problémákat a segítségükkel.

```
 1. package loops;
 2.
 3. /**
 4.  *
 5.  * @author somlyaip
 6.  */
 7. public class ForSum {
 8.
 9.     /**
10.      * @param args the command line arguments
11.      */
12.     public static void main(String[] args) {
13.
14.         int sum = 0;
15.         for(int i = 1; i < 101; i++) {
16.             sum += i; // sum = sum + i;
17.             System.out.println("i: " + i);
18.         }
19.         System.out.println("Sum: " + sum);
20.     }
21.
22. }
```

```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package loops;

/**
 * {Description}
 *
 * @author somlyaip
 */
public class ForSumEvens {

    public static void main(String[] args) {

        int sum = 0;

//        for(int i = 0; i < 101; i++) {
//            if(i % 2 == 0)
//                sum += i;
//        }
        for(int i = 0; i < 101; i += 2) {
            sum += i;
        }
        System.out.println("Sum: " + sum);
    }
}
```

```
1.  /*
2.   * To change this license header, choose License Headers in Project Properties.
3.   * To change this template file, choose Tools | Templates
4.   * and open the template in the editor.
5.   */
6.  package loops;
7.
8.  /**
9.   * {Description}
10.  *
11.  * @author somlyaip
12.  */
13. public class WhileSum {
14.
15.     public static void main(String[] args) {
16.
17.         int sum = 0;
18.
19.         int i = 1;
20.         while(sum + i < 50) {
21.             sum += i;
22.             System.out.println("i: " + i);
23.             i++;
24.         }
25.         System.out.println("Sum: " + sum);
26.     }
27. }
```

```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package loops;

/**
 * {Description}
 *
 * @author somlyaip
 */
public class DoWhileSum {

    public static void main(String[] args) {
        int sum = 0;

        int number = 1;
        do {
            sum += number;
            System.out.println("Number: " + number);
            number++;
        } while(sum < 50);

        System.out.println("Sum: " + sum);
    }
}
```

```java
package loops;

/**
 * {Description}
 *
 * @author somlyaip
 */
public class BreakAndContinue {

    public static void main(String[] args) {

//        int sum = 0;
//
//        for(int i = 0; i < 101; i++) {
//            if(i % 2 != 0) {
//                continue;
//            }
//            sum += i;
//        }
//        System.out.println("Sum: " + sum);

        int sum = 0;

        int i = 1;
        while(true) {
            sum += i;
            i++;
            if(sum + i > 50)
                break;
        }
        System.out.println("Sum: " + sum);
    }
}
```

Bevezetés a programozásba
Gyakorló feladatok

3. FEJEZET

1. Adott a külső hőmérséklet értéke egy tört szám eltárolására képes változóban. Írassa ki, hogy fagy-e odakint!

2. Adott három szakasz hossza változókban. Írassa ki, hogy lehet-e a szakaszokból háromszöget készíteni! (Egy háromszög bármely két oldalának hossza nagyobb, mint a harmadik oldal.)

3. Adott két változó: *x* és *y*. Jelenítsük meg a képernyőn, hogy a nagyobbik értékű osztható-e a kisebbikkel! Ha nem osztható, az osztási maradékot is írassuk ki! A program működjön helyesen akkor is, ha x a kisebb értékű, valamint, ha *y* az!

4. Készítsen programot, amely megjeleníti a képernyőn, egymás mellett, veszővel és szóközzel elválasztva, a 20..40 tartományban elhelyezkedő, 4-el maradék nélkül osztható számokat, növekvő sorrendben!

A kimenet részlete:

20, 24, 28, …

5. Készítsen programot, amely megjeleníti a képernyőn, egymás alatt, a -10..+10 tartományban elhelyezkedő, páratlan, egész számokat, csökkenő sorrendben!

A kimenet részlete:

9
7
5
…

6. Az indiánok 1626-ban eladták Manhattan-szigetét 24$ értékű üveggyöngyért. Mennyit érne ez az összeg ma, 5%-os kamat mellett? Az eredményt számítsa ki dollárban, majd váltsa át magyar forintra (1 USD = 279 HUF árfolyamon)! Mindkét eredményt írassa ki!

7. Adott egy egész szám, egy változóban eltárolva. Írassa ki a hatványait egymás alá, az első hatványtól kezdődően, amíg az aktuális hatványra emelt érték nem nagyobb, mint 200!

A kimenet (az egész szám: 5):

5^1 = 5
5^2 = 25
5^3 = 125

A hatványra emelést a Math.pow() függvény segítségével tudja elvégezni. Példa a használatára:

```
int hatvany = (int) Math.pow(5, 2) // öt második hatványa: 25
```

4. fejezet

Bevezetés a programozásba

Algoritmizálás

Somlyai Pál

Dunaújvárosi Főiskola

2015

Tartalom

Bevezetés a programozásba – Algoritmizálás 1

1. Algoritmus

Elsőként az algoritmus fogalmával szükséges megismerkednünk. Ehhez olvassuk el, és értelmezzük a tananyag 2–9. oldalát.

Ne ijedjünk meg a vektorok, mátrixok, halmazok illetve függvények láttán. Jelenleg nem használtuk őket, ezért a tananyag eme részét most nem fogjuk érinteni. Összegezve a leírtakat: lényegében az algoritmus a programunk terve, amelyet a forráskódból fordított programunk fog megvalósítani. Az algoritmus nem egy szerkezeti terv (mint a gépész- és építész/építőmérnökök esetében). A probléma megoldására irányuló lépéseink sorozata, egymásutániságukban, mely lépéseket elágazások és ciklusok segítségével, azaz vezérlési szerkezetek használatával szabályozhatunk (elágaztathatjuk, ismételtethetjük). Az algoritmus nagyon közel áll a forráskódhoz. Ugyan azokat az alapelemeket használhatjuk az elkészítéséhez, mint az eddig megismert Java elemek. Azonban van közöttük egy fontos különbség: a Java programkód csak a Java nyelven értelmezhető, míg az algoritmus programnyelvektől független. Ideális esetben, egy probléma megoldására először algoritmus készítünk, majd csak ezt követően készítjük el a forráskódot, természetesen az algoritmusunk alapján.

2 Algoritmizálási eszközök

Mint ahogyan már említettem, az algoritmus a programunk utasításainak tervrajza. Azonban a terveket nem csak az elménkben szoktuk tárolni, valamilyen eszköz segítségével le szoktuk őket jegyezni. Tekintsük át a tananyag 10-17. oldalait. Mi a pszeudo kódot és folyamatábrát fogjuk részletesebben tárgyalni.

2.1 Pszeudokód

A Pszeudokód nagyon közel áll a tényleges forráskódhoz. Azonban nem szabványos, szóval nincs megállapodás a szakmán belül, hogy milyen szabályok szerint kell készíteni. Mi a jegyzetben ismertetett módszert fogjuk alkalmazni.

2.2 Folyamatábra

A folyamatábra eléggé távol helyezkedik el a forráskódtól, hiszen egy vizuális algoritmizálási eszköz, viszont véleményünk szerint ennek az értelmezése a legegyszerűbb, valamint ez nyújtja a legnagyobb támogatást a forráskód elkészítésekor. Ezért mi a programok elkészítésekor a folyamatábrákból fogunk kiindulni.

3 Feladatok

Az itt felsorolt feladatokhoz szükséges elkészíteni az algoritmusokat Pszeudokód és folyamatábra segítségével. A feladatok pusztán gyakorló jellegűek, mindegyik megoldása elérhető, viszont arra kérlek titeket, hogy próbáljatok meg minél többet megoldani saját magatok. Ezek közül a feladatok közül néhányat a következő alfejezetben le fogunk programozni, azaz az algoritmust (tervet) meg fogjuk valósítani programkódban.

3.1 Négyműveletes számológép adatbeolvasással és ellenőrzéssel

Írjon egy programot, amely egy hagyományos, 4 műveletes számológépet valósít meg adatbeolvasással és ellenőrzéssel! Az adatellenőrzés azt jelenti, hogy ameddig nem megfelelő operátort ad meg a felhasználó, kérje újra az operátort!

3.2 Hárommal osztható pozitív számok

Írassuk ki egymás alá ötösével csoportosítva a 100-nál kisebb, hárommal osztható pozitív számokat!

3.3 Oszthatósági táblázat

Olvasson be két értéket, melyek között elhelyezkedő egész számokról állapítsa meg, hogy 2-vel, 3-al és 5-el maradék nélkül oszthatóak-e! Az eredményt jelenítse meg táblázatos formában! Írjon egy programot, amely adatbeolvasással és ellenőrzéssel (determináns értéke) valósítja meg a másodfokú egyenlet megoldóképletét!

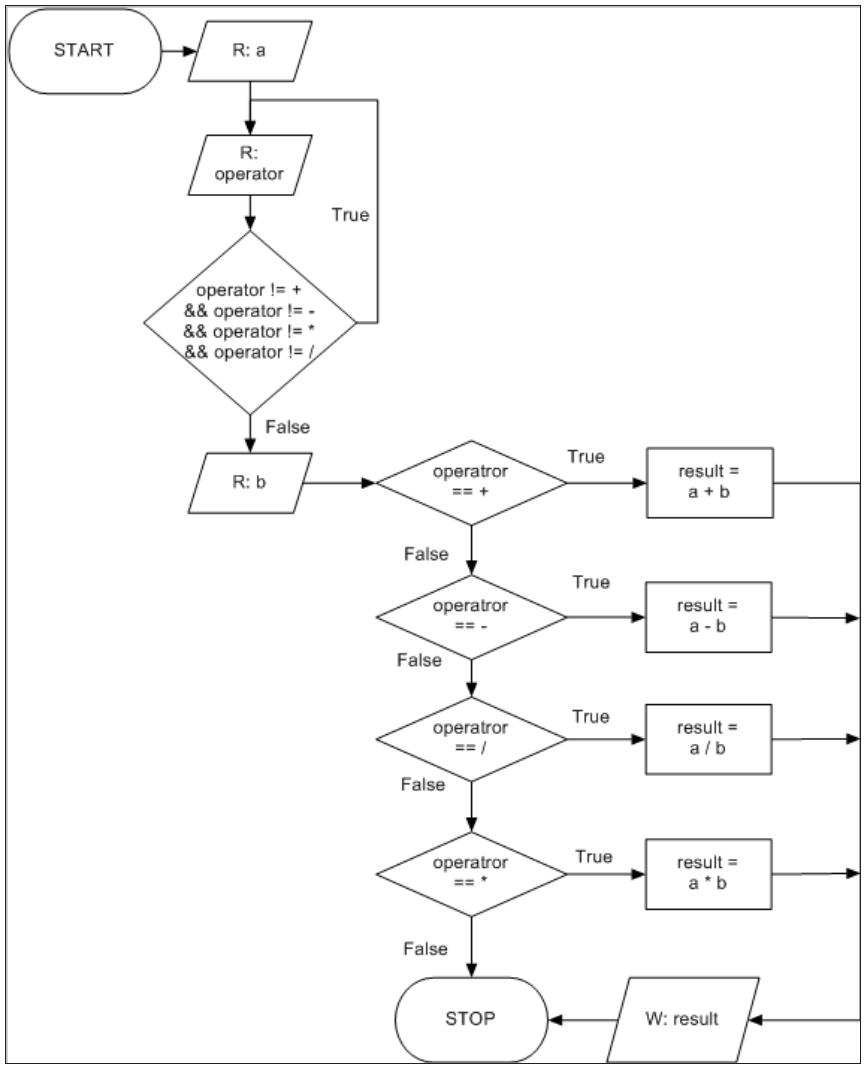
3.4 Másodfokú megoldóképlet

Írjon egy programot, amely adatbeolvasással és ellenőrzéssel (determináns értéke) valósítja meg a másodfokú egyenlet megoldóképletét!

4. Feladatok megoldásai

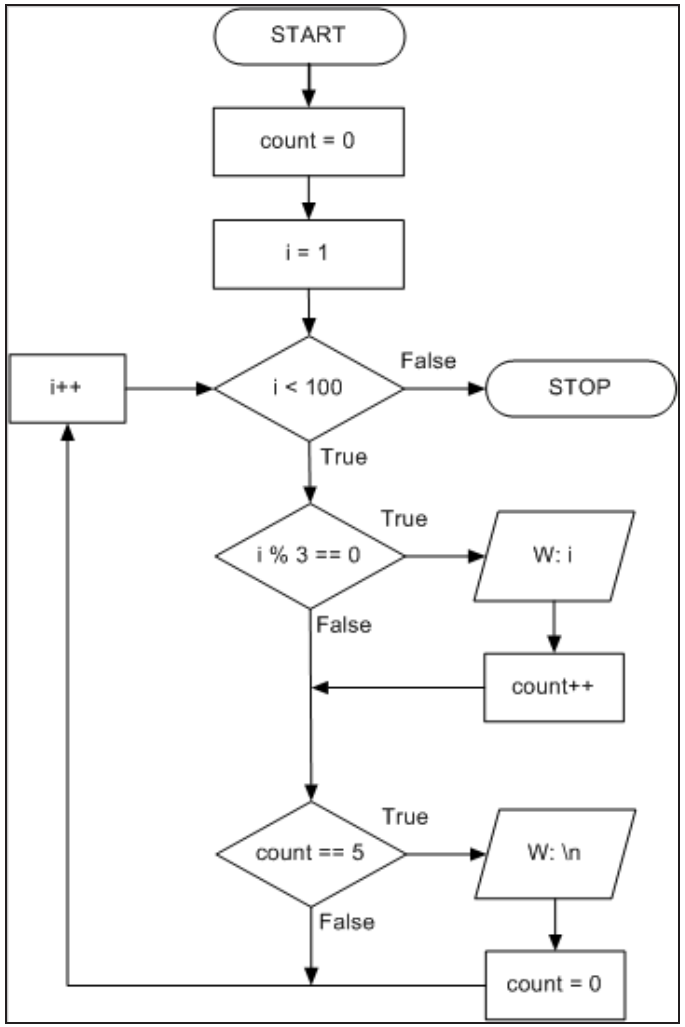
4.1 Négyműveletes számológép adatbeolvasással és ellenőrzéssel

```
Eljárás
        Be: a
        Ciklus
                Be: operator
        amíg operator != + és operator != - és operator != * és operator != / ciklus vége
        Be: b
        Ha operator == + akkor
                result = a + b
        különben Ha operator == - akkor
                result = a – b
        különben Ha operator == / akkor
                result = a / b
        különben Ha operator == * akkor
                result = a * b
        ha vége
        Ki: result
eljárás vége
```


START
R: a
R:
operator
True
operator != +
&& operator != -
&& operator != *
&& operator != /
False
R: b
operatror
== +
True
result =
a + b
False
operatror
== -
True
result =
a - b
False
operatror
== /
True
result =
a / b
False
operatror
== *
True
result =
a * b
False
STOP
W: result

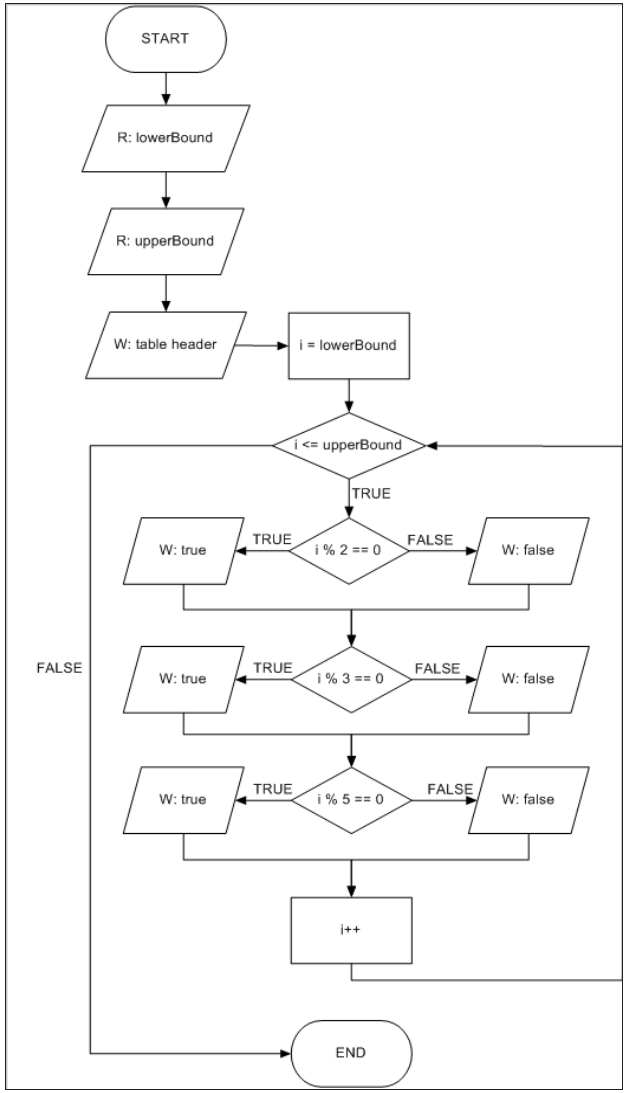
4.2 Hárommal osztható pozitív számok

```
Eljárás
        count = 0
        Ciklus i = 1-től 100-ig
                Ha i % 3 == 0 akkor
                        Ki: i
                        count = count + 1
                ha vége
                Ha count == 5 akkor
                        Ki: \n
                        count = 0
                ha vége
        ciklus vége
eljárás vége
```


START
count = 0
i = 1
i < 100
False
STOP
True
i++
i % 3 == 0
True
W: i
False
count++
count == 5
True
W: \n
False
count = 0

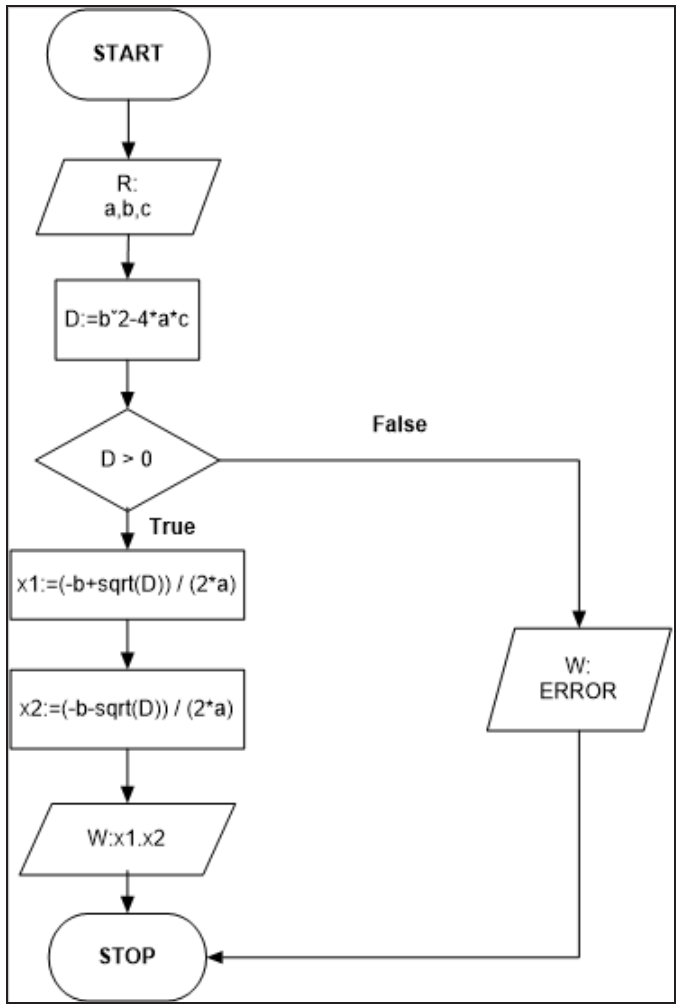
4.3 Oszthatósági táblázat

```
Eljárás
        Be: lowerBound
        Be: upperBound
        Ki: table header
        Ciklus i = lowerBound-tól upperBound-ig
                Ha i % 2 == 0 akkor
                        Ki: true
                különben
                        Ki: false
                ha vége
                Ha i % 3 == 0 akkor
                        Ki: true
                különben
                        Ki: false
                ha vége
                Ha i % 5 == 0 akkor
                        Ki: true
                különben
                        Ki: false
                ha vége
        ciklus vége
eljárás vége
```


START
R: lowerBound
R: upperBound
W: table header
i = lowerBound
i <= upperBound
TRUE
FALSE
W: true
TRUE
i % 2 == 0
FALSE
W: false
W: true
TRUE
i % 3 == 0
FALSE
W: false
W: true
TRUE
i % 5 == 0
FALSE
W: false
i++
END

4.4 Másodfokú megoldóképlet

```
Eljárás
        Be: a
        Be: b
        Be: c
        D = b ^ 2 – 4 * a *c
        Ha D > 0 akkor
                x1 = (-b + sqrt(D)) / (2 * a)
                x2 = (-b – sqrt(D)) / (2 * a)
                Ki: x1
                Ki: x2
        különben
                Ki: ERROR
        ha vége
eljárás vége
```


START
R:
a,b,c
D:=b^2-4*a*c
D > 0
False
True
x1:=(-b+sqrt(D)) / (2*a)
x2:=(-b-sqrt(D)) / (2*a)
W:x1.x2
W:
ERROR
STOP

```
 1. package drawtriangle;
 2. 
 3. /**
 4.  *
 5.  * @author somlyaip
 6.  */
 7. public class DrawTriangle {
 8. 
 9.     /**
10.      * @param args the command line arguments
11.      */
12.     public static void main(String[] args) {
13.         // unicode shapes
14.         // https://upload.wikimedia.org/wikipedia/commons/4/4c/UCB_Geometric_Shapes.png
15. 
16.         int rowCount = 5;
17.         for(int i = 0; i < rowCount; i++) {
18.             for(int j = 0; j < i; j++) {
19.                 System.out.print("\u25A0");
20.                 System.out.print(" ");
21.             }
22.             System.out.println("\u25E3");
23.         }
24.     }
25. 
26. }
```

```
1.  /*
2.   * To change this license header, choose License Headers in Project Properties.
3.   * To change this template file, choose Tools | Templates
4.   * and open the template in the editor.
5.   */
6.  package divwithtwothreefive;
7.
8.  /**
9.   *
10.  * @author somlyaip
11.  */
12. public class DivWithTwoThreeFive {
13.
14.     /**
15.      * @param args the command line arguments
16.      */
17.     public static void main(String[] args) {
18.
19.         int lowerBound = 5;
20.         int upperbound = 15;
21.
22.         System.out.println("Number\tDiv 2\tDiv 3\tDiv5");
23.
24.         for(int i = lowerBound; i <= upperbound; i++) {
25.             System.out.print(i + "\t");
26.
27.             System.out.print((i % 2 == 0 ? "true" : "false") + "\t");
28.             System.out.print((i % 3 == 0 ? "true" : "false") + "\t");
29.             System.out.print((i % 5 == 0 ? "true" : "false") + "\t");
30.             System.out.println();
31.         }
32.     }
33.
34. }
```

Bevezetés a programozásba

Adatbeolvasás

Somlyai Pál

Dunaújvárosi Főiskola

2015

Tartalom

Bevezetés a programozásba – Adatbeolvasás

1. Adatbeolvasás

Az eddig készített programjaink csupán a forráskódba „bedrótozott” (azaz literálként/változóként megadott) értékekkel dolgoztak. Azonban a valóságban nem adhatunk egy felhasználó kezébe egy olyan számológép programot, amelynek használatához a forráskód átírása, majd lefordítása szükséges. Szerencsére a programozási nyelvek – és közöttük a Java is – biztosít számunkra adatbeolvasási lehetőséget, így a felhasználóktól a program futása során – többek között – a konzolablakról kérhetünk be adatokat. Az adatbeolvasást egy parancs kiadásával tudjuk végrehajtani. Ez azt jelenti, hogy a felhasználó nem írhat be bármikor, bármilyen adatot a konzolablakba, csupán akkor, ha ezt várjuk tőle.

1.1 Adatbeolvasás a Javában

A Javában a legegyszerűbb adatbeolvasási parancs a *System.console().readLine()*. Ennek az utasításnak (valójában függvénynek) a hatására a programunk beolvas egy sort a konzolablakról, amelynek visszaadja az értékét. Mivel a felhasználó tetszőleges szöveget is be tud gépelni (hiszen a billentyűzeten számok, betűk és speciális karakterek is szerepelnek), ezért String formában adja vissza a felhasználó által megadott sort. Tekintsünk meg egy példát adatbeolvasására a Javában:

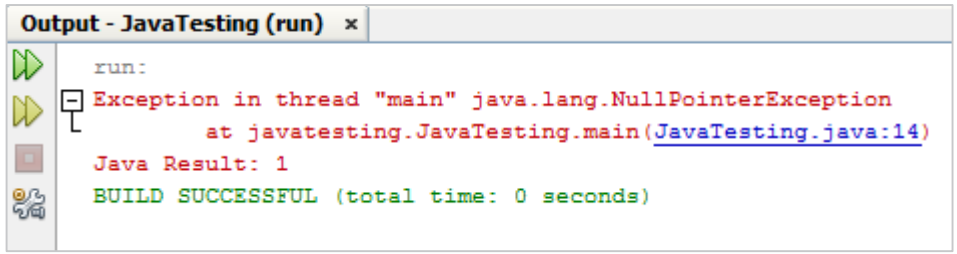
```
public static void main(String[] args) {
      System.out.println("Kérem írjon be egy tetszőleges szöveget, "
                  + "majd nyomja le az ENTER billentyűt:");

      String sor = System.console().readLine(); // az enter leütéséig
      // ezen a soron áll a programunk végrehajtása

      System.out.println(sor);
}
```

A fenti példa megjeleníti a felhasználó számára, hogy mi a teendő, majd beolvas egy sort, végül pedig kiírja a konzolablakra az eredményt. Vegyük figyelembe, hogy egészen addig a beolvasási soron fog állni a program végrehajtása, ameddig le nem nyomjuk az ENTER billentyűt. Lényegében ez azt jelenti, hogy a beolvasást követő utasítás csak az ENTER billentyű lenyomását követően fut le, azaz csak akkor folytatódik a programunk végrehajtása. Mielőtt vadul begépelnénk a fenti példakódhoz hasonló programot a Netbeansbe, térjünk ki a Netbeans adatbeolvasási sajátosságára.

1.1.1 Adatbeolvasás a Netbeansben

Ha a fenti kódot a megszokott módon lefuttatjuk a Netbeansben, a következő hibaüzenetet kapjuk:

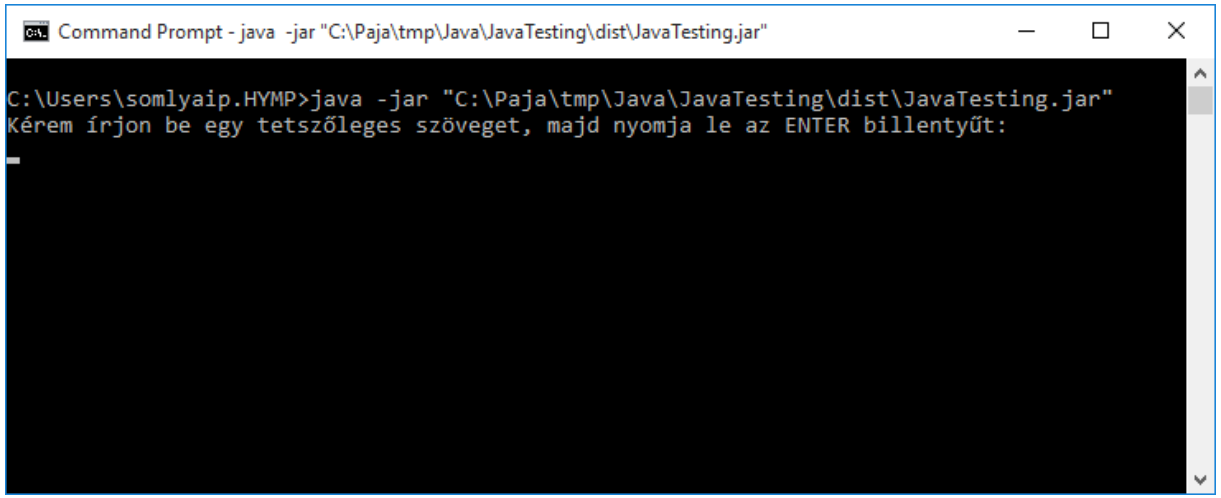
1. ábra – Beolvasási hibaüzenet a Netbeansben



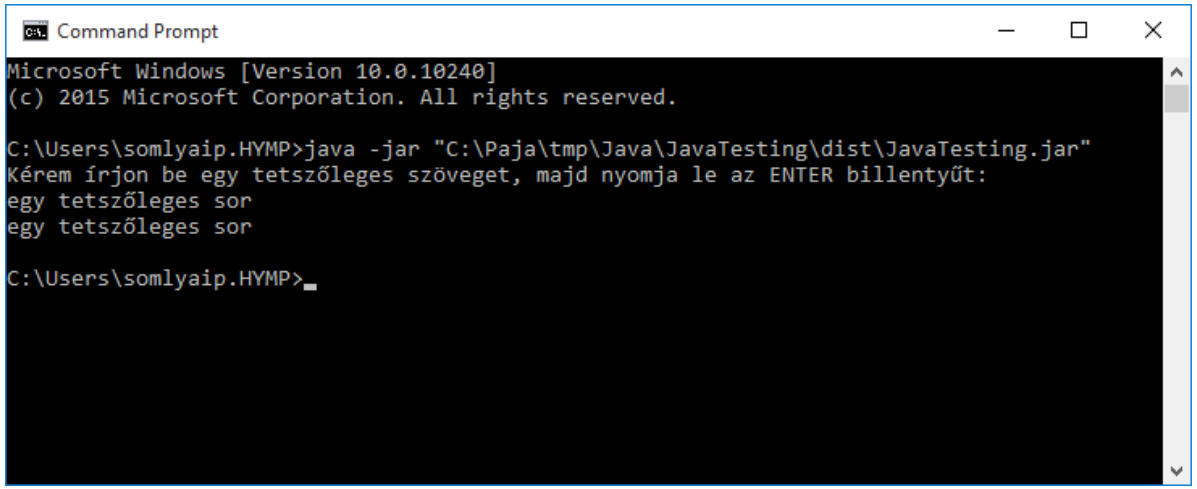
Ennek nagyon egyszerű oka van. A System.console() utasítás a számítógépünk konzolablakára (Command Line/Terminálablak) hivatkozik. Azonban – ahogy azt már a Netbeans alapfunkciók című segédletben említettem –, a Netbeans a konzolablak helyett egy az Apache Ant által emulált (utánozott) ablakot használ a kiírási eredményeink megjelenítésére. Sajnos nem fejlesztették le, hogy konzolablakként is tudjon viselkedni. Egy több, jelenleg még túlságosan bonyolult parancsból álló utasítássorozattal azonban a Netbeans output ablakáról is tudunk majd beolvasni, azonban erre még egy fél évig várni kell. Roppant sokat tanakodtunk az Informatikai Intézetben kollégáimmal, hogy milyen módon olvassunk be a Netbeansben adatokat és végül a véleményünk szerint jelenleg legegyszerűbb megoldást választottuk.

Mi is volt a probléma a beolvasással? Az, hogy az output ablak nem tud konzolként viselkedni. Akkor használjuk a Windows beépített konzolját, a Parancssort (Linuxon és Mac-en pedig a Terminált). Nyissuk meg a Parancssort a számítógépünkön (Windows gomb, vagy Windows gomb + R, majd gépeljük be a cmd szöveget, majd nyomjuk le az ENTER billentyűt. Ennek hatására megnyílik a parancssorunk. Illesszük be a Netbeansbe a fenti példakódot, majd kattintsunk a Clean and Build Project lehetőségre (a futtatás ikontól balra található kalapács és seprű ikonra). Így lefordítjuk a programunkat, valamint a közteskódot be is csomagoljuk egy jar kiterjesztésű fájlba. Ezt követően az output ablakban, alulról a 3. sorban (a legutolsó fekete sorban) található parancsot (ami a következőre hasonlít: java –jar …) másoljuk ki. Kattintsunk át a parancssorba. Amennyiben Windows 10 rendszert használunk, a CTRL + V billentyűkombinációval be tudjuk illeszteni a parancsot. Azonban ha korábbi Windows verziót használunk, akkor kattintsunk jobb gombbal a Parancssor ablakára, majd a megjelenő menüből válasszuk a Paste lehetőséget. Ezt követően adjuk ki a parancsot az ENTER billentyű lenyomásával.

A programunk elindul, majd a beolvasásánál „megáll”, aminek az eredményeként a parancssorban „elkezd villogni” a kurzorunk:



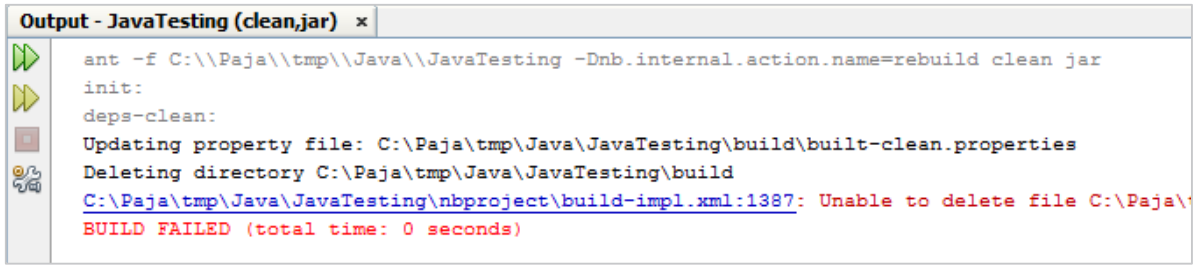
Írjuk be a kívánt szöveget, majd üssünk ENTER-t. Eredményként meg kell jelennie a beolvasott sornak a megadott sor alatt, azaz két teljesen megegyező sort kell látnunk egymás alatt:



Ha újból szeretnénk futtatni a programot, akkor csak a felfelé kurzormozgató billentyűt kell használnunk, aminek hatására megjelenik az utoljára beírt parancs, azaz a programunk futtatása.

Viszont tartsuk észben, hogy amikor módosítunk a forráskódon, akkor újra kell fordítanunk a programot (Clean and Build Project), hogy a módosítások érvényesüljenek, és a módosított forráskódból készült köztes kódot tudjuk futtatni. Ezt követően adjuk csak ki a futtatás parancsot a Parancssorban.

Előfordulhat, hogy az újrafordítási parancs hatására a következő hibaüzenet jelenik meg az output ablakban:



Ennek az az oka, hogy egy beolvasásánál áll még a programunk, azaz még fut. Ilyenkor nem tudjuk törölni a futó fájlt (a Clean and Build Project parancs először törli a régi jar fájlt, majd létrehozza az újat). Futtassuk végig a programunkat, vagy szakítsuk meg a futását a CTRL + C billentyűkombinációval, majd ezt követően Clean and Build Project, végül pedig a futtatási parancs kiadása a Parancssorban.

1.2 Adatbeolvasási tippek

Szeretnék megadni számotokra pár tippet, ami elengedhetetlen egy jól működő alkalmazás fejlesztéséhez:

- Mindig jelenítsük meg a felhasználó számára, hogy milyen adatot kívánunk bekérni tőle
  - Azaz írjuk ki a beolvasás előtt, hogy mi a teendője, így nem zavarjuk össze

- Mindig ellenőrizzük a felhasználó által megadott adatokat
  - Pl.: ha számot várunk, vizsgáljuk meg, hogy ténylegesen számot adott-e meg a felhasználó, hiszen ha nem azt olvasunk be, akkor a konvertáló függvény meghívása hibát fog okozni. Ezt nevezzük merev input oldalnak.
    - A felhasználók többsége… hogy is fogalmazzam meg? Laikus. Azaz nem mindig tudja, hogy az utasítások kiadása milyen következményekkel járhat. Képzeljük el, hogy az Excelben egy irdatlan bonyolult táblázatot szerkesztünk, tele összetett függvényekkel és véletlenül a kezdet óta nem mentettük el a fájlt (persze mi nem tennénk ilyet, de képzeljük el). Majd több cellára meghívjuk az osszeg függvényt, amelyek közül az egyik nem szám, ezért a konvertáló függvény hibát dob, és az Excel a mentetlen módosításainkkal felmondja a szolgálatot. Senki sem szeretne ilyen helyzetbe kerülni, ezért ellenőrizzük az ilyen adatokat a lehető legrészletesebb módon.

```
 1. /*
 2.  * To change this license header, choose License Headers in Project Properties.
 3.  * To change this template file, choose Tools | Templates
 4.  * and open the template in the editor.
 5.  */
 6. package input;
 7.
 8. /**
 9.  *
10.  * @author somlyaip
11.  */
12. public class Input {
13.
14.     /**
15.      * @param args the command line arguments
16.      */
17.     public static void main(String[] args) {
18.
19.         System.out.print("Please give a number: ");
20.         float number = Float.parseFloat(System.console().readLine());
21.         System.out.println("Number: " + number);
22.     }
23.
24. }
```

```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package calculatorwithreadinginput;

/**
 *
 * @author somlyaip
 */
public class CalculatorWithReadingInput {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        float a;
        float b;
        char operator;

        System.out.print("Please give the first number: ");
        a = Float.parseFloat(System.console().readLine());

        do {
            System.out.print("Please give the operator: ");
            operator = System.console().readLine().charAt(0);
        } while(operator != '+' && operator != '-'
                && operator != '*' && operator != '/');

        System.out.print("Please give the second number: ");
        b = Float.parseFloat(System.console().readLine());

        float result;
        switch(operator) {
            case '+': result = a + b;
                      break;
            case '-': result = a - b;
                      break;
            case '*': result = a * b;
                      break;
            case '/': result = a / b;
                      break;
            default: return;
        }

        System.out.printf("%.2f %c %.2f = %.4f", a, operator, b, result);
    }

}
```

Bevezetés a programozásba
Gyakorló feladatok

4. FEJEZET

1. Beolvasni két valós számot, majd kiíratni, hogy az összegük nagyobb-e, mint 10. (Kiíratás: „nagyobb, mint 10” illetve „nem nagyobb, mint 10”.)

2. Beolvasni egy valós számot és kiíratni az abszolút értékét. (Ha negatív, akkor az ellentétét, különben önmagát.)

3. Beolvasni két egész számot, majd a nagyobból kivonni a kisebbet. Kiíratni a műveletet és az eredményt is. (pl. 4-1=3)

4. Beolvasni egy négyzet és egy téglalap oldalhosszait. Kiíratni, hogy melyiknek kisebb a területe.

5. Kérjen be 5 egész számot ciklus segítségével, és vizsgálja meg, hogy van-e köztük páratlan! Írassa ki, hogy „van” vagy „nincs”!

6. Írassa ki az első N pozitív, páratlan szám négyzetét mindhárom ciklussal! Az N értékét olvassa be!

7. Olvasson be egy 0 és 100 közötti pontszámot adatellenőrzéssel! (Addig kérje az értéket újra, míg az adott intervallumba nem esik.)
A tárgyunk ponthatárait felhasználva írassa ki a megszerzett érdemjegyet! Írassa ki a jegyet számmal (if segítségével) és betűvel is (switch felhasználásával – elégtelen, elégséges, közepes, jó, jeles)!

5. fejezet

Bevezetés a programozásba
Vektorok

Somlyai Pál

Dunaújvárosi Főiskola

2015

Tartalom

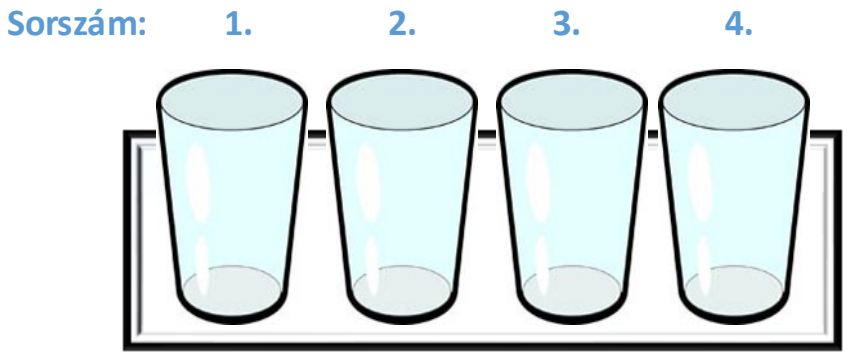
Bevezetés a programozásba – Vektorok

1. Tömbök

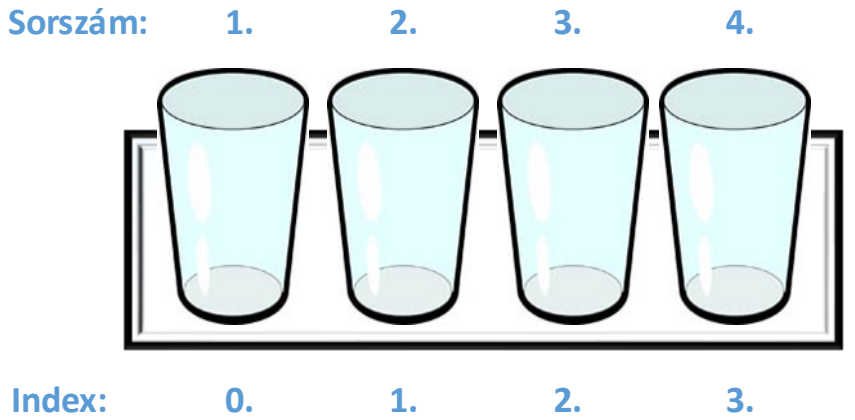
Eddig főként az egyszerű (primitív) változókkal foglalkoztunk, valamint egy objektummal, a Stringgel. Itt az ideje megismerni a második objektumunkat: a tömböt. Az objektumról most legyen elég annyi, hogy egy összetett adatstruktúra, aminek speciális értékei és viselkedései vannak. Gondoljuk végig: a String objektum valójában sok karakter típusú változót tárol, amelyek együtt egy karakterláncként tudunk kezelni a segítségével. Erre utal az összetett adatstruktúra kifejezés. A tömb (angolul array) ugyancsak egy objektum, azaz egy összetett adatstruktúra. Arra való, hogy több, azonos típusú értéket tároljon el a számunkra. Ezek az azonos típusú értékek általában egymáshoz logikailag szorosan kapcsolódó értékek is. Például egy hét napi átlaghőmérséklete, egy úszóverseny fordulójának eredményei, egy kurzus zh eredményei, stb. A tömbök első típusa a vektor, amennyiben csak tömbként hivatkozunk egy konkrét típusra, akkor a vektort értjük alatta.

1.1 Vektorok

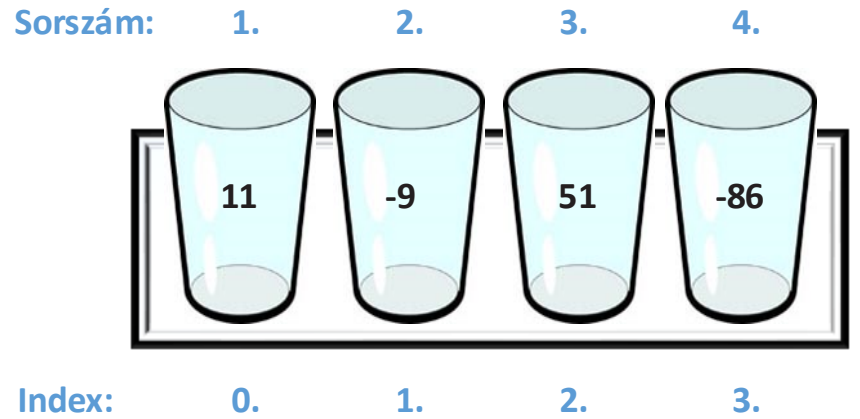
Tehát a vektor egy olyan adatstruktúra, amiben több, azonos típusú adatot tudunk eltárolni. Hogy jobban megérthessük, először visszatérek a csészehasonlathoz. Képzeljük el, hogy a vektor egy vékony tálca, amire több csészét (azaz értéket) tudunk felhelyezni. A csészék darabszáma előre megkötött, valamint a csészék csak egy sorban szerepelhetnek:



A tárolóedények balról-jobbra kerülnek sorszámozásra. A sorszám, pontosabban az index segítségével tudunk az elemekre hivatkozni. Az index szinte megegyezik a sorszámmal, annyi a különbség közöttük, hogy nem egytől, hanem nullától indul:



Az integer tömb igazából egy számsorozat. Nézzük meg, hogyan használhatjuk fel a számsorozatunkat:



Ha ki szeretnénk kérni egy elem értékét a tömbből (azaz vektorból), akkor nem mondhatunk olyanokat, hogy: „Kérem balról az első elemet!”, vagy „Kérem a középső elemet!”. Csak és kizárólag az elemek indexével hivatkozhatunk rájuk. Tehát ha a fenti vektorból kikérjük a 2. indexű elemet, akkor az 51 értéket kapjuk vissza. Nézzük meg, hogyan hozhatjuk létre ezt a vektort a Javában:

```
int[] szamok = new int[4];

int szamok[] = new int[4];
```

Mindkét módon létrehozhatjuk a tömbünket, azonban mi az első mód használatát javasoljuk, mert mi is ezt fogjuk alkalmazni. Látható, hogy nem egy int típusú változót, hanem egy int[] típusú változót definiáltunk számok azonosítóval. Amennyiben az adattípus neve után szögletes zárójeleket adunk meg, azzal jelezzük a Java fordítónak, hogy egy vektort szeretnénk létrehozni. Ezt követően rögtön inicializáltuk is. Ami még furcsább, a new kulcsszó után még egyszer leírtuk az int[] szöveget, benne megadva egy számot. Ez a szám a tömb elemeinek darabszáma. Mivel a képen látható tömbábrázolásban 4 elem szerepel, ezért adtuk meg a 4-et a tömb hosszának. Tehát egy vektor definiálásának és inicializálásának szerkezete a következő:

típus[] azonosító = new típus[tömbHossza];

Álljunk meg egy percre. Már inicializáltuk is a tömböt? Hiszen sehol sem adtuk meg az elemek értékét. A vektorok esetében az inicializálás azt jelenti, hogy megadjuk a pontos elemszámát a vektorunknak, ezáltal helyet foglalunk a memóriában a megadott darabszámú, megadott típusú változónak (hiszen a tömb „csak” egy tálca, ami összefogja a változóinkat). Tehát a vektorunk inicializált, azaz felkészítettük, hogy befogadja a megadott számú integer típusú elemet. Töltsük fel elemekkel. Az első feltöltési mód az, hogy egyesével hivatkozunk a tömb egy elemére, majd felülírjuk az értékét. Egy tömbelemre a következőképen hivatkozhatunk: azonosító[index]. Töltsük fel a megadott elemekkel:

```
szamok[0] = 11;
szamok[1] = -9;
szamok[2] = 51;
szamok[3] = -86;
```

Amennyiben ki szeretnénk íratni a vektor egy elemét, ismételten hivatkoznunk kell az adott tömbelemre:

```
System.out.println(szamok[0]); // 11
System.out.println(szamok[1]); // -9
System.out.println(szamok[2]); // 51
System.out.println(szamok[3]); // -86
```

Ha megfigyeljük az utolsó elem kiírását, láthatjuk, hogy az utolsó elem indexe mindig a tömb hosszánál egyel alacsonyabb érték.

1.1.1 Tippek a vektorokkal kapcsolatban

Tekintsük meg, hogyan tudjuk egyszerre definiálni, inicializálni és feltölteni elemekkel egy vektort:

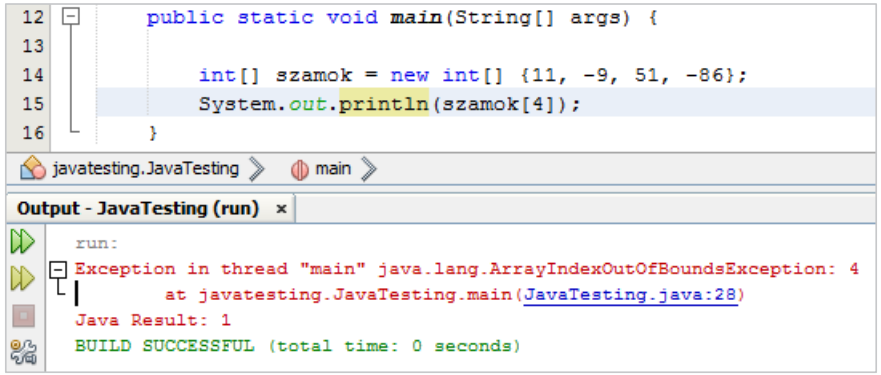
```
int[] szamok = new int[] {11, -9, 51, -86};
```

Ebben az esetben tilos megadni a tömb hosszát, hiszen utána kapcsos zárójelek között felsoroljuk őket, így a fordító meg tudja számolni az elemszámot. Ezt a megadási módot tömbinicializáló utasításnak nevezzük.

Azonban van egy még egyszerűbb megadási mód, egy vektor inicializálására:

```
int[] szamok = {11, -9, 51, -86};
```

Amennyiben nem létező index segítségével hivatkozunk egy tömbelemre, akkor a következő hibaüzenetet kapjuk:



A hibaüzenet a következő: „Kivétel keletkezett a „main” szálban: java.lang.ArrayIndexOutOfBoundsException[1]: 4[2] a javatesting.JavaTesting.main[3](JavaTesting.java:28[4])”

1 ATömbIndexHatárértékekenKívülEsikKivétel (~a tömb indexe túl magas, vagy túl alacsony)

2 A hibás index értéke 4

3 A javatesting csomagban található, JavaTesting osztály, main függvényében

4 JavaTesting.java forrásfájl 28. sora

Amennyiben olyan tömbelemre hivatkozunk, aminek még nem lett értékül adva semmi, akkor a tömb adattípusának alapértelmezett értékét kapjuk meg. Ennek az az oka, hogy amikor inicializálunk egy tömböt (az elemek megadása nélkül), akkor minden tömbelem helyére a típus alapértelmezett értéke kerül.

Természetesen nem csak integer típusú tömböket készíthetünk. Bármilyen típusból létrehozhatunk egy tömböt. Egy String objektum belsejében valójában egy karakter vektor tárolja el a karakterlánc minden egyes karakterét.

A vektorok ismertetésénél a csészehasonlatot folytattam tovább. Azonban egy másik zseniális hasonlatról is hallottam már. Képzeljük el, hogy egy tömb, egy fiókos szekrény. Annyi fiókot tartalmaz egymás alatt, amennyi a tömb elemszáma. A fiókok indexelve vannak, így ha a harmadik elemet szeretnénk kivenni, kinyitjuk a második indexű fiókot és kiolvassuk a benne lévő elemet. Amikor felül szeretnénk írni egy elemet, akkor kicseréljük a megadott indexű fiókban szereplő elemet.

Remélhetőleg ez a példa is segített megérteni és megjegyezni a vektorok működését. Ha már ismerjük a működésüket, kezdjük el bejárni őket.

1.1.2 Vektorok bejárása

Vektor bejárásának nevezzük azt a műveletsort, amikor végigiterálunk (végighaladunk) a tömb minden egyes elemén. Természetesen, mivel a tömb méretét (hosszát) ismerjük, ezért egy for ciklussal a legcélszerűbb bejárni egy vektort. A futóindexet 0-ról kell indítanunk, hiszen mindig ez a tömb első indexe, és a futóindexnek egyesével fel kell vennie minden értéket a tömb hossza – 1 értékik, hiszen ez az utolsó elem indexe:

```
int[] szamok = new int[] {11, -9, 51, -86};
for(int i = 0; i < 4; i++) {
      System.out.printf("%d, ", szamok[i]);
}
```

A fenti kódrészlet megjeleníti a tömb elemeit a képernyőn: *11, -9, 51, -86,*

Azonban a Javában van egy nagyon hasznos funkció: dinamikusan le tudjuk kérdezni egy tömb hosszát a length érték kikérésével:

```
int[] szamok = new int[] {11, -9, 51, -86};
for(int i = 0; i < szamok.length; i++) {
      System.out.printf("%d, ", szamok[i]);
}
```

1.2 Tömbök dimenziói

Egy tömböt a típusán kívül a dimenziója azonosítja. A matematikából kiindulva a dimenzió azt jelenti, hogy egy pont egyértelmű meghatározásához, hány független adatra van szükségünk.

Egydimenziós egy egyenes, amelynek egy pontját egyetlen számmal tudjuk azonosítani (lényegében a sorszámával).

Kétdimenziós a sík, amely egy pontját egyértelműen két koordinátával tudunk azonosítani (x, y).

Valamint háromdimenziós a tér, amelyik egy pontját három koordinátával tudunk beazonosítani (x, y, z).

A vektor egy egydimenziós tömb, hiszen egy elemének egyértelmű beazonosításához egy értékre van szükségünk: az indexére. Kétdimenziós tömb pedig a mátrix, amelyik a síkra hasonlít, neki két indexelője van. A mátrixról később fogunk tanulni. A többi tömböt egyszerűen a dimenziójával azonosítjuk (pl.: háromdimenziós tömb).

```
 1. package iteratevector;
 2. 
 3. /**
 4.  *
 5.  * @author somlyaip
 6.  */
 7. public class IterateVector {
 8. 
 9.     /**
10.      * @param args the command line arguments
11.      */
12.     public static void main(String[] args) {
13. 
14.         int[] vector = { 9, 15, 6, 45, 81 };
15. 
16.         System.out.println("Forward:");
17.         for(int i = 0; i < vector.length; i++) {
18.             System.out.printf("v[%d] = %d\n", i, vector[i]);
19.         }
20. 
21.         System.out.println("\nBackward:");
22.         for(int i = vector.length - 1; i >= 0; i--) {
23.             System.out.printf("v[%d] = %d\n", i, vector[i]);
24.         }
25.     }
26. 
27. }
```

Bevezetés a programozásba

Hibaüzenetek értelmezése és hibák javítása

Somlyai Pál

Dunaújvárosi Főiskola

2015

Tartalom

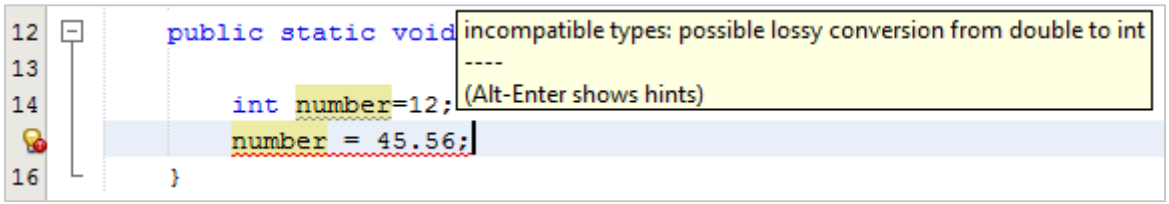
Bevezetés a programozásba – Hibaüzenetek értelmezése és hibák javítása

1. Hibák

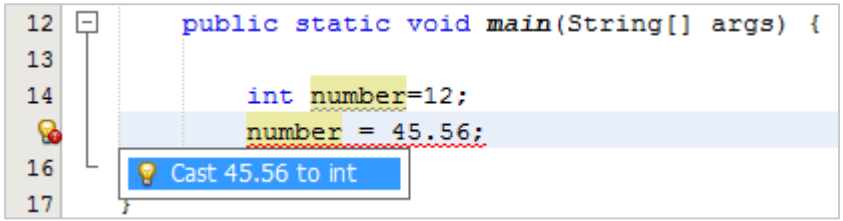
A Javában a hibákat keletkezésük ideje szerint két csoportba sorolhatjuk: fordításidejű- és futásidejű hibák közé.

1.1 Fordításidejű hibák

A fordításidejű hibák azok, amelyek már a fordítás során kiderülnek, például: két azonos nevű azonosítót használunk, vagy egy konstans változó értékét módosítjuk, esetleg egy egész típusú változóba próbálunk belegyömöszölni egy tört számot. Ezeket a hibákat a Netbeans jelzi számunkra még a futtatás előtt. Amikor a Netbeans az egyik sorában egy kifejezést piros hullámvonallal húz alá, akkor abban a sorban hibát talált. Ebben a sorban a sorszám helyén egy villanykörte ikon mellette egy piros pöttyel jelenik meg. Emellett a Projects ablakban a hibát tartalmazó forrásfájl neve mellett is ugyanezt az ikont jelenít meg. Ilyenkor, ha a hibás utasítás, vagy a sorszám helyetti hiba ikon fölé visszük az egeret, akkor megjelenik a hiba leírása:



Sajnos a furdításidejű hibaüzenetek néha elsőre semmitmondóak is lehetnek, azonban a legtöbbször elárulják a hiba okát. Az előző esetben a hibaüzenet a következőt jelenti: „inkompatibilis (azaz összeférhetetlen) típusok: lehetséges, hogy veszteséges konverzió fog történni double típusról integerre". A hibaüzenet mindig a hiba okát írja le, nem pedig a megoldás módját. Azonban ha bal egérgombbal rákattintunk a sorszám helyetti hibaikonra, vagy a hibás utasításon állva lenyomjuk az ALT + ENTER billentyűkombinációt: megjelenik egy ajánlat a hiba „megoldására":



Üssünk ENTER-t a művelet elvégzéséhez. Jelen esetben a hibát kijavította, de egyszerűbb lenne integer literált megadni a double literál típuskényszerítése helyett:

number = (int) 45.56;

1.2 Futásidejű hibák

A futásidejű hibák azok, amelyek a fordítás során nem szűrhetőek ki. Például egy szöveget beolvasunk a konzolablakról, amelyet megpróbáljuk átalakítani egész számmá, de tartalmaz egy x karaktert. Természetesen a konvertáló függvény hibát fog dobni. Valamint amikor legutóbb a tömbök helytelen indexelésénél tapasztaltuk, amikor a Javában valamilyen futásidejű hiba történik, akkor egy kivétel fog jelentkezni. A futásidejű hibákat a Javában kivételeknek nevezzük. Egy kivétel lényegében egy beszédesebb hibaüzenet, amely elárulja a hiba keletkezésének okát, valamint helyét is. A kivételek egy része megjelenik az output ablakban. A megjelenített adatok szerkezete mindig a következő:

Exception in thread <szálNeve> <osztályTeljesNeve>[: <kiegészítőInformáció>][1]
az <függvényTeljesNeve>(<forrásfájlNeve>:<kivételKeletkezésénekSorszáma>

Majd tekintsük meg a következő példát:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
at javatesting.JavaTesting.main(JavaTesting.java:19)

Mint ahogy azt már az előző alfejezetben láthattuk, a futásidejű hibák kivédése során biztosítanunkkell, hogy a hibát okozó feltétel ne lépjen fel ismét. Azaz az előbbi példakivételt úgy tudjuk megelőzni,ha nem hivatkozunk a 10-es, nem létező indexre a kódunkban. Szerencsére pontosan látjuk, hogymelyik fájlban, és melyik sorban keletkezett a kivétel, így könnyen javíthatjuk.

1 A <>-ek közötti részek behelyettesítendő értékek helyét jelentik, a []-ek közötti részek pedig tetszőlegesen elhagyható részeket.

```
 1. package arrayneighborssubtraction;
 2.
 3. import java.util.Vector;
 4.
 5. /**
 6.  *
 7.  * @author somlyaip
 8.  */
 9. public class ArrayNeighborsSubtraction {
10.
11.     /**
12.      * @param args the command line arguments
13.      */
14.     public static void main(String[] args) {
15.
16.         int[] array = { 5, 19, 4, 7, 1 };
17.
18.         for(int i = 0; i < array.length - 1; i++) {
19.             int difference = array[i + 1] - array[i];
20.             System.out.println("v[i + 1]: " + array[i + 1] + ", v[i]: " +
21.                     array[i] + ", diff: " + difference);
22.         }
23.     }
24.
25. }
```

```
 1. package recognizeandfixerrors;
 2.
 3. /**
 4.  *
 5.  * @author somlyaip
 6.  */
 7. public class RecognizeAndFixErrors {
 8.
 9.     /**
10.      * @param args the command line arguments
11.      */
12.     public static void main(String[] args) {
13.
14.         int number = Integer.parseInt(System.console().readLine());
15.         System.out.println(number);
16.     }
17.
18. }
```

Bevezetés a programozásba
Gyakorló feladatok

5. FEJEZET

1. Készítsen programot, amely beolvas egy vektorba 5 darab egész számot, majd megjeleníti a vektor tartalmát a képernyőn!

2. Készítsen programot, amely egy tetszőleges, tört számokkal feltöltött vektoron végighalad, és kiírja a tíznél nagyobb értékeket!

3. Egy String vektor egy egyszerű mondat szavait tartalmazza. A mondatban nem szerepel vessző. Jelenítse meg a mondatot a képernyőn, úgy, hogy az utolsó szó után egy pontot helyezzen el.

4. Készítsen programot, amely egy egész számokat tartalmazó vektoron végighaladva, kiírja az öttel maradék nélkül osztható elemek indexét!

5. Készítsen programot, amely egy tört számokat tartalmazó vektoron végighaladva, kiírja azokat az értékeket és az indexüket, amelyeknek a törtrésze nullától különbözik (azaz nem egész számok).

6. fejezet

Bevezetés a programozásba

Véletlenszámok, véletlenszám generálás

Somlyai Pál

Dunaújvárosi Főiskola

2015

Tartalom

Bevezetés a programozásba – Véletlenszámok, véletlenszám generálás

1. Véletlenszámok

A számítógépes programoknak sok esetben szüksége van (lásd: 1.1) véletlen generált számokra. A véletlenszámokat (random number) véletlenszám generátorokkal készítjük. „Ezek az általunk készített számok valójában „ál-véletlenszámok”, hiszen egy meghatározott szabály szerint képezzük őket egy adott véletlen kezdőértékből, a „magból”. (Ezt általában a számítógép órája alapján képzik). Lehetne „igazi” véletlen számokat is generálni a radioaktív bomlást vagy a kvantumfizika egyéb jelenségeit használva, de ezek nem volnának kellően hatékonyak.” [1] Tehát jelenleg ál-véletlenszámokat, vagy másképp nevezve pszeudo-véletlenszámokat tudunk generálni, ami azt jelenti, hogy a véletlenszámaink csak az értékük szerint véletlenszerűek, a létrehozásuk egy szabályos algoritmuson alapszik.

1.1 Véletlenszámok alkalmazása

A véletlenszámok legkönnyebben megérthető alkalmazási területe azok a valóságot szimuláló alkalmazások, amelyekben nagy szerepe van a véletlennek, esetleg a szerencsének (kinek hogy tetszik).

Az első ilyen terület: a játékok. Azon belül pedig a szerencsejátékok. Képzeljük el, amikor egy pókerjátékot programoznak, az osztó (aki természetesen az alkalmazásunk), egy algoritmus alapján keveri meg a paklit. Az algoritmusnak véletlenszerűen kell megkevernie a paklit, hogy szimuláljuk a valóságbeli eseményeket. Ki kell választanunk véletlenszerűen egy, vagy több lapot, majd egy másik pozícióra helyezni és ezt iterálni, megadott lépésszámban a keverés végrehajtásához.

Lépjünk át az RPG (role-playing game, szerepjáték, pl.: The Elder Scrolls V: Skyrim)/TBS (turn-based strategy, körökre osztott stratégiai játék, pl.: Heroes of Might and Magic) játékokra, melyekben, amikor két szereplő (vagy csapat) csatába keveredik, akkor az ellenfélnek okozott sebzések véletlenszerűek (egy adott tartományon belül). Itt a játékos/csapat alap sebzése és módosítói (fegyverek, kiegészítők, buffok: áldások) határoznak meg egy sebzés tartományt, amin belül kell véletlenszámot generálni a sebzés értékének. Emellett ha van kritikus találat a játékban (ami ismét egy százalékos valószínűséggel bír), akkor ismét véletlenszámmal kell megállapítani, hogy az adott sebzés megkapja-e a kritikus sebzés módosítóit is.

Emellett a titkosítási eljárások és bonyolult optimalizációs algoritmusok is a véletlenszámok használatán alapulnak.

1.2 Véletlenszám generálás

A Javában több módon hozhatunk létre véletlenszámokat, azonban a jelenlegi tudásunk alapján csak egy módot tudunk megismerni ebben a félévben. A Math.random() függvény segítségével van lehetőségünk generálni egy véletlenszámot a 0.0 és 1.0 közé. Az eredmény lehet pontosan 0.0, 0.0-nál nagyobb és 1.0-nál kisebb, azonban 1.0 nem lehet. A legtöbb programnyelvben, amikor meg kell adnunk egy véletlenszám generátornak a tartományt, akkor az eredmény az alsó határértéket felveheti, a felsőt pedig nem. A Math.random() függvény egy double típusú véletlenszámmal fog visszatérni.

Tehát ha 0.0..0,99. közé szeretnénk egy véletlenszámot generálni, akkor csak egyszerűen a Math.random() függvényt kell használnunk.

Azonban a legtöbb esetben két egész szám közé kell generálnunk véletlenszámokat. Ebben az esetben a következő kifejezés használata célravezető:

```
int randomNumber = (int)(Math.random() * (max - min + 1)) + min;
```

Vegyük a következő példát: a 20..40 tartományban kell véletlenszámot generálnunk. Ebben az esetben az értékadás jobb oldali kifejezésébe behelyettesítve a következő műveletet hajtjuk végre:

```
(int)(Math.random() * (40 - 20 + 1)) + 20
```

Nézzük meg, mi történik lépésről lépésre:

1. Elvégezzük a 40 - 20 + 1 műveletet, amely eredménye: 21

2. Megszorozzuk a véletlengenerált számmal, így a következő tartományban kapunk egy tört számot eredményként:
   a. 0..20,99
   b. hiszen 0 * 21 = 0 (alsó határérték), 0.99. * 21 = ~20.99

3. Viszont nekünk a 20.0 feletti értékekre nincs szükségünk, valamint mivel egész számokat szeretnénk generálni: ezért a törtrészre sincs szükségünk: így egyszerűen elhagyjuk a törtrészt, az integer típuskényszerítés: (int) segítségével. Az eredményünk a következő tartományban fog elhelyezkedni:
   a. 0..20

4. Majd a + 20 művelettel megnöveljük az eddigi értéket, aminek hatására eltoljuk a számegyenesen pozitív irányba 20 értékkel az eddigi eredmény tartományát, így bizonyosan a céltar tományban kapunk eredményt:
   a. 20..40

Mi történik, ha a tartományunk alsó- és felső határértéke is negatív szám? Vegyük példának a -100..-50 tartományt. Ebben az esetben a generált randomszámot (-50 – -100 + 1) azaz 51-el fogjuk beszorozni, majd levágjuk a törtrészt, így a 0..50 tartományban kapunk egy értéket, amelyet 100-al eltolunk negatív irányba, így megkapjuk a kívánt -100..-50 tartományt. Természetesen a képlet akkor is működik, ha a tartomány alsóértéke a negatív-, a felső pedig a pozitív tartományban helyezkedik el.

Amennyiben tört számokat szeretnénk generálni, egy megadott tartományba, akkor a következő képletet használhatjuk:

double randomNumber = Math.random() * (max - min) + min;

Azt viszont tartsuk szem előtt, hogy a double típus minden tört számjegyén nem fog megegyezni a generált maximum érték, mert a tényleges helyiértékpontos maximumnál egy kicsit alacsonyabb számot kaphatunk csak eredményként a szorzás művelet során, mivel legfeljebb 0.999. értékkel tudjuk megszorozni a max – min művelet eredményét. Valamint a számábrázolási pontatlanság miatt alacsony az esélye, hogy a minimum értékkel minden tizedeshelyen megegyező eredményt tudunk generálni. Azért ha hasonlítunk, inkább csak néhány tizedesjegy alapján próbáljunk hasonlítani. Két törtszámot a legegyszerűbben úgy tudunk összehasonlítani adott pontossággal, hogy vesszük a különbségük abszolút értékét, majd megnézzük, hogy ez az érték kisebb-e mint a precízió. Az abszolút érték miatt nem számít, hogy a nagyobból vonjuk ki a kisebbet, vagy ellenkezőleg. Ha az eredmény kisebb, mint a precízió, akkor adott tört helyiértékig megegyeznek: double number = 14.498712; double max = 14.5;

```
double number = 14.498712;
double max = 14.5;

double precision = 0.1;
if(Math.abs(number - max) < precision)
{
      System.out.println("MAX");
}

System.out.println(Math.abs(0.41235 - 0.41) < 0.01); // true
System.out.println(Math.abs(0.41235 - 0.41) < 0.001); // false
```

2. Hivatkozások

[1] Ambrus Gergely, Bárszi Gergely, Csikós Balázs, Frenkel Péter, Gács András, Gyárfás András, Hraskó András, Kiss Emil, Laczkovich Miklós, Lovász László, Montágh Balázs, Moussong Gábor, Pach János, Pelikán József, Recski András, Reiman István, „Új matematikai mozaik - 2. Véletlenszámok generálása," 2. 10. 2015. [Online]. Available: http://www.tankonyvtar.hu/hu/tartalom/tkt/uj-matematikai-mozaik-uj/ar04s02.html.

```
 1. package generaterandomnumber;
 2. 
 3. /**
 4.  *
 5.  * @author somlyaip
 6.  */
 7. public class GenerateRandomNumber {
 8. 
 9.     /**
10.      * @param args the command line arguments
11.      */
12.     public static void main(String[] args) {
13. 
14.         int min = -25;
15.         int max = 25;
16. 
17.         for(int i = 0; i < 100; i++) {
18.             int randomNumber = (int)(Math.random() * (max - min + 1)) + min;
19.             System.out.print(randomNumber);
20.             if(randomNumber == min)
21.                 System.out.println("\t-");
22.             else if(randomNumber == max)
23.                 System.out.println("\t+");
24.             else
25.                 System.out.println();
26.         }
27.     }
28. }
```

Programozási tételek
Összegzés

Dunaújvárosi Főiskola, Bevezetés a programozásba

A feladat megfogalmazása

Adott egy N elemű számsorozat: A(N). Számoljuk ki az elemek összegét! Az N elemű számsorozat azt jelenti, hogy egy tetszőleges elemszámú (N) pl.: integer vektoron kell elvégeznünk az összegzést. Tehát a feladat célja egy vektor elemeinek összegének kiszámítása. Az eredményt nem kell kiíratni, vagy bármi mást tenni vele, hiszen a programozási tételek olyan algoritmus darabok, amelyeket egy feladat megoldásában felhasználhatunk, azonban a feladat kiírásához illeszkedően ki kell egészítenünk (pl.: az összeg képernyőre írásával).

Pszeudokód

A feladat algoritmusa pszeudokódban lentebb található. Ez az általánosan elfogadott pszeudokód, vegyük figyelembe, hogy a tömb indexelése 1-től kezdődik.

```
Eljárás
      S = 0
      Ciklus I = 1-től N-ig
            S = S + A[I]
      ciklus vége
eljárás vége
```

Tehát mi történik az algoritmusban. Kezdetben nullára állítjuk az S változót, amely a sum (összeg) szóra utal. Erre azért van szükség, mert ennek a változónak az értékét növeljük meg minden vektorelem értékével. A vektor A jelölése tetszőleges, nincs semmilyen szabály rá, azonban a legtöbb pszeudokódban A-val jelölik az első vektort. A ciklus i = 1-től indul, és I <= N-ig fut. Ennek oka az, hogy az algoritmusunkat minél általánosabban kell megfogalmazni.

A józanész számára ez könnyebben érthető, azonban a leggyakrabban használt nyelvek közül a legtöbb programozási nyelv 0-tól indexeli a tömböket:

https://hu.wikipedia.org/wiki/T%C3%B6mb_%28adatszerkezet%29

Ahol n-t látunk azok főként a szkriptnyelvek, ahol a tömbök másképp működnek, mint az erősen típusos C alapú nyelveknél. Jelenleg ennek magyarázatától eltekintünk, ha tanulmányaitok során találkoztok majd ilyen nyelvekkel, akkor részletesen meg fogjuk értetni a működésüket. Ha eltekintünk az n kezdőindexű nyelvektől, akkor láthatjuk, hogy a nyelvek túlnyomó többsége 0-tól kezdi az indexelést. Ha megnézzük a leggyakrabban használt 5 programozási nyelvet (Java, C, C++, C#, Python),

http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

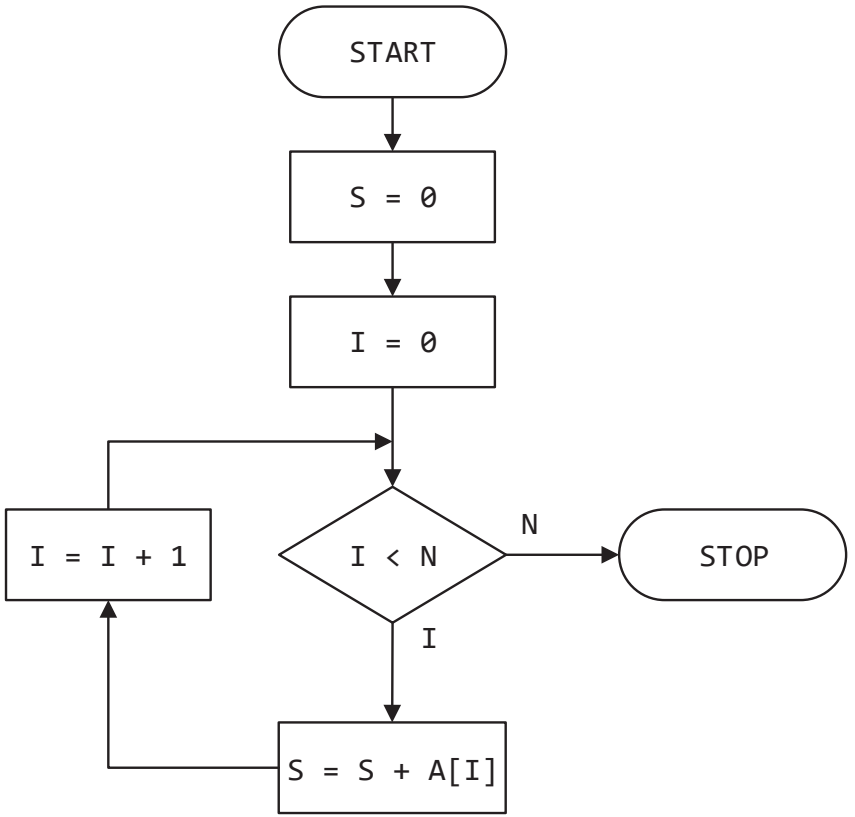
akkor látható, hogy mindegyik, kivétel nélkül a 0. indexet használja kezdőindexnek. Emellett az általunk használt Java is 0-tól indexel, ezért mi a pszeudokódok és folyamatábrák készítése során 0-tól fogunk indexelni, hiszen az az elv, hogy „az algoritmus legyen minél általánosabb", csak akkor érvényesül, ha a többségben lévő nyelvek szokásaihoz alkalmazkodunk. Nem kevésbé fontos az is, hogy így kevésbé kavarunk össze titeket, amikor implementáltok egy általunk megadott algoritmus Javában, nem kell állandóan az indexeket átváltogatnotok. Azonban tartsátok észben, hogy amennyiben egy 1-től indexelt algoritmussal találkoztok, akkor nem a készítője hibázott, egyszerűen csak más elveket követett.

Tekintsük meg az előző algoritmust 0 kezdőindex használatával:

```
Eljárás
      S = 0
      Ciklus I = 0-tól N - 1-ig
            S = S + A[I]
      ciklus vége
eljárás vége
```

Látható, hogy a ciklusunk I = 0-tól I <= N - 1-ig (azaz I < N–ig) fut. Így már teljes egészében Java kompatibilis algoritmust kaptunk.

Folyamatábra



Felhasználási területek

Bármilyen feladatnál, ahol szükség van egy vektor elemeinek összegzésére (is), pl.:

- Egy üzlet termékkészletének értéke tételesen egy vektorban van eltárolva. Azaz minden indexen, egy készleten lévő termék ára szerepel. Számítsa ki a termékkészlet összértékét!
- Egy vektorban megtalálható a 100 m-es síkfutás döntőjének mind a 10 indulójának időeredménye. Számítsa ki a futam átlagos időeredményét!

```java
package sumarray;

/**
 *
 * @author somlyaip
 */
public class SumArray {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int[] numbers = new int[10];

        int sum = 0;
        for(int i = 0; i < numbers.length; i++) {
            numbers[i] = (int)(Math.random() * 41) + 20;
            System.out.print(numbers[i] + ", ");
            sum += numbers[i]; // sum = sum + numbers[i];
        }
        System.out.println();
        System.out.println("Sum: " + sum);
        System.out.println("Average: " + (sum / (float) numbers.length));
    }

}
```

1

Programozási tételek
Megszámlálás

Dunaújvárosi Főiskola, Bevezetés a programozásba

A feladat megfogalmazása

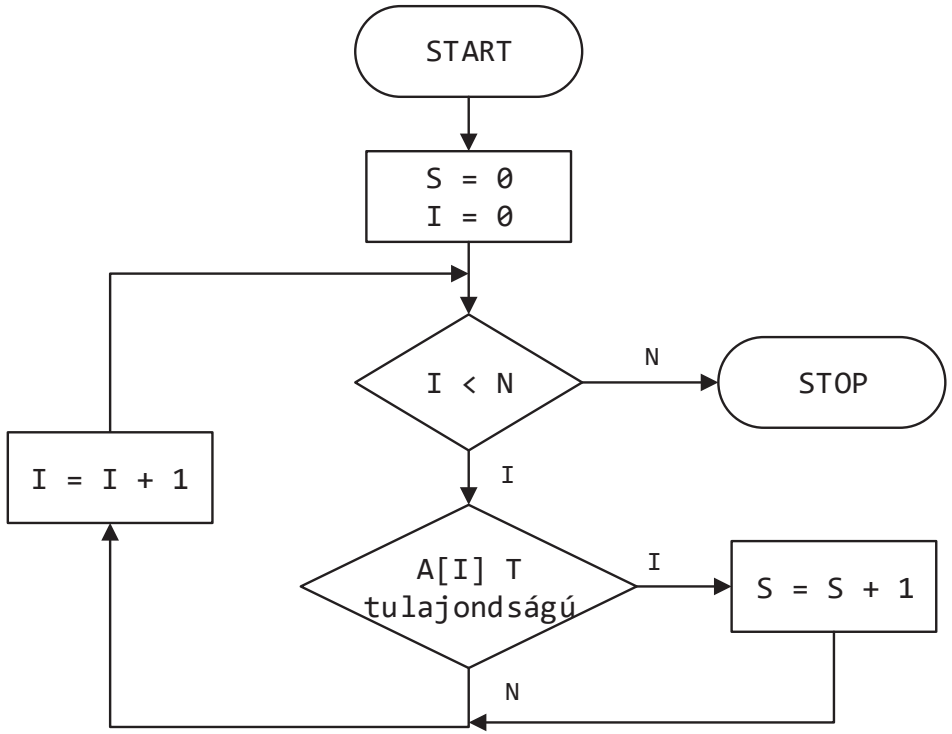
Adott egy N elemű sorozat és egy - a sorozat elemein értelmezett – T tulajdonság. Feladat a T tulajdonsággal rendelkező elemek megszámolása. Azaz ismételten van egy N elemszámú vektorunk, azonban most a típusa nem ismert. A „sorozat elemein értelmezett T tulajdonság" azt takarja, hogy van egy tetszőleges tulajdonság, amely értelmezhető a vektor elemein, pl.: egy számokat tartalmazó vektoron a tulajdonság: páros szám. Ha egy String vektoron lenne az előbbi a tulajdonság, akkor az nem lenne értelmezhető. A „T tulajdonsággal rendelkező elemek" azok az elemek, amelyek megfelelnek a T tulajdonságnak (az előző példában a páros számok). Tehát adott egy T-vel jelölt, tetszőleges feltétel, amelynek megfelelő elemek darabszámára vagyunk kíváncsiak a vektorunkban.

Pszeudokód

```
Eljárás
      S = 0
      Ciklus I = 0-tól N - 1-ig
            Ha A[I] T tulajdonságú akkor
                  S = S + 1
            ha vége
      ciklus vége
eljárás vége
```

Az „A[I] T tulajdonságú" feltétel a behelyettesítendő logikai kifejezés helyes. Ide olyan feltételvizsgálatot kellene beilleszteni, mint például A[I] % 2 == 0, vagy A[I] > 12, stb. Abban az esetben, ha a feltétel teljesül, akkor inkrementáljuk az S változó értékét, így a ciklus végeztével megkapjuk a T tulajdonságú elemek darabszámát.

Folyamatábra



Felhasználási területek

Bármilyen adatsorból megszámlálni az adott tulajdonságú elemek számát. Például:

- Adott egy kurzus százalékos zh eredményei egy vektorban. Számoljuk meg, hány hallgató teljesítette legalább 50%-os eredménnyel a dolgozatot!

- Adott egy vektorban egy tantárgyra jelentkezett hallgatók félév végi osztályzatai az adott tantárgyból. Számolja ki, hogy mekkora a tárgy teljesítésének aránya!
    - Itt meg kell számolni az elégtelentől és nem teljesítettétől különböző eredményeket, amelyet el kell osztani a jelentkezett hallgatók darabszámával, majd felszorozni 100- al, hogy megkapjuk a teljesítés százalékos arányát. Ennek a feladatnak az egyik részfeladata a megszámlálás tételének alkalmazása, de természetesen a legtöbb feladatnál – ehhez hasonlóan – a programozási tételeket ki kell egészítenünk egyéb műveletekkel a feladat teljesítéséhez.

- Egy adott napon a légszennyezettség mértékének csökkentése végett csak a páros rendszámú járművek közlekedhettek egy városban. Ismert az egyik csomópontban álló kamera által rögzített rendszámok listája. Számoljuk meg, hogy hány páratlan rendszámú autó nem tartotta be a korlátozást! Számítsuk ki, hogy mekkora volt az elhaladó, páratlan rendszámú autók aránya az összes rögzített autóhoz képest.

```
 1. package countingoddnumbers;
 2. 
 3. /**
 4.  *
 5.  * @author somlyaip
 6.  */
 7. public class CountingOddNumbers {
 8. 
 9.     /**
10.      * @param args the command line arguments
11.      */
12.     public static void main(String[] args) {
13. 
14.         int[] numbers = {5, 7, 4, 9, 6, 1, 2, 5};
15. 
16.         int oddNumbersCount = 0;
17.         for(int i = 0; i < numbers.length; i++) {
18.             if(numbers[i] % 2 != 0) {
19.                 oddNumbersCount++;
20.             }
21.         }
22. 
23.         System.out.println("There are " + oddNumbersCount +
24.                 " odd numbers in the array.");
25.     }
26. 
27. }
```

Programozási tételek

Minimum- és maximum kiválasztás
(Nulladik indexre állítás-, érték kiválasztás módszere)

Dunaújvárosi Főiskola, Bevezetés a programozásba

Minimum kiválasztás

A minimum- és maximum kiválasztás külön-külön programozási tétel. Ezért kezdetben külön fogjuk kezelni őket, viszont később összevontan fogjuk megfogalmazni őket.

A feladat megfogalmazása

Adott egy N elemű számsorozat. Válassza ki a legalacsonyabb értéket (minimumot)! Tehát meg kell keresnünk a vektorban található elemek közül a legkisebbet.

Pszeudokód

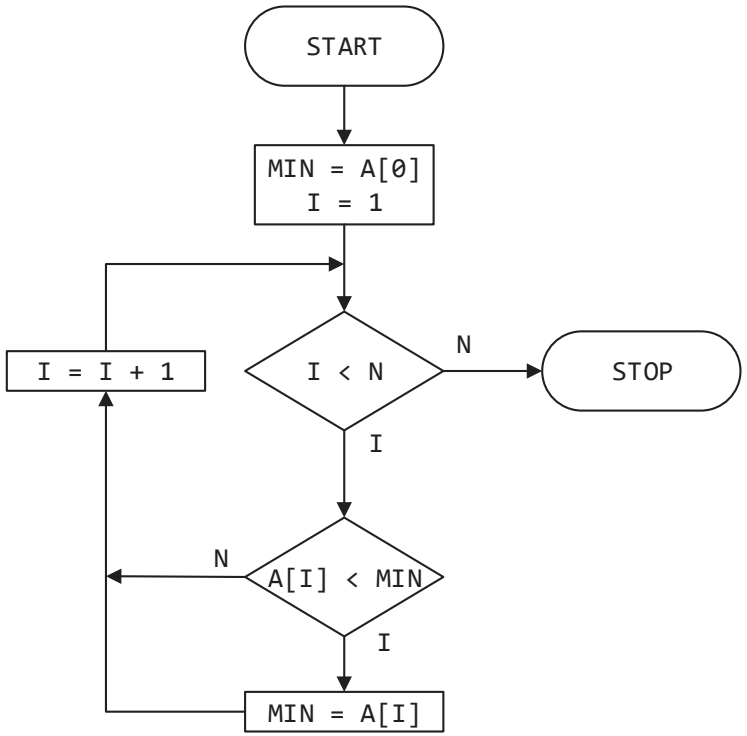
```
Eljárás
      MIN = A[0]
      Ciklus I = 1-től N-1-ig
            Ha A[I] < MIN akkor
                  MIN = A[I]
            ha vége
      ciklus vége
eljárás vége
```

Figyeljük meg, mi történik a ciklusban! Végigiteráljuk a tömbelemeket (jelen példában az első elem kivételével), majd minden elemet hozzá hasonlítjuk az eddigi minimumhoz: ha az aktuális elem kisebb, mint az eddig minimum, akkor legyen az aktuális elem a minimum.

Azonban a minimum változót valamilyen értékkel inicializálni kell. De milyen kezdőértékkel inicializáljuk? Mondjuk 0-val? A válasz helytelen. Mi történik, ha a vektorunkban csak pozitív számok vannak? Akkor mindegyik elemnél hamis lesz az A[I] < MIN feltétel, így a ciklus lefutását követően a minimum értéke 0 lesz, holott nincs ilyen érték a vektorunkban! Emiatt a leggyakrabban használt módszer az: ha az első elem értékével inicializáljuk a minimum változót. Így biztosan egy olyan értékhez fogjuk hasonlítani a többi értéket, amelyik megtalálható a vektorban, tehát az algoritmusunk nem fog helytelen eredményt adni. Azonban mivel az első (nulladik indexű) elemet átadtuk a minimum változónak, ezért az első elemet már nem kell hasonlítani saját magához, így nyugodtan indíthatjuk a ciklust I = 1-től, így megspórolunk egy teljesen felesleges ciklusiterációt. Emiatt nevezzük nulladik indexre állítás módszerének.

Az érték kiválasztás módszere megnevezést pedig azért kapta, mert az legkisebb elem értékét- és nem az indexét választjuk ki.

Folyamatábra



Felhasználási területek

Bármilyen feladatban, ahol egy számsorozatból ki kell választanunk a legalacsonyabb értéket. Pl.: Adott egy naptári napon mért legalacsonyabb hőmérsékletek 20 évre visszamenőleg egy vektorban. Határozza meg, mennyi volt a legalacsonyabb hőmérsékelt az adott naptári napon!

Maximum kiválasztás

A feladat megfogalmazása

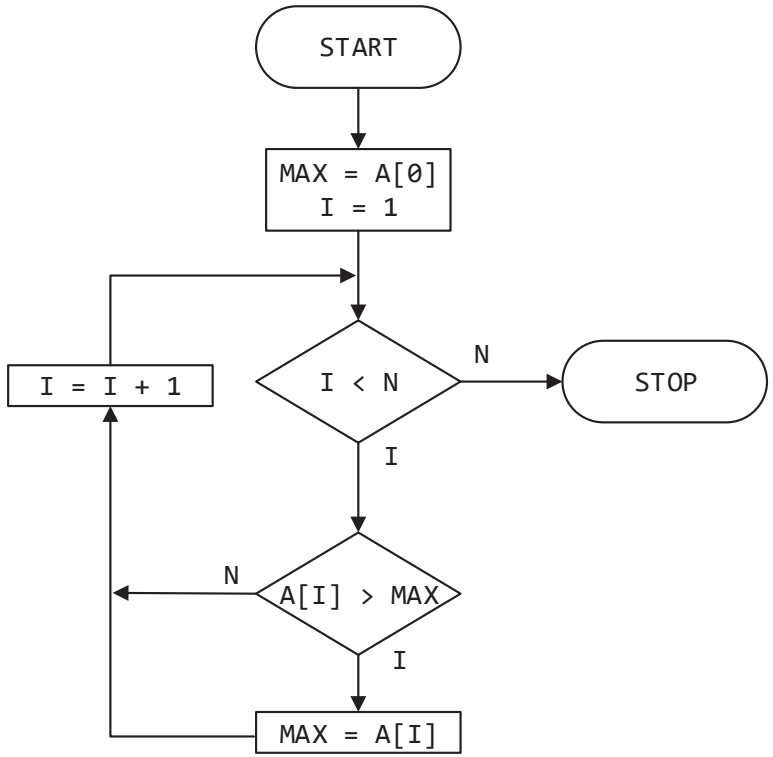
Adott egy N elemű számsorozat. Válassza ki a legmagasabb értéket (maximumot)! Tehát meg kell keresnünk a vektorban található elemek közül a legnagyobbat.

Pszeudokód

```
Eljárás
      MAX = A[0]
      Ciklus I = 1-től N-1-ig
            Ha A[I] > MAX akkor
                  MAX = A[I]
            ha vége
      ciklus vége
eljárás vége
```

Látható, hogy ismételten a nulladik indexre állítás módszerét alkalmaztuk, valamint az értéket választottuk ki. A ciklus végigiterálja a második (1. indexű) elemtől kezdődően a vektor elemeit, majd amennyiben talál magasabb értéket az eddigi maximumnál, felülírja vele a maximum értéket.

Folyamatábra



Felhasználási területek

Bármely olyan feladatban, ahol szükség van egy számsorozat maximum értékének kiválasztására. . Pl.: Adott egy naptári napon mért legalacsonyabb hőmérsékletek 20 évre visszamenőleg egy vektorban. Határozza meg, mennyi volt a legmagasabb hőmérsékelt az adott naptári napon!

Minimum- és maximum kiválasztás együtt

A többi minimum- és maximum kiválasztási módszert együttesen fogjuk ismertetni, hiszen csak a feltételvizsgálat különbözik a tételekben, azonban tartsuk szem előtt, hogy a minimum- és maximum kiválasztás két külön tétel!

A feladat megfogalmazása

Adott egy N elemű számsorozat. Válassza ki a legalacsonyabb- és legmagasabb értéket (minimumot és maximumot)! Tehát meg kell keresnünk a vektorban található elemek közül a legkisebbet és a legnagyobbat.

Pszeudokód

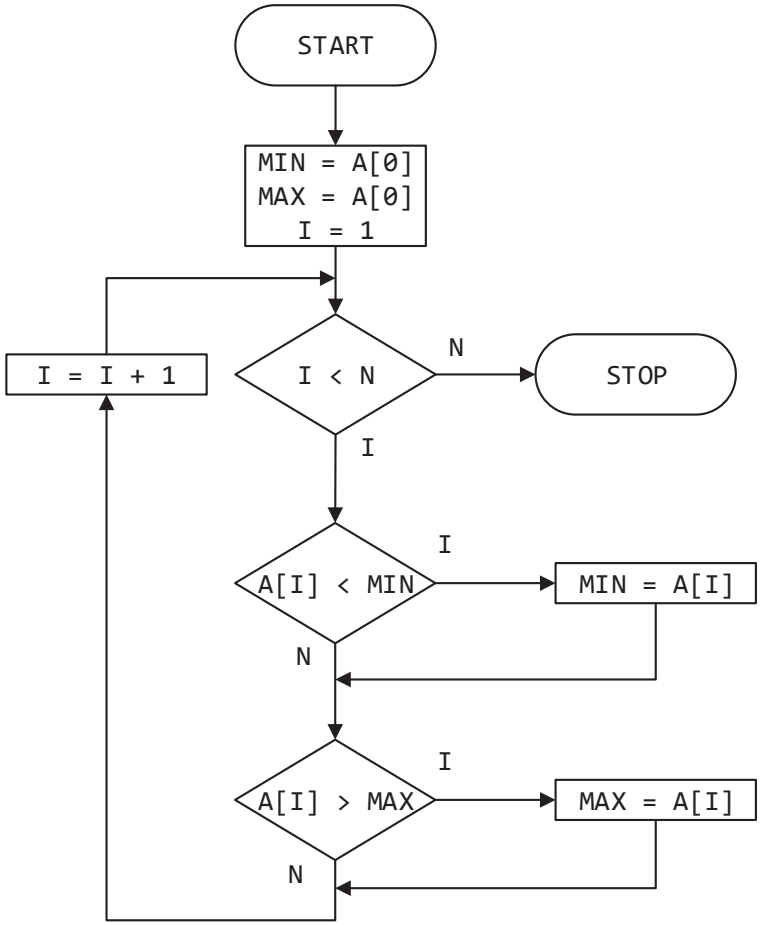
```
Eljárás
      MIN = A[0]
      MAX = A[0]
      Ciklus I = 1-től N-1-ig
            Ha A[I] < MIN akkor
                  MIN = A[I]
            ha vége
            Ha A[I] > MAX akkor
                  MAX = A[I]
            ha vége
      ciklus vége
eljárás vége
```

Az algoritmusban nincs semmilyen többlet, csupán egy cikluson belül megvalósítottuk a minimum- és maximum kiválasztását nulladik indexre állítás módszerével és érték kiválasztással. Azonban az algoritmus megvalósítható kicsivel hatékonyabban is: ha összetett if-else feltételt alkalmazunk:

```
Eljárás
      MIN = A[0]
      MAX = A[0]
      Ciklus I = 1-től N-1-ig
            Ha A[I] < MIN akkor
                  MIN = A[I]
            különben Ha A[I] > MAX akkor
                  MAX = A[I]
            ha vége
      ciklus vége
eljárás vége
```

Ez a megoldás azért hatékonyabb és elegánsabb is, mert amennyiben az első feltétel teljesül (az aktuális elem kisebb, mint a minimum), akkor a második elágazás feltétele már nem teljesülhet (hiszen ha az aktuális elem kisebb, mint a minimum, nem lehet magasabb, mint a maximum). Azonban, hogy könnyebben elválasztható legyen a két tétel egymástól, két különálló else ág nélküli elágazást fogunk használni a tételek ismertetésekor, de nyugodtan használjátok az utóbbi, hatékonyabb algoritmust, ha a minimumot és a maximumot is ki kell választani egy feladatban.

Folyamatábra



Felhasználási területek

Bármilyen olyan feladat esetén, ahol egy számsorozatban a minimumot- és a maximumot is meg kell határozni.

Programozási tételek

Minimum- és maximum kiválasztás
(Ellenkező határértékre állítás-, érték kiválasztás módszere)

Dunaújvárosi Főiskola, Bevezetés a programozásba

A feladat megfogalmazása

Adott egy N elemű számsorozat. Válassza ki a legalacsonyabb- és legmagasabb értékeket (minimumot és maximumot)! Tehát meg kell keresnünk a vektorban található elemek közül a legkisebbet és a legnagyobbat. Emellett most az ellenkező határértékre állítás módszerét fogjuk használni, valamint ismét csak az elemek értékét fogjuk meghatározni. Fontos megjegyezni, hogy az ellenkező határértékekre állítás csak akkor használható, ha ismerjük a számsorozatban elhelyezkedő számok alsó- és felső határértékét.

Például:

- Egy hét folyamán minden nap feljegyeztük az ébredésünk időpontját egész órára kerekítve, pl.:
  - 6, 10, 5, 12, 18, 10, 9
  - Válasszuk ki melyik volt a legkorábbi- és a legkésőbbi ébredés órája!

Mivel tudjuk, hogy órákat tartalmaz a számsorozatunk, ezért tudjuk, hogy az alsó határérték 0, a felső pedig 23 (tekintsünk el attól, hogy mi történik, ha mondjuk 23:31-kor ébredünk fel). Így bizonyosak lehetünk abban, hogy 0-nál alacsonyabb, valamint 23-nál magasabb értékek nem szerepelhetnek a vektorunkban.

Pszeudokód

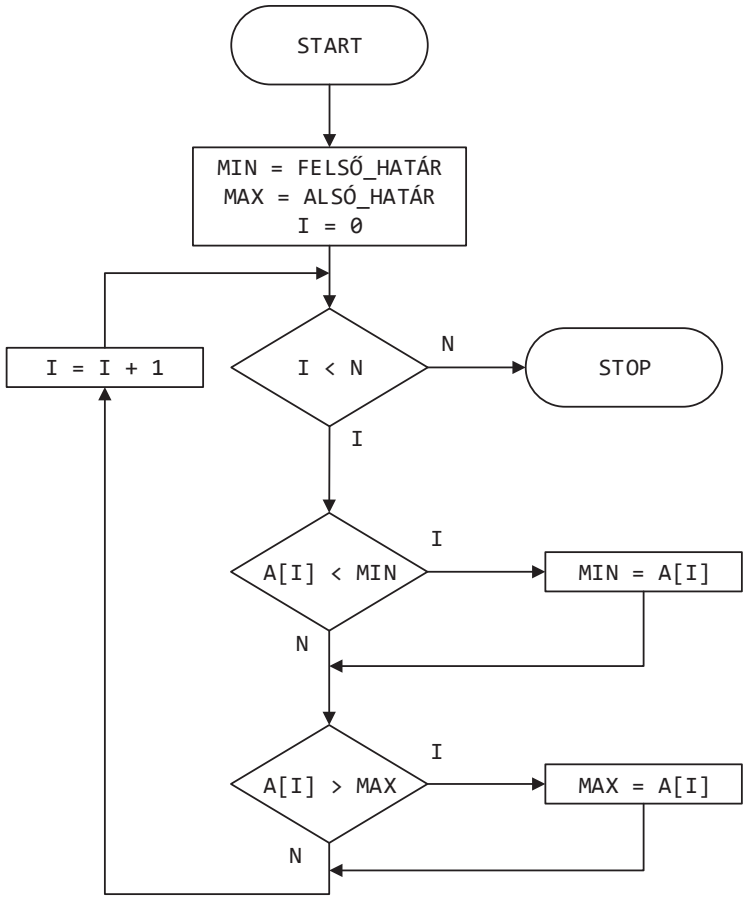
Alapozzunk az előző megállapításra. Ha a minimum értéket tároló változónk kezdőértékét a lehetséges maximum értékre (23) állítjuk, akkor nem fordulhat elő, hogy csak nála magasabb érték szerepel a számsorozatunkban és így helytelen eredményt fogunk visszaadni.

Valamint ha a maximum értékét eltároló változót a lehetséges legalacsonyabb értékről indítjuk, akkor sem lehetséges, hogy túl magas értéket fogunk megkapni, amelyik nem is szerepel a számsorozatunkban.

```
Eljárás
      MIN = FELSŐ_HATÁR
      MAX = ALSÓ_HATÁR
      Ciklus I = 0-tól N-1-ig
            Ha A[I] < MIN akkor
                  MIN = A[I]
            ha vége
            Ha A[I] > MAX akkor
                  MAX = A[I]
            ha vége
      ciklus vége
eljárás vége
```

Az algoritmus kísértetiesen hasonlít az előző alfejezetben ismertetett módszerre. A különbség csupán annyi, hogy a MIN és MAX változók értékét nem a számsorozat első elemével inicializáltuk, hanem a felső- és az alsó határértékkel, valamint a ciklusváltozónkat nem 1-től, hanem 0-tól indítottuk.

Folyamatábra



Felhasználási területek

Bármilyen olyan feladatnál, amikor egy számsorozat minimumát és/vagy maximumát kell kiszámolni, valamint ismerjük a számsorozat értékeinek alsó- és felső határértékeit.

Programozási tételek

Minimum- és maximum kiválasztás
(Nulladik indexre állítás-, index kiválasztás módszere)

Dunaújvárosi Főiskola, Bevezetés a programozásba

A feladat megfogalmazása

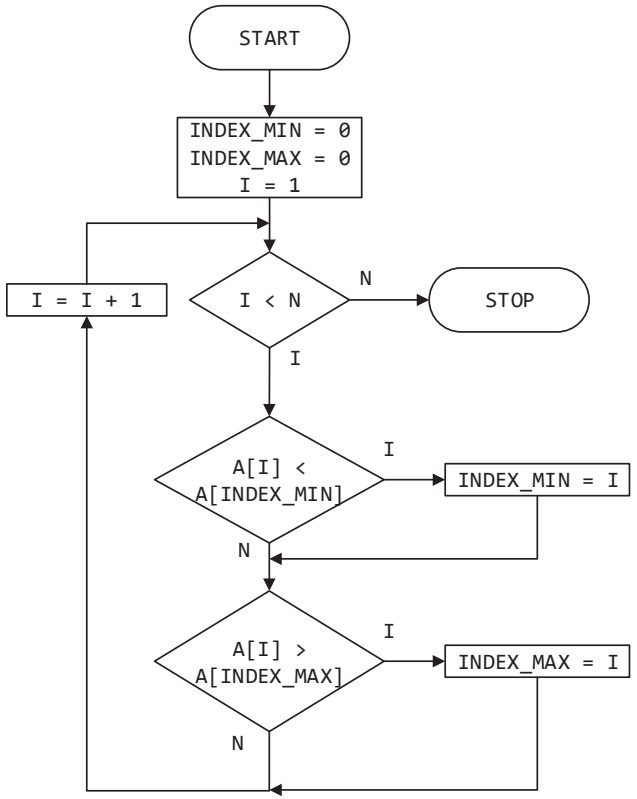
Adott egy N elemű számsorozat. Válassza ki a legalacsonyabb- és legmagasabb értékeket (minimumot és maximumot)! Az alkalmazott módszer nagyban hasonlít az először ismertetett módszerre, ugyanis ismételten a nulladik indexre állítás módszerét fogjuk alkalmazni, azonban nem a legnagyobb- és legkisebb elem értékét, hanem az indexüket fogjuk kiválasztani.

Pszeudokód

```
Eljárás
      INDEX_MIN = 0
      INDEX_MAX = 0
      Ciklus I = 1-től N-1-ig
            Ha A[I] < A[INDEX_MIN] akkor
                  INDEX_MIN = I
            ha vége
            HA A[I] > A[INDEX_MAX] akkor
                  INDEX_MAX = I
            ha vége
      ciklus vége
eljárás vége
```

Az algoritmus logikailag megegyezik az először ismertetett minimum- és maximum kiválasztási módszerrel, hiszen a minimum és a maximum értékre hivatkozó változót a nulladik indexű értékre állítjuk. Az előző példában közvetlenül az értékét tároltuk el benne, míg jelenleg csak az eddig legalacsonyabb- és legmagasabb értékű elem indexét (a segítségével tudunk hivatkozni a vektor egy elemére az indexelője segítségével). A ciklusban pedig az első indexű elemtől kezdve hasonlítottuk az elemek értékét az eddig eltárol minimumhoz és maximumhoz. A félkövérrel szedett részek jelölik a különbséget a két algoritmus között. Figyeljük meg, hogy az összehasonlítások során a minimum és maximum indexének segítségével kiolvassuk az eddigi legalacsonyabb és legmagasabb értéket, majd ahhoz hasonlítjuk az aktuális elem értékét. Amennyiben valamelyik feltétel teljesül, akkor nem az értékkel, hanem az indexxel írjuk felül a minimum vagy a maximum értékre hivatkozó változót. Úgy tűnhet, hogy lényegében ugyanazon logika alapján, csak kicsit nyakatekertebben valósítottuk meg a feladatot. Azonban előfordul, hogy nem csak a minimum értékére, hanem indexére is szükségünk van. (Lásd: Felhasználási területek)

Folyamatábra



Felhasználási területek

Bármilyen olyan feladat során, amikor nem csak a minimum és/vagy maximum értéket kell meghatároznunk, hanem szükségünk van az érték indexére is.

Például:

– Adott egy vektorban az évenkénti születések száma Magyarországon 1900 óta. A 0. indexen található az 1900-ban születettek száma, az 1. az 1901-ben, stb. Határozza meg, hogy melyik évben születtek a legtöbben, és melyikben a legkevesebben! A születések számát is írassa ki!

o Ebben a feladatban nem elég csak az értékeket meghatározni, szükségük van az indexekre is. Kiválasztást követően az 1900 + INDEX_MIN kifejezéssel megadhatjuk a legalacsonyabb születéssel bíró évet, míg az 1900 + INDEX_MAX kifejezéssel a legmagasabb születéssel bírót. Az indexek behelyettesítésével pedig kiolvashatjuk a legalacsonyabb és legmagasabb értékeket is.

Összegzés

Természetesen ellenkező határértékre állítás- és index kiválasztás módszere is létezik, azonban azeddigi tételek alapján remélhetőleg senkinek sem fog problémát okozni az algoritmus megalkotása. Azonban, mint ahogy említettük, a jelenlegi tétel az univerzálisan használható, amely segítségével szinte bármilyen probléma megoldható.

Véleményünk szerint a nulladik indexre állítás módszerét a legcélszerűbb alkalmazni, mégpedig az adott feladatnak megfelelő kiválasztással, ha szükség van az indexre, akkor index kiválasztással, ha nem, akkor elég az érték kiválasztással. Természetesen nem okoz problémát, ha az index kiválasztást alkalmazzuk olyan esetben, amikor nincs szükségünk az indexre.

```
 1. package minmax;
 2. 
 3. /**
 4.  *
 5.  * @author somlyaip
 6.  */
 7. public class MinMaxWithFirstElement {
 8. 
 9.     /**
10.      * @param args the command line arguments
11.      */
12.     public static void main(String[] args) {
13. 
14.         int[] numbers = {-2, 3, -5, -11, 8, 35, -6, 1, -7, 12};
15.         int min = numbers[0];
16.         int max = numbers[0];
17.         for(int i = 1; i < numbers.length; i++) {
18.             if(numbers[i] < min)
19.                 min = numbers[i];
20.             if(numbers[i] > max)
21.                 max = numbers[i];
22.         }
23.         System.out.printf("Min: %d, max: %d\n", min, max);
24.     }
25. 
26. }
```

```
 1. package minmax;
 2. 
 3. /**
 4.  *
 5.  * @author somlyaip
 6.  */
 7. public class MinMaxWithBoundaries {
 8. 
 9.     public static void main(String[] args) {
10. 
11.         int[] numbers = {-15, -2, 3, -5, -11, 8, 35, -6, 1, -7, 12};
12.         int min = 35;
13.         int max = -15;
14.         for(int i = 0; i < numbers.length; i++) {
15.             if(numbers[i] < min)
16.                 min = numbers[i];
17.             if(numbers[i] > max)
18.                 max = numbers[i];
19.         }
20.         System.out.printf("Min: %d, max: %d\n", min, max);
21.     }
22. 
23. }
```

```
 1. package minmax;
 2. 
 3. /**
 4.  *
 5.  * @author somlyaip
 6.  */
 7. public class MinMaxIndex {
 8. 
 9.     public static void main(String[] args) {
10. 
11.         int[] numbers = {-2, 3, -5, -11, 8, 35, -6, 1, -7, 12};
12. 
13.         int indexMin = 0;
14.         int indexMax = 0;
15.         for(int i = 1; i < numbers.length; i++) {
16.             if(numbers[i] < numbers[indexMin])
17.                 indexMin = i;
18.             if(numbers[i] > numbers[indexMax])
19.                 indexMax = i;
20.         }
21. 
22.         System.out.printf("Min: %d (numbers[%d])\n", numbers[indexMin], indexMin);
23.         System.out.printf("Max: %d (numbers[%d])\n", numbers[indexMax], indexMax);
24.     }
25. 
26. }
```

Bevezetés a programozásba
Gyakorló feladatok

6. FEJEZET

1. Véletlenszámgenerálás gyakorlása. Generáljon 20 véletlenszámot a következő tartományokban:
- -12..51
- -91..-45
- 20..57
- 5..200
- 10,2..56,78
- -12,41..60.15

2. Egy mobiltelefon előfizető másodpercalapú számlázást alkalmazó csomagot használ, amelyben bármilyen irányba fix 25 HUF percíjjal kezdeményezhet hívást. Adott egy vektorban az egy hónap altt kezdeményezett hívásainak időtartalma (hívásonként egy vektorelem), másodpercben megadva. A vektor tartalmazzon 20 elemet, melyek 1 és 3600 közötti véletlengenerált értékek lehetnek.

- Mekkora összeget „telefonált le”?

- Az előfizetése 4500 HUF havidíjú, amely teljes mértékben lebeszélhető. Fedezi-e a költségeit? Amennyiben nem, mekkora összeget kell különbözetként befizetnie?

3. Egy gépkocsi heti útnyilvántartásában adott a km-óra állása induláskor és érkezéskor. A benzin ára 340 HUF/L, a gépkocsi fogyasztása pedig 6,8 L/100 Km. Számítsa ki az autó benzinköltségét a teljes hétre!

- **[SPOILER VESZÉLY! – Ha nincs szüksége segítségre, ne olvassa tovább!]** Elismerem, ez egy kicsit combos feladat. A km-óra állását induláskor és érkezéskor – jelenlegi tudásunkhoz mérten – a legcélszerűbb két külön vektorban tárolni. Pl.: az oraIndul vektorban tárolnánk az indulási értékeket, az oraErkez vektorban pedig az érkezésekét. Az egyező indexeken, adott nap induló- és érkező állását tárolnánk. Tehát az oraIndul[0] értéke lenne a hétfői indulás értéke, az oraErkez[0] pedig a hétfői érkezés értéke.
- Ha nagyon nem megy, akkor oldjátok meg úgy, hogy egy vektorban, a napi megtett távolságok szerepelnek.

4. Adott egy vektorban egy tantárgyra jelentkezett hallgatók félév végi osztályzatai az adott tantárgyból (jeles, jó, közepes, elégtelen, elégséges, nem teljesítette). Számolja ki, hogy mekkora a tárgy teljesítésének aránya!

– **[SPOILER VESZÉLY! – Ha nincs szüksége segítségre, ne olvassa tovább!]** Itt meg kell számolni az elégtelentől és nem teljesítettétől különböző eredményeket, amelyet el kell osztani a jelentkezett hallgatók darabszámával, majd felszorozni 100-zal, hogy megkapjuk a teljesítés százalékos arányát.

5. Egy adott napon a légszennyezettség mértékének csökkentése végett csak a páros rendszámú járművek közlekedhettek egy magyarországi városban. Ismert az egyik csomópontban álló kamera által rögzített rendszámok listája. A listában csak a rendszámok második tagja, azaz a háromjegyű számok szerepelnek. (Az egyedi rendszámoktól tekintsünk el!) Számoljuk meg, hogy hány páratlan rendszámú autó nem tartotta be a korlátozást! Számítsuk ki, hogy mekkora volt az elhaladó, páratlan rendszámú autók aránya az összes rögzített autóhoz képest!

6. Adott egy naptári napon mért legalacsonyabb hőmérsékletek 20 évre visszamenőleg egy vektorban. Határozza meg, mennyi volt a legalacsonyabb hőmérsékelt az adott naptári napon!

7. Adott egy naptári napon mért legalacsonyabb hőmérsékletek 20 évre visszamenőleg egy vektorban. Határozza meg, mennyi volt a legmagasabb hőmérsékelt az adott naptári napon!

8. Adott egy vektorban az évenkénti születések száma Magyarországon 1900 óta. A 0. indexen található meg az 1900-ban születettek száma, az 1. az 1901-ben, stb. A vektor 115 elemű legyen, és véletlengenerált értékekkel töltse fel 50 000 és 100 000 között! Határozza meg, hogy melyik évben születtek a legtöbben, és melyikben a legkevesebben! A születések számát is írassa ki, ezres csoportokat használva!

– **[SPOILER VESZÉLY! – Ha nincs szüksége segítségre, ne olvassa tovább!]** Ebben a feladatban nem elég csak az értékeket meghatározni, szükségük van az indexekre is. Kiválasztást követően az 1900 + INDEX_MIN kifejezéssel megadhatjuk a legalacsonyabb születéssel bíró évet, míg az 1900 + INDEX_MAX kifejezéssel a legmagasabb születéssel bírót. Az indexek behelyettesítésével pedig kiolvashatjuk a legalacsonyabb- és legmagasabb értékeket is.

7. fejezet

Bevezetés a programozásba

Függvények bevezető

Somlyai Pál

Dunaújvárosi Főiskola

2015

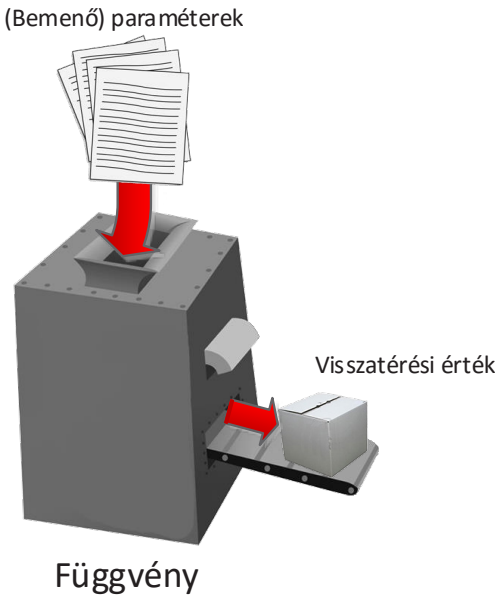
Tartalom

Bevezetés a programozásba – Függvények bevezető

A programozásban a függvények segítségével tudjuk a kódunkat rendszerezni, átláthatóbbá tenni és eltűntetni a felesleges ismétlődéseket. Mielőtt jobban kitérnénk a felhasználási területükre, ismerjük meg a működésüket.

1. Működésük

A függvények működését a legjobban úgy tudjuk elképzelni, ha úgy tekintünk rájuk, mint egy tetszőleges gépre, amely valamilyen alapanyagokon, elvégez egy adott műveletet, majd visszaadja az elkészült terméket. Most gondoljunk a függvényre egy csomagológépként:



Az alapanyagok a bemenő paraméterek, azaz jelen esetben a becsomagolandó dokumentumok. A csomagológép megkapja az alapanyagot, elvgézi rajta a kívánt műveleteket, majd visszaadja az eredményt: a késztermékét, ami a visszatérési érték, azaz a csomag.

Már biztosan mindenki használt függvényeket: az Excel függvényeket. Pontosan úgy működnek a programozási függvények is, sőt lényegében az Excelben, programozási függvényeket hívtatok meg, csak egy táblázatkezelőben, és nem a Netbeansben.

A Javában két féle függvényt különböztetünk meg: az első a saját magunk által definiált függvények (később), a másik pedig a Java fejlesztői által rendelkezésünkre bocsájtott, úgynevezett beépített függvények. A beépített függvények és az előző példa között nagyon nagy a párhuzam. Vegyük például a *Math.pow()* függvényt (pow: power of, azaz valaminek a hatványra emelése). A függvény zárójelei között felsorolt első értéket (alap), felemeli a második értékre (kitevő), majd visszaadja az eredményt:

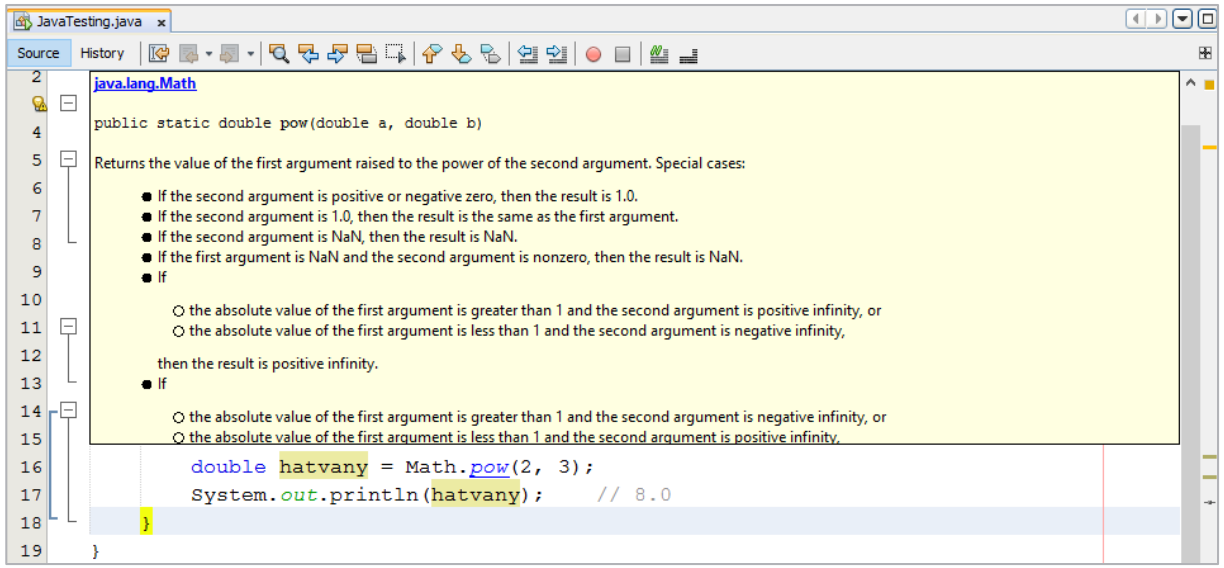
```
double hatvany = Math.pow(2, 3);
System.out.println(hatvany); // 8.0
```

Természetes kettő a harmadikon értéke nyolc. Hogy miért double, arra később kitérünk. Jelen esetben a Math.pow() függvény maga az a gép, ami megkapja az alapanyagot (2 és 3), majd elvégzi rajtuk a kívánt műveletet és visszaadja a késztermékét. Azért említettem, hogy nagy a párhuzam az ábra és a beépített Java függvények között, mert a beépített függvények működését nem feltétlenül kell ismernünk. Egyszerűen elég annyit tudnunk, hogy hogyan kell használnunk őket, ugyancsak, mint a példában említett gépeket.

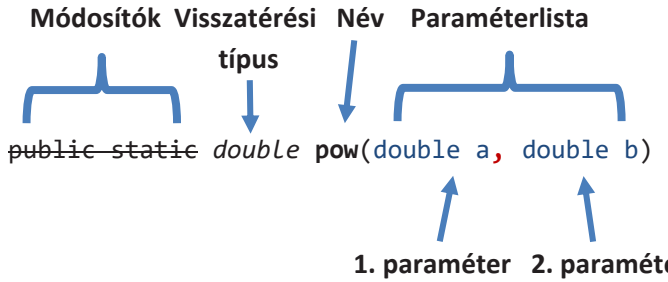
Amikor használunk egy függvényt, azt a programozásban függvényhívásnak nevezzük. A függvényhívás szabályai mindig az adott függvény tulajdonságaitól függ, amely tulajdonságokat a felépítésük megértésével ismerhetünk fel.

2. Felépítésük

Ha a Netbeansben a *Math.pow()* függvény hívásának helyén (az ábrán a 16 sor), a pow szó fölé visszük az egérkurzort, majd lenyomjuk a CTRL billentyűt, megjelenik a függvény leírása:



Számunkra a legfontosabb a második sorban található információ, amely a függvény szignatúráját (~aláírását) ábrázolja (azért nevezzük aláírásnak, mert két függvény szignatúrája nem lehet azonos, mint ahogy az aláírások is egyediek[1]):



Számunkra a *public static* módosítók jelentése jelenleg lényegtelen. Amikor szükségessé válik a használatuk (következő félévek), pontosan el fogjuk magyarázni a jelentésüket. A dőlt betűvel szedett double a függvény visszatérési típusa, azt jelöli, hogy a függvény milyen típusú értéket ad vissza a meghívását követően: azaz milyen adattípusú változóban kell eltárolni a függvényhívás eredményét, vagy egy kifejezésben elhelyezve a függvényhívást, milyen értéket fog a hívás helyére helyezni (később). A név jelenti azt az azonosítót, amivel hivatkozni tudunk a függvényre, azaz aminek a segítségével meghívhatjuk. Viszont mi nem csak egyszerűen pow-ként hívtuk meg, hanem Math.pow-ként. Térjünk vissza az előző ábra első sorára. Kékszínnel, dőlt betűvel szedve és aláhúzva ezt láthatjuk:

java.lang.Math

A java.lang azonosítók magyarázatától ismételten tekintsünk el, később részletesen meg fogjuk ismerni jelentését. Számunkra a *Math* azonosító a lényeges, miatta kell Math.pow-ként meghívnunk a függvényünket.

A függvény nevét követően, zárójelek között található a függvény paraméterlistája. Itt találhatóak felsorolva a bemenő adatok (paraméterek), a típusukkal együtt, mint ha változókat definiáltak volna. A paramétereket vesszővel kell elválasztani egymástól a paraméterlistában. A paramétereket többször argumentumként is említhetik. A két szakkifejezés között minimális jelentésbeli különbség van, amit egy későbbi alfejezetben részletesen ismertetni fogunk. Olvassuk el a 3. sort az előbbi ábrán, amely magyarul körülbelül annyit tesz:

1 Igazából ez csak egy osztályon belül érvényes. A következő félévben fogunk kitérni rá részletesen.

„Visszaadja az első argumentum, második argumentumra emelt értékét.”

Az első argumentum a, a második b, tehát a, béedik hatványát adja vissza. A következő sorokban fel vannak sorolva speciális esetek, amik többnyire a hatványozás szabályaival egyeznek meg, kivéve az NaN rövidítéseket tartalmazó sorok. A rövidítés jelentése Not A Number, azaz valamilyen olyan érték, amely nem értelmezhető számként. (A Double.NaN azonosító segítségével helyettesíthetjük be paraméterként.) Léteznek paraméter nélküli függvények, például a System.out.println(), amikor nem adunk meg semmit a zárójelek között, egy paraméter nélküli változatát hívja meg. A pontos igazság az, hogy létrehozhatunk azonos nevű függvényeket, eltérő paraméterlistában. Tekintsük meg az összes println függvényt:

```
14      public static void main(String[] args) {
15
16          System.out.println
17      }
18  }
19
```

```
println()              void
println(Object x)      void
println(String x)      void
println(boolean x)     void
println(char x)        void
println(char[] x)      void
println(double x)      void
println(float x)       void
println(int x)         void
println(long x)        void
```

Látható, hogy pontosan 10 db formája van. A Java fordító onnan fogja tudni, hogy melyik függvényt hajtsa végre, hogy milyen típusú paraméterrel hívtuk meg:

– Ha üresen: akkor az 1.

– Ha egy Stringgel: akkor a 3.

– Ha egy integerrel: akkor az utolsó előttit, stb.

Ezt a módszert függvény túlterhelésnek (function overloading) nevezzük.

3. Függvényhívás

Függvényhívásnak nevezzük azt a folyamatot, amikor a függvényhívás utasításra ér a Java fordító és „átlép” a függvény kódjának végrehajtására, majd amikor végrehajtotta azt, az eredménnyel visszatér a függvényhívás helyére. Ezt követően folytatja a programunk „hagyományos” végrehajtását. Az e heti leckében, a későbbiek során részletesebben meg fogjuk ismerni a függvényhívás folyamatát.

A függvények meghívásakor, pontosan a paraméterlistájukban felsorolt típusú, és számú paraméterrel kell meghívnunk, mégpedig a megfelelő sorrendben. Tehát a Math.pow() függvényt nem hívhatjuk meg egy double értékkel, vagy hárommal, esetleg két Stringgel. Azonban mi két integerrel hívtuk meg, hogyan működhetett? Emlékezzünk vissza arra, amit a típuskonverzió során tanultunk: egy double helyén átadhatunk egy integer változót/értéket, mivel a double egész részén képes eltárolni 4 bájtot, az integer pedig pontosan 4 bájtnyi egészérték, így kompatibilisek! Ezért készítették el double típusú paraméterekkel a függvényt, mert így integer, float, short, stb. típusú paraméterekkel is működőképes.

Természetesen számít az is, hogy melyik paraméter helyére, milyen értéket helyezünk el. Ha a Math.pow(3, 2) utasítást adjuk ki, az eredmény nem 8.0 lesz, mint a Math.pow(2, 3) esetében, hanem 9.0, hiszen a hármat emeljük a második hatványára. Emellett nagyban befolyásolja a függvényhívás módját a visszatérési típus is.

3.1 Visszatérési típus

A visszatérési típus határozza meg, hogy milyen adattípusú eredményt fogunk visszakapni a függvényhívás eredményeként. A függvényhívás egy művelet, amelynek eredménye van, hasonlóan, mint egy aritmetikai, vagy logikai műveletnek. A visszatérési típus bármilyen típus lehet: primitívek, String, tömbök, stb. Valamint lehet void (azaz üres). Az utóbbi esetben a függvény nem tér vissza semmilyen értékkel, csak a bemenő paraméterek alapján elvégez egy műveletet, pl.:

System.out.printf("%d", 12);

Amit fontos megérteni a függvényhívásról, hogy a hívás helyén, a függvény lefutását követően, a visszatérési érték fog szerepelni. Tehát hasonlóan a literálokhoz, változókhoz, konstansokhoz, aritmetikai- és logikai műveletekhez: a függvényhívás eredménye is egy érték, így ő is értéket hordoz. Ugyanúgy használhatjuk, mint az előbb felsorolt nyelvi elemeket:

```
double hatvany = Math.pow(2, 3);
System.out.println(hatvany);                    // 8.0
System.out.println(Math.pow(2, 9));             // 512
System.out.println((int)Math.pow(2, 10));       // 1024
int kettoAHatodikon = (int)Math.pow(2, 7) / 2;
System.out.println(kettoAHatodikon);            // 64
```

Egy összetett kifejezésben, amely tartalmaz függvényhívást is, ha a függvényhívás eredménye egy művelet operandusa, akkor a művelet elvégzése előtt végrehajtásra kerül a függvényhívás, így a helyén már a kikalkulált visszatérési érték fog állni (utolsó példa).

A visszatérési értéket nem kötelező felhasználni. Semmilyen hibát nem okoz, ha csak úgy lóg a levegőben egy függvényhívás visszatérési értéke:

```
Math.pow(2, 3);
```

Egyszerűen csak feleslegesen végeztük el a hatványozás műveletét, mert az eredményt kidobtuk a szemétbe.

4. Függvények feladata

A függvények a programozásban a kód strukturálására (struktúra: ~szerkezet, strukturálás: szerkezetének kialakítása) hivatottak. Feladatuk, hogy az összetartozó kódrészeket egy helyre gyűjtsék, valamint egyesítsék.

Az egyesített kódrészletek viselkedése nem (vagy csak kis mértékben) módosítható, általában csak a bemenő adatokat változtathatjuk meg. A függvény szó onnan ered, hogy a kódrészlet lefutásának eredménye (kimenő adata) a bemenő adatoktól függ.

A függvények használata csökkenti a redundanciát (a felesleges, és káros ismétlődést). Képzeljük el, mennyivel bonyolultabb lenne a másodfokú függvény megoldóképletének alkalmazása a Javában, ha a hatványokat nekünk kellene minden egyes esetben ciklusokkal kiszámítani? Valamint a ciklusok csak double hatvany = Math.pow(2, 3); System.out.println(hatvany); // 8.0 System.out.println(Math.pow(2, 9)); // 512 System.out.println((int)Math.pow(2, 10)); // 1024 int kettoAHatodikon = (int)Math.pow(2, 7) / 2; System.out.println(kettoAHatodikon); // 64 az alap és a kitevő értékében térnének el egymástól, így ordít, hogy egy függvényt kellene használni, amelyik paraméterként az alapot és a kitevőt várja.

Emellett a függvények segítségével elkülöníthetjük egymástól az egymáshoz nem tartozó kódrészleteket. Az e heti leckében erről bővebben fogunk beszélni.

5. Függvények típusai

A függvények típusai a strukturált programozási paradigmában a függvény és az eljárás. A függvény visszatérési típussal rendelkezik, az eljárás csupán void (azaz üres) visszatérési típusú: tehát nem ad vissza semmilyen értéket. A függvény bemenő paraméterek alapján ad vissza egy számítás/művelet eredményét, míg az eljárás „csak” egy adott kódrészt futtat le. Pl.:

System.out.prinln("Kiír");

6. Metódusok

Mint ahogy már említettük, a Java az objektumorientált paradigmát követő nyelv. Azonban mi ennek a tárgynak keretein belül főként strukturált programozásra használjuk. A függvény és eljárás a strukturált programozási paradigma szakkifejezései. Azért használjuk most őket, mert több helyen találkozhattok még velük (pl.: adatbáziskezelés tárolt eljárások és függvények, nem objektumorientált nyelvek: C, stb). Azonban ha egy szakkönyvet-, vagy elektronikus forrást olvastok, és a metódus, vagy tagfüggvény szakkifejezéssel találkoztok, akkor azok lényegében a függvény/eljárás szinonimái az objektumorientált környezetben.

```java
package functions;

/**
 *
 * @author somlyaip
 */
public class StringFunctions {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        // charAt
        String input = "abcd";
        System.out.println(input.charAt(1));
        System.out.println(input.charAt(3));

        // startSwith
        String comment = "Közlemény: NK-KRTZ98 Gipsz Jakab";
        if(comment.startsWith("NK-"))
            System.out.println("Valid transaction comment.");
        else
            System.out.println("Invalid transaction comment.");

        // trim
        String text = "  z ";
        System.out.printf("[%s]\n", text.trim());

        // toLowerCase
        String answer = " Y  ";
        if(answer.trim().toLowerCase().equals("y"))
            System.out.println("Yes");
        else
            System.out.println("No");

        // contains
        String username = "user%";
        if(username.contains("%"))
            System.out.println("Username sould not contains %.");
        else
            System.out.println("Username is valid.");

        // valueOf
        String numberAsText = String.valueOf(15);
        System.out.println(numberAsText);
    }

}
```

```
 1. package functions;
 2.
 3. /**
 4.  * {Description}
 5.  *
 6.  * @author somlyaip
 7.  */
 8. public class MathFunctions {
 9.
10.     public static void main(String[] args) {
11.
12.         System.out.println("Abs: " + Math.abs(-90));
13.
14.         System.out.println("round(4.999): " + Math.round(4.999));
15.         System.out.println("round(14.51): " + Math.round(14.51));
16.         System.out.println("round(14.50): " + Math.round(14.50));
17.         System.out.println("round(46.18): " + Math.round(46.18));
18.
19.         System.out.println("2^6: " + Math.pow(2, 6));
20.
21.         System.out.println("4 square root: " + Math.sqrt(4));
22.         System.out.println("8 cube root: " + Math.cbrt(8));
23.     }
24. }
```

Bevezetés a programozásba

Függvénykészítés

Somlyai Pál

Dunaújvárosi Főiskola

2015

Bevezetés a programozásba - Függvénykészítés

Eljött az ideje, hogy saját függvényeket készítsünk az ismétlődő feladatok egyszerűsítésére. Nemrég megismertük a függvények szerkezetét, pontosabban a szignatúrájuk szerkezetét. Folytassuk tovább a felépítésük megismerését. Már mindannyian készítettünk saját függvényeket, nem is csak egy alkalommal. Minden programunk tartalmaz egy main függvényt. Tekintsük át az első programunkon keresztül:

..

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

Tekintsük át a legfontosabb szabályokat:

– Minden függvénynek egy osztályon belül kell elhelyezkednie.

– Egy függvényen belül nem hozhatunk létre még egy függvényt.

– Egy osztályban nem lehet két azonos szignatúrájú függvény.
  o Lényegében csak a név és a paraméterlista számít a szignatúrából.

– Minden függvénynek kötelezően meg kell adni a visszatérési típusát, a nevét, valamint a paraméterlistáját.

– A paraméterek ugyan olyan módon használhatóak a függvényen belül, mint a lokális változók.

– A függvényeken belül készített lokális változókhoz csak az adott függvénytörzsön belül férünk hozzá!

– Ha a visszatérési típus nem void, kötelező return utasítást használni.

– A függvény visszatérési értéke minden esetben megfeleltethetőnek kell lennie a visszatérési típusával.
o Megfeleltethető: implicit típuskonverzióval át kell tudnia alakítani a fordítónak a visszatérési értéket, a visszatérési típusra.
– double visszatérési típus, integer visszatérési érték

– Egy függvény utolsó végrehajtott utasítása minden esetben egy return utasítás, kivéve, ha a visszatérési típusa void.
o Ebből következik, hogy a return utasítást követő utasítások nem futnak le.

A függvény – hasonlóan a ciklusokhoz – két részből épül fel: a függvényfejből és a függvénytörzsből. A függvényfej maga a szignatúra, a törzs pedig a szignatúrát követő utasításblokk. Az előző példában csak egy utasítást, a „Hello World!” kiírását tartalmazta a függvényünk törzse, valamint nincs visszatérési érték, mert a visszatérési típus void, azaz egy eljárásról van szó. Vizsgáljunk meg egy egyszerű függvényt:

```
public static int osszead(int a, int b) {
    return a + b;
}
```

Az érdemi különbség az előző példához képest az, hogy a visszatérési típus integer. Ebből következik, hogy a függvénytörzsben szerepelnie kell visszatérési értéknek. A visszatérési érték mindig egy kifejezés, amely a return (visszatérés) utasítást követően áll. Mivel kifejezés, ezért lehet literál, változó, konstans, művelet eredménye, esetleg függvényhívás visszatérési értéke. A visszatérési érték lesz az az érték, ami a függvényhívást követően a függvény „helyére kerül”. public class Hello { public static void main(String[] args) { System.out.println("Hello World!"); } } public static int osszead(int a, int b) { return a + b; }

A retrun utasítás után írt parancsok nem kerülnek végrehajtásra, mert a return parancs azt jelenti, hogy térj vissza a függvényhívás helyére, ezzel az értékkel. Azonban az érték elhagyható. Az üres return egy eljárásban, azaz void visszatérési típusú függvényben szerepelhet. Ott arra használhatjuk, hogy adott feltétel teljesülése esetén, szakítsuk meg az eljárás futását, mivel az utána álló utasítások nem kerülnek végrehajtásra. Ha a main függvényben adunk ki egy return utasítást, akkor az a programunk leállítását (futásának befejezését) fogja jelenteni. Lényegében a programunk a main függvényének meghívásával indul el, és a main függvény lefutásával zárul.

Jelenleg a függvényeket a main függvény alá vagy fölé, ugyan abban az osztályban kell elkészítenünk:

```
public class FuggvenyKeszites {

    public static int osszead(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        System.out.println(osszead(12, 5)); // 17
        System.out.println(eloszt(21, 10)); // 2.1
    }

    public static double eloszt(int osztando, int oszto) {
        return osztando / (double) oszto;
    }
}
```

Ha figyelmesen megnézzük a main függvényünket, láthatjuk, hogy rendelkezik egy String tömb típusú, args nevű paraméterrel. Az args az arguments, azaz argumentumok rövidítése. A programjaink parancssorban fogadhatnak argumentumokat (később), melyek az args tömbben lesznek elérhetőek a program futása során.

Mi a pontos különbség az argumentum és a paraméter jelentése között? A paramétert a függvényen belül használjuk, tehát az osszead függvény belül összeadja a és b paraméterét. Lényegében a paraméter az a speciális lokális változó, amely a paraméterlistában helyezkedik el. Ezzel szemben az argumentum az az érték, amit behelyettesítünk az adott paraméterbe. Lényegében egy paraméterekkel rendelkező függvényhívást tekinthetünk úgy, mintha a hívás legelején inicializálnánk a paraméterek értékeit az argumentumokkal.

De mi a legfontosabb érv a függvények használata mellett? Tegyük fel, hogy az előző példában az eloszt függvényben nem típuskényszerítettem az osztót, és a függvényt legalább 40 alkalommal meghívtam a programom különböző helyein. Amint észreveszem a problémát, módosítom a kódot a példában találhatóra, és az újrafordítást követően mindenhol a helyes eredményt fogja visszaadni a függvényünk. Tehát a függvények használata roppant praktikus, hiszen ha valamit módosítani kell az algoritmusunkban, akkor, ha függvényként használjuk az algoritmust megvalósító kódot, elegendő csak egy helyen módosítanunk az utasításainkat.

```
1.  package functions;
2.
3.  /**
4.   * {Description}
5.   *
6.   * @author somlyaip
7.   */
8.  public class SumIntegers {
9.
10.     public static void main(String[] args) {
11.
12.         int sum = sum(4, 6);
13.         System.out.println("Sum (4, 6): " + sum);
14.     }
15.
16.     public static int sum(int a, int b) {
17.         return a + b;
18.     }
19. }
```

```java
package functions;

/**
 * {Description}
 *
 * @author somlyaip
 */
public class PrintArray {

    public static void main(String[] args) {
        int[] vector = { 56, 79, 12 };
        printArray(vector);
    }

    public static void printArray(int[] array) {
        for(int i = 0; i < array.length; i++) {
            System.out.print(array[i] + ", ");
        }
        System.out.println();
    }
}
```

```java
package functions;

/**
 * {Description}
 *
 * @author somlyaip
 */
public class SumArray {

    public static void main(String[] args) {

        float[] vector = {12.5f, 45.6f, 8.f};
        System.out.println("Sum: " + sumArray(vector));
    }

    public static float sumArray(float[] array) {
        float sum = 0f;
        for(int i = 0; i < array.length; i++) {
            sum += array[i];
        }
        return sum;
    }
}
```

Bevezetés a programozásba

Egymásba ágyazott függvényhívás

Somlyai Pál

Dunaújvárosi Főiskola

2015

Tartalom

Bevezetés a programozásba – Egymásba ágyazott függvényhívás

1. Egymásba ágyazott függvényhívás

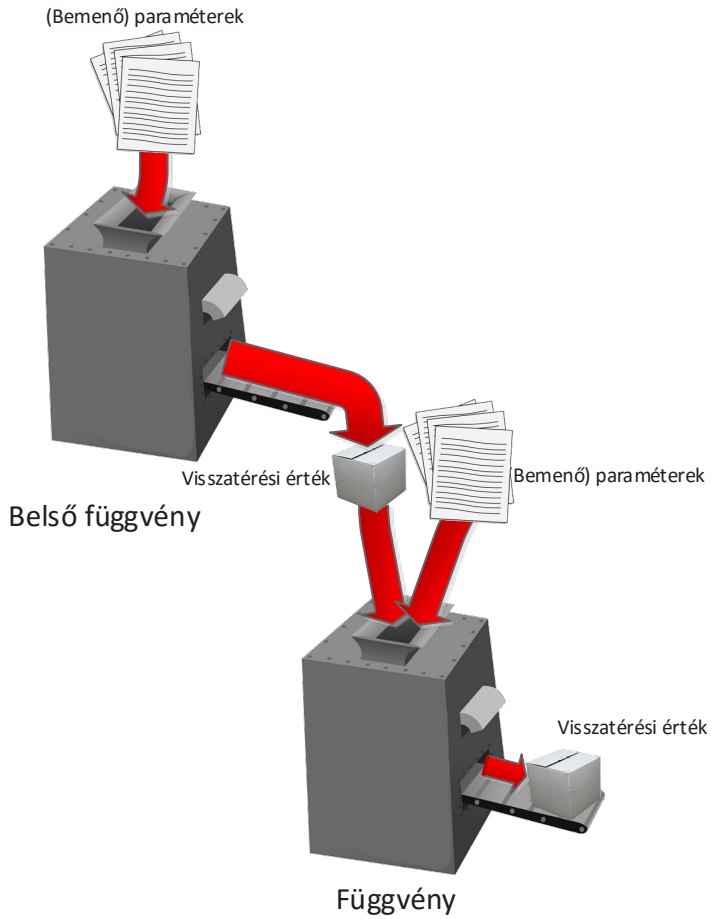
Számtalanszor előfordul, hogy egy függvény paramétereként egy másik függvényhívás visszatérési értékét szeretnénk használni. A következő példában az eloszt() függvényt az osszead(10, 11) függvényhívással, és a 10 literállal szeretnénk meghívni.

```
public class FuggvenyKeszites {

    public static void main(String[] args) {
        System.out.println(eloszt(osszead(10, 11), 10)); // 2.1
    }

    public static double eloszt(int osztando, int oszto) {
        return osztando / (double) oszto;
    }

    public static int osszead(int a, int b) {
        return a + b;
    }
}
```

Tehát az osztandó az osszead() függvényhívás visszatérési értéke lesz, az oszto pedig a 10. Mi történik ilyenkor a háttérben. Természetesen először az osszead() függvényhívás kerül kiértékelésre (végrehajtásra), majd a visszatérési értéke, a 21 kerül átadásra az eloszt() függvény számára. Ha részletesen szeretnénk megérteni a működésüket, debugolás során a függvényhíváson állva nyomjunk F7-et, vagy adjuk ki a Step Into parancsot, amivel belépünk a függvényhívásba.

Tekintsük meg az egymásba ágyazott függvényhívást ábrázoló ábrát, ami a korábbi csomagológép hasonlaton alapul.

A belső függvény az osszead(), a külső az eloszt():



A függvényhívások kiértékelésének sorrendjét a legjobban a hívási verem működésének megismerése segíthet megérteni.

2. Hívási verem

A hívási verem (call stack), egy verem[1], amelyben a meghívott függvények kerülnek be. Egy függvény akkor kerül bele, ha meghívják, és akkor kerül ki belőle, ha véget ért a futása. Tekintsük meg a következő példát:

```
1. public class HivasiVerem {
2.
3.     public static void main(String[] args) {
4.         String eredmeny = ba();
5.         System.out.println(eredmeny);
6.     }
7.
8.     public static String ba() {
9.         return "b" + a();
10.    }
11.
12.    public static String a() {
13.        return "a";
14.    }
15. }
```

Látható, hogy a main() függvényben meghívjuk a ba() függvényt, amelyik meghívja az a() függvényt. Lássuk hogyan alakul eközben a hívási verem. (A Netbeansben debugolás során, a Debugging ablakban, ami bal oldalon a projektfájlok helyére kerül debugolás során, követhetjük a hívási verem tartalmát.)

A hívási verembe a program indításával kerül be az első függvény, ami természetesen a main függvény.

1 A verem egy adatszerkezet. Képzeljük el úgy, mint egymásra helyezett ládákat. Csak a ládakupac tetejére tudunk egy új ládát helyezni, illetve csak onnan tudunk levenni egy ládát.
A verem is így működik, ha hozzá szeretnénk adni egy új elemet, akkor az a tetejére kerül, ha ki szeretnénk venni belőle egyet, akkor csak a legfelsőt vehetjük ki.

Ekkor a hívási verem tartalma:

main()

Majd a 4. sorban a main függvény meghívja a ba() függvényt, ami ennek hatására bekerül a hívási verembe:

ba()
main()

Mivel a ba() függvény bekerült a hívási verembe, azért elkezdi a fordító a ba() függvény végrehajtását. Gyakorlatilag a Java futtatókörnyzet mindig a hívási verem tetején található függvény végrehajtását végzi. A ba() függvény első sorában, a 9. sorban meghívásra kerül az a() függvény:

a()
ba()
main()

Mivel az a() függvény került a hívási verem tetejére, ezért az ő kiértékelésével folytatja a JVM. Lefuttatja, így az a() függvény kikerül a veremből, és a fordító átadja a ba() függvény számára a visszatérési értékét, amely az a() függvény hívásának helyétől folytatja az utasítások végrehajtását:

ba()
main()

Majd lefut a ba() függvény is, kikerül a veremből, és megkapja a main() függvény a visszatérési értékét, amelyet eltárol az eredmény lokális változóban:

main()

Ezt követően meghívásra kerül a System.out.println() függvény, tehát bekerül a verem tetejére:

println()
main()

A valóságban a println() függvény hívásakor több függvényhívás is megtörténik a println() függvényben, de ettől most tekintsünk el. A kiírás végeztével a println() függvény kikerül a veremből:

Majd a 6. sorban a main() függvény is véget ér, kikerül a hívási veremből: így a verem kiürül, és véget ér a programunk végrehajtása.

```
 1. package functions;
 2.
 3. /**
 4.  * {Description}
 5.  *
 6.  * @author somlyaip
 7.  */
 8. public class ComplexFunctionCall {
 9.
10.     public static void main(String[] args) {
11.
12.         System.out.println("Sum: " + sum(4, sum(5, 9)));
13.     }
14.
15.     public static int sum(int a, int b) {
16.         return a + b;
17.     }
18. }
```

Bevezetés a programozásba
Gyakorló feladatok

7. ALKALOM

1. Biztosan észrevették, hogy a Neptun rendszer csupa nagybetűs neptunkóddal, csupa kisbetűssel, illetve felváltva kis- és nagybetűsként megadva is beengedi Önöket (kis- és nagybetűk különbségét figyelmen kívül hagyó módon – azaz case insensitive módon – hasonlít). Mi történhet a háttérben? Készítsen egy programot, amelyben egy változóban eltárolja a neptunkódját (tetszőlegesen csupa kis-, vagy nagybetűs formában), majd beolvas egy neptun kódot, és meghatározza, hogy case insensitive összehasonlítással megegyezik-e az eltárolt neptunkóddal!

2. Készítsen programot, amelyik meghatározza, hogy egy beolvasott szöveg első és utolsó karaktere „a”-e? A program működjön akkor is helyesen, ha a felhasználó véletlen a szó előtt, vagy után egy, esetleg több szóközt leütött!

3. Készítsen programot, amelyik meghatározza egy beolvasott szóról, hogy tükörszó-e? Egy szó akkor számít tükörszónak, ha a fordítottan leírva megegyezik az eredetivel, pl.:

– apa

– bab

– kerek

– radar

Egy String hosszát a length() függvényével kérdezhetjük le: String s = ”a”; s.length();

4. Készítsen programot, amelyik beolvas egy tört, majd egy egész számot, és megállapítja, hogy a tört szám kerekített értéke megegyezik-e az egész számmal!

5. Készítsen programot, amely egy derékszögű háromszög két befogóját (a és b) beolvassa a felhasználótól, majd ennek ismeretében kiszámítja a derékszögű háromszög területét és kerületét!

– [**SPOILER VESZÉLY! – Ha nincs szüksége segítségre, ne olvassa tovább!**] A derékszögű háromszög területe (a * b) / 2. Kerülete a + b + c. Tehát szükségünk van az átfogó (c oldal) hosszára, amelyet a Pitagorasz-tételből meg tudunk határozni:

$$c = \sqrt{a^2 + b^2}$$

6. Készítsen két függvényt, amelyek kiszámolják egy derékszögű háromszög kerületét, valamint területét az a és b oldal ismeretében! A két függvény szignatúrája:

– double derekszoguHarmszogTerulete(double a, double b)

– double derekszoguHarmszogKerulete(double a, double b)

7. Készítsen függvényt, amely meghatározza, egy integer típusú tömb páros elemeinek számát!

8. Készítsen függvényeket, amelyek meghatározzák, egy double típusú tömb összegét és átlagát. Az átlagot kiszámító függvény törzsében az összeget kiszámító függvényt használja fel!

8. fejezet

Programozási tételek

Kiválasztás tétele

Dunaújvárosi Főiskola, Bevezetés a programozásba

A feladat megfogalmazása

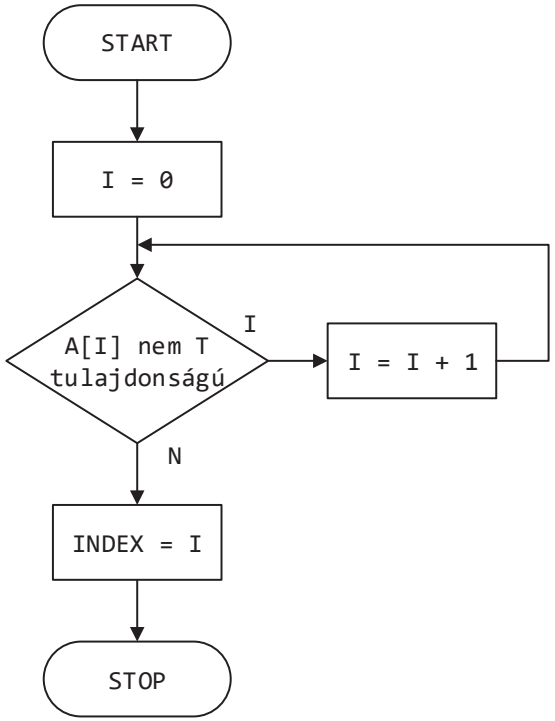
Adott egy N elemű sorozat, egy - a sorozat elemein értelmezett - T tulajdonság, és tudjuk, hogy a sorozatban van legalább egy T tulajdonságú elem. A feladat ezen elem indexének meghatározása. Amennyiben több ilyen elemet tartalmaz, akkor csak az első elem indexére van szükségünk. Tehát adott egy tetszőleges típusú elemeket tartalmazó vektor. Biztosak vagyunk benne, hogy tartalmaz legalább egy adott (T) tulajdonságú elemet. Ki kell választanunk a feltételnek megfelelő elem indexét. Amennyiben több T tulajdonságú elem is található a vektorban, akkor csak az első indexére van szükségünk.

Pszeudokód

```
Eljárás
      I = 0
      Ciklus amíg A[I] nem T tulajdonságú
            I = I + 1
      ciklus vége
      INDEX = I
eljárás vége
```

Az algoritmus arra alapoz, hogy elkezdi végigiterálni a vektor elemeit, egészen addig, ameddig el nem éri a T tulajdonságú (azaz a feltételnek megfelelő) elemet. Amint eléri: kilép, mivel a while ciklus feltételében az van megadva, hogy addig fusson, ameddig az aktuális elem nem felel meg a keresett elem feltételének. Miután kiléptünk a ciklusból, az I változó értéke fogja hordozni az első T tulajdonságú elem indexét. Amennyiben található több T tulajdonságú elem a vektorban, az sem okoz problémát, mert az algoritmus az első indexét fogja visszaadni.

Folyamatábra



Felhasználási területek

Bármilyen olyan esetben, ahol biztosak vagyunk, hogy a sorozatunk tartalmaz legalább egy adott tulajdonságú elemet és szükségünk van ennek az elemnek az indexére.

Például:

– Egy oktató pár hete zárthelyi dolgozatot íratott. Biztos abban, hogy az eredmények között volt egy maximális pontot elérő dolgozat. Határozza meg ennek a dolgozatnak a sorszámát a zh eredmények ismeretében!

```
 1. package maxscoresequencenumber;
 2. 
 3. /**
 4.  *
 5.  * @author somlyaip
 6.  */
 7. public class MaxScoreSequenceNumber {
 8. 
 9.     /**
10.      * @param args the command line arguments
11.      */
12.     public static void main(String[] args) {
13. 
14.         int[] scores = {17, 5, 0, 20, 9};
15. 
16.         System.out.println(
17.             "Max score sequence number: " + (getMaxScoreIndex(scores) + 1)
18.         );
19.     }
20. 
21.     public static int getMaxScoreIndex(int[] scores) {
22.         int i = 0;
23. 
24.         while(scores[i] < 20) {
25.             i++;
26.         }
27. 
28.         return i;
29.     }
30. 
31. }
```

Programozási tételek

Eldöntés

Egy elemre vonatkozóan

Dunaújvárosi Főiskola, Bevezetés a programozásba

A feladat megfogalmazása

Adott egy N elemű sorozat és egy a sorozaton értelmezett T tulajdonság. Van-e a sorozatnak legalább egy T tulajdonságú eleme? Tehát adott egy tetszőleges típusú vektor. Határozzuk meg, hogy a vektorban szerepel-e legalább egy feltételnek megfelelő elem?

Pszeudokód

```
Eljárás
      I = 0
      Ciklus amíg I < N és A[I] nem T tulajdonságú
            I = I + 1
      ciklus vége
      VAN = I < N
eljárás vége
```

Az algoritmusban egy while ciklus segítségével addig iteráljuk a vektorunk elemeit, ameddig:

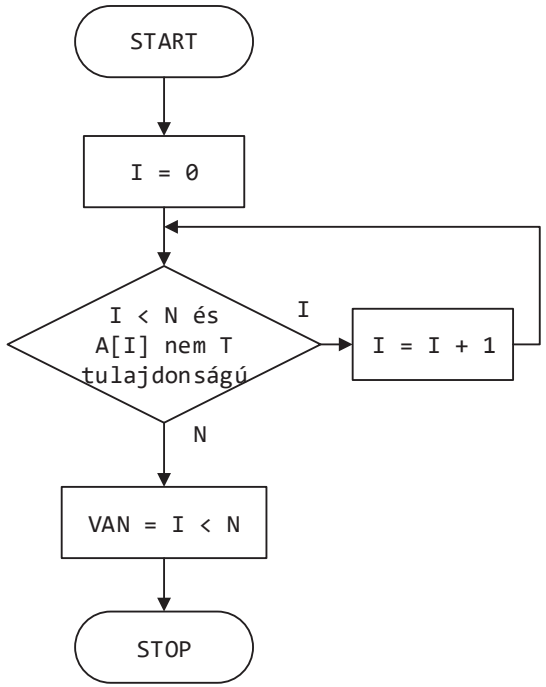
– az aktuális elem indexe kisebb, mint a vektor hossza (I < N), valamint egyszerre teljesül az is, hogy

– az aktuális vektorelem nem felel meg a keresési feltételnek (A[I] nem T tulajdonságú).

Amennyiben legalább az egyik feltétel nem teljesül, kilépünk a ciklusból. Ha az első feltétel nem teljesült, akkor végigiteráltuk a vektor összes elemét és egyik sem felelt meg a keresési feltételnek. Csak akkor lesz hamis a ciklus futási feltételének első része, ha I = N. Ezt követően rögtön kilépünk a ciklusból, tehát, ha nem találtunk benne a feltételnek megfelelő elemet, akkor az I < N feltétel hamis lesz, így a VAN változó értéke false lesz. Amennyiben nem értünk végig a vektor összes elemén, azonban találtunk a feltételnek megfelelő elemet (tehát nem teljesült az A[I] nem T tulajdonságú feltétel), akkor is kilépünk a ciklusból, ám I értéke kisebb lesz, mint a vektor elemszáma. Tehát ebben az esetben a VAN változó értéke true lesz.

Egy másik oldalról megközelítve: honnan tudhatjuk bizonyosan, hogy mi miatt léptünk ki a ciklusból? Roppant egyszerű: ha az összetett ÉS feltétel egyik tagját levizsgáljuk, és az értéke igaz: akkor a másik tagnak kellett hamis eredményt adnia, hiszen kiléptünk a ciklusból, azaz az ÉS operátor operandusainak egyike hamis eredményt hordozott. Tehát jelen esetben, ha I < N igaz, akkor az A[I] nem T tulajdonságú kifejezés értéke volt hamis, tehát negálva: A[I] T tulajdonságú, azaz az utoljára levizsgált elem megfelelt a keresési feltételnek, tehát a vektor tartalmaz legalább egy, a kereséséi feltételnek megfelelő elemet.

Folyamatábra



Felhasználási területek

Bármilyen esetben, amikor azt kell megállapítani, hogy egy vektorban található-e egy adott tulajdonságú elme.

Például:

– Egy bejelentkezési folyamat egy részét kell megvalósítania. Adott a regisztrált felhasználónevek listája, vizsgálja meg, hogy a megadott felhasználónév regisztrált-e a rendszerben!

o Azaz el kell dönteni, hogy a felhasználónevek vektorban szerepel-e egy olyan elem, amelyik értéke megegyezik a felhasználó által megadottéval.

Programozási tételek

Eldöntés
Minden elemre vonatkozóan

Dunaújvárosi Főiskola, Bevezetés a programozásba

A feladat megfogalmazása

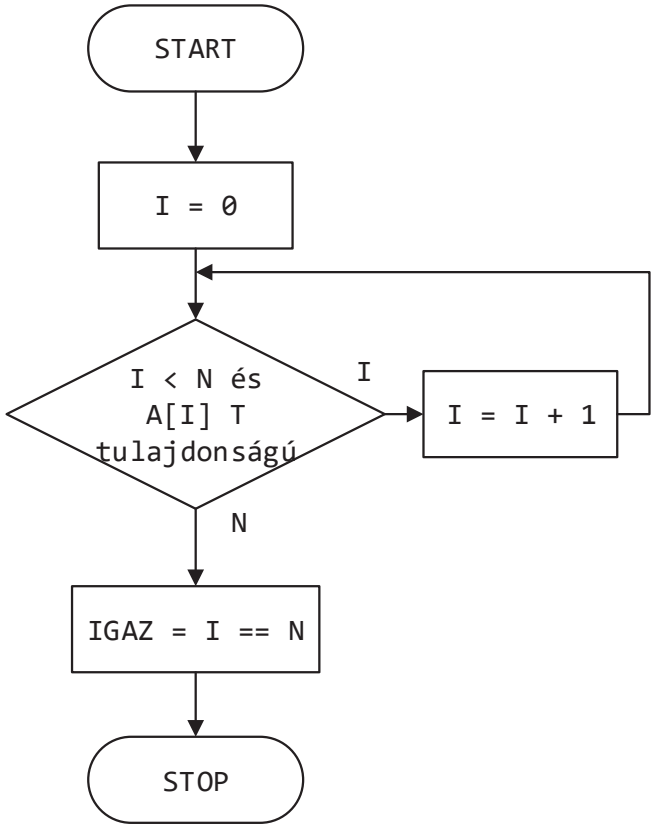
Adott egy N elemű sorozat és egy a sorozaton értelmezett T tulajdonság. Igaz-e, hogy a sorozat minden eleme T tulajdonságú? A feladat nagyban hasonlít az előző algoritmushoz, azonban jelenleg arra vagyunk kíváncsiak, hogy a vektor minden elemére igaz-e a megadott feltétel.

Pszeudokód

```
Eljárás
      I = 0
      Ciklus amíg I < N és A[I] T tulajdonságú
            I = I + 1
      ciklus vége
      IGAZ = I == N
eljárás vége
```

Az előző algoritmuson csak kis változtatásokat kell módosítanunk, hogy megkapjuk a jelenlegi feladat megoldására alkalmas algoritmust. Mivel azt vizsgáljuk, hogy minden elem megfelel-e a feltételnek, ezért addig kell iterálnunk a vektor elemeit, ameddig azokon teljesül a feltétel. Ha már egy elemen nem teljesül, akkor a vektor nem csak a feltételnek megfelelő elemeket tartalmaz. Emiatt módosítanunk kell a logikai vizsgálatot. Ha hamarabb léptünk ki, mint hogy végig értünk volna az elemeken, akkor nem minden elem T tulajdonságú. Azonban, ha I értéke egyenlő N-el, azaz végigiteráltuk az összes elemet, akkor a megadott feltétel az összes elemre igaz.

Folyamatábra



Felhasználási területek

Bármilyen olyan esetben, amikor azt szeretnénk megvizsgálni, hogy egy tetszőleges típusú vektor minden elemére igaz-e egy bizonyos feltétel.

Például:

- Adott egy vektorban a Dunaújvárosi 1. postára beérkező, Dunaújvárosban kézbesítendő levelek címzettének irányítószáma. Állapítsa meg, hogy minden irányítószám Dunaújvárost jelöli-e (2400)!
    - o Azaz végig kell iterálni a vektor elemeit és amennyiben tartalmaz legalább egy elemet, ami nem egyenlő a 2400-al, akkor legalább egy levél címzettének címe nem Dunaújvárosi cím.
    - o A valóságban ez a példa kicsit sántít, mivel léteznek kiemelt irányítószámok, amelyek részben eltérnek a város irányítószámától. Ilyenkor nem kell megadni a közterületet (utca házszám, stb.), mert az irányítószám azonosítja a pontos címet.
        - Pl.: az említett posta intézmény irányítószáma 2401
    - o Azonban ebben a feladatban tekintsünk el a kiemelt irányítószám használatától.

```java
package existsinarray;

/**
 *
 * @author somlyaip
 */
public class ExistsInArray {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        float[] numbers = {5.12f, 16.7f, 81.99f, 91.45f, .45f};

        float searchFor = 91.45f;
        System.out.println(
            "The array " +
            (
                existsInArray(numbers, searchFor)
                ? "contains"
                : "does not contain"
            ) +
            " " + searchFor
        );
    }

    public static boolean existsInArray(float[]numbers, float searchFor) {
        int i = 0;

        while(i < numbers.length && numbers[i] != searchFor) {
            i++;
        }

        return i < numbers.length;
    }
}
```

```
1.  package commentsvalidator;
2.
3.  /**
4.   *
5.   * @author somlyaip
6.   */
7.  public class CommentsValidator {
8.
9.      /**
10.      * @param args the command line arguments
11.      */
12.     public static void main(String[] args) {
13.
14.         String[] comments = {
15.             "NK-F172...",
16.             "Invalid comment",
17.             "NK-91TR..."
18.         };
19.
20.         if(areAllCommentsValid(comments))
21.             System.out.println("All comments are valid.");
22.         else
23.             System.out.println("At least one comment is invalid.");
24.     }
25.
26.     public static boolean areAllCommentsValid(String[] comments) {
27.         int i = 0;
28.         while(i < comments.length && comments[i].startsWith("NK-")) {
29.             i++;
30.         }
31.
32.         return i == comments.length;
33.     }
34. }
```

Programozási tételek

Lineáris keresés

Dunaújvárosi Főiskola, Bevezetés a programozásba

A lineáris keresés azt jelenti, hogy egy adott vektoron végigiterálva választjuk ki a feltételnek megfelelő elem indexét. Az elemek tetszőleges típusúak lehetnek és nem kell rendezettnek lennie a vektornak. Egy vektor akkor rendezett, ha elemei valamilyen rendezési elv alapján sorban helyezkednek el. Egy számokat tartalmazó vektort numerikusan (a számok értéke szerint) rendezünk általában növekvő sorrendbe. Egy String vektor elemeit pedig alfanumerikusan (azaz betűrendben) növekvő sorrendbe szokás rendezni.

A feladat megfogalmazása

Adott egy N elemű sorozat, valamint a sorozat elemein értelmezett T tulajdonság. Van-e T tulajdonságú elem és ha van, akkor mi az indexe. Lényegében a feladat kiírásából kihámozható, hogy itt az eldöntés és kiválasztás tételét kell együttesen alkalmaznunk. A „van-e T tulajdonságú elem” kérdésre az eldöntés-, a „mi az indexe” kérdésre pedig a kiválasztás segítségével adhatunk választ. Lényegében a lineáris keresés az előbb említett két tétel kombinációja. Természetesen, ha több T tulajdonságú elem található a vektorban, akkor csak az első indexére vagyunk kíváncsiak.

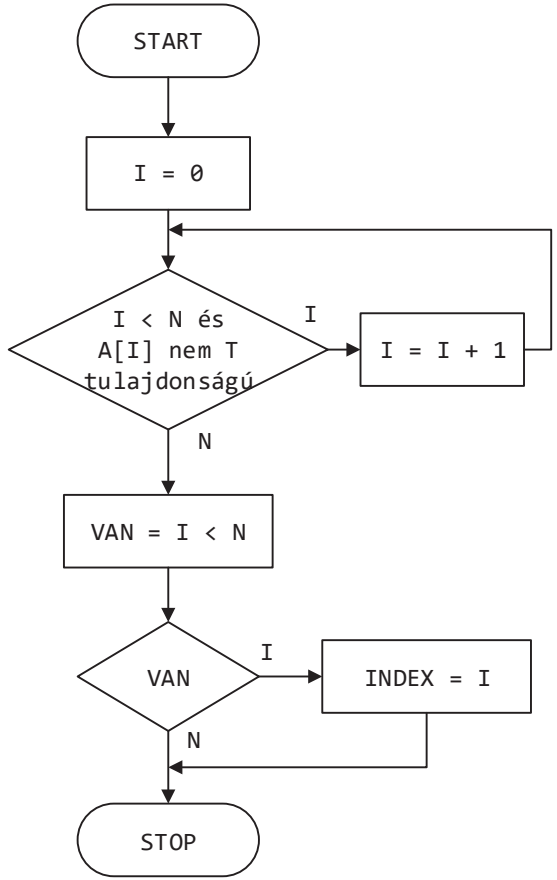
Pszeudokód

```
Eljárás
      I = 0
      Ciklus amíg I < N és A[I] nem T tulajdonságú
            I = I + 1
      ciklus vége
      VAN = I < N
      Ha VAN akkor
            INDEX = I
      ha vége
eljárás vége
```

Azonban a két tételt nem egymás után kell végrehajtanunk, hanem mivel a ciklus megegyezik, csak egy plusz elágazással kell kiegészítenünk.

Lényegében, miután kiléptünk a ciklusból, elvégezzük az eldöntés tételéből ismerős vizsgálatot, hogy van-e a feltételnek megfelelő elem? Amennyiben igen, akkor az I értéke hordozza a keresett elem indexét.

Folyamatábra



Felhasználási területek

Bármilyen feladatban, ahol nem vagyunk biztosak abban, hogy a vektorunk tartalmazza a keresett tulajdonságú elemet, valamint nem elég számunkra, hogy tartalmazza-e, hanem szükségünk van a feltételnek megfelelő elem indexére is.

Például:

– Adott egy cipőbolt polcán elhelyezkedő cipők méretei egy vektorban. A cipők csak a méretükben térnek el egymástól. Egy vásárló ebből a cipőből szeretne 46-os méretűt vásárolni. Van-e készleten megfelelő méretű, és ha igen, melyik a sorszáma a polcon?

```java
package linearsearch;

/**
 *
 * @author somlyaip
 */
public class LinearSearch {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int[] array = {35, 98, 41, 34, 67, 12};
        int searchfor = 34;

        for(int i = 0; i < array.length; i++) {
            System.out.printf("array[%d]: %d\n", i, array[i]);
        }
        System.out.println();

        int foundIndex = linearSearch(array, searchfor);
        if(foundIndex != -1)
            System.out.println("index: " + foundIndex);
        else
            System.out.println("The array doesn't contain this element: " +
                    searchfor);
    }

    /**
     *
     * @param array értékeket tartalmazó vektor
     * @param searchFor a keresett érték
     * @return Ha megtalálja: az indexe, ha nem: -1
     */
    public static int linearSearch(int[] array, int searchFor) {
        int i = 0;
        while(i < array.length && array[i] != searchFor) {
            i++;
        }

        if(i < array.length)
            return i;
        else
            return -1;
    }

    public static int linearSearchWithReturn(int[] array, int searchFor) {
        for(int i = 0; i < array.length; i++) {
            if(array[i] == searchFor)
                return i;
        }

        return -1;
    }
}
```

Programozási tételek
Logaritmikus keresés

Dunaújvárosi Főiskola, Bevezetés a programozásba

A logaritmikus (avagy bináris) keresés egy rendezett számsorozatban képes megtalálni egy adott érték indexét, mégpedig roppant hatékonyan, maximum log N + 1 lépés alatt (N kettes alapú logaritmusa + 1: a logaritmusból a logaritmikus keresés-, a kettes alapból pedig a bináris keresés név ered). Ez azt jelenti, hogy míg egy 100 elemű rendezett vektorban, ahol a 67. indexen helyezkedik el a keresett elem, akkor azt a lineáris keresés a ciklus 68. iterációjában fogja megtalálni (0-tól indexelünk), azonban a logaritmikus keresés ugyan ebben a tömbben maximum 8 lépés alatt meg fogja találni.

Az algoritmus alapja

Logaritmikus-, vagy bináris keresés mellett szokás még felezéses keresésnek nevezni. Ennek oka az, hogy a keresés során kiválasztjuk a rendezett sorozat középső elemét, majd megvizsgáljuk, hogy a keresett érték kisebb, vagy nagyobb, mint a kiválasztott indexen lévő érték. Ha kisebb, akkor a középső index alatti tartományt tekintjük keresési tartománynak (természetesen, ha nagyobb: akkor pedig a felette lévő tartományt), amelynek kiválasztjuk a középső indexét és ezt folytatjuk egészen addig, ameddig a tartomány egy elemre nem szűkül. Így mivel minden lépésben lefelezzük az előző tartományt, roppant hatékonyan fogjuk megtalálni a keresett elemet.

Az algoritmus mélyebb megértése végett olvassátok el a következő weboldalt (az algoritmus pszeudokódjának ismertetése előtt térjetek vissza a tananyaghoz):

http://www.tankonyvtar.hu/hu/tartalom/tamop425/0005_22_algoritmizalas_alapjai_scorm_05/534_a_binris_logaritmikus_keress.html

A feladat megfogalmazása

Általános feladat: N elemű rendezett sorozat; egy keresett elem (X). Szerepel-e a keresett elem a sorozatban és ha igen, akkor mi az index.

Pszeudokód

Kihasználjuk, hogy a sorozat rendezett, így el tudjuk dönteni, hogy a keresett elem az éppen vizsgált elemhez képest hol helyezkedik el (az alatta, vagy a felette lévő indextartományban). Az algoritmusban az AL az indextartomány alsó indexét-, az Fe a felső indexét jelöli. A KÖ a középső index, X pedig a keresett értéket jelöli. A többi jelölés az eddig használtakkal megegyező.

```
Eljárás
      AL = 0
      FE = N - 1
      Ciklus
            KÖ = INT((AL + FE) / 2)
            Ha A[KÖ] < X akkor
                  AL = KÖ + 1
            különben Ha A[KÖ] > X akkor
                  FE = K - 1
            ha vége
      amíg AL <= FE és A[KÖ] != X ciklus vége
      VAN = AL <= FE
      Ha VAN akkor
            INDEX = KÖ
      ha vége
eljárás vége
```

Az algoritmus kezedén az indextartomány alsó határértékét 0-ra, a felsőt az utolsó indexre állítjuk, így a teljes tartományban kereshetünk. Majd kiválasztjuk az indextartomány közepét, pontosabban: ha a tartomány alsó- és felső határértékének számtani közepe[1] egész szám, akkor ő kerül kiválasztásra, ha törtszám, akkor elhagyjuk a törtrészt az integer típuskényszerítéssel. A pszeudokódban általában INT(<érték>)-ként jelöljük az integer típuskényszerítést.

1 Számtani közép definíciója: Két nemnegatív szám számtani közepének a két szám összegének a felét nevezzük.

Ezt követően megvizsgáljuk, hogy a keresett érték a középértéktől milyen irányban helyezkedik el:

– Ha a felette lévő tartományban helyezkedik el (azaz a keresett érték nagyobb, mint a középső indexen található érték), akkor a tartomány alsó határértékét felülírjuk a középértéknél egyel magasabb indexxel. A felső határérték marad az előző.

– Amennyiben az alatta lévő tartományban található, úgy a felső határértéket a középértéknél eggyel alacsonyabb értékre állítjuk. Az alsó határérték marad a előző.

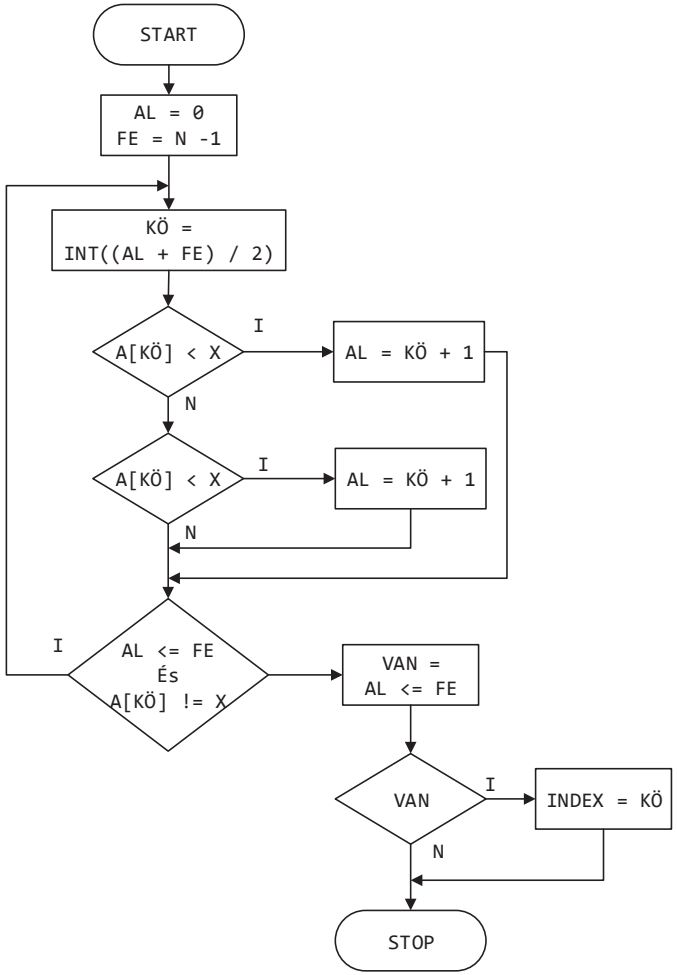
Így a keresési tartományt sikeresen lefeleztük. A do-while ciklus megvizsgálja, hogy az alsó határérték nem nagyobb-e mint a felső határérték (ha ez a kifejezés igaz, akkor a keresett érték nem található a tartományban), valamint azt is megvizsgálja, hogy a vektor középindexén elhelyezkedő eleme különbözik-e a keresett elemtől. Amennyiben mindkét feltételrész teljesül, ciklikusan folytatja a felezést.

Milyen léphetünk ki a ciklusból (mik a kilépési feltételek)?
– AL > FE: azaz, ha az alsóérték magasabb, mint a felső, akkor az utolsó tartományfelezés során helytelen tartományt alakítottunk ki, hiszen a tartomány alsó határértéke nem lehet magasabb, mint a felső (pl.: 6..5 ← helytelen tartomány)
    o Eredmény: a keresett elem nem található a vektorban. 1 Számtani közép definíciója: Két nemnegatív szám számtani közepének a két szám összegének a felét nevezzük.

– A[KÖ] == X: azaz a középérték megegyezik a keresett elemmel. Ebben az esetben nincs értelme tovább futtatni a ciklusunkat, mert megtaláltuk a keresett elemet.
    o Eredmény: a keresett elem indexe KÖ.

Honnan tudhatjuk bizonyosan, hogy melyik kilépési feltétel teljesült, azaz mi miatt léptünk ki a ciklusból? Roppant egyszerű: ha az összetett ÉS feltétel egyik tagját levizsgáljuk, és az értéke igaz: akkor a másik tagnak kellett hamis eredményt adnia, hiszen kiléptünk a ciklusból, azaz az ÉS operátor operandusainak egyike hamis eredményt hordozott.

Folyamatábra



Felhasználási területek

Bármilyen olyan keresési feladatot tartalmazó probléma esetén, ahol a keresést egy rendezett, számsorozaton kell elvégezni.

Hivatkozások

[1] T. Péter, „http://www.tankonyvtar.hu - Algoritmizálás alapjai," 2011. [Online]. Available: http://www.tankonyvtar.hu/hu/tartalom/tamop425/0005_22_algoritmizalas_alapjai_scorm_05/534_a_binris_logaritmikus_keress.html. [Hozzáférés dátuma: 3 10 2015].

```java
package logarithmicsearch;

/**
 *
 * @author somlyaip
 */
public class LogarithmicSearch {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int[] sortedNumbers = {1, 5, 9, 15, 29, 41, 65, 79, 82, 94};
        printArray(sortedNumbers);

        int searchFor = 150;
        int foundIndex;
        if((foundIndex = binarySearch(sortedNumbers, searchFor)) == -1)
            System.out.println("The array doesn't contains this element: " +
                    searchFor);
        else
            System.out.println("Index of " + searchFor + " is " + foundIndex);
    }

    public static void printArray(int[] array) {
        for(int i = 0; i < array.length; i++) {
            System.out.printf("array[%d]: %d\n", i, array[i]);
        }
        System.out.println();
    }

    public static int binarySearch(int[] sortedArray, int searchFor) {
        int indexBottom = 0;
        int indexTop = sortedArray.length - 1;
        int indexMiddle = -1;

        do {
            indexMiddle = (indexBottom + indexTop) / 2;
            if(sortedArray[indexMiddle] < searchFor)
                indexBottom = indexMiddle + 1;
            else if(sortedArray[indexMiddle] > searchFor)
                indexTop = indexMiddle - 1;
        } while(indexBottom <= indexTop
                && sortedArray[indexMiddle] != searchFor);

//        if(indexBottom <= indexTop)
//            return indexMiddle;
//        else
//            return -1;

        return indexBottom <= indexTop ? indexMiddle : -1;
    }
}
```

```java
package guessnumber;

/**
 *
 * @author somlyaip
 */
public class GuessNumber {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int boundaryBottom = 1;
        int boundaryTop = 100;
        int middle = -1;

        System.out.printf(
            "Please guess a number between %d and %d.\n",
             boundaryBottom, boundaryTop
        );

        do {
            middle = (boundaryBottom + boundaryTop) / 2;

            String answer;
            do {
                System.out.printf(
                    "Less, greater than or equal to %d [L/G/E]? ", middle
                );
                answer = System.console().readLine().trim().toUpperCase();
            } while(!answer.equals("L") && !answer.equals("G")
                    && !answer.equals("E"));

            if(answer.equals("E"))
                break;
            else if(answer.equals("L"))
                boundaryTop = middle - 1;
            else // answer = G
                boundaryBottom = middle + 1;
        } while(boundaryBottom <= boundaryTop);

        if(boundaryBottom > boundaryTop)
            System.out.println("Number not found. Please verify your answers.");
        else
            System.out.println("The number is " + middle);
    }

}
```

Bevezetés a programozásba

Gyakorló feladatok

8. ALKALOM

1. Adott egy tetszőleges egész számokat tartalmazó vektor. Válassza ki az első, héttel maradék nélkül osztható elem sorszámát! (A kiválasztás tétele miatt kötelező a vektornak tartalmaznia egy, az adott feltételnek megfelelő elemet, különben a program hibára fut.)

2. Adott egy integer vektorban, a pénztárcájában szereplő bankjegyek címletei. A fizetendő összeg 3 685 HUF. Válassza ki az első címlet indexét, amely segítségével egy címletben ki tudja fizetni az említett összeget!

3. Töltsön fel egy 10 elemű, törtszámokat tartalmazó vektort véletlengenerált számokkal 10,5..14,9 intervallumból! Határozza meg, hogy tartalmaz-e olyan elemet, amelyiket egész számra kerekítve 12-t kapunk eredményül!

4. Egy bejelentkezési folyamat egy részét kell megvalósítania. Adott a regisztrált felhasználónevek listája, vizsgálja meg, hogy a beolvasott felhasználónév regisztrált-e a rendszerben!

   – Azaz el kell dönteni, hogy a felhasználóneveket tároló vektorban szerepel-e egy olyan elem, amelyik értéke megegyezik a felhasználó által megadottéval.

5. Adott egy vektorban a Dunaújvárosi 1. postára beérkező, Dunaújvárosban kézbesítendő levelek címzettének irányítószáma. Állapítsa meg, hogy minden irányítószám Dunaújvárost jelöli-e (2400)!

6. Adott egy hajdani „100 forintos bolt” termék árainak listája. Állapítsa meg, hogy minden termék ára 100 HUF-e!

7. Töltsön fel egy 20 elemű vektort 1 és 5 közötti véletlengenerált számokkal, majd keresse meg az első 5-ös indexét!

8. Adott két String vektor. Az első országok neveit, a második a megegyező indexű ország fővárosát tartalmazza:

Index	Ország	Főváros
0	Románia	Bukarest
1	Magyarország	Budapest
2	Ausztria	Bécs
3	Németország	Berlin
4	Franciaország	Párizs

Készítsen programot, amelyik beolvas egy országnevet, majd az ország vektorban megkeresi az országnévhez tartozó indexet, amely segítségével megadja a fővárosát! Amennyiben olyan országot adott meg a felhasználó, amelyik nem szerepel a listában, kérje újra!

9. fejezet

Programozási tételek

Kiválogatás

Dunaújvárosi Főiskola, Bevezetés a programozásba

A feladat megfogalmazása

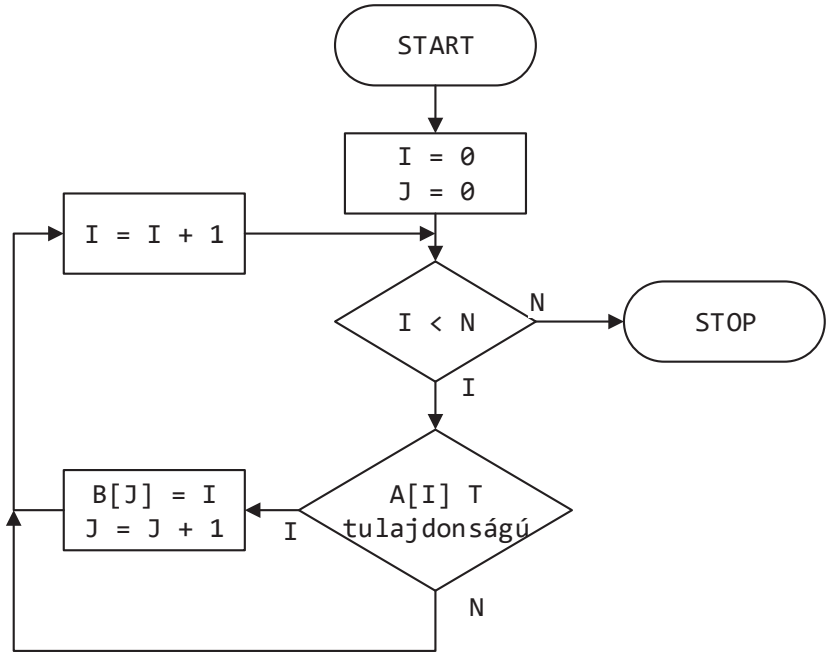
Határozza meg egy N elemű sorozat összes T tulajdonságú elemét! A kiválogatott elemek indexeit a B vektorban gyűjtse! Ez az első tétel, amelyben több vektort kell egyidejűleg használnunk. Az A vektor tartalmazza a forrásvektort, amelyből a keresési feltételnek megfelelő elemek indexeit át kell helyezni a B vektorba.

Pszeudokód

```
Eljárás
      J = 0
      Ciklus I = 0-től N-1-ig
            Ha A[I] T tulajdonságú, akkor
                  B[J] = I
                  J = J + 1
            ha vége
      ciklus vége
eljárás vége
```

Mivel a feltételnek megfelelő elemek indexét a B vektorban kell gyűjteni, ezért létre kell hoznunk egy indexet annak a vektornak is, amely a J a pszeudokódban. Az algoritmus eléggé egyszerű, végigiteráljuk az A vektor elemeit, és a feltételnek megfelelő elemek indexeit eltároljuk a B vektorban. Természetesen, miután eltároltunk egy elemet a B vektorban, az indexváltozóját inkrementálni kell, hogy ne mindig a nulladik indexen lévő elemet írjuk felül, hanem a következő szabad helyre helyezzük az eltárolandó indexet. J lényegében a következő szabad index értékét tárolja az algoritmusunkban.

Folyamatábra



Felhasználási területek

Bármelyik olyan feladatban felhasználható, ahol egy vektorból minden keresési feltételnek megfelelő elem indexére szükségünk van.

Például:

– A dunaújvárosi posta példa továbbfejlesztése: Gyűjtse ki a nem dunaújvárosi címek indexeit egy vektorba, hogy külön lehessen válogatni őket!

```java
package hightemperatures;

/**
 *
 * @author somlyaip
 */
public class HighTemperatures {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        float[] averageTemperatures = {
            23.4f, 25.6f, 28.5f, 31.2f, 32.1f, 31.5f, 29.9f
        };

        int[] highTempIndexes = new int[7];
        int j = 0;
        for(int i = 0; i < averageTemperatures.length; i++) {
            if(averageTemperatures[i] > 30) {
                highTempIndexes[j] = i;
                j++;

                // ekvivalens
                // highTempIndexes[j++] = i;
            }
        }

        System.out.println("High temeperatures:");
        for(int i = 0; i < j; i++) {
            int averageIndex = highTempIndexes[i];
            System.out.printf("%-5.1f", averageTemperatures[averageIndex]);
        }
        System.out.println();
    }

}
```

Programozási tételek

Metszetképzés

Dunaújvárosi Főiskola, Bevezetés a programozásba

A feladat megfogalmazása

Rendelkezésünkre áll egy N és egy M elemű halmaz az A és a B vektorban ábrázolva. Készítsük el a két halmaz metszetét a C vektorba! Tehát válogassuk ki a C vektorba azokat az elemeket, amelyek A-ban és B-ben is szerepelnek. A halmaz szó pedig arra utal, hogy külön-külön A és B vektorban az elemek egyediek, azaz az A vektoron belül nincs két megegyező értékű elem, ahogyan B vektorban sem.

Pszeudokód

```
Eljárás
      K = 0
      Ciklus I = 0-től N-1-ig
            J = 0
            Ciklus amíg J < M és A[I] != B[J]
                  J = J + 1
            ciklus vége
            Ha J < M akkor
                  C[K] = A[I]
                  K = K + 1
            ha vége
      ciklus vége
eljárás vége
```

Mivel három vektort kell kezelni az algoritmusban, ezért biztosan szükségünk lesz három ciklusváltozóra:

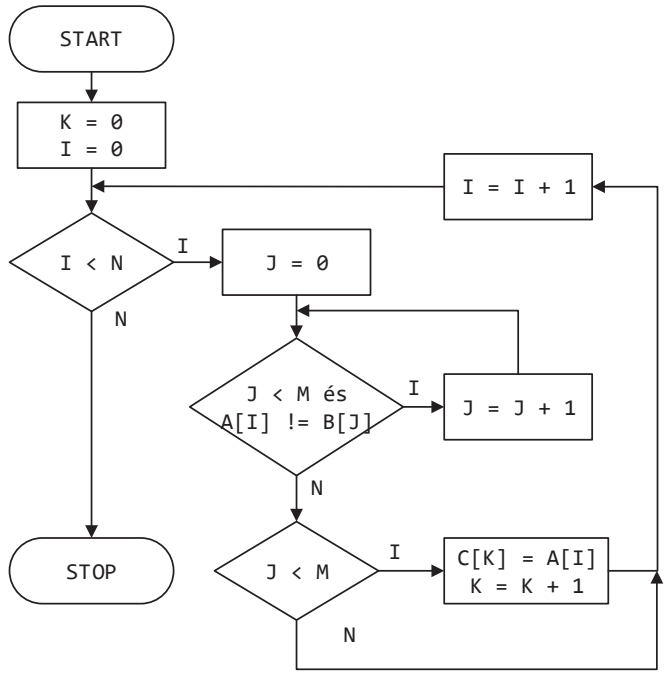
– I: Az A vektor ciklusváltozója

– J: A B vektor ciklusváltozója

– K: A C vektor ciklusváltozója

Két egymásba ágyazott ciklusra van szükségünk: a külső egy for ciklus, a belső pedig egy while. A külső ciklusban végigiteráljuk az A vektor elemeit, a belsőben pedig megvizsgáljuk, hogy található-e az A[I] elemmel megegyező értékű elem a B vektorban. Amennyiben igen: akkor hozzáadjuk az elemet a C vektorhoz. Lényegében a belső ciklus és utána a feltételvizsgálat a lineáris keresés tételének alkalmazása, kiegészítve: a mindkét vektorban megtalálható elem hozzáadásával a C vektorhoz, valamint a C vektor ciklusváltozójának (K) inkrementálásával.

Folyamatábra



Felhasználási területek

Bármilyen olyan feladatban, ahol két vektor közös elemeit kell kiválasztani egy harmadik vektorba.

Például:

– Egy középiskolai osztályban a diákoknak lehetőségükben áll választani, hogy angol vagy német nyelvet tanuljanak. Az osztály diákjainak OM azonosítóit tartalmazza egy angol és egy nemet lista, attól függően, hogy melyik nyelvet tanulja az iskolában a diák. Egy diák több nyelvet is tanulhat! Válassza ki, hogy milyen OM azonosítójú diákok tanulnak angol és német idegen nyelvet egyaránt!

– Egy beléptető rendszerrel védett épületben bűncselekmény történt. A bűncselekmény elkövetéséhez a bűnözőnek a 115-ös és a 119-es terembe is be kellett lépnie. Ismert a bűncselekmény elkövetésének napján a két terembe belépett személyek kártyáinak azonosítója két külön listában. (A belépéshez egy kártyaolvasóhoz kell érinteni a személy belépési kártyáját, amenynyiben beléphet a terembe, az ajtó kinyílik, ellenkező esetben elutasítja. A biztonsági rendszer pedig eltárolja, hogy mikor, milyen azonosítójú kártyának adott belépési engedélyt.) A nyomozás kezdetén a nyomozók elkérik a biztonsági szolgálattól, azoknak a kártyáknak az azonosítóit, amelyek segítségével mindkét terembe beléptek. A két lista ismeretében határozza meg a mindkét terembe beengedett kártyaazonosítókat!

– Adott egy középiskola végzős diákjainak OM azonosítói 2014-ben, valamint ismert egy felsőoktatási intézménybe felvételt nyert hallgatók OM azonosítói ugyan ebben az évben. Határozza meg, hogy milyen azonosítójú hallgatók nyertek felvételt az adott középiskolából, az adott felsőoktatási intézménybe!

```java
package commonsubjects;

/**
 *
 * @author somlyaip
 */
public class CommonSubjects {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        String[] subjectsMi = {
            "INF-002",
            "INF-501",
            "INF-420",
            "INF-200",
            "INF-210"
        };
        String[] subjectsGi = {
            "INF-002",
            "INF-610",
            "INF-501",
            "INF-420",
            "TKT-005"
        };

        String[] subjectsCommon = new String[5];
        int k = 0;
        for(int i = 0; i < subjectsMi.length; i++) {
            int j = 0;
            while(j < subjectsGi.length && !subjectsMi[i].equals(subjectsGi[j])) {
                j++;
            }
            if(j < subjectsGi.length) {
                subjectsCommon[k] = subjectsGi[j];
                k++;
            }
        }

        System.out.println("Common subjects: ");
        for(int i = 0; i < k; i++) {
            System.out.println("\t" + subjectsCommon[i]);
        }
        System.out.println();
    }

}
```

Programozási tételek

Unióképzés

Dunaújvárosi Főiskola, Bevezetés a programozásba

A feladat megfogalmazása

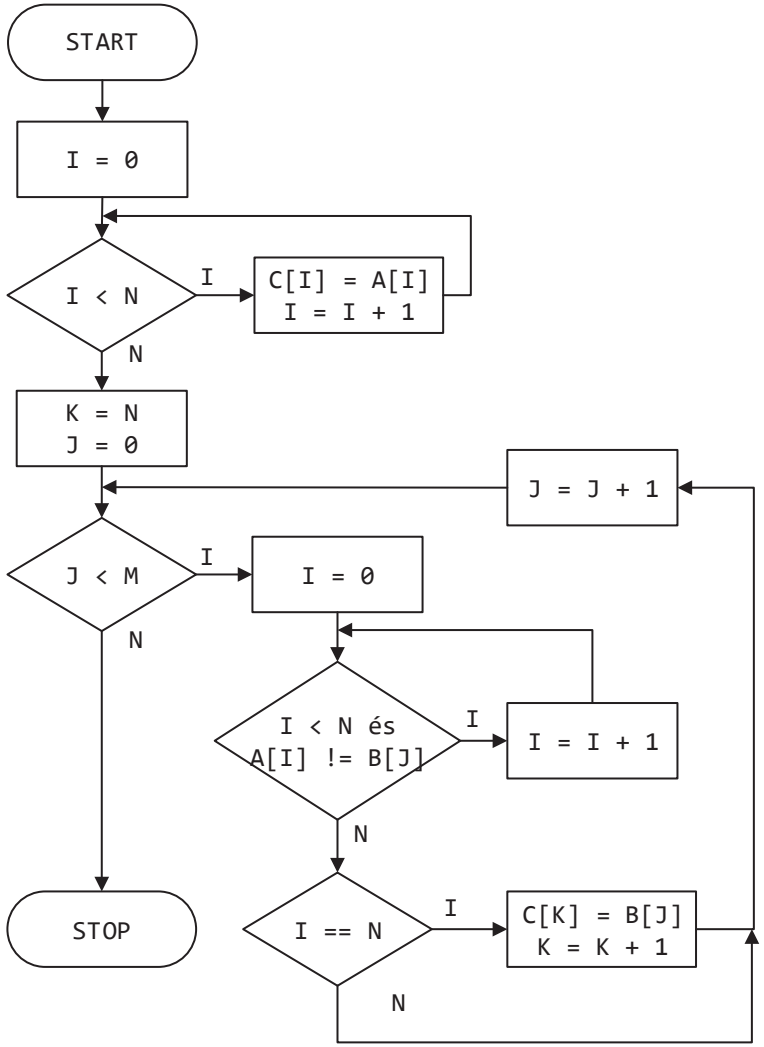
Rendelkezésünkre áll egy N és egy M elemű halmaz az A és a B vektorban ábrázolva. Készítsük el a két halmaz egyesítését (unióját) a C vektorba! Tehát a C vektorba helyezzük el az A és B vektor elemeit, de amennyiben egy elem A és B vektorban is szerepel, a C vektorba csak egyszer kerüljön be!

Pszeudokód

Első lépésben az A vektor minden elemét helyezzük el a C vektorban, majd B vektorból adjuk hozzá a C vektorhoz, azokat az elemeket, amelyek nem szerepelnek az A vektorban.

```
Eljárás
      Ciklus I = 0-tól N - 1-ig
            C[I] = A[I]
      ciklus vége
      K = N
      Ciklus J = 0-tól M – 1-ig
            I = 0
            Ciklus amíg I < N és A[I] != B[J]
                  I = I + 1
            ciklus vége
            Ha I == N akkor
                  C[K] = B[J]
                  K = K + 1
            ha vége
      ciklus vége
eljárás vége
```

Folyamatábra



Felhasználási területek

Bármilyen olyan probléma megoldásában, ahol felmerül a feladat, hogy két vektor unióját elkészítsük.

Például:

– Ismert egy autógyár németországi- és magyarországi üzemében gyártott modelljeinek listái. Határozza meg, hogy mely modelleket gyártják ebben a két üzemben!

```java
package rivers;

/**
 *
 * @author somlyaip
 */
public class Rivers {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        String[] riversRomania = {
            "Duna", "Tisza", "Fehér-Körös",
            "Fekete-Körös", "Sebes-Körös", "Maros",
            "Szeret", "Cserna", "Karas", "Feketeügy"
        };

        String[] riversHungary = {
            "Rába", "Duna", "Tisza",
            "Dráva", "Fehér-Körös", "Fekete-Körös",
            "Szamos", "Sebes-Körös", "Maros", "Berettyó"
        };

        String[] riversAll = new String[20];
        for(int i = 0; i < riversRomania.length; i++) {
            riversAll[i] = riversRomania[i];
        }

        int a = riversRomania.length;
        for(int h = 0; h < riversHungary.length; h++) {
            int r = 0;

            while(r < riversRomania.length
                    && !riversRomania[r].equals(riversHungary[h])) {
                r++;
            }

            if(r == riversRomania.length) {
                riversAll[a] = riversHungary[h];
                a++;
            }
        }

        System.out.println("All rivers (" + a + "):");
        for(int i = 0; i < a; i++) {
            System.out.println(" " + riversAll[i]);
        }
    }

}
```

Bevezetés a programozásba

Gyakorló feladatok

9. HÉT

1. Adott egy vektorban a Dunaújvárosi 1. postára beérkező, Dunaújvárosban kézbesítendő levelek címzettének irányítószáma. Gyűjtse ki a nem dunaújvárosi címek indexeit egy vektorba, hogy külön lehessen válogatni őket!

2. Töltsön fel egy 10 elemű vektort véletlenszámokkal 62..79 között, majd válogassa ki a páratlan elemek indexeit!

3. Egy középiskolai osztályban a diákoknak lehetőségükben áll választani, hogy angol vagy német nyelvet tanuljanak. Az osztály diákjainak OM azonosítóit tartalmazza egy angol és egy nemet lista, attól függően, hogy melyik nyelvet tanulja az iskolában a diák. Egy diák több nyelvet is tanulhat! Válassza ki, hogy milyen OM azonosítójú diákok tanulnak angol és német idegen nyelvet egyaránt!

4. Egy beléptető rendszerrel védett épületben bűncselekmény történt. A bűncselekmény elkövetéséhez a bűnözőnek a 115-ös és a 119-es terembe is be kellett lépnie. Ismert a bűncselekmény elkövetésének napján a két terembe belépett személyek kártyáinak azonosítója két külön listában. (A belépéshez egy kártyaolvasóhoz kell érinteni a személy belépési kártyáját, amennyiben beléphet a terembe, az ajtó kinyílik, ellenkező esetben elutasítja. A biztonsági rendszer pedig eltárolja, hogy mikor, milyen azonosítójú kártyának adott belépési engedélyt.) A nyomozás kezdetén a nyomozók elkérik a biztonsági szolgálattól, azoknak a kártyáknak az azonosítóit, amelyek segítségével mindkét terembe beléptek. A két lista ismeretében határozza meg a mindkét terembe beengedett kártyaazonosítókat!

5. Ismert egy autógyár németországi- és magyarországi üzemében gyártott modelljeinek listái. Határozza meg, hogy mely modelleket gyártják ebben a két üzemben!

10. fejezet

Bevezetés a programozásba

Mátrixok

Somlyai Pál

Dunaújvárosi Főiskola

2015

Bevezetés a programozásba - Mátrixok

A mátrixok speciális, kétdimenziós tömbök. A kétdimenzió azt jelenti, hogy két index segítségével tudjuk megadni, hogy pontosan melyik elemre gondolunk. Vegyünk egy egyszerű integer mátrixot:

12	45	-51
61	-34	98

A mátrix semmi mást nem jelent, mint összetartozó adatok, egy kétdimenziós térben történő ábrázolása. Emlékszünk még a dimenziókra? A dimenzió azt jelenti, hogy hány adat segítségével tudunk egyértelműen meghatározni egy elemet. A vektor egydimenziós tömb, azaz egy idexxel tudunk egyértelműen azonosítani egy vektorelemet. A matematikában egy szakaszhoz hasonlít, ahol a szakasz egy pontját egyértelműen meg tudjuk határozni a pont sorszámával. A mátrix kétdimenziós tömb, azaz két index segítségével tudjuk meghatározni, hogy pontosan melyik mátrixelemre gondoltunk. Matematikai hasonlattal élve, a mátrix a síkhoz hasonlít, hiszen ott is két adattal, az x és y koordinátával tudjuk meghatározni a sík egy pontját. Természetesen, ahogy a matematikai koordinátáknál, számít a koordináták sorrendje. Lényegében a vektor egy adatsort jelent, a mátrix pedig egy táblázatot. Mint ahogy a vektoroknak, a mátrixoknak is van méretük. Azonban mivel két dimenziója van, azért kettő „hosszal" rendelkezik. Ahogy említettük, a mátrixot felfoghatjuk egy táblázatként. A táblázatok sorokból és oszlopokból állnak. Egy táblázat méretét a sorainak és oszlopainak száma határozza meg. A mátrix esetében sincs ez másképp. A fenti példa mátrix kettő sorból és három oszlopból áll: azaz egy 2 x 3-as mátrix. (A mátrixok méretét a matematikában és az informatikában is <sorszám> x <oszlopszám> formában adjuk meg.) Hogyan tudjuk a fenti mátrixot létrehozni a Javában? int[][] m = new int[2][3];

```
int[][] m = new int[2][3];
```

A fenti utasítással helyet foglaltunk egy 2 x 3-as méretű, integer típusú mátrixnak a memóriában, azonban még nem töltöttük fel elemekkel. Ahogy azt már említettük, a mátrixok elemeire két index segítségével tudunk hivatkozni:

mátrixAzonosító[<indexSor>][<indexOszlop>]

Tekintsük meg, hogy a példa mátrix elemeinek mi az indexe. Az alábbi táblázatban az első oszlopban találjuk a sorok indexeit, az első sorban pedig az oszlopok indexeit.

	0	1	2
0	12	45	-51
1	61	-34	98

Tehát a mátrix elemeinek indexei a következőek:

Érték	Hivatkozás
12	m[0][0]
45	m[0][1]
-51	m[0][2]
61	m[1][0]
-34	m[1][1]
98	m[1][2]

A példamátrixunkat, melyet az előbbi példában hoztunk létre, a következőképpen tudjuk feltölteni elemekkel:

```
m[0][0] = 12;
m[0][1] = 45;
m[0][2] = -51;
m[1][0] = 61;
m[1][1] = -34;
m[1][2] = 98;
```

Természetesen a vektorokhoz hasonlóan, inicializáló blokk segítségével is fel tudjuk tölteni elemekkel a vektort, a definiálás helyén:

```
int[][] m = new int[][]
{
   {12, 45, -51},
   {61, -34, 98}
};
```

Figyeljük meg, hogy ebben az esetben sem adhatjuk meg a mátrix méretét (a tömbökhöz hasonlóan), hiszen a felsorolt elemekből a Java fordító meg tudja határozni a mátrix méretét. Az előző példában látható, hogy a mátrix felfogható egy olyan vektorként, amely elemei vektorok. Minden indexén egy vektor található, tehát az előző példában egy kételemű vektort láthatunk, amelynek a nulladik indexű helyén egy háromelemű integer vektor található, valamint az első indexén is egy ugyancsak háromelemű vektor helyezkedik el. Ezeket a vektorokat ábrázolja az inicializáló blokkon belüli vektorok inicializáló blokkjára emlékeztető sordefiníció.

Természetesen a mátrix méretét is le tudjuk kédezni. Az előző példában az m.length visszaadja a mátrix sorainak számát, az m[0].length, pedig az első sorban található elemek számát, ami megegyezik az oszlopok számával. Azonban ez nem mindig igaz a Javában, hiszen lehetőségünk van olyan kétdimenziós tömböt készíteni, amelynek soraiban különböző számú elemek helyezkednek el:

```
int[][] m = new int[][]
{
      {12, 45, -51},
      {61, -34, 98, 89}
};

System.out.println(m.length);        // 2
System.out.println(m[0].length);     // 3
System.out.println(m[1].length);     // 4
```

Ebben az esetben a „mátrix” első sorában három elem szerepel, a másodikban pedig négy. Azonban mi a mátrixfeladatok során tényleges (matematikailag helyes) mátrixokat fogunk alkalmazni, amelyek minden sorában egyenlő számú elem fog szerepelni, ezért a mátrix bejárása során nyugodtan használhatjuk az oszlopok számának lekérdezésére a mátrix nulladik indexű sorhosszának lekérdezését. A mátrix bejárásához két, egymásba ágyazott for ciklusra van szükségünk, melyek közül a külső a sorokat lépteti, a belső pedig az oszlopokat:

```
int[][] m = new int[][]
{
      {12, 45, -51},
      {61, -34, 98, 89}
};

for(int sorIndex = 0; sorIndex < m.length; sorIndex++) {
      for(int oszlopIndex = 0; oszlopIndex < m[0].length; oszlopIndex++)
{
            System.out.printf("%5d", m[sorIndex][oszlopIndex]);
      }
      System.out.println();
}
```

A fenti kódrészlet futtatásának kimenete:

12 45 -51

61 -34 98

A külső for ciklus minden egyes iterációjában teljes egészében lefut a belső for ciklus, azaz az indexek a következőképen alakulnak a két ciklus lefutása során:

Külső ciklus iterációszáma	Belső ciklus iterációszáma	sorIndex	oszlopIndex	m[sorIndex][oszlopindex]
1.	1.	0	0	12
	2.	0	1	45
	3.	0	2	-51
2.	1.	1	0	61
	2.	1	1	-34
	3.	1	2	98

Figyeljük meg, hogy a mátrix alakjának kialakítása, a belső for ciklust követő sortörés miatt megfelelő. Ha elhagynánk, a mátrix sorai egymás mellé kerülnének, így inkább tűnne egy vektornak, mint egy mátrixnak.

```
 1. package printmatrix;
 2.
 3. /**
 4.  *
 5.  * @author somlyaip
 6.  */
 7. public class PrintMatrix {
 8.
 9.     /**
10.      * @param args the command line arguments
11.      */
12.     public static void main(String[] args) {
13.
14.         int[][] matrix = {
15.             {16, 13, 15, 14},
16.             {12, 39, 11, 10},
17.             {85, 16, 77, 43}
18.         };
19.
20.         System.out.printf("m[0][0]: %d\n", matrix[0][0]);
21.         System.out.printf("m[2][3]: %d\n", matrix[2][3]);
22.         System.out.printf("m[1][2]: %d\n", matrix[1][2]);
23.
24.         for(int row = 0; row < matrix.length; row++) {
25.             for(int column = 0; column < matrix[0].length; column++) {
26.                 System.out.printf("%4d", matrix[row][column]);
27.             }
28.             System.out.println();
29.         }
30.
31.         System.out.println();
32.         for(int column = 0; column < matrix[0].length; column++) {
33.             for(int row = 0; row < matrix.length; row++) {
34.                 System.out.printf("%4d", matrix[row][column]);
35.             }
36.             System.out.println();
37.         }
38.     }
39.
40. }
```

```java
package matrixsumcountminmax;

/**
 *
 * @author somlyaip
 */
public class MatrixSumCountMinMax {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        int[][] matrix = {
            {-12, 59, 14},
            {45, 67, -51}
        };

        int sum = 0;
        int count = 0;
        int min = matrix[0][0];
        int max = matrix[0][0];
        for(int i = 0; i < matrix.length; i++) {
            for(int j = 0; j < matrix[0].length; j++) {
                System.out.printf("%4d", matrix[i][j]);
                sum += matrix[i][j];
                if(matrix[i][j] % 5 == 0)
                    count++;
                if(matrix[i][j] < min) {
                    min = matrix[i][j];
                }
                if(matrix[i][j] > max) {
                    max = matrix[i][j];
                }
            }
            System.out.println();
        }
        System.out.println();
        System.out.printf("Sum: %d\n", sum);
        System.out.printf("Count: %d\n", count);
        System.out.printf("Min: %d\n", min);
```

```java
package matrixrowscolumns;

/**
 *
 * @author somlyaip
 */
public class MatrixRowsColumns {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        int[][] matrix = new int[4][4];

        for(int i = 0; i < matrix.length; i++) {
            for(int j = 0; j < matrix[0].length; j++) {
                matrix[i][j] = (int)(Math.random() * 101) - 25;
                System.out.printf("%5d", matrix[i][j]);
            }
            System.out.println();
        }

        // Első sor összege
        int sum = 0;
        for(int j = 0; j < matrix[0].length; j++) {
            sum += matrix[0][j];
        }
        System.out.println("Sum of first row: " + sum);

        // Második oszlop átlaga
        sum = 0;
        for(int i = 0; i < matrix.length; i++) {
            sum += matrix[i][1];
        }
        System.out.println("Average of second column: " +
                (sum / (float)matrix.length));

        // Harmadik sor minimuma
        int min = 75;
        for(int j = 0; j < matrix[0].length; j++) {
            if(matrix[2][j] < min)
                min = matrix[2][j];
        }
        System.out.println("Minimum of third row: " + min);

        // Páros számú sorok maximuma
        int max = -25;
        for(int i = 0; i < matrix.length; i++) {
            for(int j = 0; j < matrix[0].length; j++) {
                if((i + 1) % 2 == 0 && matrix[i][j] > max)
                    max = matrix[i][j];
            }
        }
        System.out.println("Maximum of even rows: " + max);
    }

}
```

Bevezetés a programozásba
Gyakorló feladatok

10. FEJEZET

1. Töltsön fel egy 3 x 4-es mátrixot véletlengenerált számokkal -50 és +200 között, majd végezze el rajta a következő feladatokat:

    a. Számítsa ki a mátrix elemeinek összegét!

    b. Határozza meg, hogy tartalmaz-e -45 alatti értéket!

    c. Számítsa ki az 1. indexű sor minimumát!

    d. Számítsa ki a 2. indexű oszlop maximumát!

2. Adott egy hőmérséklet-mátrix, amely egy hét, egész órákban mért hőmérsékletét tartalmazza. A mátrix 24 x 7-es méretű. Az oszlopok felelnek meg a napoknak, a sorok pedig az óráknak. A hőmérsékletek tört számként vannak megadva. Végezze el a következő műveleteket a hőmérséklet táblázaton:

a. Számítsa ki a hét átlaghőmérsékletét!

b. Határozza meg, hogy volt-e fagypont alatti hőmérséklet!

c. Határozza meg, hogy délben, minden nap 10 °C feletti volt-e a hőmérséklet?

d. Számítsa ki a keddi nap legalacsonyabb hőmérsékletét!

e. Számítsa ki a csütörtöki nap legmagasabb hőmérsékletét!

3. Adott egy mátrix, amely egy általános iskolás diák órarendjét ábrázolja. Az oszlopokban szerepelnek a napok, a sorokban pedig az órák. A 0. indexben található a 8:00-kor kezdődő első óra. A mátrix 5 x 5-ös méretű. Töltse fel tetszőlegesen tantárgyakkal! Amikor nincs órája a tanulónak, ott helyezzen el egy üres Stringet! Végezze el a következő feladatokat az órarenden!

a. Számolja meg, hogy összesen hány órája van a diáknak egy héten!

b. Számolja meg, hogy hány helyen nincs órája az órarendben!

c. Melyik nap végez a legkorábban, és a legkésőbb? Jelenítse meg a napok nevét, és az utolsó óra kezdetét!

11. fejezet

Bevezetés a programozásba

Továbbfejlesztett for ciklus

Somlyai Pál

Dunaújvárosi Főiskola

2015

Bevezetés a programozásba

Továbbfejlesztett for ciklus

A for ciklust legtöbbször egy összetett adatstruktúra (vektor, mátrix, stb.) elemeinek végigiterálására használjuk. Ezért a Java fejlesztőiben felmerült az ötlet, hogy készítsenek egy olyan ciklust, amelyik direkt egy összetett adatstruktúra bejárására hivatott. (Valójában valószínűleg más programozási nyelvekből vették át ezt a gyakorlatot, de a lényeg az, hogy bekerült a Java nyelvbe is). Továbbfejleszett for ciklusnak (enchanced for loop) nevezték el. Sokszor előfordulhat, hogy foreach-ként fogunk utalni rá, hiszen számos más nyelvben foreach-ként (~haladjon végig minden egyes elemen) nevezik az erre a célra hivatott ciklusokat. A továbbfejelsztett for ciklus szerkezete a következő:

for(<elemTípus> <elemAzonosító> : <összetettAdatstruktúraAzonosítója>) {
…
}

A for kulcsszó után meg kell adnunk az aktuális elem típusát, majd hogy milyen néven szeretnénk rá hivatkozni a cikluson belül, ezt követően egy kettőspont után áll a végigiterálandó adatstruktúra azonosítója.

```
int[] vektor = new int[] {12, 45, -51};

for(int elem : vektor) {
      System.out.println(elem);
}
```

Az előző példa kimenete a következő:

```
12
45
-51
```

Látható, hogy a foreach ciklus minden lefutása során felvette a vektor következő elemének értékét, amit így egyszerűen, indexelés nélkül felahsználhattunk. A foreach lényegében egy kényelmi megoldás, hogy ne kelljen a ciklusváltozók segítségével behivatkoznunk a vektor elemeit.

```
 1. package foreach;
 2. 
 3. /**
 4.  *
 5.  * @author somlyaip
 6.  */
 7. public class Foreach {
 8. 
 9.     /**
10.      * @param args the command line arguments
11.      */
12.     public static void main(String[] args) {
13. 
14.         int[] arrayInt = {2, 5, 12, -89, 72};
15.         System.out.println("Integer array:");
16.         for(int number : arrayInt) {
17.             System.out.print(number + " ");
18.         }
19.         System.out.println("\n");
20. 
21.         String[] someAvengers = {"Iron Man", "Captain America", "Thor", "Hulk"};
22.         System.out.println("Some Avengers:");
23.         for(String superhero : someAvengers) {
24.             System.out.println(" - " + superhero);
25.         }
26.     }
27. 
28. }
```

```java
package foreachsumminmax;

/**
 *
 * @author somlyaip
 */
public class ForeachSumMinMax {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        double[] array = new double[20];

        double rangeMin = -2.3;
        double rangeMax = 5.6;
        for(int i = 0; i < array.length; i++) {
            array[i] = Math.random() * (rangeMax - rangeMin) + rangeMin;
        }

        double sum = 0;
        double min = rangeMax;
        double max = rangeMin;
        for(double element : array) {
            System.out.printf("%4.2f\n", element);
            sum += element;
            if(element < min)
                min = element;
            else if(element > max)
                max = element;
        }
        System.out.printf("Sum: %.1f\n", sum);
        System.out.printf("Min: %.1f\n", min);
        System.out.printf("Max: %.1f\n", max);
    }

}
```

```
1.  package foreachmatrix;
2.
3.  /**
4.   *
5.   * @author somlyaip
6.   */
7.  public class ForeachMatrix {
8.
9.      /**
10.      * @param args the command line arguments
11.      */
12.     public static void main(String[] args) {
13.
14.         double[][] matrix = {
15.             {12.5, 45.7},
16.             {578.1, 226.317}
17.         };
18.
19.         double sum = 0;
20.         System.out.println("Double matrix:");
21.         for(double[] row : matrix) {
22.             for(double element : row) {
23.                 System.out.printf("%8.2f", element);
24.                 sum += element;
25.             }
26.             System.out.println();
27.         }
28.         System.out.printf("Sum: %.4f\n", sum);
29.     }
30.
31. }
```

```
 1. package foreachstring;
 2.
 3. /**
 4.  *
 5.  * @author somlyaip
 6.  */
 7. public class ForeachString {
 8.
 9.     /**
10.      * @param args the command line arguments
11.      */
12.     public static void main(String[] args) {
13.
14.         System.out.print("Please enter a line: ");
15.         String line = System.console().readLine();
16.         for(char character : line.toCharArray()) {
17.             System.out.println(character);
18.         }
19.     }
20.
21. }
```

Bevezetés a programozásba
Gyakorló feladatok

11. FEJEZET

Minden feladat megoldása során használjon foreach ciklust!

Az egyes részfeladatokat külön-külön függvényekkel készítse el!

1. Adott egy naptári napon mért legalacsonyabb hőmérsékletek 20 évre visszamenőleg egy vektorban. Határozza meg, mennyi volt a legalacsonyabb hőmérsékelt az adott naptári napon!

2. Adott egy naptári napon mért legalacsonyabb hőmérsékletek 20 évre visszamenőleg egy vektorban. Határozza meg, mennyi volt a legmagasabb hőmérsékelt az adott naptári napon!

3. Egy adott napon a légszennyezettség mértékének csökkentése végett csak a páros rendszámú járművek közlekedhettek egy magyarországi városban. Ismert az egyik csomópontban álló kamera által rögzített rendszámok listája. A listában csak a rendszámok második tagja, azaz a háromjegyű számok szerepelnek. (Az egyedi rendszámoktól tekintsünk el!) Számoljuk meg, hogy hány páratlan rendszámú autó nem tartotta be a korlátozást! Számítsuk ki, hogy mekkora volt az elhaladó, páratlan rendszámú autók aránya az összes rögzített autóhoz képest!

4. Egy mobiltelefon előfizető másodpercalapú számlázást alkalmazó csomagot használ, amelyben bármilyen irányba fix 35 HUF percíjjal kezdeményezhet hívást. Adott egy vektorban az egy hónap altt kezdeményezett hívásainak időtartalma (hívásonként egy vektorelem), másodpercben megadva. A vektor tartalmazzon 30 elemet, melyek 10 és 4000 közötti véletlengenerált értékek lehetnek.

a. Mekkora összeget „telefonált le”?

b. Az előfizetése 4000 HUF havidíjú, amely teljes mértékben lebeszélhető. Fedezi-e a költségeit? Amennyiben nem, mekkora összeget kell különbözetként befizetnie?

5. Töltsön fel egy 3 x 4-es mátrixot véletlengenerált számokkal -150 és +100 között, majd végezze el rajta a következő feladatokat:

a. Számítsa ki a mátrix elemeinek összegét!

b. Határozza meg, hogy tartalmaz-e -100 alatti értéket!

c. Számítsa ki az 2. indexű sor minimumát!

d. Számítsa ki a 1. indexű oszlop maximumát!

12. fejezet

Bevezetés a programozásba

Parancssori argumentumok

Somlyai Pál

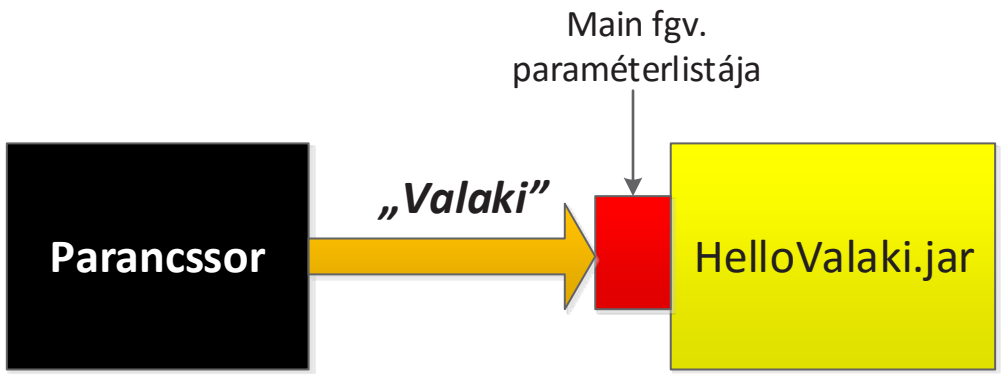
Dunaújvárosi Főiskola

2015

Bevezetés a programozásba – Parancssori argumentumok

Az adatbeolvasást követően a következő mód, ahogy adatokat adhatunk át egy parancssori alkalmazásunknak: a parancssori argumentumok használata. Az első példában készítsük el a klasszikus Hello alkalmazást, azonban most parancssori argumentumként adjuk át számára, hogy kit üdvözöljön.

Az alkalmazás használata során a következő történik:



Létrehozzuk a HelloValaki projektet, amely fordítását követően előáll a HelloValaki.jar fájl. A parancssorban, a

java –jar /path/to/HelloValaki.jar Valaki

Parancs kiadását követően a Valaki karakterlánc a main függvényünk args paraméter tömbének 0. indexére kerül. A */path/to/* egy sűrűn használt jelölés, mely egy tetszőleges elérési út megadását szimbolizálja. Esetünkben a */path/to/HelloValaki.jar a HelloValaki.jar* fájl teljes elérési útvonalát jelöli. Nézzük meg, hogyan kell elkészítenünk a main függvényünket, hogy a Hello Valaki! szöveg jelenjen meg a képernyőn az előző parancs kiadását követően: A parancssori argumentumok használatának módja nem igazán bonyolult. A .jar kiterjesztésű fájl parancssori meghívása során a fájl nevének megadását követően egy szóközt kihagyva, szóközökkel tagolva, felsorolhatjuk az átadandó parancssori argumentumokat, amelyek a program futása során a main függvény args paraméterének fognak átadásra kerülni, pontosan olyan sorrendben, ahogyan felsoroltuk őket.

Ha kiadjuk az előző parancsot, azonban a Valaki szót lecseréljük a saját nevünkre, akkor minket fog köszönteni a programunk. Viszont legyünk körültekintőek, hiszen ha nem adunk meg parancssori argumentumokat a programunk futtatásakor, akkor az args tömb üres lesz, és az args[0] hivatkozás ArrayIndexOutOufBoundException hibát fog okozni.

Rendben, szóközökkel kell tagolni az átadandó argumentumokat. Lényegében ilyenkor a main függvény hívását végezzük el, a hívási argumentumokat felsorolva a .jar fájl mögött. De hogyan adhatunk meg olyan argumentumot, amelyik szóközt tartalmaz? Ha átadjuk a következő argumentumot:

a nevem

akkor az args tömb 0. indexű eleme az a, az 1. indexű pedig a nevem karaktersor lesz. Ilyen esetben az aktuális argumentumot idézőjelek között kell átadni: ”a nevem”. Ilyenkor tetszőleges számú szóköz szerepelhet az idézőjelek között, a main függvényünk egy paraméterként fogja megkapni a teljes idézőjelek között felsorolt karakterláncot.

```
1.  package cmdgeneraterandomnumber;
2.
3.  /**
4.   *
5.   * @author somlyaip
6.   */
7.  public class CmdGenerateRandomNumber {
8.
9.      /**
10.      * @param args the command line arguments
11.      */
12.     public static void main(String[] args) {
13.
14.         if(args.length != 3) {
15.             System.out.println("Arguments count should be 3.");
16.             return;
17.         }
18.
19.         int min = Integer.parseInt(args[0]);
20.         int max = Integer.parseInt(args[1]);
21.         int count = Integer.parseInt(args[2]);
22.
23.         if(min >= max) {
24.             System.out.println("Min should be less than max.");
25.             return;
26.         }
27.
28.         if(count < 1) {
29.             System.out.println("Count should be greater than 0.");
30.             return;
31.         }
32.
33.         for(int i = 0; i < count; i++) {
34.             int randomNumber = (int) (Math.random() * (max - min + 1)) + min;
35.             System.out.println(randomNumber);
36.         }
37.     }
38.
39. }
```

Bevezetés a programozásba

Rekurzív függvényhívás

Somlyai Pál

Dunaújvárosi Főiskola

2015

Bevezetés a programozásba – Rekurzív függvényhívás

A rekurzió egy programozási módszer, melyben egy függvény önmagát hívja meg. Lényegében úgylenne a legpontosabb a megfogalmazás, ha azt mondanánk, hogy önmagát is meghívja. A működéselényegében a következő, elvégez egy adott feladatot, majd meghívja saját magát és ezt ismételi,ameddig egy bizonyos feltétel teljesül. Lényegében, amit meg tudunk oldani rekurzív függvényhívással,azt meg tudjuk oldani ciklusokba szervezéssel is. Néhány esetben azonban sokkal egyszerűbb egyrekurzív függvényhívás használata, mint ciklusokba szervezni a megvalósítást.

Tekintsünk meg egyegyszerű, szemléltető példát rekurzív függvényhívásra:

```
public static void main(String[] args) {
	kiirKettoig(0);
}

public static void kiirKettoig(int i) {
	if(i < 3)
	{
		System.out.print(i + ", ");
		kiirKettoig(i + 1);
	}
	System.out.printf("%nvége, i = %d", i);
}
```

Az előbbi programkód futtatásának kimenete:

```
0, 1, 2,
vége, i = 3
vége, i = 2
vége, i = 1
vége, i = 0
```

Mi történik a háttérben:

1. Meghívásra kerül a kiirKettoig() függvény 0 argumentummal, aminek hatására

2. Kiírja a képernyőre a „0, ” karaktersort, majd meghívja a kiirKettoig() függvényt 1 argumentummal
   – A végrehajtása nem szakad meg, csak mivel a hívási verem tetejére az 1 argumentummal meghívott kiirKettoig() függvény került, ezért annak a végrehajtásával folytatja a JVM.

3. Kiírja a képernyőre az „1, ” karaktersort, majd meghívja a kiirKettoig() függvényt 2 argumentummal.

4. A függvény kiírja a képernyőre a „2, ” karaktersort, majd meghívja a kiirKettoig() függvényt a 3 argumentummal.

5. A függvényben szereplő feltétel nem teljesül, mert i értéke pontosan 3, azért nem hívja meg ismételten magát, hanem tovább fut, és kiírja a „vége, i = 3” szöveget a képernyőre, majd mivel véget ért a végrehajtása, kikerül a veremből, és a JVM az őt meghívó függvény végrehajtásával folytatja.

6. Visszaugrik arra a függvényhívásra, ahol i értéke 2, kilép az if igaz ágából, kiírja „vége, i = 2” karaktersort, majd véget ér a függvényhívása, kikerül a hívási veremből, a végrehajtás onnantól folytatódik, ahol az i = 1 értékkel meghívott függvény meghívta önmagát.

7. Mivel a függvényhívás végrehajtásra került, ezért kilép az elágazásból, kiírja a „vége, i = 1” karaktersort, majd kikerül a hívási veremből, amiért a végrehajtás az i = 0 argumentummal meghívott függvényre kerül.

8. Amelyik kilép az elágazásból, kiírja a „vége, i = 0” szöveget a képernyőre, kikerül a hívási veremből, amiért a program végrehajtása a main függvénybe kerül.

9. Amely végrehajtotta az egyetlen utasítását (a rekurzív függvényhívást), így befejeződik a program futása.

A rekurzió érdekességét talán az alábbi internetes programozói „igazság” szemlélteti:

„Ahhoz, hogy megértsd a rekurziót, először is meg kell értened a rekurziót.”

Ha értettük a mondat csattanóját, akkor valószínűleg sikerült megértenünk a rekurzív függvényhívás működését.

```java
package factorial;

/**
 *
 * @author somlyaip
 */
public class Factorial {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int number = 3;

        if(number < 0) {
            System.out.println("Number should be non-negative!");
            return;
        }

        // Factoriális for ciklussal
        int fact = 1;
        for(int i = 2; i <= number; i++) {
            fact *= i; // fact = fact * i;
        }
        System.out.println("Factorial of " + number + " is " + fact);

        // Faktoriális rekurzív függvényhívással
        System.out.println("Factorial of " + number + " is " + factorial(number));
    }

    public static int factorial(int number) {

        if(number < 0)
            return -1;

        if(number < 1)
            return 1;

        int factorial = number * factorial(number - 1);
        return factorial;
    }

}
```

Bevezetés a programozásba

Statikus függvények osztályokba szervezése

Somlyai Pál

Dunaújvárosi Főiskola

2015

Bevezetés a programozásba – Statikus függvények osztályokba szervezése

Eddig többször esett már szó az osztályokról, azonban mindig abban maradtunk, hogy később fogjuk részletesen megismerni őket. Viszont most egy rövid ízelítőt adunk arról, mire is használhatjuk őket. Le kell szögeznem, hogy a mostani használatuk távol áll az objektumorientált megközelítéstől, azonban a függvények készítését követően ez a következő lépés a forráskód strukturálásában.

Jelenleg maradjunk annyiban, hogy az osztályok segítségével összefoghatunk összetartozó függvényeket. Például készíthetünk egy Matek osztályt, amelyben elhelyezhetjük a négy alapműveletet megvalósító függvényeket, majd a Matek.osszead(2, 4) utasítás kiadásával kiszámíthatjuk 2 + 4 értékét. A lényege a statikus függvények osztályokba szervezésének, hogy ha a későbbiekben valamelyik programunk során ismételten szükségünk lesz a négy alapműveletre, csak ezt az osztályt kell tovább vinnünk a programunkba, és már használhatjuk is őket.

Tekintsük meg a Matek osztályt és a használatát:

```
public class JavaTesting {

    public static void main(String[] args) {
        System.out.println(Matek.osszead(9, 2));
        System.out.println(Matek.kivon(9, 2));
    }
}

class Matek {
    public static int osszead(int a, int b) {
        return a + b;
    }

    public static int kivon(int kisebbitendo, int kivonando) {
        return kisebbitendo - kivonando;
    }
    //...
}
```

A class kulcsszóval hozhatunk létre egy új osztályt, amelynek meg kell adni a nevét, valamint kapcsos zárójelek között kell elhelyezni a függvényeit. Az osztálynak a main függvényünket tartalmazó osztály blokkján kívül kell elhelyezkednie. Az osztályba szervezett függvényekre pedig az:

<osztályNeve>.<függvényNeve>()

formában tudunk hivatkozni. Remélem mindenkinek ismerős ez a módszer a Math.abs(), Math.pow(), stb. függvényekből. Ők is osztályokba rendezett statikus függvények. Eddig minden függvényünk statikus volt, a következő tárgy keretein belül fogunk áttérni a „nem statikus” függvények használatára.

```
 1. package staticclasses;
 2.
 3. /**
 4.  *
 5.  * @author somlyaip
 6.  */
 7. public class StaticClasses {
 8.
 9.     /**
10.      * @param args the command line arguments
11.      */
12.     public static void main(String[] args) {
13.         int[] numbers = {5, 92, 78, 10, 250};
14.         IntArray.print(numbers);
15.         System.out.println("Sum: " + IntArray.sum(numbers));
16.         System.out.println("Min: " + IntArray.min(numbers));
17.
18.         float[] floatArray = {12.45f, 45.12f, 6.7f, 98.35f};
19.         FloatArray.print(floatArray);
20.         System.out.println("Sum: " + FloatArray.sum(floatArray));
21.     }
22.
23. }
24.
25. class IntArray {
26.     public static void print(int[] array) {
27.         if(array.length > 0) {
28.             System.out.print(array[0]);
29.
30.             for(int i = 1; i < array.length; i++) {
31.                 System.out.print(", " + array[i]);
32.             }
33.             System.out.println();
34.         }
35.     }
36.
37.     public static int sum(int[] array) {
38.         int sum = 0;
39.         for(int number : array) {
40.             sum += number;
41.         }
```

```
42.         return sum;
43.     }
44.
45.     public static int min(int[] array) {
46.         int min = array[0];
47.         for(int i = 1; i < array.length; i++) {
48.             if(array[i] < min)
49.                 min = array[i];
50.         }
51.         return min;
52.     }
53. }
54.
55. class FloatArray {
56.     public static void print(float[] array) {
57.         if(array.length > 0) {
58.             System.out.printf("%.2f", array[0]);
59.
60.             for(int i = 1; i < array.length; i++) {
61.                 System.out.printf(", %.2f", array[i]);
62.             }
63.             System.out.println();
64.         }
65.     }
66.
67.     public static float sum(float[] array) {
68.         float sum = 0;
69.         for(float number : array) {
70.             sum += number;
71.         }
72.         return sum;
73.     }
74. }
```

Bevezetés a programozásba
Gyakorló feladatok

12. FEJEZET

1. Készítsen programot, amely az első parancssori argumentumában megadott darabszámban jeleníti meg a második paraméterként megkapott szöveget, egymás alatt a képernyőn!

2. Készítsen programot, amely parancssori argumentumok alapján, bizonyos String műveleteket végez el, az ugyancsak paraméterként átadott szövegen. Az alkalmazást a következő képen lehessen használni:

java –jar Strings.jar -<kapcsoló> <inputSzöveg>

Tehát a Strings.jar program meghívásakor átadott első argumentum egy kapcsoló, amelyik a program működését befolyásolja, ettől függ, hogy a bemenő szövegen milyen műveletet kell elvégeznünk. A kapcsolók a következőek lehetnek, valamint a következő művelet elvégzését jelentik:

- -l toLowerCase
- -u toUpperCase
- -t trim

Tehát a program a következő argumentumokra, a következő eredményeket írja ki a képernyőre:

java –jar String.jar –l LOwERCaSe	lowercase
java –jar String.jar –u uppercase	UPPERCASE
java –jar String.jar –t ” a ”	a

3. Készítsen statikus függvényeket összegyűjtő osztályokat a következő feladatokra:

a. A véletlengenerált számok előállítására.
   i. Adott intervallumban, egész számok generálására
   ii. Adott intervallumban, tört számok generálására

b. Az eddig ismertetett programozási tételek alkalmazása
   i. Tulajdonságvizsgálatot tartalmazó tételek esetén (eldöntés, megszámlálás), adja meg az adott feltételt a függvény nevében, pl.:

int megszamlalParosak(int[] v)

Bevezetés a programozásba
INF-501

Algoritmusok és adatábrázolás

Kógelmann Gábor

Dunaújvárosi Főiskola,
Informatikai Intézet

1. Az algoritmusszerkesztés alapfogalmai

Az elektronikus számítógépek által végrehajtandó feladatokat a gép számára érthető nyelven kell megfogalmazni. Erre a feladatra számítógépi programokat használunk.

A számítástechnika történetének legjelentősebb programnyelvei, kialakulásuk időrendjében:

- ALGOL (**ALGO**rithmic **L**anguage)
- FORTRAN (**FOR**mula **TRAN**slator)
- COBOL (**CO**mmon **B**usiness **O**riented **L**anguage)
- PL/I (**P**rogramming **L**anguage **I**)
- PASCAL
- C, C++
- Modula stb...

A fenti, úgynevezett **magasszintű programnyelv**ek mellett minden számítógép architektúra (mikroprocesszor) számára létezik egy gépközeli programnyelv, amit **ASSEMBLY szintű programnyelv**nek hívnak.

A program utasítások sorozata, amelyeket az adott programnyelv **formai (szintaktikai)** szabályainak betartásával kell alkalmazni. A programok tartalmi ("működési") helyessége a **szemantikai** programhelyesség.

Az **algoritmikus gondolkodásmód,** a **programozási logika** elsajátításához nincs szükség konkrét programnyelv ismeretére.

Tananyagunk e fejezete, az algoritmuskészítés alapjainak megismertetésére vállalkozik.

1.1 Az algoritmus fogalma

A számítógép általános megfogalmazás szerint egy adatok feldolgozására készített eszköz. Ilyen megközelítés alapján az adatok két csoportba sorolhatók:

– A program

– A program által kezelt adat

1.1.1 A program és az algoritmus

A program a megoldáshoz vezető utat tartalmazza, azaz azt írja le, hogy milyen műveleteket kell elvégezni a **tényleges adatokkal,** hogy a kívánt eredményhez jussunk.

A **program** egy adott programnyelven megvalósított végrehajtási **algoritmus.** Az algoritmus általában nem kötődik semmilyen konkrét programnyelvhez. Az a módszer vezet jó eredményre, ha előbb elkészítjük a feladat megoldásának algoritmusát, majd ehhez választunk megfelelő programnyelvet.

Definíció:

Algoritmusnak nevezzük az aritmetikai, logikai, stb... műveletek olyan célszerűen összeállított sorozatát, amely a kitűzött feladat egyértelmű megoldásához vezet.

A feladatok általában bonyolultak, azaz sok esetben több száz, sőt több ezer elemi lépést tartalmaznak. Ezért a feladatot célszerű részfeladatokra bontani. Ebben az esetben mindig van egy **főprogram,** és több-kevesebb **alprogram.** Az alprogramokat szokás **eljárásnak, szubrutinnak** is nevezni. A feladat megoldásnak ezt a megközelítését strukturált programozásnak nevezzük.

A főprogram eljárást, vagy eljárásokat hív meg, melyek aztán további eljárá- sokat hívhatnak. Az eljárások befejeztével a vezérlés a hívás utáni utasításra kerül vissza. Az eljáráshíváskor **paraméter**ek adhatók át a hívottnak. Ennek segítségével lehet az általános feladatmegoldást az aktuális igényhez igazítani.

Követelmények az algoritmusokkal szemben:

– Legyen általános, amennyire ezt az adott feladat megengedi.

– Szélső esetekben is helyes eredményt adjon.

– Véges számú lépés (idő) után fejeződjön be.

Az algoritmusok fajtái:

– Verbális algoritmus

– Jel algoritmus

A **verbális algoritmus** szövegesen adja meg a feladatot. (Élőszóban, vagy írásban.)

Az írásban megadott feladatmegoldást szokás **pszeudó kód**nak nevezni.

A **jel algoritmus** a feladat megfogalmazására geometriai szimbólumokat, matema- tikai jelöléseket használ.

A leggyakrabban használt jel algoritmusok:

– Folyamatábra

– Struktogram

– Szerkezeti ábra

1.1.2 A program által kezelt adatok

Az adatok egyik nagy csoportját a **konstans**ok alkotják. **Értékük** a program futtatása során **nem változhat** meg.

A másik csoportot a **változók** képezik. Ezek **értéke** a program futtatása során **megváltozhat.** A változó is kétféle lehet: **egyszerű** (skalár) és **összetett**.

Az egyszerű változók három jellemző tulajdonsága:

– Név
Általában az angol ABC betűiből, a számokból, és néhány különleges jelből képezzük. Célszerű a tartalomra utaló elnevezést választani. Egy memóriaterületet szimbolizál.

– Típus
Ez a jellemző utal arra, hogy a memóriában az adatot milyen formában tároljuk, illetve azt is behatárolja, hogy milyen jellegű műveletet végezhetünk vele.

– A változó címe a memóriában
Azt írja le, hogy az adat milyen kezdőcímtől helyezkedik el a memóriában.

Az összetett változók jellemző tulajdonságai:

– Megegyezik az előző hárommal

– A kezelt adatcsoport jellemzői
Eszerint beszélünk **tömb**ről és **halmaz**ról.
– Tömb
Jellemző tulajdonsága a **dimenzió**.
Egy, két, három, esetleg több dimenzió.
Az egydimenziós tömb neve: **vektor**
A[3] ; B[12]
A kétdimenziós tömb neve: **mátrix**
C[3,2] ; D[5,5]

A tömb elemeire az **indexel** hivatkozunk.
C[1,1] - Az első sor, első oszlopa
C[3,2] - A harmadik sor második oszlopa

– Halmaz
Jellemző tulajdonsága, hogy elemeit mely adatok közül választhatjuk ki.
H1 = [12, 34, 7, 23, 12]
H2 = [A, B, a, b]
H3 = [ ] - Üres halmaz.

Az egyes adat típusokon értelmezett műveletek:

– Numerikus (valós, egész)
  – Anégyalapművelet (+ - * /)
  – A hatványozás (^ vagy **)
  – Egyváltozós műveletnél az előjel (-)

– Csak egész típusú adatok esetén
  – Egészosztás (DIV) 7 DIV 2 ⇨ 3
  – Maradékképzés (MOD) 7 MOD 2 ⇨ 1

– Karakter és sztring típusú adatok
  – Egymásutánírás (+) (konkatenálás)
  "A" + "1" ⇨"A1"
  "ABCD" + "EFG" ⇨"ABCDEFG"

– Logikai adatok
  – ÉSművelet (AND)
  – VAGY művelet (OR)
  – KIZÁRÓ VAGY művelet (XOR)
  – TAGADÁS művelet (NOT)

– Az igazságtáblázat:

a	b	a AND b	a OR b	a XOR b	NOT a
1	1	1	1	0	0
1	0	0	1	1	
0	1	0	1	1	1
0	0	0	0	0	

0 - Hamis (FALSE)
1 - Igaz (TRUE)

Relációs műveletek:

– Két azonos típusú konstans, illetve változó(k) között értelmezhető.

– =,>,<,>=,<=,<>(Nemegyenlő)

– A művelet eredménye egy logikai érték (1 , 0)

A = 5; B = 6; A > B → 0
A < B → 1
A = B → 0

A halmazokon értelmezett műveletek:

– Mindig kétoperandusúak.

– Halmazok összege (+)
Az azonosak csak egyszer kerülnek az eredmény halmazba.

– Halmazok szorzata (*)
Azok kerülnek az eredmény halmazba, melyek mindkét halmazban előfordulnak.

– Halmazok különbsége (-)
Az első operandus elemeiből elmaradnak azok az elemek, amelyek a másodikban is megvannak.

A változók függvények segítségével is kaphatnak értéket.

Példaként néhány, majdnem minden programnyelvben előforduló, beépített függvény:

– Matematikai
  – ABS(x) x numerikus
  – COS(x) x numerikus
  – SQRT(x) x nemnegatív numerikus

– Konverziós
  – CHR(x) x numerikus (0–255) x numerikus
  – STR(x) x numerikus
  – VAL(x) x sztring

– Sztring-kezelő
– LEFT(x,y) x sztring, y nemnegatív egész
– LENGTH(x) x sztring

Értékadó kifejezések:

Azt, hogy milyen adatokon, milyen műveletet kell elvégezni **kifejezés**sel adjuk meg.

A kifejezés bal oldalán változónak kell lennie, a jobb oldalon előfordulhatnak változók, konstansok, függvényhívások (visszatérő értékkel) és műveleti jelek. A két oldal között az egyenlőség-jel áll.
A jobboldali változó(k) értéke a műveletvégzés során nem változik meg.

A műveletek elvégzési sorrendje kötött. A sorrend **zárójel**ekkel befolyásolható. Több zárójel esetén a kiértékelés a belső zárójelpártól indul. A műveleteknek **precedenciá**ja (prioritása) van.

A precedencia-szabály részben programnyelv függő, de általában az alábbi:

– előjel

– negáció

– hatványozás

– szorzás, osztás

– összeadás, kivonás

– reláció műveletek

– stb.

A felsorolás csökkenő prioritás szerinti.

Az azonos prioritásszinten a kiértékelés általában balról jobbra irányú.

Példa:

```
a=-b+c*2 ; a=-(b+c)*2 ; a=(-b+c)*2
z = ABS(x) + y
i=i+1
```

Az utóbbi példa olvasása: Olvasd ki i értékét, növeld meg eggyel, és az eredményt helyezd vissza **i**-be.

1.2 Az algoritmusokban előforduló tevékenységek

Alaptevékenységek:

– Adatok beolvasása megadott változókba.
(**Input** tevékenység)

– Adatok kiírása. Az adat lehet konstans, változó tartalom, és kifejezés értéke.
(**Output** tevékenység)

– A kifejezések értékének kiszámítása.
(**Értékadó** tevékenység)

Vezérlőtevékenységek:

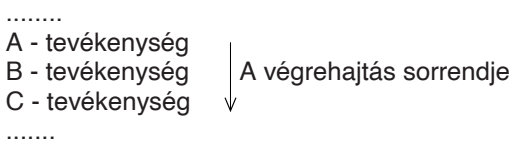
– A számítógép az egyes műveleteket az utasítások leírásának sorrendjében hajtja végre.
(**Szekvenciális** vezérlőtevékenység)

– Egy kifejezés értékétől függően kiválasztja a következő tevékenységet.
(**Szelekciós** vezérlőtevékenység)

– Egy vagy több tevékenységet, egy feltételtől függően ismétel.
(**Iterációs** vezérlőtevékenység)

1.2.1 Szekvenciális vezérlőtevékenység

Másnéven szekvenciális (soros) végrehajtási folyamat. A legtöbb adatfeldolgozási folyamat ezt a logikát követi.



1.2.2 Szelekciós vezérlőtevékenység

Feltételtől függő végrehajtási folyamat. Olyan feltételt tartalmaz, amely több alternatív tevékenység közül egynek a kiválasztásához vezet.

Típusai:

- Egyszerű alternatíva

```
IF      feltétel      THEN
        [A - modul]
(Az IF szerkezet vége)
```

- Kettős alternatíva

```
IF      feltétel      THEN
        [A - modul]
ELSE
        [B - modul]
(Az IF szerkezet vége)
```

- Összetett alternatíva

```
IF      feltétel(1)           THEN
        [A(1) - modul]
ELSE  IF  feltétel(2)     THEN
        [A(2) - modul]
        ......
        ......
ELSE  IF  feltétel (n)    THEN
        [A(n) - modul]
ELSE
        [B - modul]
(Az IF szerkezet vége)
```

A modul egy, vagy több tevékenységet (utasítást) foglal magába.

1.2.3 Iterációs vezérlőtevékenység

Feltételtől függő, ismétlődő végrehajtási folyamat, másnéven **ciklus.** Az ismétlődő utasítás(ok) alkotják a **ciklustörzset** (ciklusmagot). Az iterációs vezér- lőtevékenység másik része a **ciklusfej**.

Általános esetben a ciklusfej résztevékenységei:

– Ciklusváltozó kezdeti értékadása (inicializálás)

– A ciklusváltozó tesztelése (a ciklusból való kilépés érdekében)

– A ciklusváltozó értékének módosítása

Konkrét programnyelvtől függően az első, illetve harmadik résztevékenység el is maradhat.

Típusai:

- Előltesztelő - WHILE típusú ciklus
  WHILE feltétel
  [modul (ciklusmag)]
  (A WHILE szerkezet vége)

- Előltesztelő - FOR típusú ciklus („számlálós ciklus”)
  FOR i = k **TO** v **BY** m
  [modul (ciklusmag)]
  (A FOR szerkezet vége)

A betűk (változók) jelentése:

i - Ciklusváltozó
k - A ciklusváltozó kezdőértéke
v - A ciklusváltozó végértéke
m - A ciklusváltozó módosító értéke

- Hátultesztelő - DO - WHILE típusú ciklus

```
DO
        [modul (ciklusmag)]
WHILE  feltétel
(A WHILE szerkezet vége)
```

A hátultesztelő ciklus esetén a ciklusmag legalább egyszer végrehajtódik.

Ha a ciklusváltozó tesztelése elmarad, (esetleg hibás) úgynevezett **végtelen ciklust** kapunk.

A ciklusok a gyakorlati feladatok megoldásakor sok esetben egymásba ágyazottak.

```
x = 0
FOR  i = 1  TO  2  [BY  1]
    FOR  j = 1  TO  3  [BY  1]
        x = x + 1
    NEXT
NEXT
```

Az x értéke az egymásbaágyazott ciklusok végrehajtása után 6 lesz.
A [...] jelölés arra utal, hogy a módosító (lépésköz) megadása elmaradhat. Ekkor a feltételezett érték: +1.

2. Az algoritmusok megfogalmazására felhasználható segédeszközök

Ebben a fejezetben a leggyakrabban használt algoritmus leíró módszereket tekintjük át:

– Pszeudókód

– Folyamatábra

– Struktogram

– Szerkezeti ábra

2.1 Pszeudókód

Az algoritmus szövegszerű (mondatszerű) leírása. Hátránya, hogy nincs szabványosítva. Minden könyv, jegyzet szerzője az ízlésének legjobban megfelelő jelölésrendszert használ, ezért mindig definiálni kell az egyes alap és vezérlőtevékenységek éppen aktuálisan alkalmazott jelölésrendszerét.

A nyelve lehet angol, de adott esetben akár a szerző anyanyelvének megfelelő kulcsszavak is alkalmazhatók. E jegyzetben mi is ez utóbbit tesszük.

Operátorok:

– Aritmetikai operátorok: +, -, *, /

– Relációs (összehasonlító) operátorok: <, <=, >, >=, =, <>

– Logikai operátorok: **és, vagy, nem**

Tevékenységek (műveletek):

– Értékadás: változó = kifejezés

– Beolvasás: **Be:** <változó lista>

– Kiírás: **Ki:** <változó lista>

– Szelekciós tevékenység:
  – Egyszerű alternatíva
    Ha <feltétel> **akkor**
    <műveletek>
    ha vége.

  – Kettős alternatíva:
    Ha <feltétel> **akkor**
    <műveletek1> **különben**
    <műveletek2>
    ha vége.

  – Összetett alternatíva:
    Ha <feltétel1> **akkor**
    <műveletek1>
    különben Ha <feltétel2> **akkor**
    <műveletek2>
    különben Ha <feltétel3> **akkor**
    <műveletek3>
    [**különben**
    <műveletek4>]
    ha vége.

– Iterációs tevékenység:
  – Előltesztelő – **WHILE** típusú ciklus:
    Ciklus amíg <feltétel>
    <műveletek>
    ciklus vége.

  – Előltesztelő – **FOR** típusú ciklus:
    Ciklus <kezdőértéktől> <végértékig> [<lépésköz>]
    <műveletek>
    ciklus vége.

  – Hátultesztelő – DO - WHILE típusú ciklus:
    Ciklus
    <műveletek>
    amíg <feltétel> **ciklus vége.**

Eljárás (szubrutin, függvény):

<az eljárás neve> (<formális paraméterlista>) <műveletek>
<az eljárás neve> **vége.**

Példa:

Számoljuk ki egy N elemű sorozat összegét!

Megoldás:

1. Olvassuk be az összegzendő elemek számát (N);
2. Olvassuk be a következő összegzendő elemet (A);
3. Végezzük el az összesítést;
4. Irassuk ki az eredményt (S)!

```
S = 0
Be: N
Ciklus I=1-től N-ig
   Be: A
   S = S + A
ciklus vége.
Ki: S
```

2.2 Folyamatábra

A folyamatábrát szokás blokkségének is nevezni.

Két alaptípusa:

– Szervezői

– Programozói

A **szervezői folyamatábra** a felhasznált adatállományok nevét, típusát, és az adatokat kezelő programok megnevezését tartalmazza.
A **programozói folyamatábra** (blokk-diagram) a megoldandó feladat részletekbe menő megfogalmazása.

A folyamatábra blokkokból áll, ezek tartalmazzák az egyes utasításokat.
Hogy egy feladatnak mekkora részét tekintjük egy blokknak, nincs előírás.
Összetett, nagyobb méretű feladat esetén, mindenképpen alkalmazzunk eljárásokat.

2.2.1 Folyamatábra szimbólumok

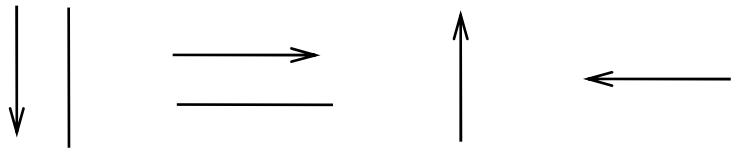
Az aritmetikai (általános) utasítás szimbóluma:

Elsősorban aritmetikai utasítások jelölésére használatos, de minden olyan esetben is használható, amelyre nincs külön szimbólum.

A művelet végrehajtási sorrend:

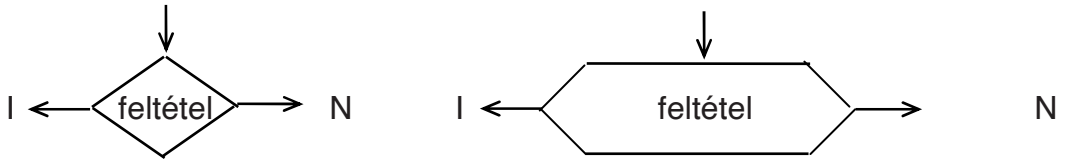
A sorrendet nyilakkal jelöljük. A fentről lefelé, illetve a balról jobbra mutató nyilat nem kell kitenni.

A folyamatvonalak nem keresztezhetik egymást.

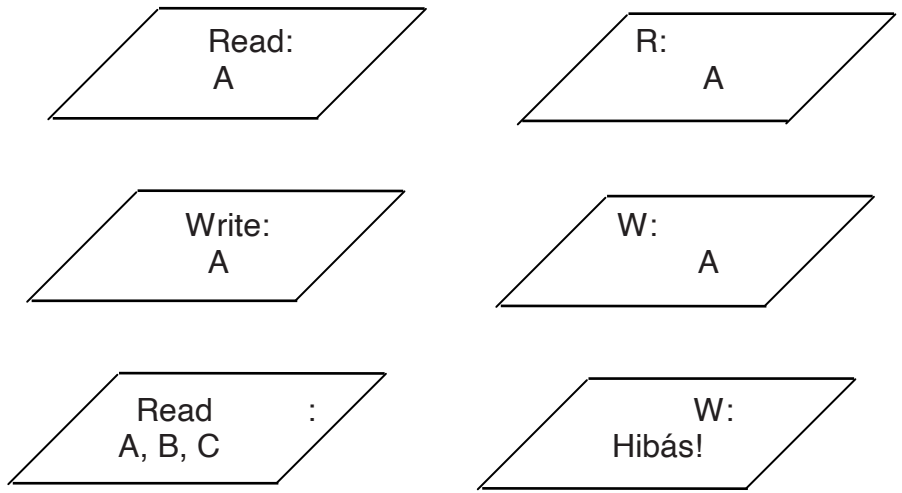


A döntés, elágaztatás (szelekció) szimbóluma:

A szimbólum belsejébe írt feltétel (kifejezés) teljesülésekor az **Igen**, ellenkező esetben a **Nem** ágon folytatódik a feladat végrehajtása.



A beolvasó (input) és kiíró (output) művelet szimbóluma:
A szimbólum nem határozza meg az adott periféria típusát, csak az adatmozgás irányát.
A beolvasás speciális értékadásnak tekinthető. A kiíró művelet esetén konstans érték is megadható.



Az eljárás hívás szimbóluma:

Amennyiben a feladat mérete és bonyolultsága megkívánja használható.

Az algoritmus kezdés és befejezés szimbóluma:

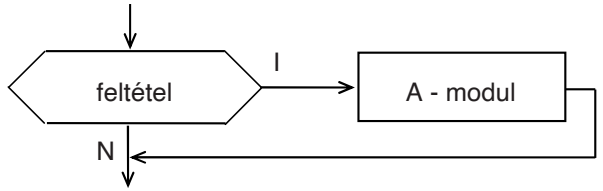
Minden folyamatábrának egy kezdő és egy végpontja lehet. Az eljárásoknál a szimbólumok az eljárás nevét is tartalmazzák.

A csatlakozópont szimbóluma:

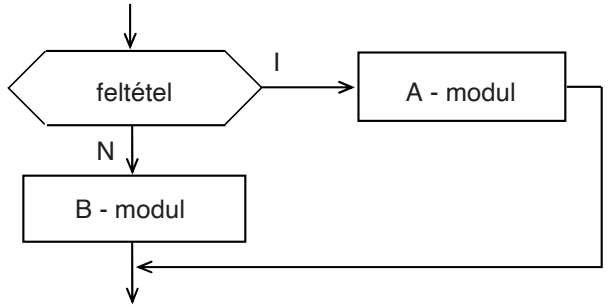
Amennyiben a folyamatábra nem fér el egy lapon,vagy a folyamatvonalak kereszteznék egymást, akkor kell alkalmazni.

2.2.2 A vezérlőtevékenységek megvalósítása folyamatábrával

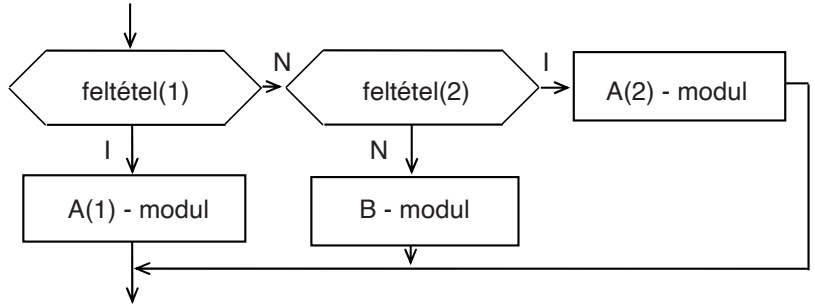
Szelekciós vezérlőtevékenység:



- Kettős alternatíva

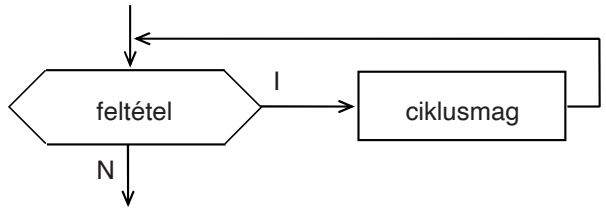


- Összetett alternatíva

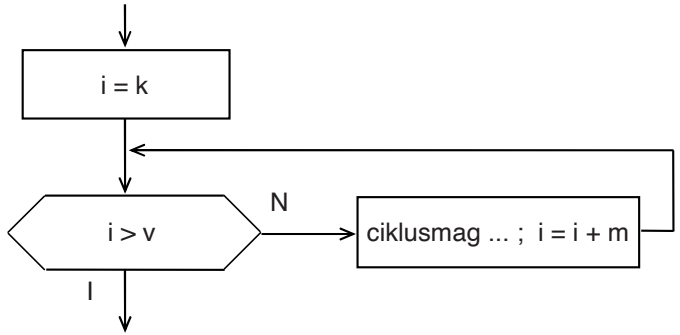


Iterációs vezérlőtevékenység:

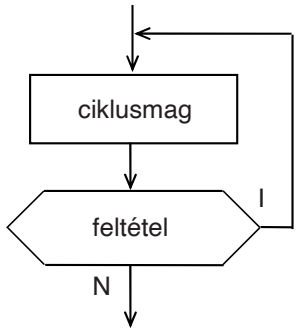
- Előltesztelő - WHILE típusú ciklus



- Előltesztelő - FOR típusú ciklus



- Hátultesztelő - DO - WHILE típusú ciklus



2.2.3 Példák folyamatábrára

1. Példa

Olvasson (kérjen) be számokat és válassza ki közülük a maximális értékűt! A maximumot jelenítse meg!

Az első megoldás:

A legegyszerűbb, de csak bizonyos feltételek mellett működik helyesen.

Ezek a feltételek:

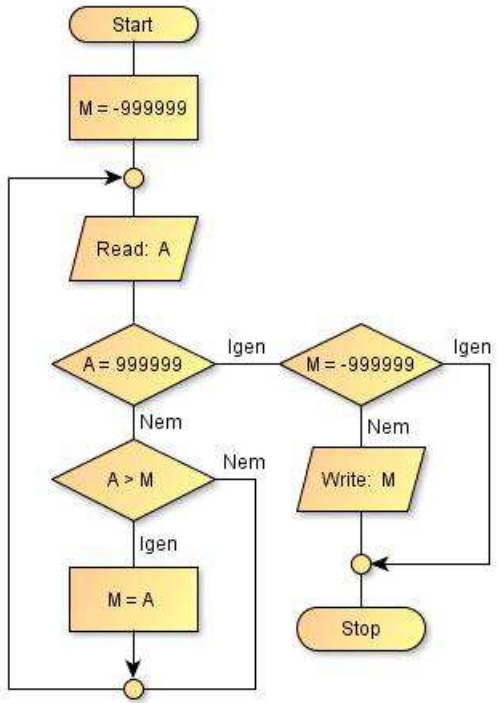
– A számok maximum hat jegyűek

– A számsorozat végét a 999999 jelzi

– A számok között nem lehet a -999999 és a 999999

A változók: A – A beolvasott szám
M – A maximum

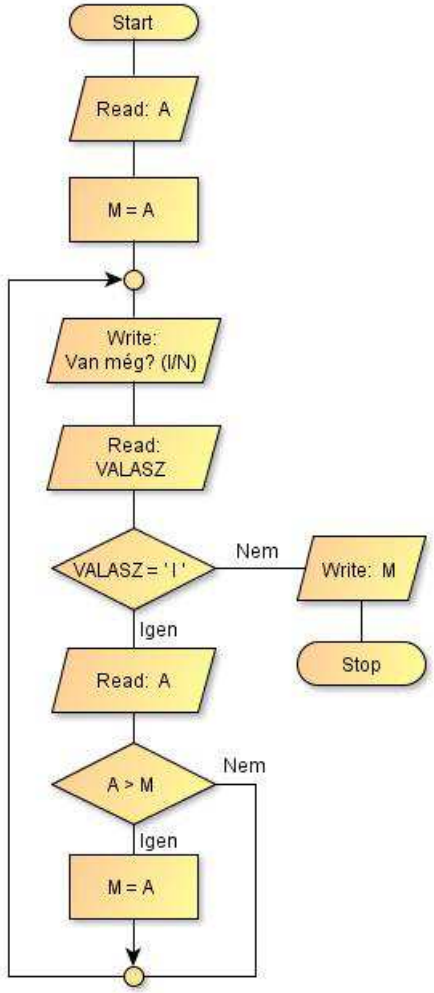
1. példa / 1. megoldás



A második megoldás:

Ebben a továbbfejlesztett megoldásban minden újabb adatbekérése előtt megkérdezzük, van-e még további szám?
A változók: A – A beolvasott szám
M – A maximum
VALASZ – A feltett kérdésre adott válasz

1. példa / 2. megoldás



2. Példa

Olvasson (kérjen) be (**N**) darab számot, és gyűjtse a párosakat a (**B**) vektorba, a páratlanokat a (**C**) vektorba! A két tömb elemeit írassa ki!

Miután először a páros, majd a páratlan számokat szeretnénk kiíratni, a vektorokba való ideiglenes tárolás elkerülhetetlen.

A szám páros, illetve páratlan voltának eldöntésére két módszert is választhatunk:

– Alkalmazzuk a moduló (**MOD**) utasítást, vagy függvényt.

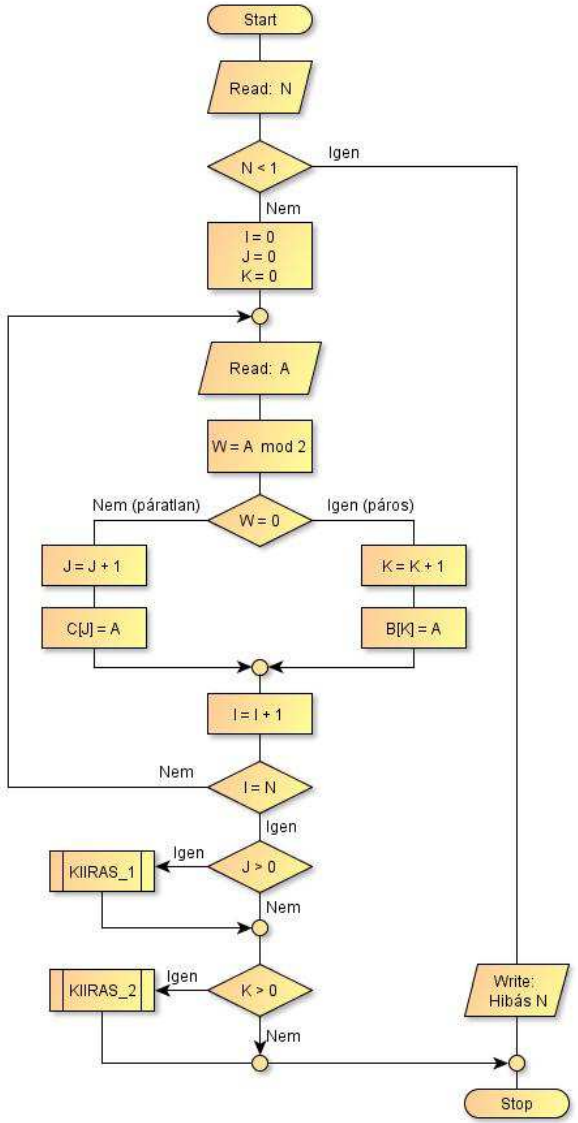
– Egészosztást (**DIV**) végzünk kettővel, és a kapott eredményt kettővel visszaszorozva hasonlítjuk az eredeti értékhez.

Az algoritmusban a páros, illetve páratlan számok kiírására eljárásokat használunk.

A változók:
N– Az elemek száma
A – A beolvasott szám
I – A futó index
J – A páratlan számok tömbjének indexe
K – A páros számok tömbjének indexe
W – Munkaváltozó

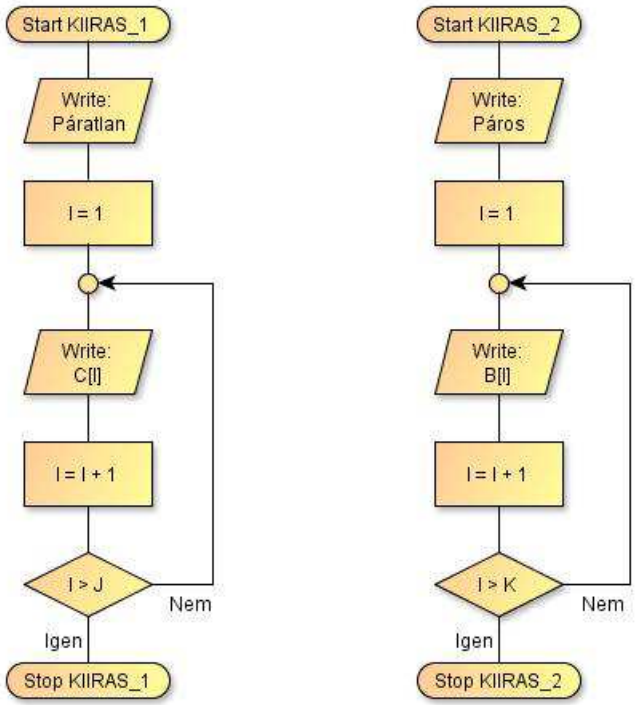
A következő oldalon láthatják a főprogram („főalgoritmus”) folyamatábráját:

2. példa / 1. lap


Start
Read: N
N < 1
Igen
Nem
I = 0
J = 0
K = 0
Read: A
W = A mod 2
Nem (páratlan)
W = 0
Igen (páros)
J = J + 1
K = K + 1
C[J] = A
B[K] = A
I = I + 1
Nem
I = N
Igen
Igen
KIIRAS_1
J > 0
Nem
Igen
KIIRAS_2
K > 0
Write:
Hibás N
Nem
Stop

Az eljárások:

2. példa / 2. lap

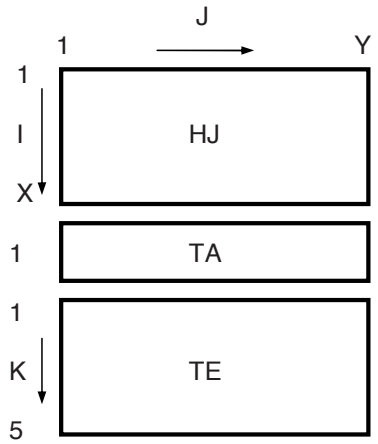


3. Példa

Adott (**X**) hallgató (**Y**) tantárgyának érdemjegye.
Határozza meg a tantárgyankénti jegyeloszlást, illetve átlagot!

A feladat eléggé összetett ahhoz, hogy egy főprogram mellett több eljárásra bontsuk. A legegyszerűbb (és egyben legáltalánosabb) megoldást akkor kapjuk, ha tömböket használunk a feladat megoldására.

A felhasznált tömböket és ciklusváltozókat az alábbi ábrán szemléltetjük:



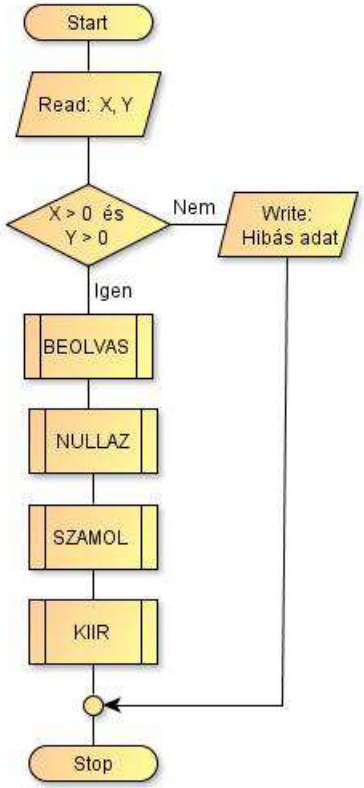
HJ - Hallgatók jegyei mátrix

TA - Tantárgy átlag vektor

TE - Tantárgy jegyeloszlás mátrix

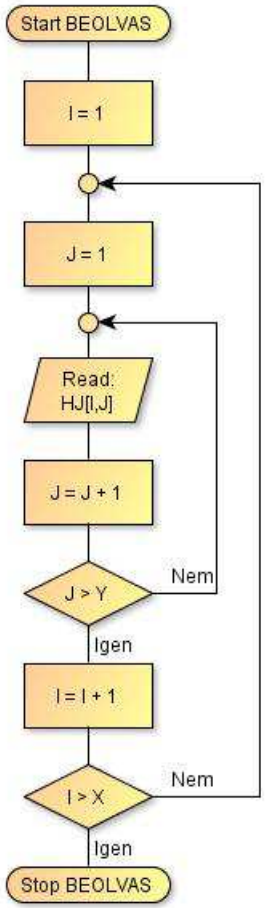
A főprogram („főalgoritmus”):

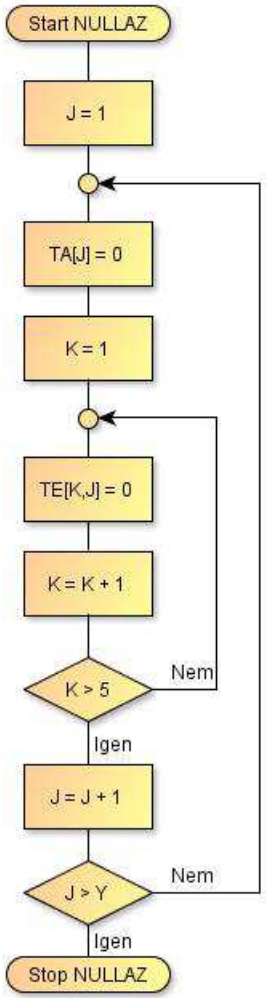
3. példa / 1. lap



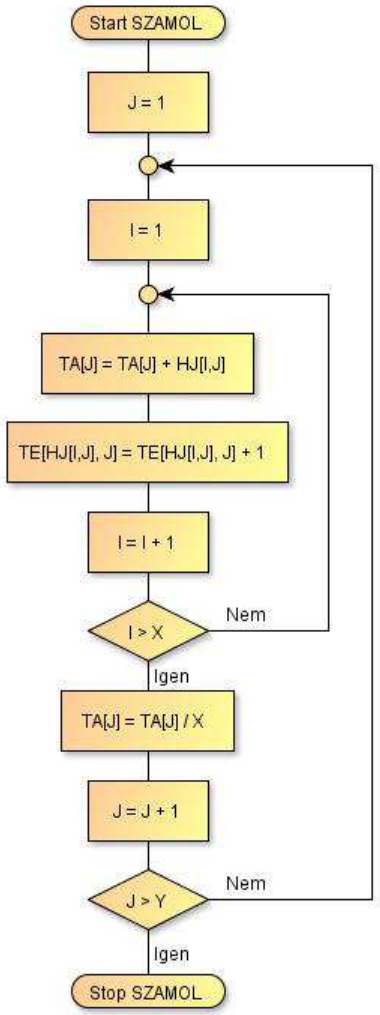
Az eljárások:

3. példa / 2. lap

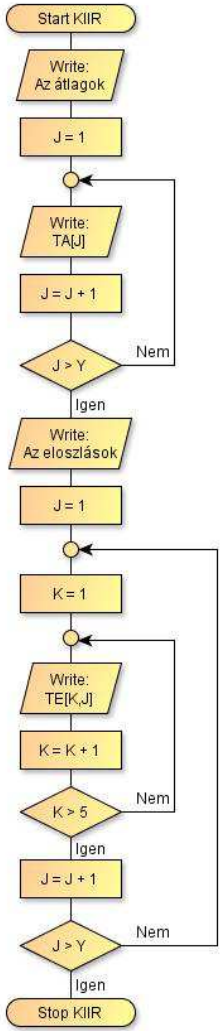




3. példa / 3. lap


Start SZAMOL
J = 1
I = 1
TA[J] = TA[J] + HJ[I,J]
TE[HJ[I,J], J] = TE[HJ[I,J], J] + 1
I = I + 1
I > X
Nem
Igen
TA[J] = TA[J] / X
J = J + 1
J > Y
Nem
Igen
Stop SZAMOL

3. példa / 4. lap


Start KIIR
Write:
Az átlagok
J = 1
Write:
TA[J]
J = J + 1
J > Y
Nem
Igen
Write:
Az eloszlások
J = 1
K = 1
Write:
TE[K,J]
K = K + 1
K > 5
Nem
Igen
J = J + 1
J > Y
Nem
Igen
Stop KIIR

2.3 Struktogram

A jel algoritmus másik gyakran használt formája. Jellemzője, hogy az algo- ritmus megtervezésére és megfogalmazására csak téglalapokat használ. Egy feladat megoldását az őt jelképező téglalapba rajzolt - szintén téglalappal jelképezett - részfeladatok megoldásával, vagy alaptevékenységek végrehajtásával kapjuk meg.

2.3.1 A vezérlőtevékenységek megvalósítása struktogrammal

Szelekciós vezérlőtevékenység:

– Egyszerű alternatíva

// feltétel //	
igen	nem
A - modul	-

– Kettős alternatíva

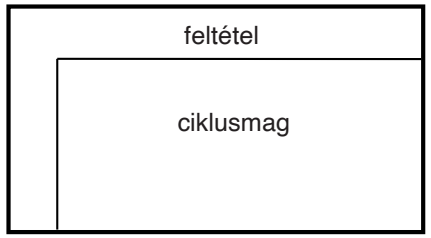
// feltétel //	
igen	nem
A - modul	B - modul

– Összetett alternatíva

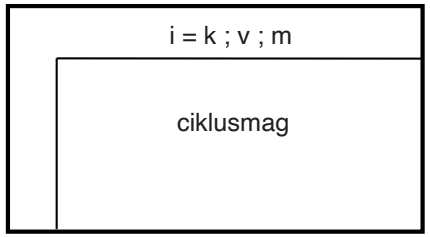
// feltétel(1) //		
igen	nem	
A(1) - modul	// feltétel(2) //	
	igen	nem
	A(2) - modul	A(3) - modul

Iterációs vezérlőtevékenység:

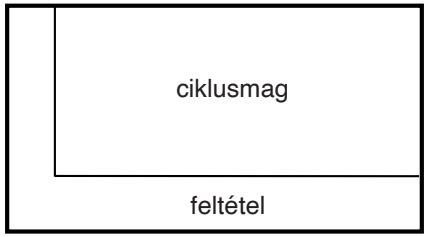
– Előltesztelő - WHILE típusú ciklus



– Előltesztelő - FOR típusú ciklus



– Hátultesztelő - DO - WHILE típusú ciklus



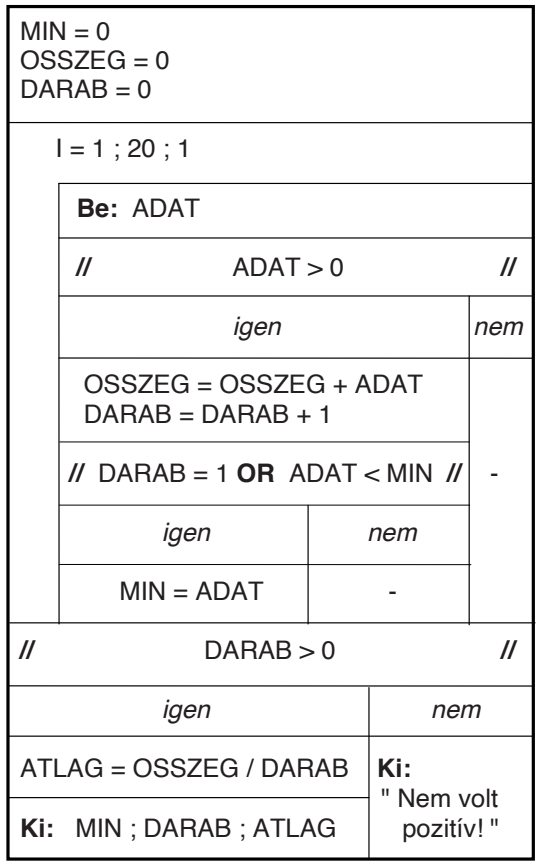
2.3.2 Példa struktogramra

Olvassunk (kérjünk) be 20 számot! Keressük ki a számsorozat pozitív eleme- inek minimumát, számítsuk ki ezek összegét és átlagát!

A feladatban felhasznált változók:

A változók: MIN – A pozitív számok minimum értéke
OSSZEG – A pozitív számok összege
DARAB – A pozitív számok darabszáma
ATLAG – A pozitív számok átlaga
I – Ciklusváltozó

A feladat struktogramja:

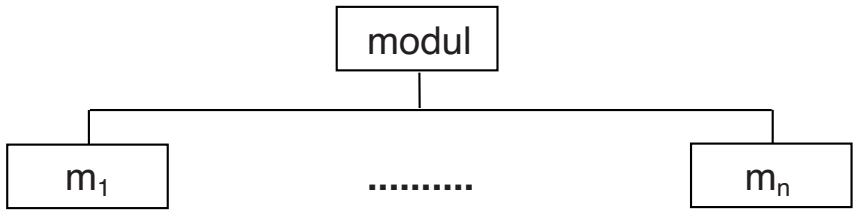


2.4 A szerkezeti ábra (Jackson-diagram)

A szerkezeti ábra modulokból épül fel. A modulok jelentik az elvég- zendő feladatokat. Jelölésére téglalapokat használunk, amelybe a modul nevét, vagy az elvégzendő feladatot írjuk.

2.4.1 A vezérlőtevékenységek megvalósítása szerkezeti ábrával

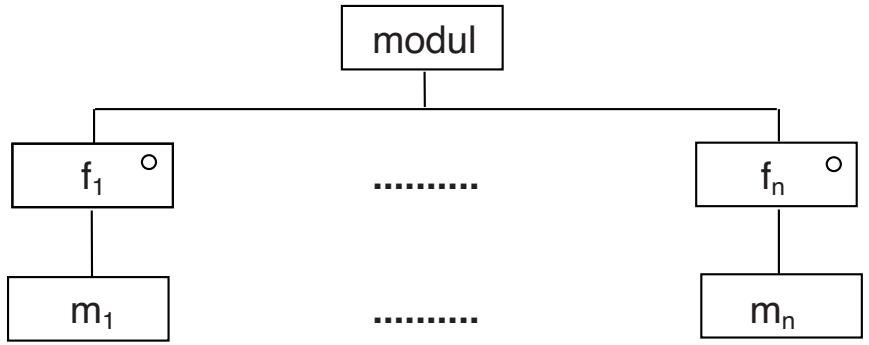
Szekvenciális vezérlőtevékenység:



Jelentése:

A **modul** nevű feladat úgy oldandó meg, hogy egymás után megoldjuk az $\mathbf{m_1}$**.....** $\mathbf{m_n}$ nevű feladatokat, vagy ha valamelyik mi modul alaptevé- kenység, azt végrehajtjuk.

Szelekciós vezérlőtevékenység:

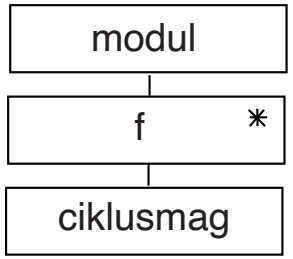


Jelentése:

A modul nevű feladatot úgy kell megoldani, hogy a feltételeket megvizsgálva, meg- oldjuk, vagy végrehajtjuk azt a modult, melynél az $\mathbf{f}_i$ feltétel igaz. Kikötés, hogy a feltételek közül csak egy, vagy egyik sem lehet igaz.

Iterációs vezérlőtevékenység:

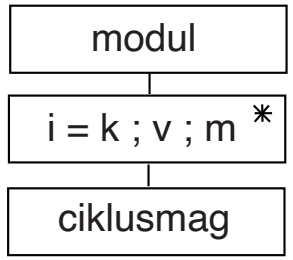
– Előltesztelő - WHILE típusú ciklus



Jelentése:

A **modul** nevű feladatot úgy oldjuk meg, hogy a **ciklusmagot** addig ismételjük, míg az **f** feltétel igaz.

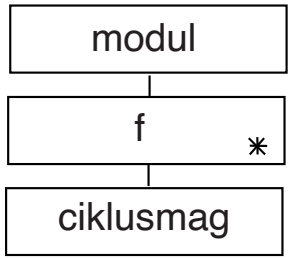
– Előltesztelő - FOR típusú ciklus



Jelentése:

A **modul** nevű feladat megoldása **i = k** értékkel kezdődik 2 értékig tart, és **m** értékkel módosul minden ciklusmag végrehajtás után.

– Hátultesztelő - DO - WHILE típusú ciklus



Jelentése:

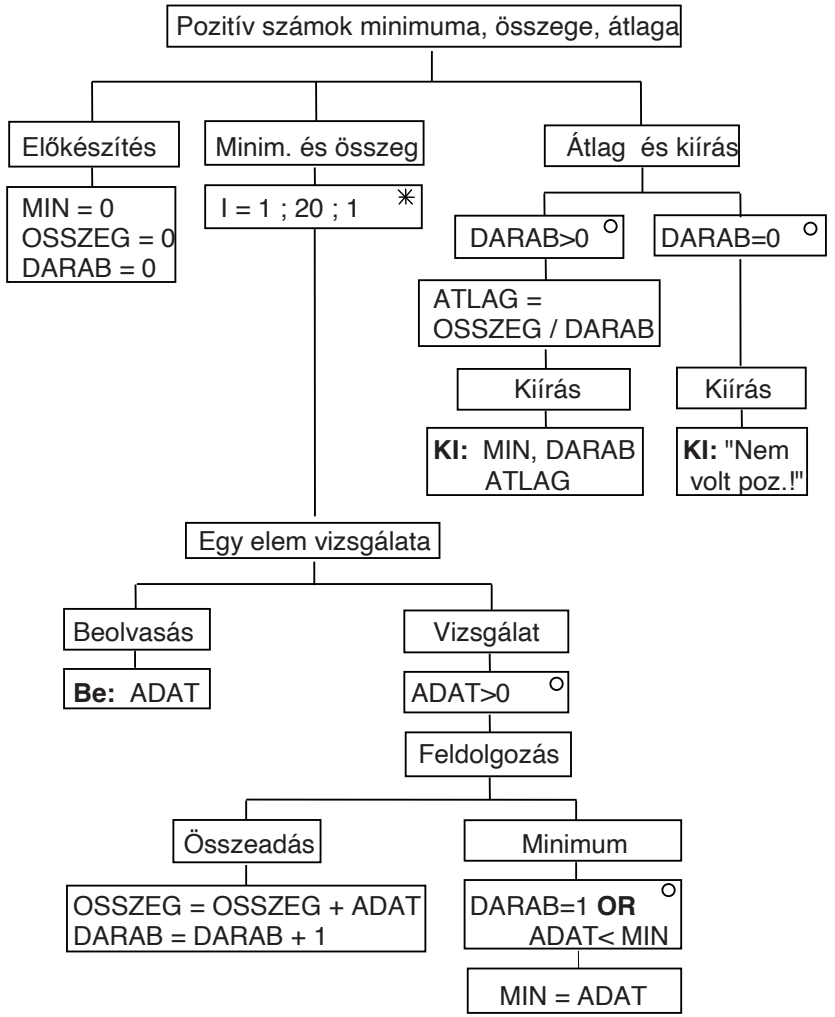
A **modul** nevű feladatot úgy oldjuk meg, hogy a **ciklusmagot** addig ismételjük, míg az **f** feltétel igaz.
A feltétel vizsgálat a ciklusmag végrehajtása után történik.

2.4.2 Példa szerkezeti ábrára

Olvassunk (kérjünk) be 20 számot! Keressük ki a számsorozat pozitív eleme- inek minimumát, számítsuk ki ezek összegét és átlagát!
A feladatban felhasznált változók:

MIN	– A pozitív számok minimum értéke
OSSZEG	– A pozitív számok összege
DARAB	– A pozitív számok darabszáma
ATLAG	– A pozitív számok átlaga
I	– Ciklusváltozó

A feladat szerkezeti ábrája:



3. Adatábrázolás

A számítógépi programok utasításai döntően valamilyen adatra vonatkozó műveletet hajtanak végre. Az adatnak a műveletvégzés idején a memóriában kell lennie, azaz valamilyen **"szabványos" formátumban a memóriában kell tárolni.**

Az adatok **felhasználási területük** szerint a következő csoportokba sorolhatók:

– Numerikus

– Logikai

– Karakteres

Az informatikában a megszokott tizes számrendszer mellett még további három számrendszerrel találkozhatnak:

Kettes (bináris)
Számjegyek: 0 , 1
Jelölés: 10101110_2

Tizenhatos (hexadecimális)
Számjegyek: 0-9,A,B,C,D,E,F
Jelölés: $1AF3_{16}$

Nyolcas (oktális)
Számjegyek: 0,1,2,3,4,5,6,7
Jelölés: 1732_8

3.1 Alapműveletek különböző számrendszerekben

Átalakítás decimálisból binárisba:

$51.75_{10} \longrightarrow 110011.11_2$

51 / 2	
25	1
12	1
6	0
3	0
1	1
0	1

	75 * 2
1	50
1	00

Átalakítás binárisból hexadecimálisba:

$1\ 0110\ 0010.001_2 \longrightarrow 162.2_{16}$

Összeadás (Bin):

$$\begin{array}{r} 1011101101 \\ 101110 \\ +\quad 11011 \\ \hline 1100110110_2 \end{array} \qquad \begin{array}{r} 749 \\ 46 \\ +\quad 27 \\ \hline 822_{10} \end{array}$$

Kivonás (Bin):

$$\begin{array}{r} 1011101101 \\ -\quad 11011111 \\ \hline 1000001110_2 \end{array} \qquad \begin{array}{r} 749 \\ -\quad 223 \\ \hline 526_{10} \end{array}$$

Kivonás komplemens szám segítségével (Bin):

1011101101 - 11011111 = ?

0011011111 Egyes komplemense: 1100100000
Kettes komplemense: + 1
1100100001

$$\begin{array}{r} 1011101101 \\ +\ 1100100001 \\ \hline \cancel{1}1000001110_2 \end{array}$$

Átalakítás decimálisból hexadecimálisba:

$51.75_{10} \longrightarrow 33.C_{16}$

51 / 16	
3	3
0	3

	75 * 16
12	00

Átalakítás hexadecimálisból binárisba:

$33.C_{16} \longrightarrow 11\ 0011.11_2$

Összeadás (Hex):

$$\begin{array}{r} 1AF \\ +\ \ 3A \\ \hline 1E9_{16} \end{array} \qquad \begin{array}{r} 431 \\ +\ \ 58 \\ \hline 489_{10} \end{array}$$

Kivonás (Hex):

$$\begin{array}{r} 2BA \\ -\ 1E2 \\ \hline D8_{16} \end{array} \qquad \begin{array}{r} 698 \\ -\ 482 \\ \hline 216_{10} \end{array}$$

Kivonás komplemens szám segítségével (Hex):

2BA - 1E2 = ?

1E2 Komplemense:

$$\begin{array}{r} E1D \\ +\ \ 1 \\ \hline E1E \end{array}$$

$$\begin{array}{r} 2BA \\ +\ E1E \\ \hline \cancel{1}0D8_{16} \end{array}$$

3.2 Adattípusok

A konkrét adattípusok ismertetése előtt érdemes még néhány fogalommal megismerkedni.

– Bájt (Byte)
A számítógépek legkisebb címezhető egysége.
A memóriában nyolc egymás melletti bit.

– Félszó (Half Word)
Két egymás melletti bájt.

– Szó (Word)
Négy egymás melletti bájt.

– Duplaszó (Double Word)
Nyolc egymás melletti bájt.

– Zónarész, Magas (High) bitek
Egy bájt balról számolt négy bitje. (Tetrád)

– Számrész, Alacsony (Low) bitek
Egy bájt jobbról számolt négy bitje.

Egy bájt értéke tehát a következő formában adható meg:

$$\underset{\substack{\textbf{Zónarész}\\\textbf{High}}}{1110}\quad \underset{\substack{\textbf{Számrész}\\\textbf{Low}}}{1010_2} = EA_{16}$$

Karakterek ábrázolása

– Az ASCII kódrendszert használja
(American Standard Code for Information Interchange)

– 7 bites, 8 bites változat

– Nemzeti karakterkészletek
(437, 852, 1250 kódlapok)

– Magyar szabvány: MSZ 7794-3

– Nemzetközi szabvány: ISO-8859-x
x = 2 Kelet Európa

– Több bájtos változat: (UNICODE, UTF-8)

Egész típusú adatok ábrázolása

– Fixpontos bináris típus

– A kettedespont a szám után képzelendő (Csak egész érték)

– A negatív számok kettes komplemens formában

– A bináris aritmetika számol vele

– A hossz alapján:
  – Szavas egész (2 bájt)
    A számábrázolási tartomány:

$$-2^{15} \quad - \quad +2^{15}-1$$
$$-32768 \quad - \quad +32767$$

A szám (Decimális)	**Memória tartalom** (Hexadecimális)
1000	03E8
3	0003
–1	FFFF
32767	7FFF
–32768	8000
0	0000

– Dupla egész (4 bájt)
A számábrázolási tartomány:

$$-2^{31} \quad - \quad +2^{31}-1$$

Több mint ±2 milliárd

A szám (Decimális)	Memória tartalom (Hexadecimális)
–1	FFFFFFFF
1	00000001

– Hosszú egész (8 bájt)
A számábrázolási tartomány:

$$-2^{63} \quad - \quad +2^{63}-1$$

A szám (Decimális)	Memória tartalom (Hexadecimális)
–1	FFFFFFFF FFFFFFFF
5	00000000 00000005

Valós típusú adatok ábrázolása

– Minden szám felírható a következő formában:
$$\pm m^{*}r^{\pm e}$$

– Ahol m a mantissza, r a számrendszer alapszáma (radixa), **e** a kitevő (exponens)

– Bináris számrendszerben tehát:
$$\pm m^{*}2^{\pm e}$$

– A törtszámok is ábrázolhatók

– Az exponens 8, 11, 15 bit, típustól függően

– A tárolás normalizált formában:
$$\pm 1.m^{*}2^{\pm e}$$

– IEEE 754 szabvány

– Tudományos, gazdasági számításokhoz

– A lebegőpontos aritmetika használja

– A hossz alapján:
  – Egyszeres pontosságú lebegőpontos szám (Single precision, 4 bájt)
    – A számábrázolási tartomány:
      $\pm 10^{-38}$ - $\pm 10^{+38}$
    – A pontossága: 6–7 decimális jegy
  – Felépítése:

Me Exponens (e_k) Mantissza (m)

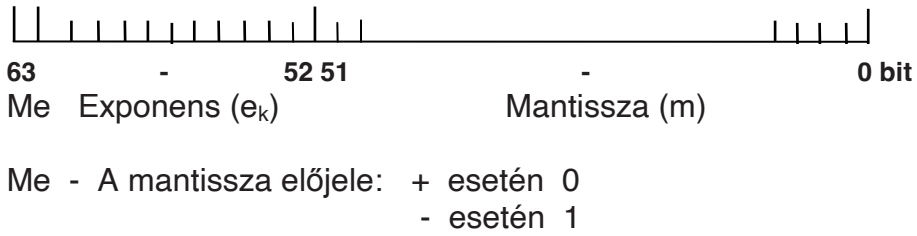
Me - A mantissza előjele: + esetén 0
- esetén 1

Az átalakítás lépései példán bemutatva:

a. A decimális szám átalakítása binárissá
   $26.75_{10} \longrightarrow 11010.11_2$
b. Normalizálás
   $11010.11 \longrightarrow 1.101011 * 2^4$

c. Az exponens korrigálása (nulla pont eltolás)
   $e_k = e + 127 = 4 + 127 = 131_{10} = 83_{16}$
d. Ábrázolás
   0 10000011 10101100000000000000000
   4 1 D 6 0 0 0 0

– Dupla pontosságú lebegőpontos szám
(Double precision, 8 bájt)
  – A számábrázolási tartomány:
    $\pm 10^{-308}$ – $\pm 10^{+308}$
  – A pontossága:
    15–16 decimális jegy
  – A korrigált exponens:
    e_k =e+1023
– Felépítése:



– Kiterjesztett pontosságú lebegőpontos szám
(Extended precision, 10 bájt)

A teljes hossz 80 bit, amiből 1bit mantissza előjel, 15 bit az exponens, és 64 bit a mantissza.
  – A számábrázolási tartomány:
    $\pm 10^{-4932}$ – $\pm 10^{+4932}$
  – A pontossága:
    18 – 19 decimális jegy
  – A korrigált exponens:
    e_k =e+16385

A felhasznált irodalom

1. N. Wirth: Algoritmusok + adatstruktúrák = programok
könyv
Műszaki Könyvkiadó, Budapest

2. Lipschutz: Adatszerkezetek
könyv
Panem Kft, Budapest

3. Dr. Marton László - Pukler Antal - Pusztai Pál: Bevezetés a programozásba
könyv
NOVADAT BT, Győr

Bevezetés a programozásba

Debugolás a Netbeans segítségével

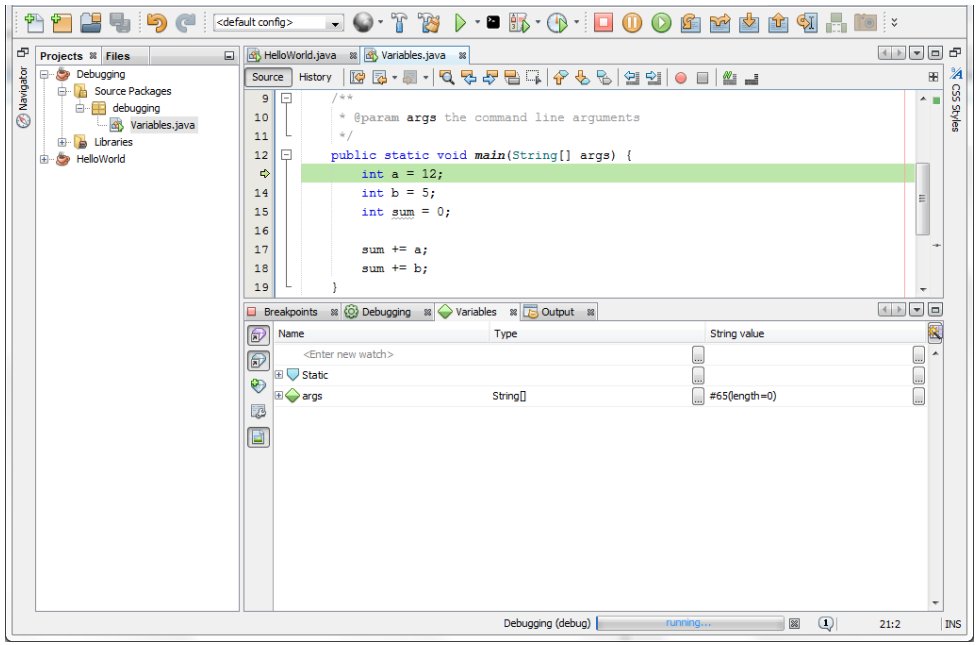
Dunaújvárosi Főiskola,
Informatikai Intézet

Debugolás a Netbeans segítségével

A debugolás folyamata (programozás során) a program helytelen működésének okait feltáró/megelőző tevékenység. A debugolás során utasításról-utasításra (~kódsorról-kódsorra) végrehajtattatjuk a programunkat és valós időben követhetjük változóik aktuális értékeit, melynek segítségével a szemantikai hibák gyorsabban kijavíthatóak. Szerencsére a modern fejlesztőkörnyezetek fejlett debugolást támogató lehetőségekkel vannak felvértezve, amely alól a Netbeans sem képez kivételt.

Debugolás elindítása

Minden hibakereséssel kapcsolatos parancs a főmenü Debug menüpontja alatt érhetők el (ALT + D), azonban érdemes megtanulni a gyorsbillentyűket, a parancsok lehető leggyorsabb kiadása végett. A hibakeresést a Debug ⇨ Step Into (~bele lépés)/F7 paranccsal kezdhetjük meg. A zöld színnel kiemelt sor jelenti az aktuális sort, amelyben található utasítást **még nem hajtotta végre** a program *(1. ábra)*.



1. ábra - Debugolás elindítása

A debugolás indítását követően megjelenik az Output Windows mellett/felett három ablak:

– Breakpoints: töréspontok megjelenítése

Bővebben: Amikor végrehajtattuk az utolsó utasítást is, az aktuális sor a main függvény záró kapcsos zárójelére ugrik. Itt Step Over parancsot kiadva befejezhetjük a programunk végrehajtását. Időközben is befejezhetjük a programunk debugolását a Finish Debugger Session (SHIFT + F5) paranccsal.

Utasítás végrehajtás folytatása

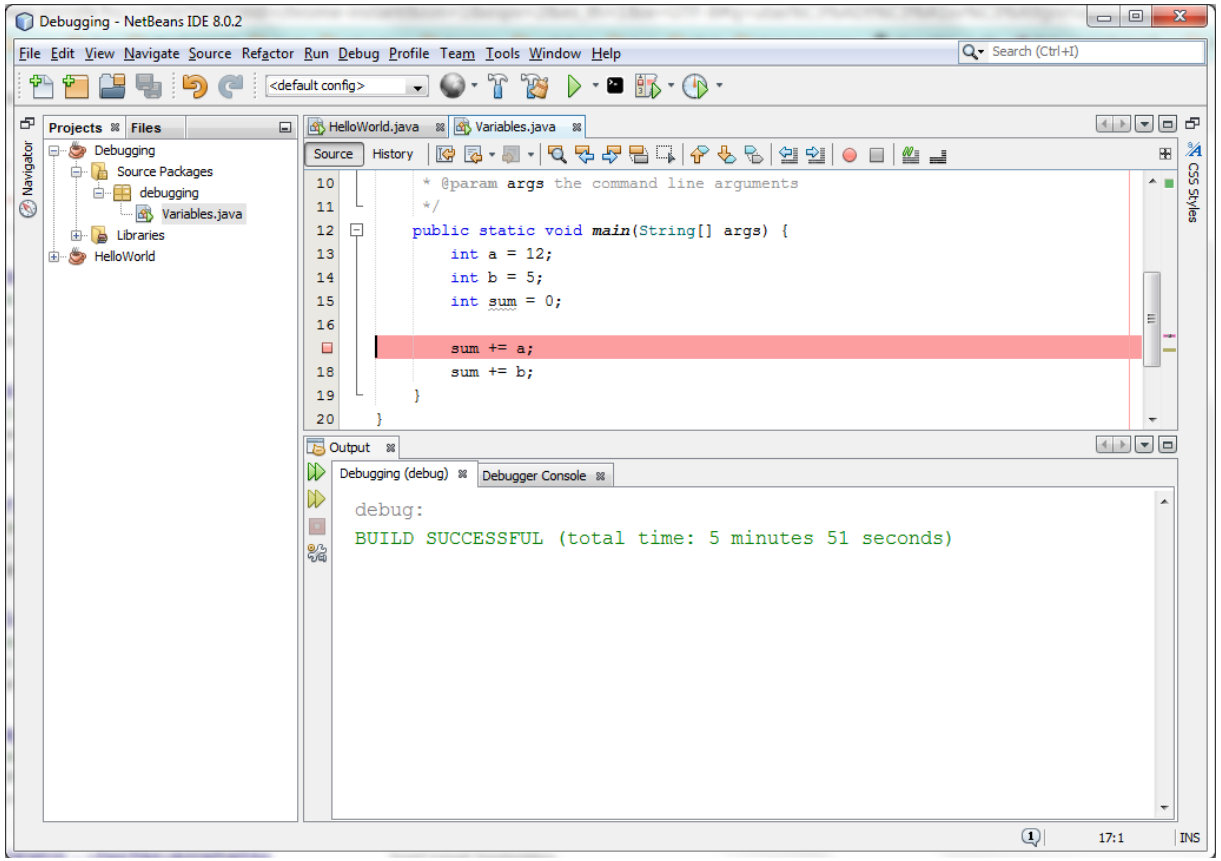
Előfordulhat, hogy a programkódunk elején leellenőriztük a szükséges változóink értékeit, viszont szükséges lefuttatnunk a teljes programot, azonban nem szeretnénk utasításról-utasításra végiglépkedni rajta. Ilyen esetben használhatjuk a Continue (F5) parancsot, amely folyamatában végrehajtja az utána következő utasításokat. Azonban az utasítás végrehajtás megállítható töréspontokkal.

Töréspontok

Számos alkalommal előfordul, hogy a programunk egy adott részén szeretnénk megvizsgálni a változók értékeit. Amennyiben nem nyúlfarknyi példaprogramokról, hanem „valódi” több ezer/tízezer/… kódsorból álló programok bizonyos részeit szeretnénk megvizsgálni. Ilyenkor nem túl célszerű utasításról-utasításra végrehajtani a programkód irreleváns részét.

Ilyen esetekben vannak nagy segítségünkre a töréspontok (breakpoints). Töréspontot az adott sor sorszámára kattintva, vagy a Debug ⇨ Toogle Line Breakpoint lehetőséget választva, illetve a CTRL + F8 billentyűkombinációt használva helyezhetünk el *(3. ábra).*

3. ábra – Töréspont



A töréspontok sora piros színnel kerül kiemelésre. Az első/következő töréspontra futtatáshoz válasszuk a Debug ⇨ Debug Project lehetőséget, vagy használjuk CTRL + F5 billentyűkombinációt, illetve kattintsunk a futtatás ikontól jobbra (nálatok eggyel, az itteni képernyőképeken kettővel) található ikonra kattintva. Ennek eredményeként az utasítás végrehajtás a töréspontra fut és természetesen az előtte lévő összes utasítás végrehajtásra kerül.

A töréspontok kikapcsolhatóak, a sorukban állva, ugyan azon a módok valamelyikén, ahogy elhelyeztük őket.

o Függvények debugolása fejezet

– Debugging: hívási verem/fa megjelenítése

o Függvényhívások során biztosít releváns információt

– Variables: az eddig deklarált változók értékeit jeleníti meg számunkra

o Az első utasítás (pontosabban az első változó deklarálása/definiálása előtt csak az args változó értékét láthatjuk (mely a parancssorban – esetlegesen – átadott paramétereket tartalmazza)

o Mi ezt az ablakot fogjuk legtöbbször használni

A debugolási parancsok a következő módokon elérhetőek el:

– Debug menü

– Gyorsbillentyűk

– Menüsor alatti ikonsor jobb oldalán

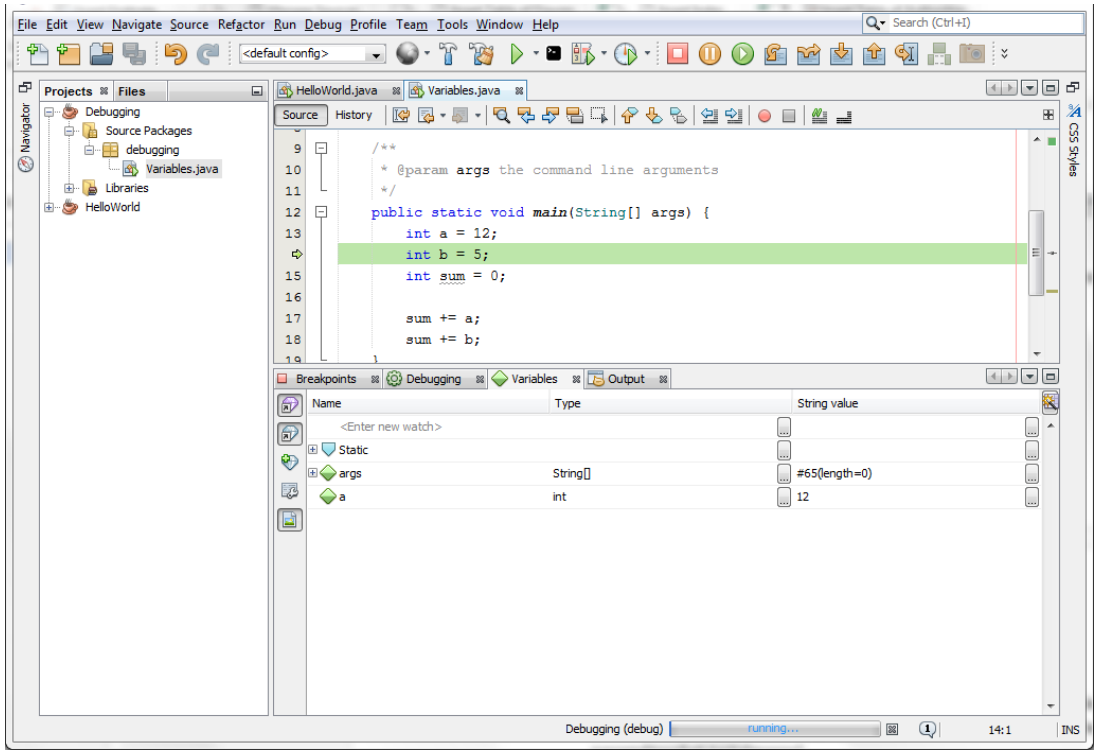
o Piros STOP, Sárga PAUSE, Zöld Continue és a léptetőműveletek

o **Csak a debugolás elindítását követően jelennek meg**

Utasítások végrehajtása

A programkódot utasításról-utasításra ugyancsak a Step Into, illetve a Step Over (átugrás, F8) paranccsal hajthatjuk végre. (A két parancs kimenete csak függvények végrehajtásakor különbözik.) Ilyenkor megjelennek a deklarált változók a Variables ablakban, valamint figyelemmel kísérhetjük értékeik változásait *(2. ábra)*.

2. ábra - Variables window



Amikor végrehajtattuk az utolsó utasítást is, az aktuális sor a main függvény záró kapcsos zárójelére ugrik. Itt Step Over parancsot kiadva befejezhetjük a programunk végrehajtását. Időközben is befejezhetjük a programunk debugolását a Finish Debugger Session (SHIFT + F5) paranccsal.

Utasítás végrehajtás folytatása

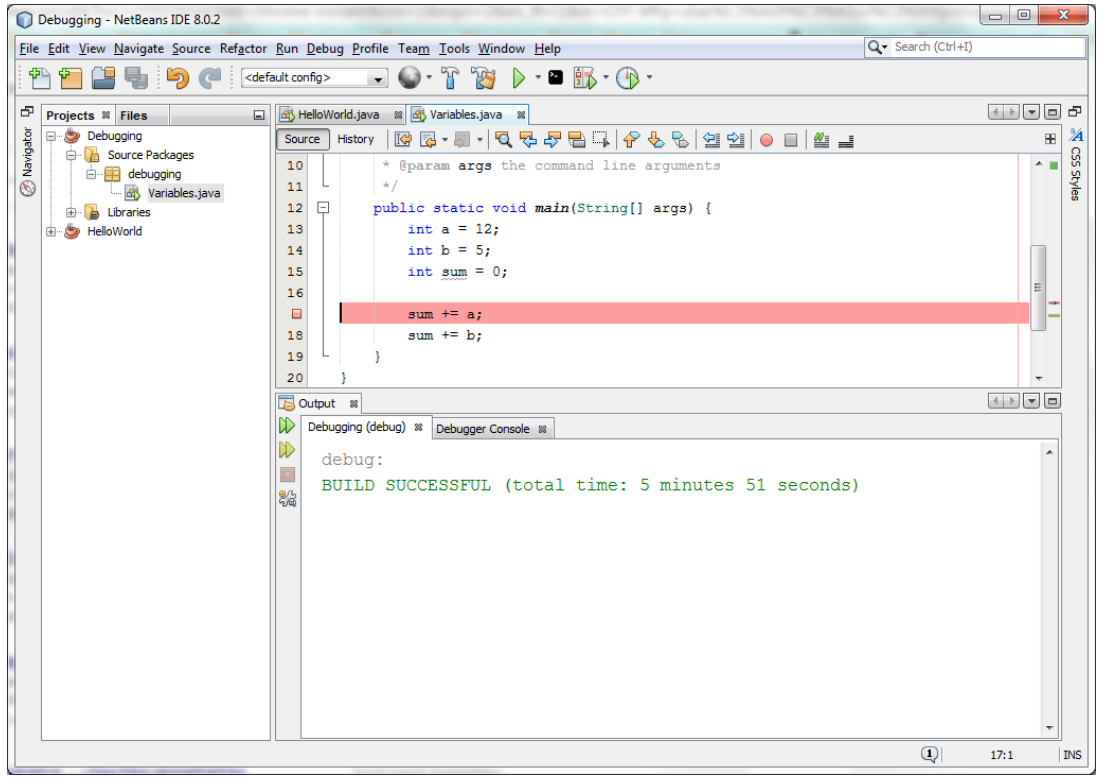
Előfordulhat, hogy a programkódunk elején leellenőriztük a szükséges változóink értékeit, viszont szükséges lefuttatnunk a teljes programot, azonban nem szeretnénk utasításról-utasításra végiglépkedni rajta. Ilyen esetben használhatjuk a Continue (F5) parancsot, amely folyamatában végrehajtja az utána következő utasításokat. Azonban az utasítás végrehajtás megállítható töréspontokkal.

Töréspontok

Számos alkalommal előfordul, hogy a programunk egy adott részén szeretnénk megvizsgálni a változók értékeit. Amennyiben nem nyúlfarknyi példaprogramokról, hanem „valódi” több ezer/tízezer/… kódsorból álló programok bizonyos részeit szeretnénk megvizsgálni. Ilyenkor nem túl célszerű utasításról-utasításra végrehajtani a programkód irreleváns részét.

Ilyen esetekben vannak nagy segítségünkre a töréspontok (breakpoints). Töréspontot az adott sor sorszámára kattintva, vagy a Debug ⇨ Toogle Line Breakpoint lehetőséget választva, illetve a CTRL + F8 billentyűkombinációt használva helyezhetünk el *(3. ábra).*

3. ábra – Töréspont



A töréspontok sora piros színnel kerül kiemelésre. Az első/következő töréspontra futtatáshoz válasszuk a Debug ⇨ Debug Project lehetőséget, vagy használjuk CTRL + F5 billentyűkombinációt, illetve kattintsunk a futtatás ikontól jobbra (nálatok eggyel, az itteni képernyőképeken kettővel) található ikonra kattintva. Ennek eredményeként az utasítás végrehajtás a töréspontra fut és természetesen az előtte lévő összes utasítás végrehajtásra kerül. A töréspontok kikapcsolhatóak, a sorukban állva, ugyan azon a módok valamelyikén, ahogy elhelyeztük őket.

Függvények debugolása

Függvényhíváskor a függvények utasításának végrehajtásához a programnak át kell ugrania a függvény törzsére. Amennyiben egy függvényhíváson állva Step Over parancsot adunk ki, a függvény törzsébe nem lépünk be, csupán végrehajtjuk a háttérben. Amennyiben a Step Into parancsot adjuk ki, belépünk a függvény törzsébe, ahol utasításról-utasításra végigléptethetjük a függvényt. Akkor érdemes belépnünk egy függvény törzsébe, ha kíváncsiak vagyunk a függvény lokális változóinak értékváltozásaira. Amennyiben irrelevánsak számunkra, csak a visszatérési érték lényeges, esetleg eljárásról van szó (a „függvénynek” nincs visszatérési típusa), akkor nyugodtan átugorhatjuk.

Amennyiben véletlenül beléptünk a függvénybe, vagy már kigyűjtöttük belőle a lényeges információkat, a Step Out (kiugrás) paranccsal futhatunk ki az aktuális függvényből.

Fontos megemlíteni a Step Over Expression parancsot is, amely egymásba ágyazott függvényhívások, illetve feltétel kiértékelésekben halmozott függvényhívások során válhat hasznunkra, abban az esetben, ha nem mindegyiket szeretnénk végig debugolni. A parancs balról jobbra fog átfutni a paraméterek helyén szereplő függvényhívásokon.

Kurzorhoz futtatás

A Run To Cursor parancsot kiadva az utasítások végrehajtását a kurzor aktuális helyzetéig folytatja a Netbeans.

Dunaújváros, 2015 .07. 09 Somlyai Pál

Bevezetés a programozásba

Java programok fordítása és futtatása

Dunaújvárosi Főiskola,
Informatikai Intézet

Java programok fordítása és futtatása

A számítógépes programok készítésének első lépése a forráskód elkészítése. Később, a forráskód fordítását követően áll elő a számítógép által végrehajtható parancsokat tartalmazó, úgynevezett futtatható alkalmazás. (Általános leírás, nem minden esetben pontosan így zajlik.)

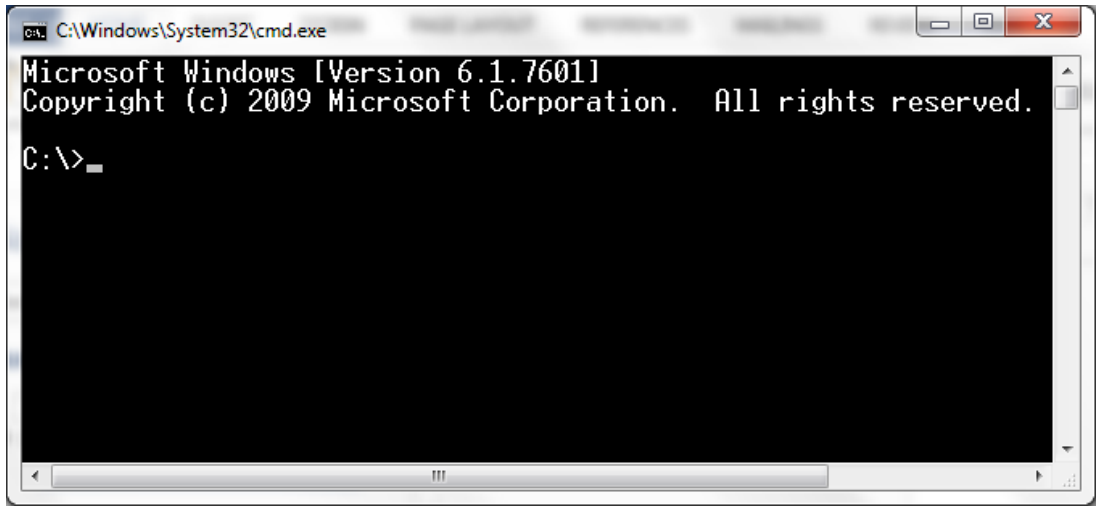
Manapság számos IDE-hez férhetünk hozzá, amelyek egy kattintást követően a Java forráskódunkat futtatható állománnyá képesek fordítani, majd lefuttatni az előállított fájlt. Természetesen a tantárgy keretein belül IDE segítségével fogjuk elvégezni a fordítást és futtatást, mégsem nélkülözhetjük ezen műveletek manuális végrehajtásának ismeretét.

Forráskódtól a futtatható alkalmazásig

Fordítás

A Java forráskódok kivétel nélkül .java kiterjesztésű fájlokban szerepelnek. Egy java forráskód lefordításához (Windows rendszeren) a következőket kell tennünk:

1. Nyissuk meg a Start menüt/Kezdőképernyőt, majd kezdjük el gépelni a „cmd” szöveget. Amint megjelent a "cmd.exe” találat, nyomjunk entert. A következőhöz nagyon hasonló ablaknak kell megjelennie.



2. Ezt követően keressük meg a „javac.exe” nevű fájlt, a „Program Files” könyvtár egyik alkönyvtárában. Kezdjük el gépelni a cmd-ben a következő szöveget: „C:\Prog”, majd nyomjuk le a tabulátor billentyűt. Ennek eredményeként a következő képernyőnek kell fogadnia:

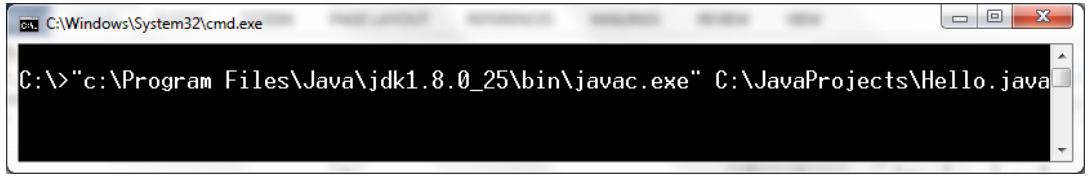
A tabulátor hatására az eddig beírt elérési útra elsőként egyező könyvtárnév jelenik meg, újbóli megnyomására a második és így tovább. (Fordított sorrendben a SHIFT + TAB együttes lenyomásával „léptethetünk”.) Figyeljük meg, hogy az elérési út idézőjelek közé került. Ennek az oka, hogy a könyvtárnév szóközt tartalmaz. Ilyen esetben mindig idézőjelek között kell megadni a könyvtár nevét.

Folytassuk az elérési út megadását:

– \Java, majd tabulátor
– \jdk, majd tabulátor
– \bin, majd tabulátor
– \javac, majd tabulátor

Ennek eredményeként eljutottunk a javac.exe fájlhoz, amelyik a Java fordító fájl (Java Compiler).

3. Írjunk be egy szóközt, majd kezdjük el a tabulátor segítségével megadni a java forrásfájl elérési útját. Egy lehetséges eredmény a következő képernyőképen látható:



Majd nyomjuk le az enter billentyűt. A parancs kiadásának hatására (amennyiben nem látunk hibaüzentet), a forrásfájl mellett létre kell jönni egy <forrásfájlNeveKiterjesztésNélkül>.class fájlnak (pl.: Hello.class). Ez a „futtatható állományunk" amelyet kizárólag egy speciális paranccsal tudunk futtani.

Futtatás

Első lépésként lépjünk át a forrásfájl könyvtárába a cd parancs segítségével (pl.: cd C:\JavaProjects). Amennyiben a fájl egy másik meghajtón helyezkedik el (pl.: E:\), a cd parancs és az elérési út közé helyezzünk el egy „/D" úgynevezett kapcsolót. (pl.: cd /D E:\DUF\Prog\Hello.java)

Gépeljük be a parancssorba „java" parancsot követően a .class kiterjesztésű fájl nevét a kitejesztés nélkül (Hello.class => java Hello), majd nyomjuk le az enter billentyűt. Ennek hatására meg kell jelennie a kívánt kimenetnek.

Leggyakrabban felmerülő problémák

Amennyiben mégsem jelent meg a kívánt kimenet, akkor nagy eséllyel a következő problémák valamelyike (peches napon többük együttesen) lépett fel:

– A forrásfájlban található fájl neve nem egyezik meg a forrásfájl nevével, a .java kiterjesztés nélkül.
  o Javítsuk az osztály nevét, adjuk ki újra a fordítási parancsot, majd a futtatásit is. A parancssorban állva a felfelé- és lefelé kurzormozgató billentyűkkel navigálhatunk az eddig beírt parancsok között.

– Hibás elérési utat adtunk meg a .class fájlhoz vezetően.
  o Javítsuk az elérési utat.

– A „java” parancs és az elérési út közé nem helyeztünk szóközt
  o Tegyük meg

– „A java parancs nem feleltethető meg belső vagy külső parancsnak illetve kötegfájlnak.” hibaüzenethez nagyon hasonló hiba jelent meg.
  o A java parancs helyett gépeljük be a javac.exe szülőkönyvtárának elérési útját, majd egészítsük ki java.exe-vel. (pl.: „c:\Program Files\Java\jdk1.8.0_25\bin\java.exe”). Futtassuk a parancsot.

Dunaújváros, 2015. 07. 02

Somlyai Pál

Bevezetés a programozásba

JDK telepítés

Dunaújvárosi Főiskola,
Informatikai Intézet

JDK telepítés

A Java fejlesztéshez és később a Netbeans telepítéséhez is szükségünk lesz a Java Development Kit-re (JDK, Java Fejlesztési Eszköztár), ezért telepítsük fel számítógépünkre a JDK-t.
JDK telepítése

Töltsük le a legfrissebb JDK-t, amely jelen dokumentum készítésének pillanatában az alábbi linkről tölthető le:

http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Az első táblázat fejlécében válasszuk az „Accept License Agreement" lehetőséget. Ezt követően ugyancsak ebből a táblázatból válasszuk ki az operációs rendszerünkhöz megfelelő lehetőséget:

– Windows rendszer esetén:
  - o 32 bites rendszer esetén: Windows x86 sorában található letöltési link
  - o 64 bites rendszer esetén: Windows x64 sorában található letöltési link

Hogyan tudhatom meg, hogy 32, vagy 64 bites Windows rendszert futtatok?

– Gyorsbillentyű: Windows + Pause, vagy

– Start menü -> Vezérlőpult -> Rendszer és biztonság -> Rendszer

Futtassuk a letöltött fájlt. A telepítést végezhetjük „Next-next-finish" módon, azaz nem kell egyedi beállításokat végeznünk és nem fognak kéretlen böngésző-eszköztárak települni alatta(legalább is a dokumentum készítésének pillanatában a telepítő így működött).

Bevezetés a programozásba

Java kódolási konvenciók

Dunaújvárosi Főiskola,
Informatikai Intézet

Java kódolási konvenciók

A kódolási konvenciók a programkód szerkesztési szabályait meghatározó megállapodás. A Dunaújvárosi Főiskolán legnagyobb arányban a Java közösség által meghatározott konvenciókat fogjuk alkalmazni (http://www.oracle.com/technetwork/java/codeconvtoc-136057.html).

Változók

A változókat camelCase (~púpos írásmód) módon nevezzük el, például:

– int elsoSzam;

– int i;

– int hosszuNevuValtozo;

Az írásmód használata egyszerű: az összetartozó szavakból, mondatrészekből elhagyjuk a betűközöket/szóközöket, majd az második szótól kezdődően a szavak kezdőbetűjét nagybetűs megfelelőjükre cseréljük. A kéttagú mássalhangzók első karakterét emeljük csak nagybetűssé. Tartózkodjunk a rövidítések nagybetűs jelölésének megtartásától, például:

– String ~~HTMLKod~~; => String htmlKod;

Konstansok

A konstansok nevét UPPER_SNAKE_CASE módon jelöljük, például:

– final int ORA_PERCEI = 60;

Szóközök használata az olvashatóság végett

Az operátorok, tömbelemek és függvény paraméterek és argumentumok közé mindig helyezzünk el egy szóközt is. Például:

– operátorok:
    o aritmetikai:
        – int sum = a + b;
    o logikai:
        – if(a > 12)

– tömbelemek:
    o int vector = {4, 5, 10, 51};

– függvény paraméterek:
    o public static void add(int a, int b)

– függvény argumentumok:
    o int sum = add(5, 12);

Ezek a szóközök segítenek jobban elhatárolni a kifejezések elemeit, így gyorsabban értelmezhetünk egy összetett aritmetikai, vagy logikai műveletet, függvényhívást, stb.

Utasításblokkok

Az utasításblokkok kezdete előtt ne helyezünk el sortörést, csak egy szóközt, viszont utánuk sortörést alkalmazunk. Az utasításblokk zárását követően helyezzünk el sortörést, amennyiben utasítás és nem kulcsszó következik.

Például:

```
if(szam % 2 == 0) {
      System.out.print("Páros");
} else {
      System.out.print("Páratlan");
}
System.out.println();
```

Behúzások

Minden logikailag egy szinthez tartozó utasítást (~amelyek egy végrehajtási ágban szerepelnek) egymás alatti oszlopban kezdünk. Amikor a logika elágazik, vagy iterációba kezd stb, egy szinttel beljebb kezdjük (egy Tabulátor szélességgel) a következő utasítást.

Például:

```
System.out.print("-------");
for(int i = 1; i < 11; i++) {
     if(szam % 2 == 0) {
           System.out.print("Páros");
     } else {
           System.out.print("Páratlan");
     }
}
```

Sorhossz

Tartózkodjunk a 80 karakternél hosszabb soroktól, melynek okai a következőek:

– Alacsonyabb felbontású monitoron, nagyobb betűmérettel valószínűleg horizontálisan görgetni kell a kód olvasása közben
o kényelmetlen és zavaró

– Több terminálablak 80 karakter széles

– Normál, vagy magas felbontáson elhelyezhetünk egymás mellett két forráskódot (két külön szerkesztőben), így könnyebb összehasonlítani a két kódot, esetleg módosítani egy régebbi kódot, felhasználni egy más által megírt programot, stb.

– A tartalom összehasonlító eszközök is egymás mellett jelenítik meg az összehasonlított két fájl tartalmát

– A képernyő egyik felén elhelyezhető a kódszerkesztő, a másikon Java dokumentáció vagy egy magyarázó videó

– Stb.

Szerencsére a Netbeans szerkesztőterületén található függőleges piros vonal pont ezt a határt jelöli (igazából a 81. karaktert követően húzódik).

Az ideális maximum sorhossz 70 karakter, szóval törekedjünk rá, hogy ezt tartsuk, de a 80 karaktert semmi képen se lépjük túl.

Hosszú, összetett műveletek és karakterláncok tördelése

Ha egy sorban nem fér egy összetett művelet (akár logikai, akár aritmetika) a műveletjelet a következő sorban helyezzük el.

```
if(beolvasottValtozo > 0 && beolvasottValtozo % 3 == 0
   && beolvasottValtozo % 5 == 0) {

    System.out.println("Hárommal és öttel maradék nélkül osztható "
          + "egész szám");
}
```

Ugyan ez igaz egy hosszú karakterlánc tördelésére is: a konkatenálás operátor a következő sor elejére kerüljön. Amikor nehezen átlátható végeredményt kaptunk, próbáljuk meg egy sortöréssel megjelölni az utasítások határát.

Függvények

A függvényeket a változókkal megegyező módon nevezzük el. A függvénytörzs kezdő utasításblokkját tartsuk a függvényfej sorában.

Hosszú függvényhívás tördelése

A hosszú függvényhívások során a paraméterlistát tördeljük, méghozzá a paramétereket elválasztó vesszőt követően.

Például:

```
int eredmeny = hosszuFuggveny(egyHosszuArgumentumNeve,
                    egyMasikHosszuArgumentumNeve);
```

A fenti példa a Java ajánlás, viszont véleményem szerint sokkal olvashatóbb és így könnyebben értelmezhetőbb a következő formában, főként 3-4 paraméter esetén:

```
int eredmeny = hosszuFuggveny(
                    egyHosszuArgumentumNeve,
                    egyMasikHosszuArgumentumNeve
                );
```

Dunaújváros, 2015. 07. 10

Somlyai Pál

Bevezetés a programozásba

Netbeans alapfunkciók

Dunaújvárosi Főiskola,
Informatikai Intézet

Netbeans alapfunkciók

Mint minden mesterembernek, a programozóknak is ismerniük kell az eszközeiket. A programozók elsődleges eszköze az Integrált Fejlesztőkörnyezetük, amivel a programkódot készítik. Jelen dokumentum a Netbeans IDE 8.0.2 alapfunkcióinak bemutatására hivatott.

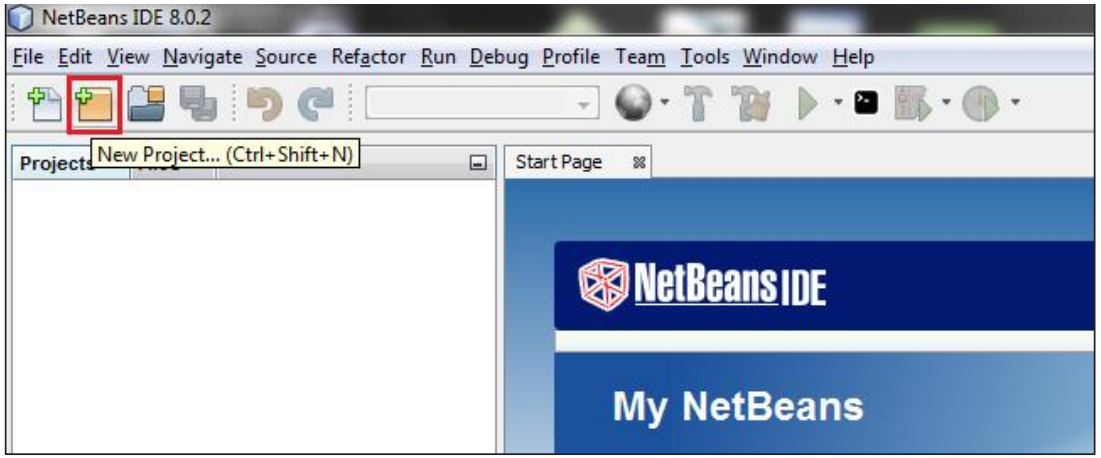
Új projekt létrehozása

A Netbeans (alapértelmezetten) minden programot egy külön projektként kezel. Tehát amikor új programot szeretnénk készíteni, szükségünk van egy új projektre, amit a következőképpen hozhatunk létre:

1. File ->New Project, vagy

2. CTRL + SHIFT + N gyorsbillentyű-kombináció, illetve

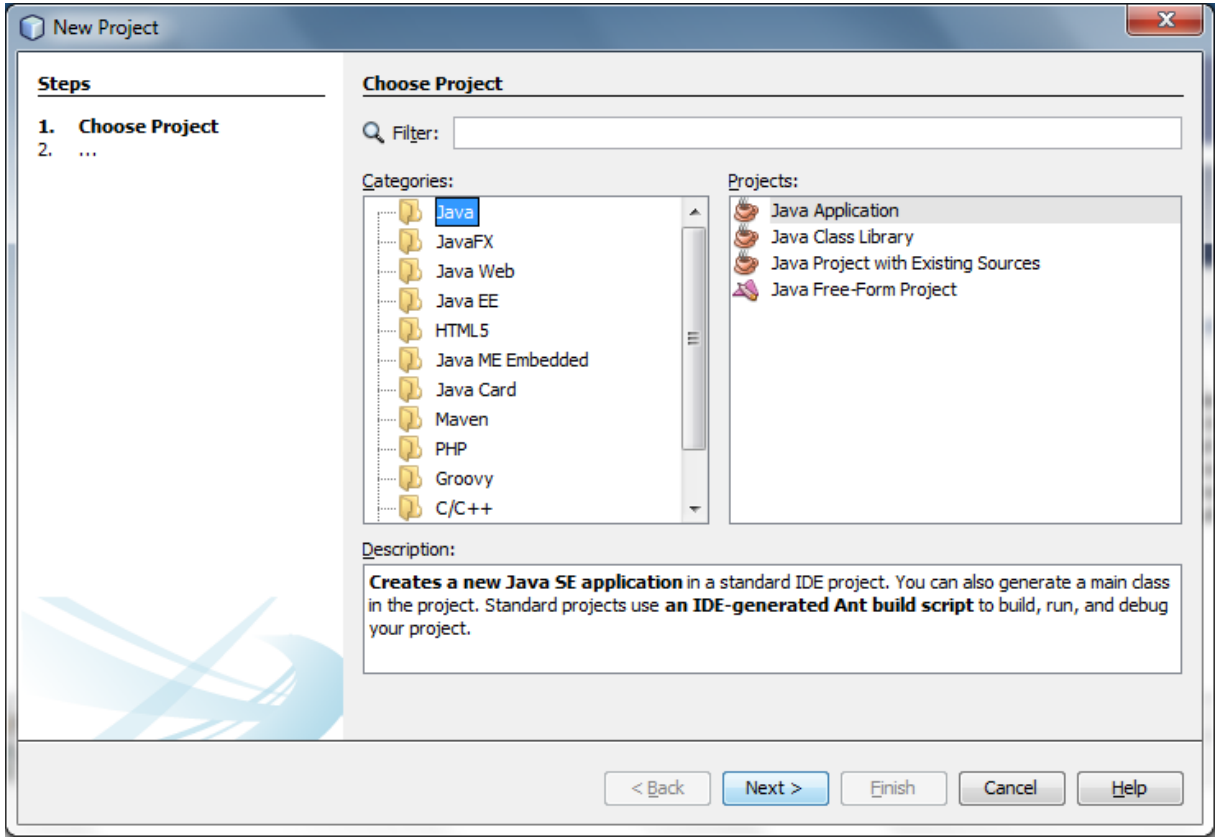
3. Az új projekt létrehozása ikonra kattintva *(1. ábra)*



1. ábra - Új projekt létrehozása

Bármelyik lehetőséget választjuk, a következő ábrán *(2. ábra)* látható ablak fog megjelenni.

2. ábra - Projekttemplate kiválasztása

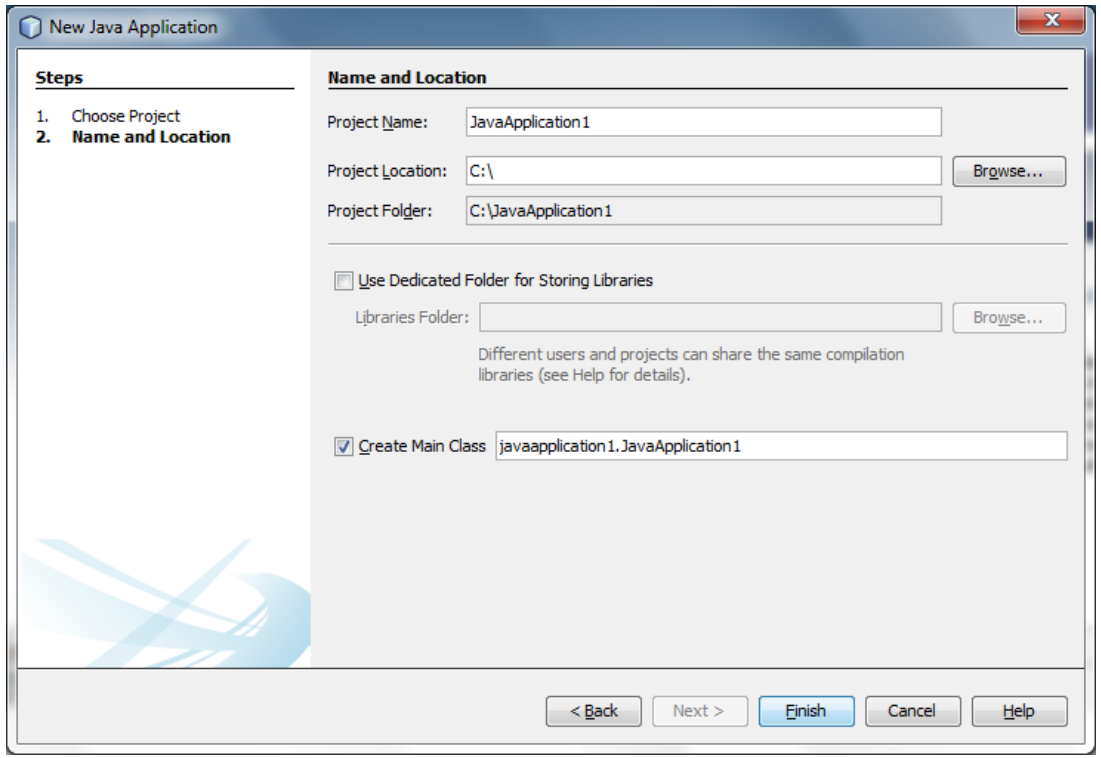


Nálatok valószínűleg kevesebb kategória jelenik meg (amennyiben a Netbeans Java SE csomagot telepítettétek, de abban a csomagban is szerepel az általunk használt Projekttemplate).

Bal oldalon választhatjuk ki a Projekttemplate[1] kategóriát, jobb oldalon pedig a kiválasztott kategórián belüli, konkrét Projekttempalte-et. Egy Projekttemplate kiválasztásával a Netbeans előre elkészített beállításokkal építi fel a projektünket, valamint létrehozza a tempate-hez szükséges forrásfájlt.

Mi ezen tantárgy keretein belül kizárólag a Java kategóriát és a Java Application projektet fogjuk használni. Válasszuk ki az említett lehetőségeket, majd kattintsunk a „Next" gombra, aminek hatására az új projekt létrehozása varázsló a következő lapra lép *(3. ábra)*.

3. ábra - Új projekt beállítása



1 A template lényegében sablont jelent.

Ezen a lapon van lehetőségünk beállítani a következő adatokat:

– Project Name:
  - o A projektünk neve
  - o Ezen a néven fog megjelenni a Projects ablakban a létrehozott projektünk, valamint
  - o A kiválasztott könyvtárban egy ezzel megegyező nevű alkönyvtárat fog létrehozni a projekt számára
  - o Minden esetben használjunk olyan projektnevet, amivel a későbbiekben könnyen beazonosíthatjuk a programunkat
    - – Ne elégedjünk meg a Netbeans által felkínált lehetőséggel, mely a későbbiekben nem fogja megfelelően azonosítani a projektünket

– Project Location:
  - o A projektünk helye a fájlrendszerben
  - o Az itt megadott könyvtárban fogja létrehozni a projekt könyvtárát

– Use Dedicated Folder for Storing Libraries
  - o Külön könyvtár használata a külső programkönyvtárak használatára
  - o Későbbi tárgyak során kerülhet sor a használatára
  - o Jelen tárgy keretein belül sose legyen kiválasztva

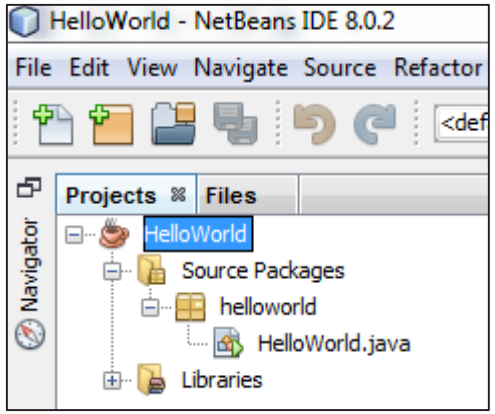
– Create Main Class
  - o Fő osztály létrehozása
  - o Egy olyan osztály létrehozása, amely tartalmaz egy main függvényt (a program belépési pontját), így ezen osztály lesz a programunk indítóosztálya
  - o Minden esetben válasszuk ki, így nem kell begépelnünk az osztályunk alapszerkezetét, valamint kiválasztani a futtatandó osztályt
  - o A Netbeans alapértelmezetten a projekt nevéből képzi:
    - – <projektNeveCsupaKisbetusen>.<projektNeve>
    - – ami valójában a következőt takarja: csomagNév.OsztalyNév
  - o Neve tetszőlegesen módosítható

Amikor végeztünk a beállításokkal, kattintsunk a „Finish” gombra, melynek következtében a projektünk elkészül, majd meg is nyílik. Minden esetben, amikor új projektet hozunk létre, a Projects ablakban megjelenik az új projekt, azonban a régebbiek nem záródnak be. Amennyiben nem szeretnénk, hogy továbbra is megjelenjen egy projekt az említett ablakban: kattintsunk jobb gombbal a nevére, majd válasszuk a „Close” lehetőséget a bezárásához. Ennek eredményeként a projekt nem törlődik a merevlemezről, csak nem fog tovább megjelenni az említett ablakban.

Projektfájlok

Amint létrehoztuk a projektet, megjelenik a Projects ablakban *(4. ábra)*.

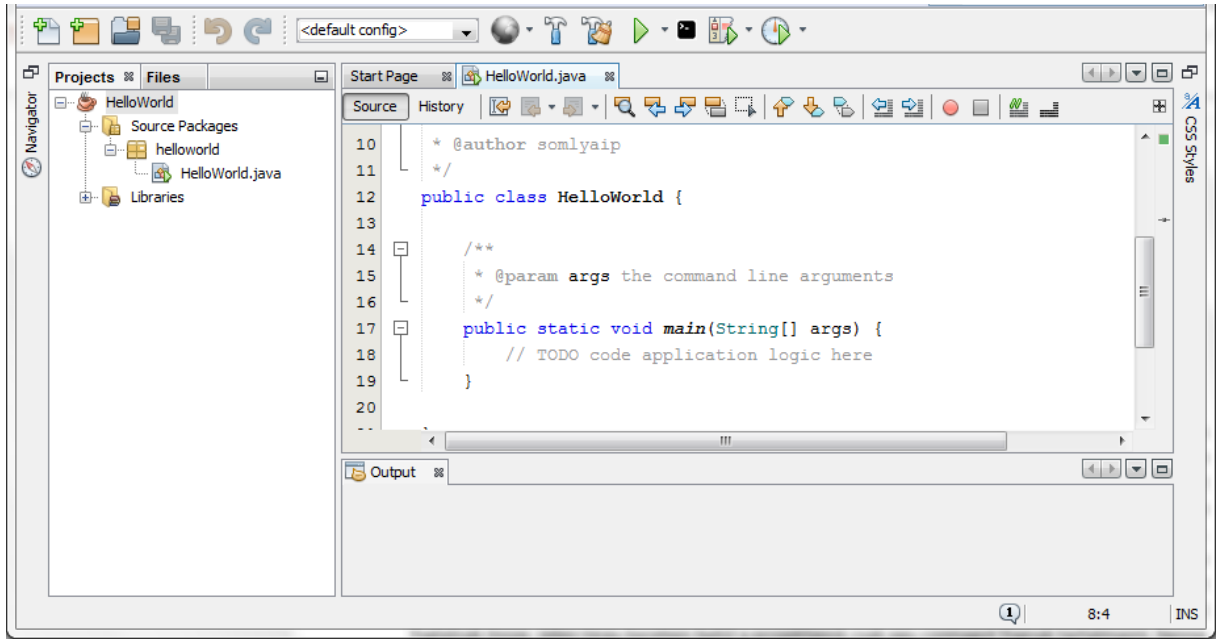
4. ábra - Projekt struktúra



(Az itt látható HelloWorld projekt természetesen egy HelloWorld projektnevű beállítás eredménye.) A Projekteket úgy tudjuk megkülönböztetni, hogy ők találhatóak a legjobban „balra igazítva". Mintha egy grafikus könyvtárstruktúra megjelenítőt használva, ők lennének az alkönyvtárai a megtekintett könyvtárnak. Emellett a nevüktől balra egy kávéscsészére emlékeztető ikon található. Minden Java Application projekt neve mellett ilyen ikont láthatunk. Amennyiben + jelet látunk mellette, nyissuk le. Így megjelennek a Source Packages és a Libraries „könyvtárak"[2]. A „Libraries" számunkra csak későbbi tárgyakban lesz lényeges, most csak a „Source Packages" könyvtárral fogunk foglalkozni, ugyanis itt találhatóak a projektünk csomagjai. A csomagok összetartozó forráskód-fájlokat (osztályokat) foglalnak össze. Jelen tárgy keretein belül a projektjeink csak egy csomagot fognak tartalmazni, benne a main függvényt hordozó egyetlen forrásfájllal. Amennyiben nem nyílt meg automatikusan a forrásfájl, kattintsunk rá duplán. Ennek hatására megjelenik a szerkesztőterületen *(5. ábra)*.

2 Valójában nem könyvtárak a fájlrendszerben, pontosabban nem ezen a néven szerepelnek a projekt könyvtárstruktúrájában.

5. ábra - Megnyitott forrásfájl



A forrásfájl fölött látható, hogy a szerkesztőterület lapokra (tab-okra) bontva jelenik meg. A lapok általában egy forrásfájlt jelölnek, de tetszőleges ablakok is kerülhetnek ide (pl.: Start Page).

A forrásfájlok módosításait a következőképen menthetjük el:

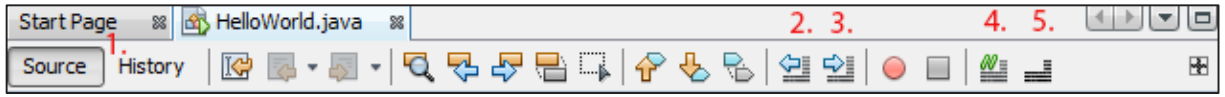
– CTRL + S gyorsbillentyű
  o Az aktuális forrásfájl módosításait menti

– Menüsor alatti ikonsorban balról a 4. (két egymást fedő floppy lemezt ábrázoló) ikonra (Save All) kattintva
  o Minden megnyitott forrásfájl módosításait eltárolja
  o Amennyiben az ikon inaktív (szürke, nem kattintható): nincs mentetlen módosításunk

Forrásfájl szerkesztő ablak

Amennyiben az aktuálisan megnyitott lap forrásfájl, a lapok alatti területen több lehetőség jelenik meg (6. ábra).

6. ábra - Forrásfájl szerkesztő lehetőségek



Az ablakon található legfontosabb elemek fölött sorszámokat láthatunk:

– 1. Source/History
  o Nyomógombos választási lehetőség a forrásfájl tartalma (Source) és az adott fájl korábbi változatainak megjelenítése (History) között
    – A History-t választva láthatjuk a korábbi állapotokat és visszaállhatunk egy előző állapotra.

– 2. Kijelölt sorok kezdetének mozgatása balra (~Behúzás csökkentése)
  o A kijelölt sorok kezdetét egy tabulátornyi távolsággal balra mozgatja
  o Egyenértékű a SHIFT + TAB gyorsbillentyűvel

– 3. Kijelölt sorok kezdetének mozgatása jobbra (~behúzás növelése)
  - o A kijelölt sorok kezdetét egy tabulátornyi távolsággal jobbra mozgatja
  - o Egyenértékű a TAB gyorsbillentyűvel

– 4. Kijelölt sorok kommentezése
  - o A kijelölt sorok mindegyikét, egyesével egysoros kommentté alakítja

– 5. Kijelölt sorok kommentezésének megszűntetése
  - o A kijelölt egysoros kommentet tartalmazó sorokat (a teljes sor komment) visszaalakítja utasítássá, azaz megszünteti a kezdetükön a komment parancsot (//)

Autocomplete

Egy változó, illetve parancs egy részének begépelését követően a CTRL + SPACE billentyűkombinációt leütve megjelenik az egyező nevű változók/parancsok listája. (A begépelt szövegrésznek a változó /parancs nevére az elejétől kezdve kell illeszkednie. Pl.: f => for, Syst => System) A kurzormozgató billentyűk segítségével válogathatunk az illeszkedő lehetőségek közül, majd ha a megfelelőn állunk: üssünk ENTER-t! Amennyiben nem található a listában a kívánt elem üssük le az ESC-et és ellenőrizzük a változó/parancs nevét, mert valószínűleg helytelenül gépeltük be.

Output (kimeneti) ablak

A Netbeans számos egyéb ablakot is tartalmaz, melyeket a Window menüpont alól tudunk megjeleníteni, amennyiben nem láthatóak.

Nekünk az Output window-ra lesz szükségünk, amelyik a projekt futtatásáról és fordításáról jelenít meg adatokat számunkra. Amikor a Netbeans segítségével futtatjuk az alkalmazásunkat itt jelenik meg a program kimente (azaz a program „ide írja” ki a print utasítások „eredményét”). Emellett az Output window segítségével adatbeolvasást is végrehajthatunk, ezt a funkciót azonban egy későbbi tárgy keretein belül fogjuk csak kihasználni, mert jelenleg nem fog rendelkezésünkre állni a megfelelő tudás hozzá (komplikáltabb a használata mint CMD ablakból beolvasni). Összességében az Output window egy CMD ablakot „emulál” (~utánoz).

Futtatás

Netbeansben a futtatás előtt nem kell manuálisan kiadnunk a fordítási parancsot, mivel a háttérben a forrásfájl módosításakor lefordítja a bytecode-ot.

A következő módon futtathatjuk a segítségével megírt programokat:

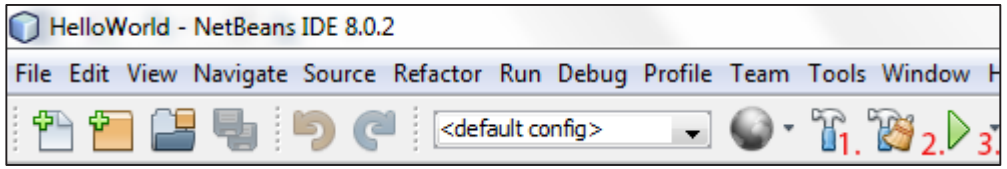
– Run ⇨Run Project (<projectNeve>), vagy

– F6 gyorsbillentyű, illetve

– Menüsor alatti zöld nyílra *(7. ábra, 3.)* kattintva.

Ne aggódjunk, amikor elmentjük a forrásfájl módosítását: a Netbeans minden esetben, azonnal lefordítja belőle a bytecode-ot, így a futtatásra kattintva az aktuális forráskódból fordított verzió fog elindulni.

7. ábra - Fő parancsikonok



Futtatás során a projektbe beállított Main osztályt futtatja le, aminek kimenete az Output window-ban jelenik meg.

Amennyiben a következő (zöld színű) üzenet jelenik meg az Output window-ban: „BUILD SUCCESSFUL (total time: x seconds)”, akkor a fordított fájl futtatása sikeres volt. A programunk kimenete az említett szöveg fölött jelenik meg.

Amennyiben piros üzenetet (hibaleírás) látunk az Output window-ban: akkor valamilyen probléma lépett fel a fájl fordítása/futtatása során. A hibaleírást átnézve találhatjuk meg a számunkra releváns hibaüzenetet. A hibaüzenet többnyire a forráskódban is megjelenik a hiba keletkezésének helyén (amennyiben fordítási hibáról van szó).

Fordítás

Amennyiben konkrét fordítási parancsot adunk ki: a Netbeans a bytecode-ainkat egy .jar kiterjesztésű fájlba csomagolja.

Amennyiben újra szeretnénk fordítani a .jar fájlt, adjuk ki az újrafordítási parancsot (Run ⇨ Clean and Build Project; SHIFT + F11; 7. ábra, 2.), aminek során a Netbeans eltávolítja az előző fordítás eredményét a merevlemezről és újrafordítja a .jar állományt

Amennyiben csak a fordítás parancsot adjuk ki (Run⇨ Build Project; F11; 7. ábra, 1.) és már korábban elkészült a .jar állomány: egy értesítőüzenet jelenik meg. Válasszuk a Clean and Build-et.

Alapértelmezett projektbeállításoknál mindig a Clean and Build Project-et használjuk, a Build Project speciális beállítások mellet nyer csak értelmet.

Amennyiben a fordítás sikeres volt, a következő üzenethez hasonló jelenik meg az Output window-ban:

Building jar: C:\HelloWorld\dist\HelloWorld.jar
To run this application from the command line without Ant, try:
java -jar "C:\HelloWorld\dist\HelloWorld.jar"

Az utolsó utasítást parancssorban kiadva lefuttató a programunk manuális módon.

Tegyük a következőt:

1. Nyissunk meg egy parancssort (Windows gomb⇨ „cmd” begépelése ⇨ ENTER

2. Jobb egérkattintás a CMD ablakon ⇨Paste

3. Enter

Értelmezzük az előbbi üzenet második sorát, amely magyarra fordítva a következőt jelenti:
„Az alkalmazás parancssorban, Ant nélkül történő futtatásához próbálja ki a következő parancsot:”
Mi az az Ant? Az Apache Ant egy fordítási- és futtatási műveleteket automatizáló szoftver segédeszköz. A Netbeans ennek segítségével futtatja az alkalmazásainkat és emiatt jelenik meg a kimenetük az

Output window-ban. Viszont nekünk néha szükségünk van a parancssori futtatásra, hiszen a legegyszerűbb beolvasási módot csak így tudjuk használni (bővebben az előadáson és a laboron).

Meglévő projekt megnyitása

Előfordulhat, hogy a projektablakban korábban bezártunk egy projektet, azonban később ismételten meg szeretnénk nyitni. Erre használhatjuk a Start Page ⇨Recent Projects listáját, ahol a legutolsó megnyitott projektek közül kiválaszthatjuk a megfelelőt megnyitásra. Amennyiben a kívánt projekt nem jelenik meg a Recent Projcets listában, úgy a következőképpen nyithatjuk meg a korábbi projektet:

– File ⇨ Open Project, vagy

– CTRL + SHIFT + O, illetve

– Az új projekt létrehozására szolgáló parancsikon mellett jobbra elhelyezkedő ikonra kattintva *(7. ábra,* bal oldalról a 3. ikon) majd tallózzuk be a projekt könyvtárát.

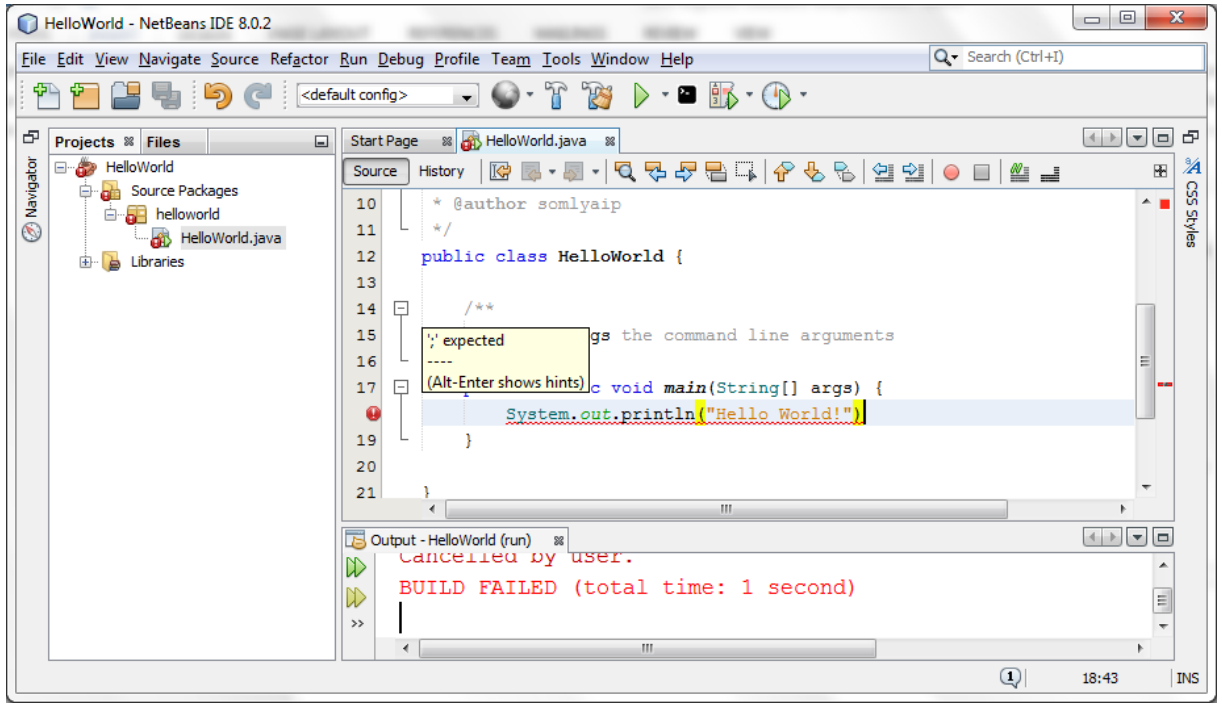
Hibajelzések

A Netbeans két féle hibaszintet különböztet meg:

– Warning:
  o Figyelmeztetés
  o A futtatást nem akadályozó hibák
    – amelyeket azért illik megoldani
  o Jele:
    – sárga háromszögben egy felkiáltójel a sorszám helyén
    – a kódrészlet sárgával aláhúzva jelenik meg

– Error:
  o Hiba
  o A futtatást megakadályozó hibák
  o Jele:
    – piros körben egy felkiáltójel
    – a hibás kódrészlet pirossal aláhúzva jelenik meg

8. ábra - Hiba jelzése a forráskódban



Amennyiben a sorszám helyett megjelenő hiba ikon, vagy az aláhúzott forráskód fölé visszük az egeret, megjelenik a hiba leírása.

Főmenü elérése gyorsbillentyűvel

A főmenü-pontok az ALT + <MenüpontAláhúzottBetűje> billentyűzetkombinációval érhetőek el. Például a File menü az ALT + F billentyűkombinációval nyitható le. Ezt követően a kurzormozgató billentyűkkel navigálhatunk a lenyitott menüpont elemei között, amellyel értékes időt spórolhatunk meg akár az otthoni gyakorlás során, akár a zárthelyi dolgozat írása alatt.

Code templatek

A kód templatek gyakran használt, előre elkészített kóddarabkák, amelyeket egy rájuk utaló karaktersor begépelése, majd a tabulátor billentyű lenyomása hatására megjelennek a forráskódban.

Számunkra a két legfontosabb:

– sout
    o System.out.println

– psvm
    o public static void main

Próbáljuk ki őket, egy csupasz osztályban állva írjuk be: psvm, majd nyomjuk le a tabulátor billentyűt. Ezt követően a main függvényben írjuk be: sout, majd ismét tabulátor.

A kód templatek használata roppant módon meggyorsítja munkánkat, ám ezen tárgy keretein belül csak akkor kezdjük el őket alkalmazni: amikor már reflexszerűen írjuk be az utasításukat, illetve készítjük el a szerkezetüket.

Aki további kód templatek, illetve gyorsbillentyűket szeretne tanulmányozni, nézze át a következő dokumentumot: https://netbeans.org/project_downloads/www/shortcuts.pdf

A dokumentum a 7.0-ás verzióhoz készült, ezért lehetséges, hogy nem fog minden leírt parancs egyezni a 8.0.2-es verzióval.

Dunaújváros, 2015. 07. 06

Somlyai Pál

Bevezetés a programozásba

Netbeans telepítés

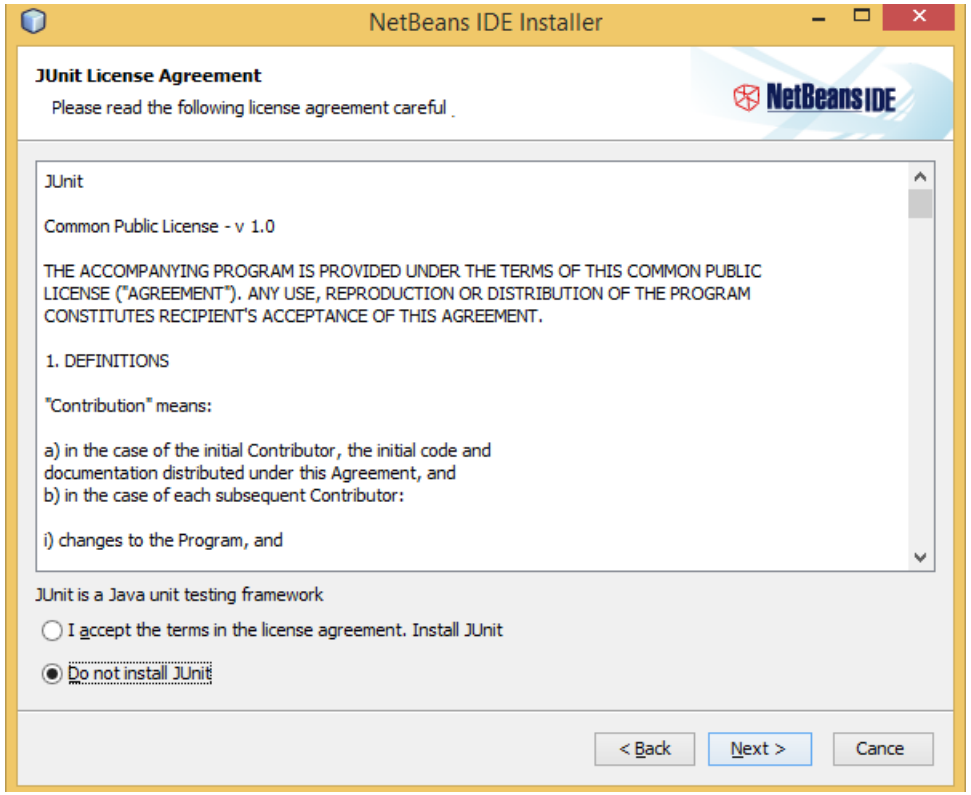
Dunaújvárosi Főiskola,
Informatikai Intézet

Netbeans telepítés

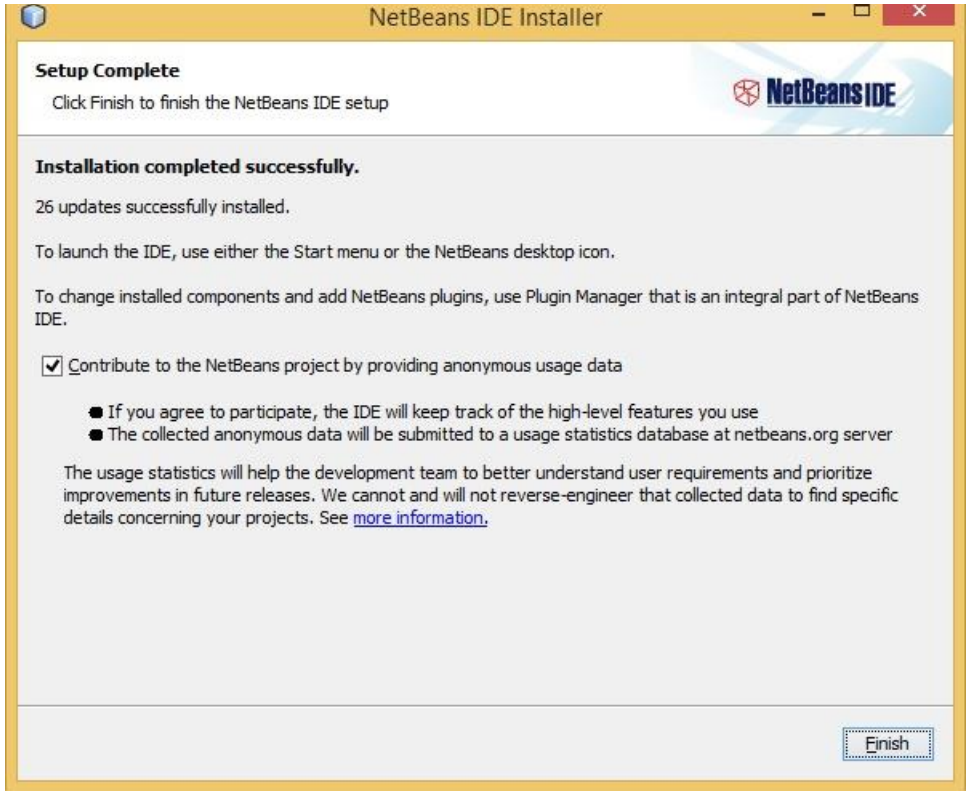
A legfrissebb Netbeans az alábbi linkről tölthető le:

https://netbeans.org/downloads/

A megjelenő csomagtípusokból válasszuk a Java SE oszlop alján található Download lehetőséget. A letöltést követően indítsuk el a telepítőt. A telepítés második lépéseként a következő képernyő jelenik meg.



A JUnit egy automatizált tesztelésre szolgáló keretrendszer, melyet csak a későbbi félévek során fogunk használni. Válasszuk a „Do not install JUnit” lehetőséget. Ezt követően csak az alább szereplő képernyő megjelenésekor van szükségünk döntést hozni.



A Netbeans egy közösség[1] által fejlesztett fejlesztői környezet, amelynek fejlesztéséért a programozók, tervezők, designerek, stb. nem kapnak fizetést, pusztán elkötelezettségből és nagyvonalúságból teszik. Valószínűleg nem használják fel semmilyen kártékony célra a tőlünk (elvileg névtelenül) gyűjtött, IDE használati adatokat (amik csak a Netbeans használatára vonatkoznak), valamint nem próbálják meg visszafejteni a projekteink adatait, de mindenki döntsön belátása szerint. Amennyiben „bepipálva" marad a négyzet, a telepített Netbeans használati adatokat küld a netbeans.org szervereinek, hogy az IDE fejlesztői számunkra legmegfelelőbben fejlesszék tovább a Netbeans IDE-t.

Ezt követően indítsuk el a Netbeans-t és elkezdhetjük a Java programozás elsajátítását.

Dunaújváros, 2015. 07. 03

Somlyai Pál

1 A közösség szoftverfejlesztési berkekben egy olyan (főként) programozókból álló csoportot jelent, amely aktívan részt vesz fejlesztői eszközök, keretrendszerek, akár operációs rendszerek, stb. fejlesztésében. Ebben az esetben a Netbeans fejlesztésében közreműködő programozókra utal.

Bevezetés a programozásba

Netbeans projekt könyvtárstruktúra

Dunaújvárosi Főiskola,
Informatikai Intézet

Netbeans projekt könyvtárstruktúra

Amikor a Netbeans segítségével létrehozunk egy projektet, az alábbi képen látható könyvtárstruktúra egy része automatikusan létrejön (a build és a dist könyvtár viszont csak az első fordítási parancs kiadását követően).

Könyvtárstruktúra

A képen látható könyvtárstruktúra a HelloWorld projekt könyvtárstruktúrája. Amennyiben a Netbeansben a Projects fül mellett található Files fülre kattintunk, a forrásfájlok helyett megjelenik a projekt teljes könyvtárstruktúrája. A projekt könyvtárának alkönyvtárai a következő fájlokat tartalmazzák:

–build:
- o a fordítás során előálló bytecode-ot tartalmazó .class kiterjesztésű fájlok
- o egyéb fordítás során előálló segédfájlok

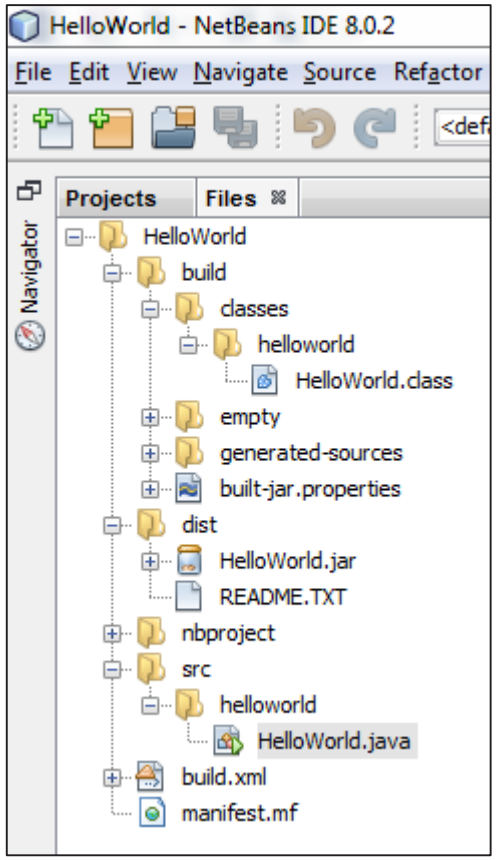
–dist (~distribution, azaz terjesztés):
- o <projektNeve>.jar fájl
  - – a legtöbb Java program több osztályból áll
  - – a jar (Java Archive) fájl egy tömörített „könyvtár” amelyik tartalmazza a projekthez tartozó összes osztály bytecode-ját a könnyebb terjesztés (~felhasználókhoz eljuttatás) végett

–nbproject (Netbeans Project):
- o a netbeans projekt beállításait tartalmazza

–src:
- o a projekt forrásfájlait tartalmazó könyvtár
- o a Projects fül alatt a „Source Packages” könyvtár alatt megjelenő fájlok

–build.xml
o a fordítási folyamatot leíró xml

–manifest.mf
o a készítendő jar fájl metaadatait (pl. verzió, cím, készítő, stb.) leíró beállítófájl.



Dunaújváros, 2015. 07. 02

Somlyai Pál

Bevezetés a programozásba

Kurzusinformációk

Dunaújvárosi Főiskola,
Informatikai Intézet

Rövid célkitűzés Képesek lesznek egy informatikai feladat algoritmusát előállítani és leírni. Képesek lesznek ezt az algoritmust C programozási nyelven megírni. Képesek lesznek egyszerű C programot írni, futtatni, ellenőrizni. Képesek lesznek egy egyszerűbb C program működését megérteni. Képzési előzmény: A középiskolában elsajátított műszaki és informatikai tudás. Ráépülő fejlesztési cél: Informatikai eszközhasználat bővítése, programozási és algoritmus készítési ismeretek és jártasság kialakítása és fejlesztése.
Képzési előzménye, fejlesztési célok
A követett képzési alapmódszer, különösen a gyakorlat / szeminárium stb. megoldása és ha különleges, akkor annak célja. Mindez hogyan “támasztja alá” a szak szemléletet, fő célját. Frontális oktatás az előadásokon. Számítógépes munkavégzés a laborokon. Önálló feladatmegoldás és team munkavégzés váltakozása. Programozási problémák és feladatok közös megbeszélése.
Tudás
Ismeri az algoritmus készítés alapelveit, lépéseit, határait. Ismeri a C nyelvű programozás alapelveit, lépéseit, határait. Ismeri az ehhez szükséges matematikai és számítógép-kezelői alapokat. Ismeri a C fejlesztői környezethez és a programozáshoz szükséges angol nyelvű fogalmakat.
Képesség
Képes fejben elkészíteni egy egyszerűbb informatikai feladat algoritmusát. Ezt az algoritmust képes megírni C programozási nyelven, lefuttatni, a végeredményt elemezni, hibát keresni. Képes egy egyszerűbb C program felépítését és működését megismerni, abban hibát keresni.
Attitűd
Erős motivációval rendelkezik a programozás irányában, mert a Gazdaságinformatikus BSc szakon ez komoly elvárás.
Autonómia és felelősségvállalás
Önálló munkavégzés. Önálló problémamegoldás képessége. Csoportmunka képessége.
Előadás: A C programnyelv kialakulása. A programnyelv alapelemei. A változók típusai. Konstansok, szimbolikus konstansok. Skalárok és tömbök definiálása, deklarálása. A kezdeti értékadás. Utasítások és blokkok. Függvények definíciója, deklarációja, prototípusa. Az argumentum, a paraméter és a visszatérési érték Operátorok és kifejezések. Labor gyakorlat: A számítógépes problémamegoldás alapfogalmai: algoritmus, algoritmizálás, jel algoritmusok (folyamatábra, szerkezeti ábra, struktogram), program. Szintaktika, szemantika. A strukturált programozás. Adatok ábrázolása (tárolása) a memóriában. Az integrált fejlesztői környezet használata. Egyszerű feladatok (algoritmusok) kódolása, tesztelése, javítása.
- Hallott szöveg feldolgozása jegyzeteléssel 20% - Információk feladattal vezetett rendszerezése 30% - Feladatok önálló feldolgozása 50%

A tantárgy neve		magyarul	**Bevezetés a programozásba**								Szintje	**A (alap)**	
		angolul	Introduction to programming									**DFAN-INF-501** **DFAL-INF-501**	
2015/16/1													
Felelős oktatási egység			**Informatikai Intézet**										
Kötelező előtanulmány neve													
Típus		Heti óraszámok Előadás			Gyakorlat		Labor			Követelmény	Kredit	Oktatás nyelve	
Nappali	**150/75**		**2**			**0**		**3**		**F**	**5**	**magyar**	
Levelező	**150/25**	Féléves	**10**		Féléves	**0**	Féléves	**15**					
Tárgyfelelős oktató					neve		**Kógelmann Gábor**				beosztása	**Óraadó**	
A kurzus képzési célja, indokoltsága (tartalom, kimenet, tantervi hely)			**Rövid célkitűzés** Képesek lesznek egy informatikai feladat algoritmusát előállítani és leírni. Képesek lesznek ezt az algoritmust C programozási nyelven megírni. Képesek lesznek egyszerű C programot írni, futtatni, ellenőrizni. Képesek lesznek egy egyszerűbb C program működését megérteni. Képzési előzmény: A középiskolában elsajátított műszaki és informatikai tudás. Ráépülő fejlesztési cél: Informatikai eszközhasználat bővítése, programozási és algoritmus készítési ismeretek és jártasság kialakítása és fejlesztése.										
			Képzési előzménye, fejlesztési célok A követett képzési alapmódszer, különösen a gyakorlat / szeminárium stb. megoldása és ha különleges, akkor annak célja. Mindez hogyan "támasztja alá" a szak szemléletet, fő célját. Frontális oktatás az előadásokon. Számítógépes munkavégzés a laborokon. Önálló feladatmegoldás és team munkavégzés váltakozása. Programozási problémák és feladatok közös megbeszélése.										
Jellemző átadási módok			Előadás	Frontális előadás.									
			Gyakorlat										
			Labor	Számítógépes feladatmegoldás.									
			Egyéb										
Követelmények (tanulmányi eredményekben kifejezve)			**Tudás** Ismeri az algoritmus készítés alapelveit, lépéseit, határait. Ismeri a C nyelvű programozás alapelveit, lépéseit, határait. Ismeri az ehhez szükséges matematikai és számítógép-kezelői alapokat. Ismeri a C fejlesztői környezethez és a programozáshoz szükséges angol nyelvű fogalmakat.										
			Képesség Képes fejben elkészíteni egy egyszerűbb informatikai feladat algoritmusát. Ezt az algoritmust képes megírni C programozási nyelven, lefuttatni, a végeredményt elemezni, hibát keresni. Képes egy egyszerűbb C program felépítését és működését megismerni, abban hibát keresni.										
			Attitűd Erős motivációval rendelkezik a programozás irányában, mert a Gazdaságinformatikus BSc szakon ez komoly elvárás.										
			Autonómia és felelősségvállalás Önálló munkavégzés. Önálló problémamegoldás képessége. Csoportmunka képessége.										
Tantárgy tartalmának rövid leírása			Előadás: A C programnyelv kialakulása. A programnyelv alapelemei. A változók típusai. Konstansok, szimbolikus konstansok. Skalárok és tömbök definiálása, deklarálása. A kezdeti értékadás. Utasítások és blokkok. Függvények definíciója, deklarációja, prototípusa. Az argumentum, a paraméter és a visszatérési érték Operátorok és kifejezések. Labor gyakorlat: A számítógépes problémamegoldás alapfogalmai: algoritmus, algoritmizálás, jel algoritmusok (folyamatábra, szerkezeti ábra, struktogram), program. Szintaktika, szemantika. A strukturált programozás. Adatok ábrázolása (tárolása) a memóriában. Az integrált fejlesztői környezet használata. Egyszerű feladatok (algoritmusok) kódolása, tesztelése, javítása.										

Tanulói tevékenységformák	- Hallott szöveg feldolgozása jegyzeteléssel 20% - Információk feladattal vezetett rendszerezése 30% - Feladatok önálló feldolgozása 50%
Kötelező irodalom és elérhetősége	C nyelvvel kapcsolatos elektronikus tananyagok. Elérhetőség a Moodle rendszeren keresztül.
Ajánlott irodalom és elérhetősége	1. Lipschutz: Adatszerkezetek Panem Kft, Budapest, 1993. 2. Marton László – Fehérvári Arnold: Algoritmusok és adatstruktúrák NOVODAT, Győr, 2002. 3. Stephen G. Kochan: Programfejlesztés C nyelven Kiskapu SAMS, Budapest, 2008. 4. Benkő Tiborné és társai: Programozzunk C nyelven ComputerBooks, Budapest, 2010. 5. B. W. Kernighan, D. M. Ritchie: A C programozási nyelv Műszaki könyvkiadó, Budapest, 1985. 6. Benkő Tiborné, Dr. Poppe András: Együtt könnyebb a programozás (C) ComputerBooks, Budapest, 2004.
Beadandó feladatok/mérési jegyzőkönyvek leírása	Nincsenek kötelezően beadandó feladatok. Esetenként házi feladat kiírása előfordul.
Zárthelyik leírása, időbeosztása	Egy zárthelyi az elméleti anyagból az előadáson. Egy zárthelyi, és három önálló programozási feladat a labor foglalkozásokon. - 1 db. zh az előadáson a 13. héten. (25 pont) - 1 db. zh a nyitott laboron az 5. héten. (15 pont) - 2 db. program a nyitott laboron, a 7., és 9. héten. (2x15 pont) - 1 db. komplex program a laborgyakorlatokon, a 12. héten. (30 pont)

Tantárgyi Követelményrendszer

A foglalkozásokon való részvétel követelményei és a távolmaradás pótlásának lehetősége, a jelenlét ellenőrzésének módja és rendszeressége	
Távollét esetén az igazolás módja	
Félévközi jegy esetén megszerzésének feltételei és módja, valamint vizsgaidőszakban történő javítás lehetősége	
Vizsgajegy esetén a vizsgán, ill. a szorgalmi időszakban teljesített követelmények milyen módon és milyen arányban számítanak bele a végső érdemjegy kialakításába	
A vizsgaidőszakban nem pótolható azon részfeladatok, amelyek a követelményrendszer szerint a teljes félév összefüggő munkájával készíthetők el, a vizsa típusa (írásbeli és/vagy szóbeli)	
A tananyag elsajátításához felhasználható egyéb jegyzetek, segédletek, irodalmak listája	
Egyéb általános tudnivaló	

Nappali ütemezés nézet

Bevezetés a programozásba beosztás, alapütemezés				
Hét	Típus	Tartalom	Hivatkozások	Fejlesztett kompetencia
1	Előadás			
	Gyakorlat			
	Labor			
2	Előadás			
	Gyakorlat			
	Labor			
3	Előadás			
	Gyakorlat			
	Labor			
4	Előadás			
	Gyakorlat			
	Labor			
5	Előadás			
	Gyakorlat			
	Labor			
6	Előadás			
	Gyakorlat			
	Labor			
7	Előadás			
	Gyakorlat			
	Labor			
8	Előadás			
	Gyakorlat			
	Labor			
9	Előadás			
	Gyakorlat			
	Labor			
10	Előadás			
	Gyakorlat			
	Labor			
11	Előadás			
	Gyakorlat			
	Labor			
12	Előadás			
	Gyakorlat			
	Labor			
13	Előadás			
	Gyakorlat			
	Labor			
14	Előadás			
	Gyakorlat			
	Labor			
15	Előadás			
	Gyakorlat			
	Labor			

Levelező ütemezés nézet

Bevezetés a programozásba beosztás, alapütemezés				
Konzultációs alkalom	Típus	Tartalom	Hivatkozások	Fejlesztett kompetencia
1	Előadás			
	Gyakorlat			
	Labor			
2	Előadás			
	Gyakorlat			
	Labor			
3	Előadás			
	Gyakorlat			
	Labor			
4	Előadás			
	Gyakorlat			
	Labor			
5	Előadás			
	Gyakorlat			
	Labor			
6	Előadás			
	Gyakorlat			
	Labor			