9 Algoritmizálás és a C# programozási nyelv használata

Előszó

Ez a jegyzet a Digitális kultúra tantárgy 9. évfolyamos tananyagának az Algoritmizálás és programozási nyelv használata témát dolgozza fel. Tematikájában, legtöbb feladatában és megjelenési formájában az állami tankönyv* és online tananyag az alapja, ahol lehet, ott a szöveg is azonos, de a tankönyvben bemutatott Python nyelv helyett a C# nyelvet, illetve a Visual Studio 2019 Community fejlesztőkörnyezetben történő programozást mutatja be.

A jegyzet – kihasználva, hogy nincs terjedelmi korlátozás – a tankönyvhöz képest jelentős kiegészítéseket tartalmaz:

- Alternatív megoldásokkal, a megoldások összehasonlításával segíti az érdeklődők igényeinek kielégítését, de tisztázza azt is, hogy mi a továbbhaladáshoz szükséges minimum.
- Az algoritmusok elemzése részletesebb – négyféle elágazásra és négyféle ciklusra mutat példát, alternatív megoldásokat.
- Az önálló tanulás támogatása érdekében tárgyalja a program lépésenkénti futtatását, hibák értelmezését, az adatok beviteli módjait.
- Az elemi adatok mellett a tömb, a lista és a szöveg típusú adatsorozatok használatát is tanítja.
- Az algoritmus elemeit mondatszerű leírással és folyamatábrával is bemutatja.
- Az OOP alapjait a használattal – és az IDE kódszínezésével – összhangban tárgyalja.
- A programozói gondolkodásmódot, a szokásokat is bemutatja.
- Tanulásmódszertani javaslatokat ad.
- Programozás- és kódolástechnikai ötleteket, módszertani ajánlásokat tesz.
- A fogalmak szemléltetése után a szaknyelv kifejezéseit használja.

A jegyzet teljes megtanulása elsőre soknak tűnhet, a többoldalú megközelítések miatt is ajánlott húzni belőle és az egyes részekre akkor visszatérni, amikor szükség van rá. Ugyanakkor, a lehetőségek áttekintése hozzájárulhat az egyéni preferenciák érvényesítéséhez.

Sikerekben gazdag tanulást kívánok:

Budapest, 2021.

Szalayné Tahy Zsuzsanna

* Digitális kultúra tankönyv 9. Oktatási Hivatal 2020.

Tartalom

Mi az a programozás? 3
- Mi a „program”? 3
- Hol vannak a programok? 3
- Mi van egy programfájlban? 4
- Csak ennyiből áll egy szoftverfejlesztő munkája? 6

Első programjaink 7
- A programozási környezet 7
- Legelső programunk 8
- Programozás IDE nélkül (csak erős idegzetűeknek) 13

Változók, kiíratás, adat bekérése 15
- Szöveg, karakter és szám – adattípusok 15
- Változók 16
- Adat bekérése a felhasználótól 18

Számok és karakterláncok a programunkban 20
- Hány éves a felhasználó? 20

Segíts magadon, az IDE is megsegít! 25
- A színek jelentése 25
- Kódkiegészítés és helyi súgó 27
- Tanulást is támogató eszközök 29

Elágazások 31
- Gondoljunk egy számra 31
- Az összehasonlítás jelölése 34
- Összetett feltétel 35
- Összetett feltétel logikai szabályai 36
- Véletlenszám-előállítás 37
- Elágazások és véletlenek alkalmazása 38

Ciklusok 39
- A feltételes ciklus (while-ciklus) 40
- Következő órára leírod százszor, hogy … (for-ciklus) 41
- A logikai típus 45
- Összetett ciklusfeltétel 46

Ciklusok és véletlenek 47

Ciklusok oda-vissza, illetve egymásba ágyazva 48

Összetartozó adatok kezelése 49
- Adatok sorozata 49
- A tömb és használata 49
- A lista és használata 52
- A szöveg karakterlánc 55
- Bekért adatok ellenőrzése (do-while-ciklus) 57

Adatsorok kezelése 61
- Szerencsejáték esélyek 63
- A bejárós ciklus (foreach-ciklus) 66
- Adatsorozatok függvényei 68

Adatsorozatok és ciklusok 69

Tárgymutató 71

Mi az a programozás?

Mi a „program”?

A számítógép, a telefon, az összes olyan eszközünk, amiben valamilyen „számítógép” van, önmagában képtelen ellátni azt a feladatot, amire készült. Csak egy darab „vas” – azaz hardver. Csak akkor képes igazán működni, ha fut rajta egy (vagy sok) program, alkalmazás – azaz szoftver. Szoftver, program, alkalmazás – nagyjából ugyanazt jelenti: azt a programozó, a szoftverfejlesztő által megírt valamit, ami elmondja a hardvernek, hogy mikor mit „csináljon”.

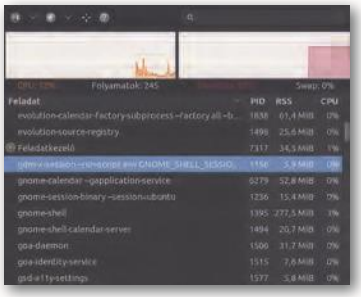
Program mondja meg

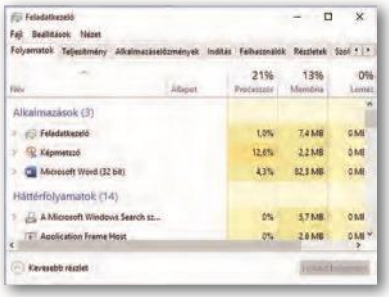
- a kenyérsütő-gépnek, hogy meddig gyúrja a tésztát, meddig hagyja kelni, és mikor kezdje sütni, mennyire legyen meleg a fűtőszál, hányat sípoljon a sütő, amikor kész a kenyér;
- a mosógépnek, hogy mikor és mennyi vizet szívjon be, mennyire melegítse fel, meddig és merre forogjon benne a dob, mikor és meddig kell centrifugálnia.

Ezek a számítógépek egyetlen programot futtatnak. Az informatikaórán bennünket jobban érdekelnek a hagyományos értelemben vett számítógépek (laptopok, asztali gépek, szerverek) és a mobileszközök. Ezek csak bekapcsoláskor futtatnak egyetlen programot, ami azt mondja el nekik, hogy honnan és hogyan kell betölteniük a „fő” programjukat: az operációs rendszert. A többi program (a böngésző, az üzenetküldő, a játék, a képszerkesztő, a szövegszerkesztő, a filmvágó stb.) pedig az operációs rendszerből, annak felügyelete alatt indul el, amikor rákattintunk az egérrel vagy rábökünk az ujjunkkal az indítóikonjára, esetenként automatikusan.

Hol vannak a programok?

Az elindított, pontosabban a futó programok a számítógép memóriájában vannak. Nemcsak a program van itt, hanem az általa éppen használt adatok is: a szövegszerkesztő által szerkesztett szöveg, a képszerkesztőbe betöltött kép. Az operációs rendszerünk feladatkezelőjében megnézhetjük az épp futó programokat. Látjuk, hogy a legtöbbet nem mi indítottuk el, sőt nem is látjuk őket – a háttérben futnak.





Egy Linuxon futó feladatkezelő és a Windows feladatkezelője – Indítsunk el egy programot, keressük meg a feladatkezelőben és állítsuk meg innen! *

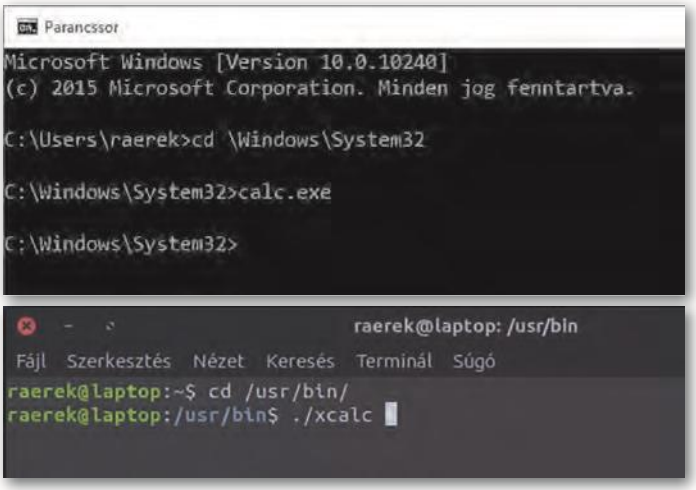
Amíg a programot nem indítjuk el, a számítógép háttértárán, például a laptop SSD-jén, az asztali gép vagy szerver winchesterén, vagy a telefon memóriakártyáján van, ugyanolyan fájlként,

* Digitális kultúra 9. tankönyv 93. oldalán látható kép

mint a képek, a zenék vagy a szövegek. Az egyszerű programok egyetlen fájlból állnak, az összetettek sokszor nagyon sokból.

A grafikus felületű operációs rendszerek elterjedése előtt (az 1990-es évekig) a programokat úgy indítottuk el, hogy a parancssoros felületben beléptünk abba a mappába (könyvtárba), amelyikben a programunk volt és beírtuk a program nevét. A módszer ma is működik, bár többnyire csak a számítógépekhez jobban értő emberek, rendszergazdák, rendszermérnökök és szoftvereket készítők használják. Lévén e fejezet célja épp az, hogy kicsit mi is szoftvert készítsünk, ismerkedjünk meg ezzel a módszerrel!

1. Nyissunk a gépünkön parancssoros felületet: Windowson indítsuk el a Parancssor nevű alkalmazást, macOS-en és Linuxon pedig valamelyik terminált!
2. „`cd`” mappaváltó paranccsal lépkedjünk abba a mappába, ahol a programfájl van (minden sor begépelése után ENTER-t nyomunk)!
3. Írjuk be a program nevét, és nyomjuk meg az ENTER-t!



Program indítása parancssorból Windows 10-en és Ubuntu Linuxon *

A program ilyenkor betöltődik a számítógép memóriájába és a benne lévő utasítások végrehajtódnak, azaz a program futni kezd.

Mi van egy programfájlban?

Ha már úgyis a parancssorban, az imént elindított programunk mappájában vagyunk, adjuk ki:

- Windowson a `type`
- macOS-en és Linuxon a cat

parancsot, és írjuk utána a programfájl nevét (például `type calc.exe`, `cat xcalc`)!

Rengeteg krikszkrakszot ír a parancssori ablakba a gép. Ha némileg hihetetlen is, a számítógép ezt érti, ebből tudja, hogy mit kell csinálnia. Ez a program egyik alakja, az úgynevezett *gépi kódú* program, amely most a képernyőn karakterek formájában jelenik meg.

* Digitális kultúra 9. tankönyv 94. oldalán látható kép

Szerencsére a legtöbb szoftverfejlesztőnek nem így kell megfogalmaznia a gép teendőit. Rendelkezésünkre állnak programozási nyelvek, azaz az angol nyelv szavait használó magasabb szintű nyelvek. Ilyen például a C, a C++, a Python, a Pascal, a Ruby, a Go, a Perl, a JavaScript, a Java és még sorolhatnánk. Ilyen az érdeklődésünk fókuszába emelt **C#** (ejtsd: szí sharp) is. A szoftverfejlesztő többnyire valamelyik programozási nyelven írja a program **forráskódját**.

Egy egyszerű forráskódot többé-kevésbé már most is tudunk értelmezni. Mit csinál az alábbi, C# nyelvű program?

```
1.  using System;
2.  namespace Valami
3.  {
4.    class Program
5.    {
6.      static void Main()
7.      {
8.        Console.WriteLine("Üdv néked!");
9.        Console.Write("Hány éves vagy?");
10.       string ev = Console.ReadLine();
11.       int evek_szama = int.Parse(ev);
12.       if (evek_szama < 14)
13.         Console.Write("Jé, hogyhogy már középiskolás vagy?");
14.       else
15.         Console.Write("Egy év múlva {0} éves leszel.", evek_szama + 1);
16.     }
17.   }
18. }
```

Nos, ilyen és ehhez hasonló programokat fogunk mi is írni az elkövetkezendő órákon. Látjuk, hogy az angol szavak mellett van még a programban írásjel, műveleti jel, kerek és kapcsos zárójel – ezek mind a program részei, nem hagyhatók el. C#-ban az utasítások végét pontosvessző jelzi. A programot lehetne egy sorba is írni, de az olvashatóságot, a kód értelmezését segíti az egységek kezdetét és végét jelző kapcsos zárójelek külön sorba írása és a belső tartalom beljebb kezdése.

Természetesen ezt a programkódot a számítógép ebben a formában nem érti, és nem tudja futtatni. A fenti forráskódot egy másik program (C# esetén a `csc.exe`) előbb elemzi, gépközeli nyelvre fordítja, linkeli – **build**-eli – és létrehozza a futtatható fájlt, a példában `Valami.exe` néven. Azok a programozási nyelvek, amelyek fordítóprogramja a háttértárra is menti a bináris kódot (`.exe`-t) a **compiler**ek (fordítók. Ettől eltérően, csak a memóriában hozzák létre a bináris kódot (nem készül `.exe` fájl) az **interpreter**ek (értelmezők). Ezeknek a működése hasonló a böngészőkhöz, amelyek a szöveges HTML és CSS kódot értelmezve jelenítik meg a weblapot. A C# compiler, a Python interpreter nyelv.

Feladatok

1. Az alábbi Python, illetve C++-nyelvű kódok a fenti C# nyelvűvel megegyező működésű programot eredményeznek. Keressük meg a hasonlóságokat, mutassunk rá a különbségekre!

Python:

```python
print('Üdv néked')
évek_száma = input('Hány éves vagy?')
évek_száma = int(évek_száma)
if évek_száma < 14:
  print('Jé, hogyhogy már középiskolás vagy?')
else:
  print('Egy év múlva', évek_száma+1, 'éves leszel')
```

C++:

```cpp
#include <iostream>
using namespace std;
int main()
{
  cout << "Üdv néked!" << endl;
  int evek_szama;
  cout << "Hány éves vagy?";
  cin >> evek_szama;
  if (evek_szama < 14)
    cout << "Jé, hogyhogy már középiskolás vagy?";
  else
    cout << "Egy év múlva" << evek_szama + 1 << "éves leszel.";
  return 0;
}
```

2. Rakjunk össze egy másik programot az alábbi részletekből! (Használjunk fel minden darabot! Egy-egy darab többször is használható.)

Csak ennyiből áll egy szoftverfejlesztő munkája?

Nos, igen, a fejlesztői munka legismertebb része az új programok írása.

- Sokkal többen vannak azok, akik meglévő programokat alakítgatnak át az új követelményeknek megfelelően (nekik köszönhetők például a telefonjainkra letöltendő frissítések).
- Van, aki azzal foglalkozik, hogy egy meglévő program fusson másféle gépen is – ez néhány esetben gyorsan megoldható, más programoknál nehéz és kimerítő feladat.
- Vannak, akik azért dolgoznak, hogy egy meglévő program legyen gyorsabb.
- Van, aki programokat tesztel: megnézi, hogy biztosan jól működnek-e minden helyzetben.

- Van, aki azzal foglalkozik, hogy programot, szoftverrendszert tervez. Ő már nem ír kódot, hanem azért felel, hogy a szoftver különböző részei minél jobban tudjanak együttműködni.
- Van, aki biztonsági ellenőrzést végez programokon, például azért, hogy számítógépes bűnözők ne tudják a banki szoftverekkel átutaltatni a pénzünket másik számlára.

A következő leckében mi is megírjuk első programunkat.

Kérdések

1. Mik azok a számítógépes vírusok?
2. Milyen más feladatok merülhetnek fel egy szoftver elkészítésekor? Milyen képzettségű munkatársai vannak a szoftver fejlesztőjének?
3. Milyen kép él benned a programozókról? Milyen előítéletek kapcsolódnak hozzájuk?
4. Milyen világszerte ismert oldalakon foglalkoznak programozási kérdések megválaszolásával? Hány programozással kapcsolatos videó készül naponta?

Első programjaink

A programozási környezet

A programozási (fejlesztői) környezet arra való, hogy benne írjuk meg programjainkat, használatával a programot gépi kódúvá alakítsuk és tesztelhessük, dokumentálhassuk a kész programot. Ezek közül a legfontosabb a gépi kódúvá alakítás, a többit vagy meg tudjuk oldani egy adott környezetben, vagy nem.

A programozási környezetet telepítenünk kell a gépünkre. A C# nyelvhez legjellemzőbben használt környezet a *Visual Studio*, ami a `visualstudio.microsoft.com` webhelyről tölthető le. Válasszuk az operációs rendszernek is megfelelő közösségi (Community) – ingyenes – verziót! Linux operációs rendszerhez a *Mono* (`www.mono-project.com`) használható. A Visual Studio telepítésekor figyeljünk arra, hogy elsőre csak a szükséges összetevőket telepítsük *(.Net Core* és esetleg a *.Net desktop development* modulok). A csábító – játék, webes, mobil – alkalmazások készítésére összeállított kiegészítések és további programozási nyelvek telepítésére később is van mód. Bármilyen kis részét telepítjük a Visual Studionak, az eszköz alkalmas lesz csoportmunkában programozásra is és a képernyő megosztására is. Ehhez azonban be kell jelentkeznünk egy Microsoft-fiók címmel (akár készíthetünk is egyet). A címet használva felhőben is tárolhatjuk a programunkat és a Live Share kiterjesztés módot ad a képernyő megosztására.

Az IDE

A Visual Studio (és a fapadosabb Visual Studio Code is) **integrált fejlesztői környezet** (Integrated Development Environment), azaz **IDE**. Egy program elkészítéséhez általában elegendő egy egyszerű szövegszerkesztő és egy fordítóprogram. Azonban ezekkel az eszközökkel nehéz gyorsan és jól programozni, megtalálni az elírásokat, hibákat. Ezért az egyes nyelvekhez programozást támogató eszközöket készítettek, amelyek egy alkalmazásba foglalása az IDE. Ilyen támogató eszközök:

- a kód funkció szerinti színezése;
- a fordító számára nem érthető kódrészlet (hiba) jelzése;
- a program lépésenkénti futtatása, pillanatnyi állapotának a kijelzése;
- a helyérzékeny javaslat a kódra, kódkiegészítésre;

- rövid szóból összetett kódrészletek megjelenítése (snippetek);
- nem használt, valószínűleg felesleges kódrészletek jelzése;
- nem szabványos, esetleg problémás, de működő kódra („code-smell”) figyelmeztetés;
- memóriafoglalás és futási idő figyelése, elemzése;

a fentieket rendszerint több programozási nyelvre, többféle programtípus készítésére tudja alkalmazni.

A Visual Studio csak egy a programozókat segítő IDE-k: a Code::Blocks, a Delphi, az Eclipse, az IDLE, a Lazarus és az XCode néhány további az ismertebb fejlesztőkörnyezetek közül.

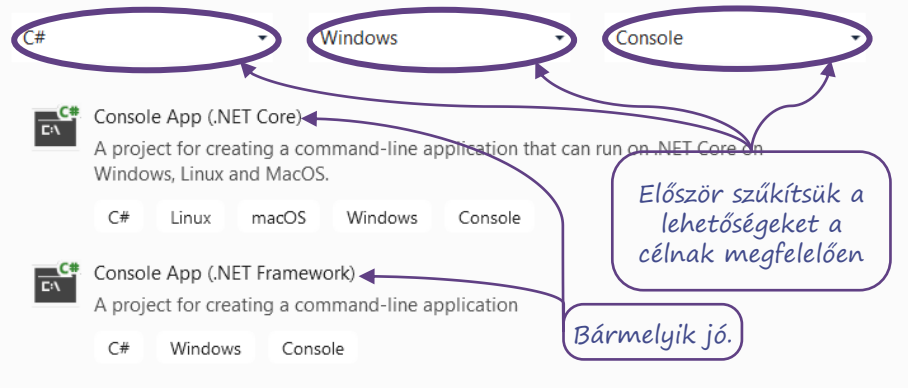
Feladat

1. Keressünk leírást a felsorolt IDE-kről! Nézzünk utána, hogy milyen operációsrendszerre lehet telepíteni, milyen hardverfeltételek szükségesek a futtatásukhoz, illetve melyik programozási nyelvekhez adnak fejlesztői támogatást!

Legelső programunk

Az IDE használatára jellemző, hogy elindítás (és bejelentkezés) után pontosítani kell, hogy mit szeretnénk:

- Létrehozni egy új programot (projektet) vagy megnyitni egy régit, esetleg csatlakozni más projekthez. Elsőre nem sok választásunk van, hozzunk létre és adjuk meg a programunk jellemzőit:



- Ki kell választanunk, hogy az IDE eszköztárából melyik eszközkészletet szeretnénk használni, azaz melyik programozási nyelven, melyik operációs rendszerre milyen típusú programot szeretnénk készíteni. Jellemzően **C#** nyelven, a **saját gépünk operációs rendszerén** futó **Console** (konzol) alkalmazást fogunk készíteni.
- A megadott céloknak megfelelő eszközök listájából bármelyiket választhatjuk. A lista hossza a telepített moduloktól függ. Ha nincs ajánlat, akkor az interneten kereshetünk megfelelő modult (Install more tools and features).
- Ezután meg kell adnunk, hogy melyik háttértáron hol legyen a program mappa és mi legyen a programunk neve. A megadott név lesz többek között a teljes programot, kódot és segédfájlokat tartalmazó mappa neve és – ha megírtuk a kódot és lefordítottuk, akkor – a programfájlunk neve is. Ezért nem célszerű sem hosszú, sem szóközöket vagy ékezetes betűket tartalmazó nevet megadni.

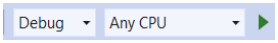
- A sok előkészület után az IDE létrehozza a megfelelő fájl struktúrát és egyben egy minimális programot is.

Igaz, hogy az IDE telepítése sok helyet igényel és a használatát is meg kell tanulni (egy kicsit), de cserébe a legelső programunkhoz nem kell semennyit kódolnunk, mert azt előállítja az IDE. Így programunkat rögtön futtathatjuk. 😊

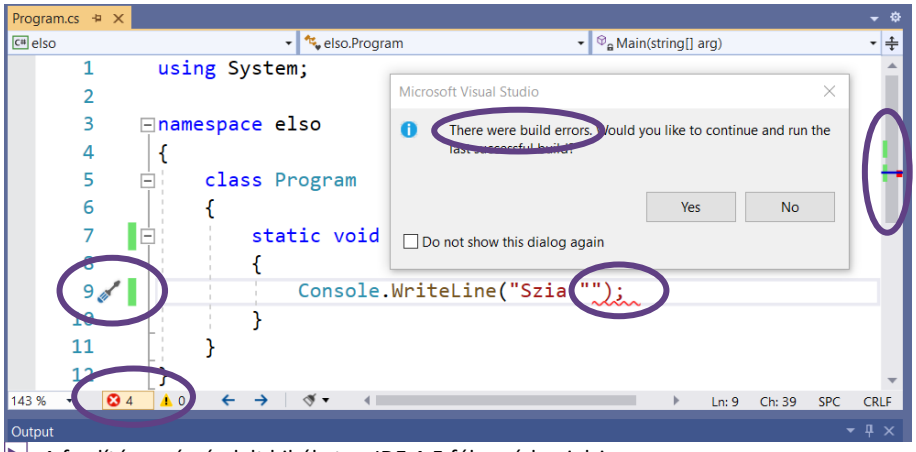
Fejlett IDE a menüsorban és gombon is felkínálja a program fordítását és futtatását. Ezt a két lépést lehet külön is indítani, de általában megtalálható a fordítás utáni azonnali futtatás, a „**Build&Run**” lehetősége is, például **Start** néven.

Pusztán azért, hogy elmondhassuk: programoztunk, módosítsuk az IDE által létrehozott kódot úgy, hogy a „Szia.” szöveget írja ki! Teendőnk: ha a kapott kódban szerepel a „Hello World”, akkor ezt lehet módosítani. Ha nincs ilyen, akkor fentebb (vagy az interneten) láthatunk példát a megfelelő „Hello World” kódra, ezt lehet bemásolni.

A fordítás és futtatás egy – a menüsoron látható zöld háromszögön – kattintással vagy az **F5** billentyű lenyomásával indítható. Kis időbe telik, míg az IDE mindent ellenőriz, de utána egy villanás alatt lefut a programunk.

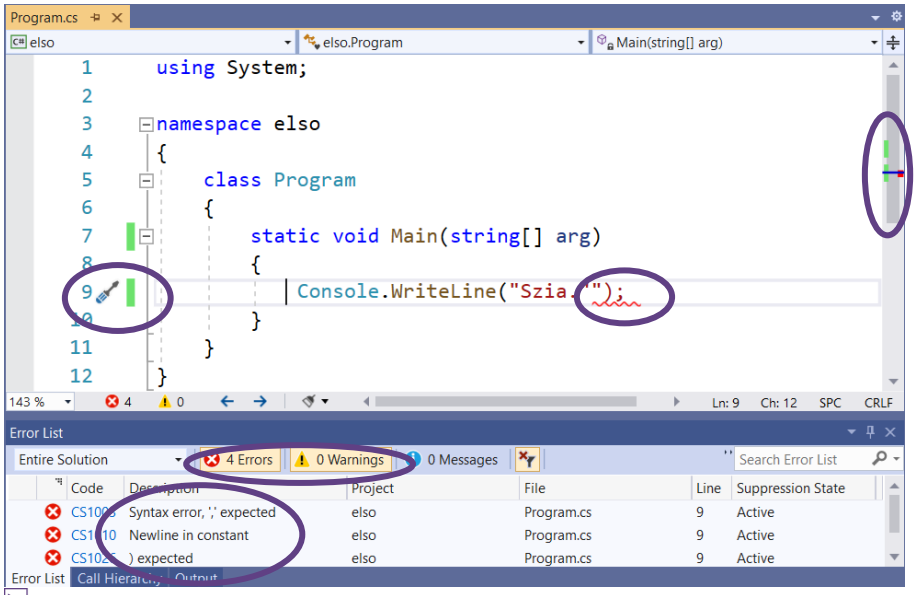


… vagy mégsem, ha valamit elrontottunk.



A fordítás során észlelt hibákat az IDE 4-5 féle módon jelzi.

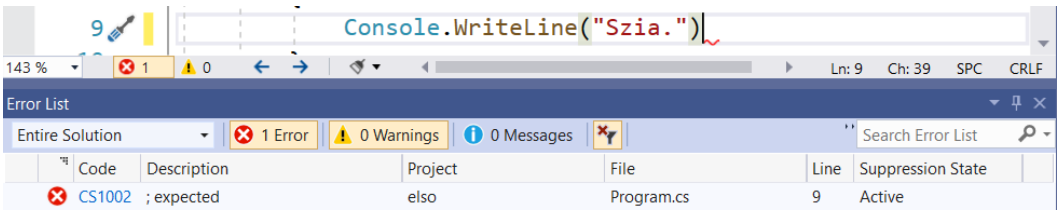
A fordítóprogram csak a **szintaktikának** (nyelvtani és formai szabályoknak) megfelelő helyes kódot tudja binárisra fordítani. Ha elírunk valamit, meg sem próbálja kitalálni, hogy mire gondolhattunk. (Szerencsére, mert így minimalizálható a félreértés esélye). Az IDE azonban több módon is segíti a hiba helyének és okának a meghatározását.



A fordítás (és futtatás) megszakítása után – és később folyamatosan frissítve – láthatjuk a hibajelzéseket és az *Error List* ablakban a hiba leírását is.

Hamar észrevehetjük, hogy bár a hibák jelzésének megvan az oka, gyakran nem tudja pontosan megmondani se a fordítóprogram se az IDE, hogy mit rontottunk el. Ezért mindig – már a kód írása során – figyelni kell a jelzésekre és minden hibás kódsort rögtön a beírást követően ellenőrizni, szükség esetén javítani kell. Kevés módosítás során fellépő hibát a gép pontosabban tud beazonosítani és az ember is könnyebben jön rá a hiba okára.

Ugyanebből a megfontolásból, a programot nem szabad karaktersorozatként írni. Mindig teljes szerkezeti egységeket kódoljunk! A kapcsos zárójelek és a tabulálás is jelzi a kód-blokkok egymáshoz való viszonyát. Mindig írjuk meg a tejes blokkot (nyitó és záró jeleket) és ezután egészítsük ki a jelek között a kódot!



A leggyakoribb hiba a pontosvessző hiánya, amit a „**; expected**” leírás jelez.

A helyes kódolás a helyes gondolkodási móddal is összekapcsolódik. Ezért hibás – és gyakran sikertelen is – a programkód soronkénti másolása, megjegyzése. Megtanulni egy kódot – az egyes részek szerepe és egymáshoz való viszonyának megértése nélkül – szinte lehetetlen.

Egy pillanat és eltűnt

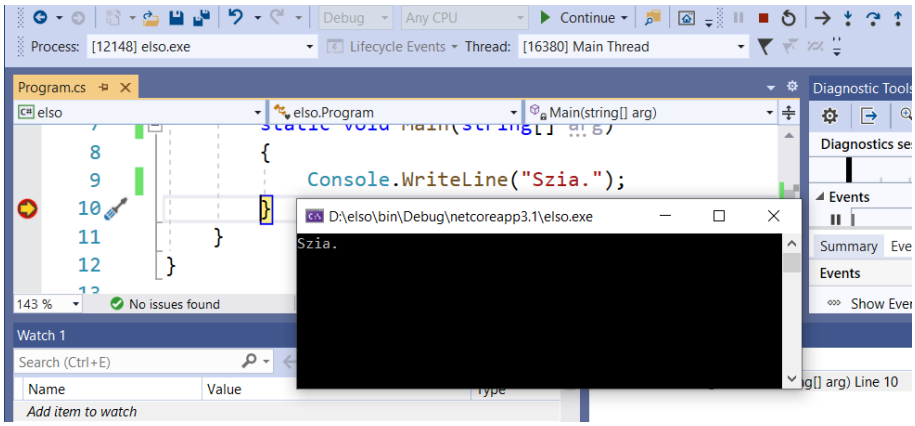
Felvillant a programunk, de nem láttunk semmit. Jól van ez így? Igen, a programokat azért írjuk, hogy gyorsan elvégezzék a feladatukat. Az pedig kimondottan zavaró lenne, ha a dolguk végeztével nem záródnának be, ha – például – egy böngésző bezárása után ottmaradna egy

konzolablak. Most azonban jó lenne látni, hogy mi is történt a programunkban. Ilyenkor megoldást jelenthet, ha

- ellenőrző módban futtatjuk a programunkat. Ez a **Debug** mód, az indítás előtt látható és beállítható a menüsoron.
- a programunk vége előtt megállítjuk a program futását, elhelyezünk egy **breakpoint**ot.

A *debug* hibakeresést jelent. Arra is van mód, hogy futtatás közben lépésenként megfigyeljük, mi történik a programban, esetleg módosítsunk a kódon. A véglegesített programot Release módban fordítással tudunk létrehozni. Ennek optimálisabb bináris kódja.

A programunkba egyes IDE-k esetén kissé eltérő helyen, de mindig a kód sorszámozása környékén lehet kattintással elhelyezni a piros pöttyöt, a breakpointot. A programunk itt meg fog állni. Minden aktív kódsor elé lehet breakpointot tenni. Ezeket ismételt rákattintással törölni is lehet, jobb gombbal rákattintva még feltételhez is köthetjük a megállítást.



A program futása minden breakpoint elérésekor, azaz a jelzett utasítás végrehajtása előtt megszakad. A helyet, ahol a végrehajtás éppen tart, sárga nyíl jelzi. Eközben a futtatás gombja átalakul, a program folytatását lehet kérni (F5), de a mellette megtalálható piros négyzettel (SHIFT+F5 billentyűkombinációval) a futtatás megszakítható. Emellett található általában az újraindítás, a különböző részletekben továbblépés lehetőségeinek gombjai.

A hibakeresés (debugolás) talán legnagyobb segítsége, hogy vizsgálhatjuk a program pillanatnyi állapotát. A Diagnostic Tools csak szakmabelieknek mond valamit, de a Watch ablak (alul) kezdő programozóknak is hasznos, érthető betekintést enged a programba.

A Visual Studio, a megállított program kódjában, az adatok fölé vitt egérmutató hatására előugró buborékban megmutatja a mutatott változó értékét (memóriatartalmát).

Az elkészült szoftver futtatása

A C# *compiler* nyelv, ezért a fordítás során elkészíti a futtatható állományt. Ezt az operációs rendszerből indítva is futtathatjuk.

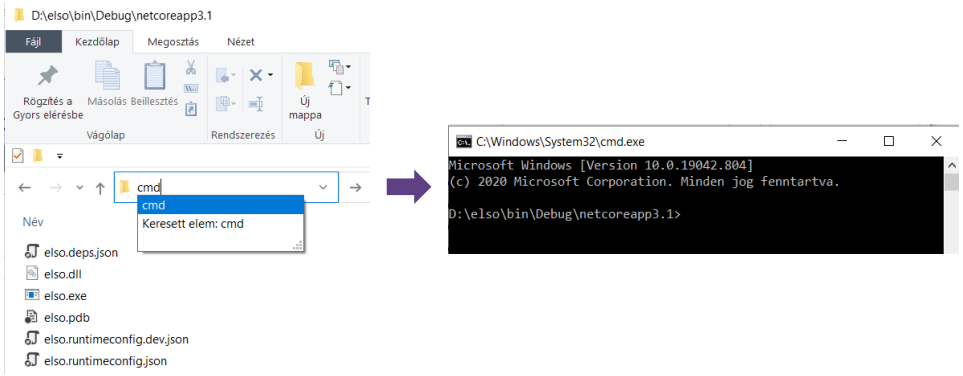
Nyissunk meg egy konzolablakot és lépjünk be a programunk mappájába, azon belül a bin/Debug (vagy, ha Release módban fordítottuk, akkor bin/Release) mappába!

- A konzolablak Windowson a Parancssor indításával vagy a `cmd` parancs futtatásával indítható.

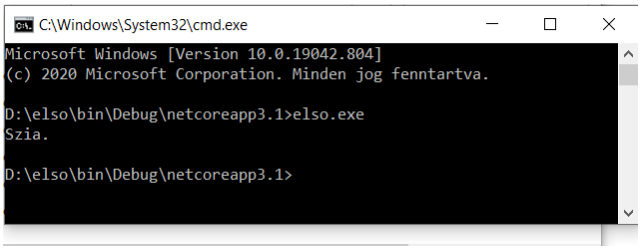
- A program mappájának kiválasztásához először a megfelelő meghajtóra váltsunk át, majd a `cd` parancs után írjuk be az elérési útvonalat.
- Ha a Debug (Release) mappában nem találjuk az `.exe` fájlt, akkor a fejlesztői verziónak megfelelő almappába tovább kell lépni.

Ez a megoldás eléggé nehézkes, ezért álljon itt egy Windows fájlkezelő trükk:

- Keressük meg fájlkezelőben az `.exe` fájlt! Rákattintva elindul a programunk, azonban semmi sem fogja megállítani ezért nem látható a működése.
- A fájlkezelő címsorába az elérési út helyére írjuk be: `cmd` és üssünk ENTER-t. A `cmd` parancs megnyit egy konzolablakot és – mivel az elérési út helyére írtuk – épp a megadott mappában várja az utasításunkat.



A konzolablakban beírva a program nevét (`elso.exe`) a programunk lefut. Mivel a konzolablakon belül indítjuk a futtatást, a program befejezése után is nyitva marad a konzol, a kiírt szöveg is látható marad.



Ha többször szeretnénk a programunkat konzolablakban futtatni, akkor nem érdemes bezárni az ablakot. A program nevét újra beírva, vagy felfelé kurzormozgató nyíl nyomkodásával visszaírva, a program (esetleg idő közben frissített verziója) újra futtatható. Ha „törölni” szeretnénk a konzolablak tartalmát, ezt a `cls` utasítással tehetjük meg.

A felhasználó mondja meg, mikor legyen vége

Amikor a kész programot futtatjuk – mivel konzolprogramot írunk – egy konzolablakban, akkor az összes kiírás látható marad. Hasonló módon lehetne az eredményt fájlban tárolni vagy átküldeni egy másik programnak helyben vagy interneten keresztül. De programozás közben sem így, konzolban futtatva, sem breakponinttal megállítva nem kényelmes ellenőrizni az eredményt. Ezért jobb lehet, ha kóddal meg tudnánk állítani a program futását egy, a felhasználótól várt billentyűleütésig. Ez a sokszor látott „Press any key to continue...”, amit most nem az IDE ír ki, hanem mi.

```
1.  using System;
2.  namespace Valami
3.  {
4.    class Program
5.    {
6.      static void Main()
7.      {
8.        Console.WriteLine("Szia!");
9.        Console.WriteLine("Press any key to continue...");
10.       Console.ReadKey();
11.     }
12.   }
13.  }
```

Mit csináltunk?

Létrehoztunk egy projektet, eközben keletkezett egy mappa a háttértáron. Ebben a mappában 2 + 1 fontos fájl van.

1. Az `sln` kiterjesztésű fájlnak ugyanaz a neve, mint a mappának. Ezzel lehet legközelebb megnyitni a fejlesztőkörnyezetben a projektünket.
2. A `Program.cs` fájlban van a kódunk, amit akár egy szövegszerkesztőben is megnyithatunk. De ez a lényeg! Általában ezt csatoljuk e-mailhez, ennek másolatát(!) adjuk be ellenőrzésre.
3. A projekt nevével azonos nevű `exe` fájl a lefordított programkód. Ez az, ami futtatható, ez maga a program, a szoftver, a termék. Nagyon örülünk neki, ha jól működik, de nem baj, ha töröljük, mert a kódból újra (és újra) elő lehet állítani. Mivel ez egy futtatható állomány, a vírusvédelem érzékenyen reagál a másolására, küldésére, más környezetben futtatására, távoli használatra. Jellemző, hogy e-mailben nem tudjuk elküldeni, ha mégis, akkor a „termékünk” karanténba kerül vagy tisztítás áldozatául esik a címzettnél. Ha tömörítéssel próbáljuk elrejteni, akkor a teljes tömörített mappára vár ugyanez a sors.
4. … és a többi fájl, ami a fejlesztőkörnyezet, a elemző szoftver és a fordító program működése során keletkezik. Folyamatos munka során nem érdemes ezeket a fájlokat törölni és semmiképp sem érdemes belenyúlni, módosítani.

Programozás IDE nélkül (csak erős idegzetűeknek)

A programozás lényege, hogy beírjuk az utasításokat egy szövegfájlba (hibamentesen), ezt a fájlt átadjuk a fordítóprogramnak, ami az utasításokat gépi kódra fordítja, amit ezután futtathatunk. Ehhez mindössze három dologra van szükség:

- egy parancssoros ablakra,
- egy szöveg készítésre alkalmas programra és
- a fordítóprogramra.

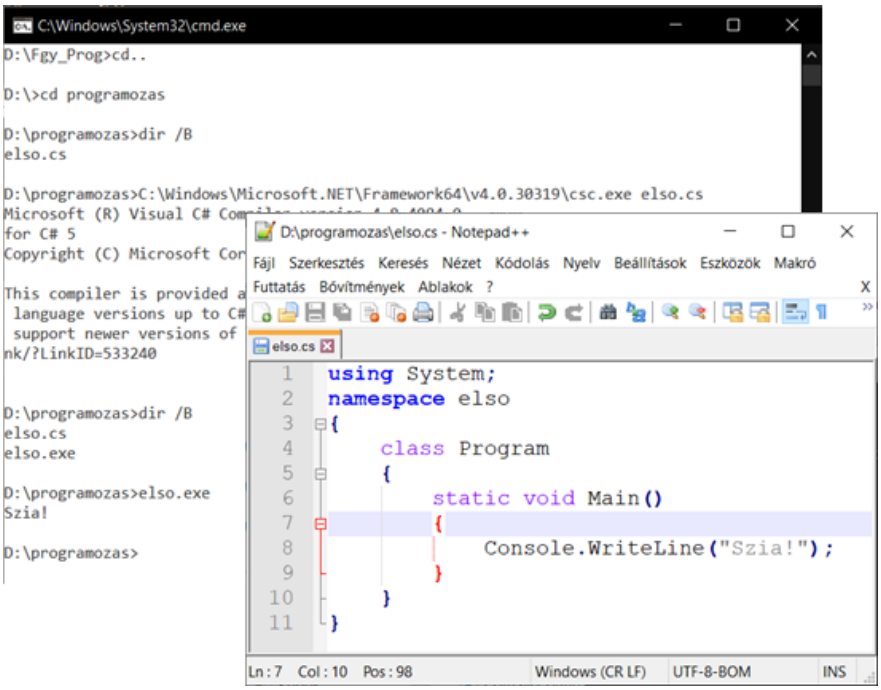
Windows operációs rendszeren a parancssoron belül a `copy` (másolás) parancs alkalmas arra, hogy a beírt szöveget fájlba mentsük. A `copy con elso.cs` utasítás a konzolról, azaz a billentyűzetről az `elso.cs` fájlba másolja a tartalmat. A parancs kiadása után ENTER-t ütve kezdhetjük a kód gépelését. A **CTRL+Z** billentyűkombinációval zárjuk le az adatsort (a „fájlt”).

Kódolás, fordítás és futtatás Parancssor ablakban (konzolalkalmazásban).

Ezt követően a `type elso.cs` parancs segítségével ki is írhatjuk a fájl tartalmát, de módosítani csak úgy lehet, ha elölről kezdjük. Ezért szoktak inkább egy szövegszerkesztő alkalmazást használni a kód megírására.

A fordításhoz meg kell keresni a `csc.exe`-t a gépen, a fájlkezelőből kimásolt útvonalat kell beilleszteni, és kiegészíteni a program nevével, majd szóközzel elválasztva a programkód neve következik. A fordító elkészíti a futtatható állományt, a futtatás eredménye, hogy a képernyőre (kimeneti konzol) kiíródik a „Szia!” szöveg.

Egyszerű – vagy kicsit többet tudó – szövegszerkesztővel elkészített fájl az előzővel azonos módon fordítható és futtatható konzolablakban.



Kódolás szövegszerkesztőben, fordítás és futtatás Parancssor ablakban.

Talán érdemes egyszer kipróbálni így is a programozást, de csak azért, hogy lássuk, mennyivel jobb egy IDE. Mostantól egy IDE (a Visual Studio 2019) segítségével fogjuk elkészíteni a kódot, ez fogja meghívni a fordítót, hogy elkészítse a futtatható állományt és általában futtatást is az IDE-ből indítjuk.

Változók, kiíratás, adat bekérése

Módosítsuk a programunkat úgy, hogy „Szia” helyett írja ki születésünk évét!

Szöveg, karakter és szám – adattípusok

A születésünk éve egy szám. A programozási nyelvekben az a szokás, hogy a szöveges adatokat idézőjelek közé kell írni, ha egyetlen karaktert adunk meg, azt aposztrófok közé írjuk, a számoknak kiegészítő jelölése. A születésünk évéről mi tudjuk, hogy szám, de megadhatjuk szövegként is, sőt – tagolva – karakterenként is esetleg számjegyenként szövegként. Ha több adatot szeretnénk egymásután kiírni, akkor ezeket a **+** jellel fűzhetjük össze.

Érdemes már most megjegyezni, hogy a szöveg karakterek sorozatát jelenti, de lehetséges olyan szöveg, amiben nincs egyetlen karakter sem. Ezt egymásmelletti idézőjelekkel adjuk meg: `""`.

Az aposztrófok között mindig pontosan egy karakter lehet, egymás mellé írva nincs értelmük. Például: `'s'` `'z'` egy-egy karakter. Speciális esetben jelezhet több karakter egyetlen karaktert. Ilyen például az ENTER `'\n'` karaktere, amit másképp nem tudnánk beírni. Hasonlóan lehet megadni a TAB billentyűleütésnek megfelelő `'\t'` karakterét és más vezérlő karaktereket, illetve a billentyűzeten nem elérhető betűket. Minden ilyen, speciális karakter a `'\'` backslash jellel kezdődik, ez a C#-ban (és sok más nyelvben is) az „**escape character**”

Kiírásokban nagyon gyakori, hogy a végére kell egy ENTER. A „Szia!” szöveg kiírását követően most azért lép új sorba a programunk, mert a `Console.WriteLine()`-ba ez bele van kódolva. Lezárhatjuk úgy is enterrel a kiírást, hogy beírjuk az idézőjelek közé: `"Szia!\n"`, vagy a szöveggel összefűzve: `"Szia!" + '\n'`. Ilyenkor a dupla enter elkerülése érdekében a kiíráshoz a `Console.Write()` formát használjuk.

A számokat is írhatjuk idézőjelbe, de ebben az esetben a C# szövegként kezeli az adatot Mit jelent ez? A legegyszerűbb, ha kipróbáljuk! A `+` jel a számokat összeadja, a szövegeket összefűzi. Nézzük meg, mi lesz az eredménye, ha a `"3"` `'3'` illetve 3-hoz hozzáadjuk az `"5"`, `'5'` és 5 adatokat!

A kiírás kódolása a cw snippettel: Console.WrilteLine() helyett elegendő beírni: cw és kétszer leütni a TAB billentyűt.

Az alábbi kódot beírva és futtatva látható néhány szabály és néhány talány.

```
   static void Main()
   {
1.    Console.WriteLine("3" + "5");
2.    Console.WriteLine(3 + 5);
3.    Console.WriteLine('3' + '5');
4.    Console.WriteLine("3" + 5);
5.    Console.WriteLine("3" + '5');
6.    Console.WriteLine(3 + "5");
7.    Console.WriteLine(3 + '5');
8.    Console.WriteLine('3' + 5);
9.    Console.WriteLine('3' + "5");
   }
```

```
35
8
104
35
35
35
56
56
35
```

Szövegként a 3 és 5 összefűzése 35 lesz, számként összeadva 8. Ha a 3-as és 5-ös karaktert adjuk össze, akkor egy harmadik eredményt kapunk. Ez a két karakter ASCII kódjának az összege, 51 + 53. Nagyon sok programozási nyelvben tapasztalhatjuk, hogy a karakter néha a karakterképét mutatja, máskor a bináris szám kódjának értékét. A 3. kiírt sorban azt látjuk, hogy az összeadásjel számként értelmezte a karakterek kódját. A 4–6. sorban látható, hogy a szöveg „erősebb” a számnál, mert az összefűzés biztosabban végrehajtható, mint az összeadás. Ha a példánkban "BK" + 5 szerepelne, akkor a BK5 egy lehetséges értelmezés, de a "BK" számként nem értelmezhető. Ezután már egyértelmű, hogy a karakter egy szám társaságában a + jelet összeadásnak tekinti és kódjának értéke hozzáadódik a számhoz, míg szöveg mellett karakterként összefűződik a szöveggel.

Kísérletünkből az is kiderül, hogy egy számjegyekből álló adat a kódolástól függően lesz szöveg vagy szám, illetve, ha egy számjegyű akkor lehet karakter is.

Változók

Íjunk új programot! A kódot mindig a `static void Main()` kapcsoszárójelei {} közé írjuk, ezért a lényeg körülbelül a 7. sortól lesz

```
7.  string allat = "ló";
8.  Console.WriteLine("allat");
9.  Console.WriteLine(allat);
```

Az első (7.) sor új elemet tartalmaz. Az `allat` itt egy *változó*, aminek azt adtuk értékül, hogy „ló”. Legegyszerűbb, ha a változókra olyan dobozként gondolunk, amibe bele tehetünk valamit. Itt egy szöveget tehetünk bele, ezt jelzi a változó neve előtt a `string` szó. A fordítóprogram számára a változó a memóriában lefoglalt területet nevesíti. A változó típusa alapján a fordító program foglalja le a megfelelő méretű memóriaterületet, figyel arra, hogy oda más ne kerüljön, csak az, amit a változóba beleteszünk. Amikor a változó (memóriabéli doboz) nevét írjuk le (idézőjelek nélkül), akkor a doboz *tartalmát* helyettesíti be. Ha a változó nevét idézőjelek közé írjuk, akkor a fordítóprogram szövegként tekint rá, amit meg sem próbál értelmezni.

A változókat azért hívjuk változóknak, mert az értékük változtatható. Ha új értéket adunk nekik, a régi egyszer s mindenkorra nyomtalanul eltűnik. Gondoljuk végig az alábbi program kimenetét, aztán futtassuk a programot, hogy kiderüljön, jól tippeltünk-e.

```
 7.  string allat = "ló";
 8.  Console.WriteLine(allat);
 9.  allat = "nandu";
10.  Console.WriteLine(allat);
11.  allat = "cickány";
12.  Console.WriteLine(allat);
```

változó létrehozása és értékadás

ugyanaz a változó, érték módosítása

Szabály, hogy a C# változónevei

- betűvel vagy alávonással (_) kezdődhetnek;
- betűvel, számmal vagy alávonással folytatódhatnak (írásjel és szóköz nem lehet bennük), azaz `anyu kora` helyett használjuk az `anyuKora` alakot (ez szokás C#-ban), vagy aláhúzással írjuk egybe a szavakat: `anyu_kora`;
- a kis- és a nagybetű használatára figyelnek, azaz `Majom`, `majom` és `majoM` három külön változó (a C# case sensitive);
- nem egyezhetnek meg az úgynevezett „foglalt szavakkal” és „kulcsszavakkal” – ilyen például az adattípusok megnevezése (`string`, `int`), a vezérlő szerkezetek megnevezései (`for`, `if`, `while`).

Nem szabály, de érdemes akként tekinteni rá: a programnak mindegy, hogy miként nevezzük el a változókat. Ha a fenti programban az „allat” helyett *mindenhol* „noveny” szerepelne, a program hibátlanul működne. A *programozónak* fontos a jó változónév, hogy ha holnapután előveszi a programját, még mindig el tudjon igazodni rajta. A változónevek választásával a program értelmezését segítjük.

Az ékezetmentes írásmód szintén nem szabály, hanem óvatosság kérdése. Ha a forráskódot olyan gépen szeretnénk megnézni (pl. külföldön), ahol nincs telepítve a használt ékezetes karakterkészlet, bajban leszünk az ékezetes változónevekkel.

A példán jól látszik, hogy a változó típusát egyszer, a legelső használatkor adjuk meg, pontosabban akkor, amikor a változót létrehozzuk, helyet kérünk számára a memóriában. Ezt később – amíg az adott változó számára a hely le van foglalva, azaz, amíg „él” a változó, – nem tudjuk megváltoztatni és nem lehet újra létrehozni ugyanolyan néven másik változót. A változó tartalmának módosításakor már csak a változó nevét kell beírni, a típust nem.

Arra is lehetőség, hogy a változót első lépésben csak megnevezzük – úgy mondják, hogy deklaráljuk. Programnyelvtől és adattípustól függően ezzel együtt a deklarált adatnak lesz lefoglalt helye a memóriában és abban valami. Lehet, hogy csak az, ami az előző használatkor

ottmaradt, de egyes programozási nyelvekben – mint a C++ is – egyes adattípusoknál a memóriafoglaláshoz kezdőérték beállítása is társul. Amikor nincs kezdeti értékbeállítás, akkor az adatot konstruktorral tudjuk létrehozni, amivel kezdőértéket is adunk, azaz inicializáljuk.

Így is lehet:

```
7.  string allat;
8.  allat = "ló";
9.  Console.WriteLine(allat);
```

típus és név deklarálása, kezdőérték: ""

érték módosítása

Karakterekből konstruktorral is lehet:

```
10. string csillagok = new string('*',10);
11. Console.WriteLine(csillagok);
```

létrehozás (konstruálás): típus, név, „new'' érték

Próbáljunk ki többféle módon adatot deklarálni és kezdőértéket adni neki! A programot állítsuk meg breakpointtal az elején és soronként léptessük. Minden lépés után vigyük az egérmutatót a változónév fölé, nézzük meg, milyen értéket ír ki!

Adat bekérése a felhasználótól

A legtöbb program kér adatokat a felhasználótól (de legalábbis a környezetéből: egy mérőműszertől, fájlból, hálózatról). A telefonunkba be kell írni az új telefonszámot, vagy egy listából kiválasztani a már rögzítettet. A böngészőnkbe beírjuk, hogy melyik webhelyet nyissa meg. A gépünknek megadjuk a jelszavunkat.

C#-ban a felhasználótól a `Console.ReadLine()` függvény fogadja a konzolablakban beírt adatot. Írjunk be ennyit egy programba, és futtassuk le! Azt látjuk, hogy a program vár. Ha nyomkodjuk a billentyűket, akkor amit lenyomtunk, kiíródik, ha ENTER-t nyomunk, a program futása befejeződik.

A felhasználó általában nem tudja, hogy a program éppen vár valamilyen adatra, vagy dolgozik valamin. Ezért, ha embertől, humán felhasználótól vár a program adatot, akkor illik a programban egy kiírással tudatni a felhasználóval, hogy az ő aktivitására van igény. Ezért az adatbekérés `Console.ReadLine()` **függvényét** általában megelőzi a `Console.Write()` **eljárás**.

A `Console.Write()` azért eljárás, mert csinál valamit: kiírja a képernyőre (konzolra) a zárójelei között megadott szöveget. A már korábban használt `WriteLine`-tól abban tér el, hogy a kiírás után nem lép új sorba – nem teszi ki az ENTER karaktert a kiírás végére. Az elnevezésben a „Line” egy ENTER-rel lezárt sorra utal.

A `Console.ReadLine()` az ENTER leütéséig gyűjti a karaktereket, a programunk azonban nem jegyzi meg, amit válaszolunk, mert nem mondtuk neki, hogy tárolja el. A `Console.ReadLine()` függvénysége abban nyilvánul meg, hogy van eredménye, azaz átadható, felhasználható „visszatérési értéke”, amit egy `string` típusú változóban el tudunk tárolni. Így tudjuk erre megkérni:

```
Console.Write("Hogy hívnak? ");
string nev = Console.ReadLine();
Console.WriteLine(nev);
```

A `Consol.Write()` és `Consol.WriteLine()` utasítás több dolgot is ki tud írni egymás után vagy akár egy szövegbe behelyettesítve is. Amit ki akarunk íratni, azt a zárójelen belül a ’+’ jellel összefűzhetjük vagy idézőjelek között megadhatjuk a kiírandó szöveget, amiben *helyőrzővel*

jelezzük az adatok helyét (és megjelenítési formáját) és ezt követően vesszővel elválasztva felsoroljuk a helyőrzőkbe behelyettesítendő adatokat (változókat). Ez a C# *„composite formatting”*kiírási módja. Az angol kifejezés arra utal, hogy a fordítóprogram a listában megadott kifejezéseket egyesíti az idézőjelek között megadott szöveggel.

Például: `Console.Write("Ezt " + "egymás mellé" + " írom");` Bármelyik szöveg helyett írható változó, ahogy azt már korábban láttuk. Ha a kiírandók egyike szöveg, akkor az eredmény az összefűzött szöveg lesz. A kiírásnál mindig figyelni kell a szóközök megfelelő elhelyezésére is, mert ami az idézőjeleken kívül van, az nem lesz benne a kiírásban.

Ennél áttekinthetőbb lehet a helyőrző használata. Ha középső részt könnyen változtathatóvá szeretnénk tenni, akkor a `Console.Write("Ezt {0} írom","egymás mellé");` formában is írhatjuk. Az „egymás mellé” ebben a formában könnyebben módosítható például „szépen”-re.

Egészítsük ki a fenti programot úgy, hogy a C# nevünkön szólítva bennünket, köszönjön nekünk! Próbáljuk ki mindkét kiírási módot:

```
 7.  Console.Write("Hogy hívnak? ");
 8.  string nev = Console.ReadLine();
 9.  Console.WriteLine("Szia " + nev + "!");
10.  Console.WriteLine("Szia {0}!", nev);
```

Több különböző adat a helyőrzőkbe írt számmal azonosítható. A 0 helyén a vessző utáni első adat lesz, de ezt követően vesszővel elválasztva megadhatjuk az 1, 2,… helyőrzőkbe behelyettesítendő adatokat is. Próbáljuk ki így is:

```
Console.WriteLine("Jó {1} {0}!", nev, "napot");
Console.WriteLine("Jó {0} {1} {2}!", "estét", nev, "úr");
```

Kérdések, feladatok

1. Mit írnak ki az alábbi programok? Gondoljuk végig, aztán próbáljuk ki a programokat, nézzük meg, hogy igazunk volt-e!

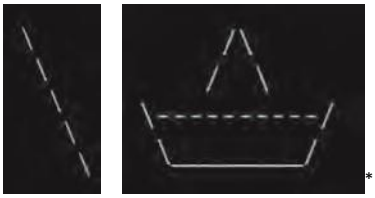
```
string allat = "kutya";
string kutya = "Bogi";
allat = "macska";
Console.WriteLine(allat);
```

```
string gyumolcs = "alma";
gyumolcs = "szilva";
string alma = "dinnye";
int dinnye = 3;
gyumolcs = alma;
Console.WriteLine(gyumolcs);
```

2. Rajzoljuk ki a jobb oldalon látható ábrát karakterek használatával! Figyeljünk arra, hogy a backslash („\”) különleges szerepű, az utána lévő karakter vezérlőkarakter, aminek a C# speciális értelmet adna, nem írná ki. Egy backslash kiírásához, kettőt kell írnunk – a `'\\'` egy darab ’\’ karakterként jelenik meg a konzolon.*

* Digitális kultúra 9. tankönyv 101. oldalán látható kép

3. Rajzoljuk ki az alábbi ábrákat karakterek használatával! Próbáljuk meg egyetlen kiíró utasítással megadni a teljes ábrát! Ehhez használhatjuk a `\n` és `\t` vezérlőkaraktereket is.

 *

4. Kérdezzük meg a felhasználótól (a program használójától) a vezetéknevét! Kérdezzük meg a keresztnevét is, és köszönjünk neki, a teljes nevén szólítva!

Számok és karakterláncok a programunkban

Eddig még csak karakterláncot (szöveget) tároltunk a változóinkban. Ebben a leckében változtatunk ezen, és számokat is használunk.

Hány éves a felhasználó?

Olyan programot fogunk írni, amely választ ad a fenti kérdésre.

1. Az első részfeladat egy olyan program megírása, amely megkérdi, hogy mikor születtünk, és ezt ki is írja nekünk. Ez eddigi tudásunk alapján megoldható, úgyhogy készítsük el önállóan az adat bekérését!

A `Console.ReadLine()` `string` típusú adatot ad át, amit emiatt csak `string` típusú változóban tudunk tárolni. Most ez nem lesz jó, mert a `string`-hez már hozzáadni sem tudtunk számot, így a kivonás során sem lesz használható. Az egész számok tárolásához egy másik adattípusra van szükség, aminek általános neve **integer**. Matematikából azt tanuljuk, hogy az egész számok halmaza végtelen, de a processzor nem tud végtelen nagy számmal dolgozni. Ezért a program írásakor fel kell mérnünk, hogy mekkora számokat kell a változónkban eltárolni és ehhez kell igazítani a választott típust. C#-ban a következő típusok közül választhatunk: `Byte`, `SByte`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`.

A `Byte` 0–255 értékek tárolására alkalmas, ahogy a neve is mutatja, 1 bájt helyet igényel. Gyakran hasznos, ezért kisbetűs változata is van a nevének: `byte`. Az `SByte` 'S' kezdete az előjelességet (signed) jelöli, így –128–127 értékek tárolására alkalmas.

Az „Int"-ek végén a szám a tárolt adatméret bitjeinek számát mutatja, azaz 2, 4 és 8 bájtos adatokról van szó. Az U kezdetű csak nemnegatív (unsigned) számokat tárol, de ezért kétszer olyan nagyot is, mint az U nélküli, ahol a számok fele negatív. A leggyakrabban használt egész típus az `Int32`, ezért ennek is van egy egyszerűbb neve: `int`. Egész típusként ezt fogjuk használni.

A bekért születési év egész számként tárolásához meg kell oldanunk még egy problémát: A `string`-ből, amit a `Console.ReadLine()` ad, valahogy `int`-té kell alakítani az adatot. Erre több mód is van. Általánosan, minden számtípusnak van értelmező függvénye, ami a megfelelő szövegből előállítja az adott típusú számot. A függvény neve `Parse()`. Például létezik

* Digitális kultúra 9. tankönyv 101. oldalán látható képek

int.Parse(), Int32.Parse(), Byte.Parse() függvény. A sok Parse() függvényt összefogták egy csomagba (egy osztályba), aminek a neve Convert. Ezen belül például a Convert.ToInt32() ugyanaz, mint az Int32.Parse(). Mivel az Int32 másik neve int, ezért az int.Parse() is szinonimája ezeknek. Néhány felhasználási mód:

```
int n = int.Parse("3000");
Int32 m = Int32.Parse("-3");
int k = Convert.ToInt32("400");
string Ft = Console.ReadLine();
int ar = Convert.ToInt32(Ft);
int jegy = int.Parse(Console.ReadLine());
```

Ha tudjuk, hogy csak egy számot fog beírni a felhasználó, akkor érdemes a legutolsó minta alapján eltárolni az értéket, mert ez a legrövidebb. A beírt kód hatására a konzolon beírt adatot a ReadLine() string típusként odaadja az int értelmezőnek, ami átalakítja egész számra és az eredményt az '=' jel „beteszi" a változóba.

2. A következő részfeladat a felhasználó korából a születés évének a meghatározása.

Ehhez a programunknak tudnia kell, hogy melyik évben futtatják. A jelenlegi év megkérdezhető az operációs rendszertől, de egyelőre megelégszünk azzal, hogy egy változóba beírjuk.

Azokat az értékeket, amelyek a program léte során nem változnak, **konstans**nak nevezzük, és a C# nyelvben a változó létrehozáskor jelöljük is a típus elé írt const jelzővel. Egyes nyelveknél lehetőség van arra, hogy a konstans értékét a futtatás során első alkalommal a felhasználó adja meg, de a C# ennél szigorúbb. A konstans értékét a kódban kell rögzíteni, ezt később nem lehet módosítani. A konstans változókat érdemes a program elején megadni, és sokan csupa nagybetűvel írják a változónevet, hogy később se felejtsék el, hogy fordítási hibát okoz, ha valami meg akarná változtatni az értékét.

Az eddigiek alapján a programunkban az alábbiaknak kell szerepelnie:

- értékadás egy konstansnak;
- a születési évére vonatkozó kérdés kiírása;
- a beírt adat (tárolása, majd … vagy „röptében" …) egésszé alakítása és tárolása;
- születés évének kiszámítása;
- eredmény kiírása.

Módosítsuk és egészítsük ki a kódot a fentieknek megfelelően:

```
 7.  const int IDEIEV = 2021;
 8.  Console.Write("Mikor születtél? ");
 9.  int szEv = int.Parse(Console.ReadLine());
10.  int felhasznaloKora = IDEIEV - szEv;
11.  Console.WriteLine("{0} éves vagy. ", felhasznaloKora);
```

Ha megfigyeljük a mondatainkat, látjuk, hogy kódoláskor előbb foglalkozunk a lejegyzett tennivalók jobb oldalán (végén) lévő dolgokkal és csak utána a bal oldallal. Ezt a kód begépelésekor nyugodtan kövessük. A 3. sort például lehet úgy írni, ahogy gondoljuk: „kell egy Console.ReadLine(), azután ezt be kell tenni az int.Parse()-ba, az eredményét el kell tárolni (mondjuk) a felhasznaloKora – int típusú – változóban. Igaz, hogy így a kurzor oda-vissza cikázik, vagy az egérrel sokszor kell az eddig megírtak elé kattintani, de mégis hatékonyabban tudunk dolgozni, mert a gondolatainkat nem kell fejben előre megformáljuk, hanem a lejegyzés során áll össze. Idővel, sok gyakorlás után egyes sorok kigondolása automatizmus lesz, akkor már természetes, ha egyvégtében, balról jobbra írjuk a kódot.

Ismét látható, hogy a kód másolása miért nem alkalmas a programozás tanulására: az értelmes másoláshoz tudni kellene, hogy hol kezdődik a gondolat, ehhez ki kellene találni a szerző gondolatát. Más fejével gondolkodni még a saját megoldás kitalálásánál is nehezebb.

Nem egész számok

3. Most, hogy már tudjuk a felhasználó korát. Adjuk meg, hogy mennyibe kerülne a gyertya a születésnapjára! Egy gyertya ára Euroban 0,18–0,35 €, a pontos árat kérjük be!

Természetesen tudunk tizedestörtekkel is dolgozni programjainkban. A tizedesjel lehet pont vagy vessző, hogy éppen melyik, az az operációsrendszertől, a fejlesztőkörnyezettől és a programozási nyelvtől is függ. Valószínű, hogy ha a felhasználóként írjuk be az adatot, akkor az operációsrendszer beállítása miatt tizedesvesszőt kell írnunk, de a programkódba a programozási nyelv szabványa szerinti tizedespontot lesz a megfelelő.

A tizedestörteket a programozók lebegőpontos számnak is nevezik, ami angolul *floating point number*, innen származik a típus neve: *float*. A lebegőpontos számokat normál alakhoz hasonlóan ($m*2^k$) tárolja a program, ahol a szám pontosságát az határozza meg, hogy a mantissza (m) hány bites. A `float` típusnál kétszer pontosabb a `double` típus, ezért általában ezt használjuk.

Mindkét típusnak van átalakító függvénye: `float.Parse()` és `double.Parse()`, illetve létezik a `Convert.ToDouble()` megadási mód is.

A szövegből megfelelő `Parse()` függvénnyel tudunk számmá alakítani, de szükség lehet különböző számtípusok közötti átalakításra is. Azokban az esetekben, amikor egyértelmű az átalakítás és ebből nem származhat adatvesztés, akkor ez „implicit”, azaz rejtetten végbemegy. Például, egy egész szám bármikor használható `double` típusú adatként. Ugyanakkor visszafelé már jelezni kell, hogy át szeretnénk értelmezni az adattípust és néha az egész számról is közölnünk kell, hogy most tizedesjegyes számként szeretnénk használni. Ezt az „explicit”, azaz kijelentett átalakítást úgy adjuk meg, hogy az érték elé zárójelben megmondjuk, hogy milyen adattípusra akarjuk átalakítani. Például, az `(int)`3.9 értéke 3, egész szám; a `(double)`4 értéke 4.0, dupla pontosságú lebegőpontos szám.

A gyertyák árának meghatározása:

```
12.  Console.Write("Hány Euro egy gyertya? ");
13.  double egyGyertya = double.Parse(Console.ReadLine());
14.  double gyertyakAra = felhasznaloKora * egyGyertya;
15.  Console.Write("A gyerták {0} Euróba kerülnek.", gyertyakAra);
16.  int euro = (int)gyertyakAra;
17.  int cent = (int)((gyertyakAra - euro) * 100);
18.  Console.WriteLine("A gyertyák ára: {0} Euro {1} cent", euro, cent);
```

Legyen egyenlő

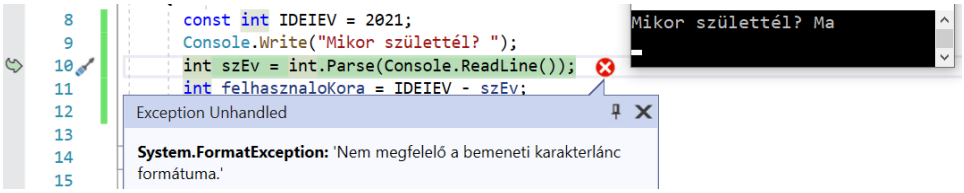
Eddig három műveleti jelünk volt, a „+” a „*” és a „–” jel, de mostanra el kell fogadnunk, hogy programozáskor az egyenlőségjel is műveleti jel. Alapvetően mást jelent ilyenkor az egyenlőségjel, mint matematikaórán. Ott állításokat, kijelentéseket fogalmaztunk meg vele (Például: kettő egyenlő háromból egy; hatszor hat egyenlő harminchattal.) Programozáskor az egyenlőségjel egy művelet elvégzésére való felszólítás: *legyen* a változó értéke…

Programíráskor az egyenlőségjel az értékadás műveletének a jele, olvashatjuk „**legyen egyenlő**” formában. Értékadáskor mindig az egyenlőségjel jobb oldalát értékeli ki a program elsőként, majd a kapott eredményt értékül adja az egyenlőségjel bal oldalán lévő változónak.

Az értékadás egyenlőségjelét egyes programozási nyelvekben emiatt egy kettősponttal kezdik. A jelölés ismertségét mutatja, hogy a – három ponthoz hasonlóan – egyesített formája is létezik a jelölésnek és több szövegszerkesztő írás közben át is alakítja. Külön írva „a := b”, egyben „a ≔ b”. Egyesek szerint még szemléletesebb, ha nyíllal jelöljük az értékadást: a ← b.

És amit nem lehet átalakítani számmá?

Abból hibaüzenet lesz. De nem a fordításkor dobja fel az IDE a „szokásos” ablakot, hanem a program futása közben:



A 10. sorban az int.Parse() zárójelei között olyan szöveg van, amit az nem tud számmá átalakítani

Természetesen kezelhetők az ilyen jellegű hibák, csak mi még nem tanultuk meg a módját. Még sokáig abból indulunk ki programkészítéskor, hogy a felhasználótól érkező bemenetet nem kell ellenőriznünk, *validálnunk*. Egy „igazi” alkalmazás esetében a hibákra való felkészülés (felhasználótól kapott rossz bemenet, elfogyó háttértár, megszakadó hálózati kapcsolat stb.) a programnak igen jelentős része.

Kiegészítés: Karakterlánccá alakítás

A C#-ban kétféle karakterlánc van. Az egyszerűbb típus pusztán a karakterek sorozatát jelenti, de van egy „okosabb” adattípus is, a `string`, amit már használtunk is. Láttuk, hogy két `string` összefűzhető, és azt is láttuk, hogy a `Console.Write()` és `Console.WriteLine()` utasításokban a kiírás szövegében helyőrzővel jelezhetjük több kiírnivaló megjelenítését, amelyeket ezt követően vesszővel elválasztva sorolunk fel. Ilyenkor a vesszővel felsorolt adatok bármilyen típusúak lehetnek, a kiírásban szöveges formában jelennek meg.

Egy-egy adat szöveggé alakítását első sorban az adattípus `ToString()` függvénye végzi. Ezt a függvényt úgy írták meg, hogy a fordítóprogram akkor is lefuttatja, ha nem írjuk ki, de szükség van rá. A korábban már kipróbált példánkban is valójában ez történt:

```
Console.Write("Ekkor születtél: {0}.", szEv.ToString());
```

A `ToString()` függvény azontúl, hogy képes „magától” működni, további „csodákra” is képes: a zárójelek között megadhatjuk a szöveggé alakítás formátumát. A lehetőségek tanulmányozásához a *C# tostring format* kulcsszavakkal érdemes a dokumentációban, illetve más forrásokban példákat keresni, itt most csak néhány mintát adunk meg:

- Egész szám adott számú jeggyel kiírása: pl. 7.`ToString("D3")`
- Szám százalékformátumban: pl. 7.`ToString("P0")`.
- Nem negatív egészek hexadecimális alakja: pl.: 32.`ToString("X")`
- Egyéni számformátum: pl. 1234.`ToString("# ##0")`.

```
Console.WriteLine(7.ToString("D3"));
Console.WriteLine(7.ToString("P0"));
Console.WriteLine(32.ToString("X"));
Console.WriteLine(1234.ToString("# ##0"));
```

```
007
700%
20
1 234
```

A megjelenítés formátumát nemcsak az adat `ToString()` függvényével állíthatjuk be, hanem a helyőrzőnél is. Ilyenkor a helyőrző az adott formátumot „átadja” a `ToString()`-nek, ezért a megadás módja sok szempontból hasonló, de a kód rövidebb, mert a `ToString()`-et nem kell kiírni.

```
Console.WriteLine("{0:D3}", 7);
Console.WriteLine("{0:P0}", 7));
Console.WriteLine("{0:X}", 32);
Console.WriteLine("{0:# ##0}",1234);
```

```
007
700%
20
1 234
```

Elmondhatjuk, hogy a `ToString()` a szöveggé alakítás során szinte minden igényünket kielégíti, de ha lenne még valami – például, hogy közben csengessen is –, annak sincs akadálya, mert programozással felülírható a működése.

Kiegészítés: String igazítása

A számok szöveggé alakítását követően gyakran felmerül az igény a számok kiírásának igazítására. Például jó lenne helyiértékesen, egymás alá kiírni a számokat; jó lenne egymás alá úgy kiírni neveket, hogy a vezetéknév után a keresztnevek is egymás alatt (oszloposan) helyezkednek el.

A C#-ban erre is van lehetőség, de ez már nem a számnak, hanem a szövegnek, a `string`-nek a függvénye. A `PadLeft(int)` függvény a szöveg elé annyi szóközt tesz, hogy a szöveg hossza legalább a zárójelek közé írt egész szám legyen. Hasonlóan a `PadRight(int)` függvény a szöveg után teszi a kiegészítő szóközöket. Középre igazítás nincs, de a szöveg hosszának ismeretében – ez a szöveg `Length` tulajdonsága – és a fenti két függvény kombinációjából és egy kis matekozással megvalósítható. Néhány példa a szélesség beállítására:

```
Console.WriteLine("almafa".PadLeft(10));
Console.WriteLine("{0}|{1}|{2}", "Ó".PadRight(8),
"Pál".PadRight(8), 7.ToString().PadLeft(3));
Console.WriteLine("Ó".PadRight(8) + '|' +
"Pál".PadRight(8) + '|' + 7.ToString().PadLeft(3));
```

```
    almafa
Ó       |Pál     |  7
Ó       |Pál     |  7
```

Ennél az igazítási módszernél a karakterek száma lehet számított érték. De ha konstans (állandó) értékkel is megelégszünk, akkor a szélességet a helyőrzőben is megadhatjuk. A pozitív érték balra, a negatív jobbra igazítja a szöveget a megadott szöveghosszon belül:

```
Console.WriteLine("{0,10}" , "almafa");
Console.WriteLine("{0,-6}|{1,-6}|{2,3}",
"Ó", "Pál", 7);
Console.WriteLine("{0,-6}|{1,-6}|{2,3:D3}",
"Ó", "Pál", 7);
```

```
    almafa
Ó     |Pál   |  7
Ó     |Pál   | 007
```

Kérdések, feladatok

1. Keressük meg és próbáljuk ki, hogyan lehet a 3,14159265 számot két tizedesjeggyel kiírni!
2. Próbáljuk ki a helyőrzőkben a szöveg igazítását és a formátumok megadását! Írjuk ki a szövegeket összefűzéssel, illetve a formátum és szövegigazító függvények használatával is!
3. Kérjünk be egy kilométerben mért távolságadatot a felhasználótól, és írjuk ki tengeri mérföldre átváltva! (Egy tengeri mérföld 1852 méter.) Ha elkészültünk, megírhatjuk a feladat megfordítását.
4. Kérjünk be a felhasználótól két számot, tároljuk őket egy-egy változóban!
   a) Adjuk össze őket, és írjuk ki az eredményt!

b) Írjuk ki az eredmény elé, hogy „Az összegük:”!
c) Írjuk ki magukat a számokat is! Ha például 2-t és 3-at adott meg a felhasználó, akkor a kimenet legyen ilyen: `2 és 3 összege: 5`.
d) Írjuk át a programot szorzásra! (A szorzás jele a *.)
e) Írjuk meg a többi matematikai számítást is! (Az osztás jele a /, a kivonásé a –.)
f) Kihívást jelentő feladat: A hatványozás és a gyökvonás megírásához, a `Math` függvényeit lehet használni.

5. Vegyük elő a korábban megírt, nevünket kiíró programot! Ebben a feladatban már megoldottuk a vezeték- és a keresztnév kiírását. Bővítsük úgy a programot, hogy kérdezze meg a születési évünket is, és írja ki a nevünkkel egy sorban: `Kék Blamázs, 2011`. A születésiév-megállapító programunkkal egybegyúrva készíthetünk olyan programot is, amelyik megkérdezi a neveinket, a születési évünket, és a bulvárcikkekben szokásos formában, a korunkkal együtt írja ki a nevünket: `Fehér Karnis (21)`.
6. Adott óra, perc, másodperc hármassal megadott időt váltsunk át másodpercre! (Az adatokat nem kell mindenképp a felhasználótól kérni, beírhatjuk őket a programba is.) Sikeres megoldás után készítsük el a feladat megfordítottját!
7. Kihívást jelentő feladat: Milyen szöget zár be egymással a kis és a nagy mutató adott időpontban? Kérjük be az óra és perc értéket, írjuk ki a mutatók közötti szöget fokban!

Mint a legtöbb programozási nyelvben, a C# nyelvben is meg tudjuk mondani egy osztás maradékát. Ez az úgynevezett modulo osztás, vagy egyszerűbben csak *mod*, a jele a `%`. Így a `9 % 4` értéke 1, mert kilencet néggyel osztva egy a maradék. Ugyanakkor az egész számok (`int`) körében a `/` jel a bennfoglaló osztás jele, az eredménye egész szám. A `9 / 4` értéke 2, azaz kilencben a négy kétszer van meg (a maradékot a másik, a `%` jellel elvégzett művelet eredményeként adhatjuk meg). A lebegőpontos (`double`) értékek között a `/` jel részekre osztást végez. Emiatt `9.0 / 4.0` értéke 2.25.

Ha két egész (`int`) szám hányadosát lebegőpontos számként szeretnénk megkapni, akkor az egyiket (tipikusan a számlálót) explicit módon át kell alakítani lebegőpontossá. A `(double)9/4` értéke 2.25, mert a 9 átalakult 9.0 értékre, emiatt a ’/’ a részekre osztást el tudja végezni.

Különösen arányok és százalékok számításánál („hányad rész”) figyeljünk, mert bennfoglalással, egészek közötti osztással a 9/10 értéke 0, ami 100-zal szorozva: 9/10*100 is nulla. Ha a számításban van szorzás és osztás is, akkor általában érdemes a szorzásokat előre venni: 100*9/10 értéke 90, de ilyenkor is problémát jelenthet, hogy az eredmény egész szám.

Segíts magadon, az IDE is megsegít!

Az integrált fejlesztői környezetek, így a Visual Studio is, többféle eszközzel segíti a programozót a programjának a megírásában. Ezek közül most a kódkiegészítés, a helyérzékeny kódajánlás, a kódszínezés, a helyi súgó és a snippet-ek használatát nézzük át.

A színek jelentése

Az IDE általában lehetővé teszi, hogy többféle színvilág közül válasszunk. Van, aki a világos, más a sötét színsémát szereti, ez ízléstől, megszokástól és a környezet megvilágításától is függ. A kódok színezése az egyes színsémákban eltérő, de a programozási nyelv elemei alapján a kódon belül egységes jelentéssel bír. A C# nyelv objektumorientált, ezért a kód színezése azt mutatja, hogy az adott kifejezés (szó) az objektumok szempontjából milyen funkciójú. A

kódszínezés olyan, mint nyelvtanból a mondatelemzés. Nem a színnek van jelentése, hanem az azonosan megjelenőknek van közös jelentése.

▶

Amikor a programot létrehozzuk, rögtön látható néhány kulcsszó, foglalt szó: `using`, `namespace`, `class`, `static`, `void`. Azután ez kiegészül a `string`, `int`, `double` egyszerű változók tíousaival, de majd még bővül a lista…

A programunkban minden „`using`” után egy-egy megnevezés van. Ezek már létező névterek, „`namespace`”-ek.. Olyanok, mint amit épp mi is írunk, aminek a neve a `namespace` után van írva. Ha a `using` utáni megnevezés színe más (fakóbb), mint a `namespace` utáni szó színe, akkor abból a névtérből még nem használunk semmit, pillanatnyilag feleslegesen van ott, kitörölhető. Hasonló fakó megjelenést, esetleg zöld hullámos jelzést tapasztalhatunk egy-egy változónk esetén is, ha az adott változót még nem használtuk.

▶

A `class` szó után kötelezően a `Program` szó jön, a `Program`, tehát, egy osztály. Az osztályok az objektumorientált programozásban az objektumoknak (és működésüknek) a leírásai. Objektum például:

- egy játékban a harcos, a pálya, a fegyver, a bábu;
- az űrlapon a beviteli mező, a címke, a lista;
- egy rajzolóprogramban a téglalap, a vonal, az ellipszis;
- szövegszerkesztőben a karakter, a bekezdés, a táblázat …

A sort a végtelenségig lehet folytatni, mivel az OOP alapokon írt programokban minden objektum. A `Program` azért különleges, mert ez az osztály egyetlen objektumot ír le, a futtatandó programunk objektumát. Ugyanakkor a mások által megírt osztályok objektumait – a színek alapján – használjuk: `Console`, `String`, `Int32`, `Double` és majd még jön néhány. Ezek olyan valamik, mint a `Program`, azzal a különbséggel, hogy nem írjuk, hanem a megírtat felhasználjuk, emellett nem az éppen írt `namespace`-ben vannak megírva, hanem a System névtérben. (Ha a `using` System; sort töröljük, akkor a fentiek mindegyike hibás, ismeretlen lesz.)

Érdekes kapcsolat van az `int` és `Int32`, a `string` és `String`, a `double` és `Double` között. A foglalt szavakat a System névtér nélkül is érti a fordítóprogram, míg az objektumokat nem. Ugyanakkor a működésük teljesen azonos. Az `Int32` az `int` szinonimája. Egyúttal sejthető, hogy a változóink is objektumok (legalábbis objektumszerűen viselkednek).

▶

Az OOP programozási nyelvek körében jellemző, hogy az objektumnak a dolgait (adatait, részeit …) **pont**tal kapcsoljuk az objektumhoz. Ezt láthatjuk a `Console`.`Write`(), 7.`ToString`() esetén. A `Write` a `Consol`-**nak a** kiírója, a `ToString`() a 7-es szám**nak a** szöveggé alakítója.

Már korábban volt arról szó, hogy a `Console`-nak az egyik kiírója a `WriteLine`() *eljárás*, a beolvasója a `ReadLine`() *függvény*. A színük azonos, és ugyanilyen színű a `Main`() is, ami a `Program` osztálynak a fő *eljárása*. A színbéli azonosság nem véletlen. Az eljárás valójában egy speciális függvény. A `Main`() eljárásnak nincs „visszatérési értéke”, csak csinál valamit. Ezt jelzi a neve elé írt `void` szó. A függvényeknél a `void` helyett valamilyen adattípus szerepel, például `int`. Ugyanakkor, közös jellemzőjük, hogy a nevük utáni kerek zárójelben megadhatunk *paramétereket*.

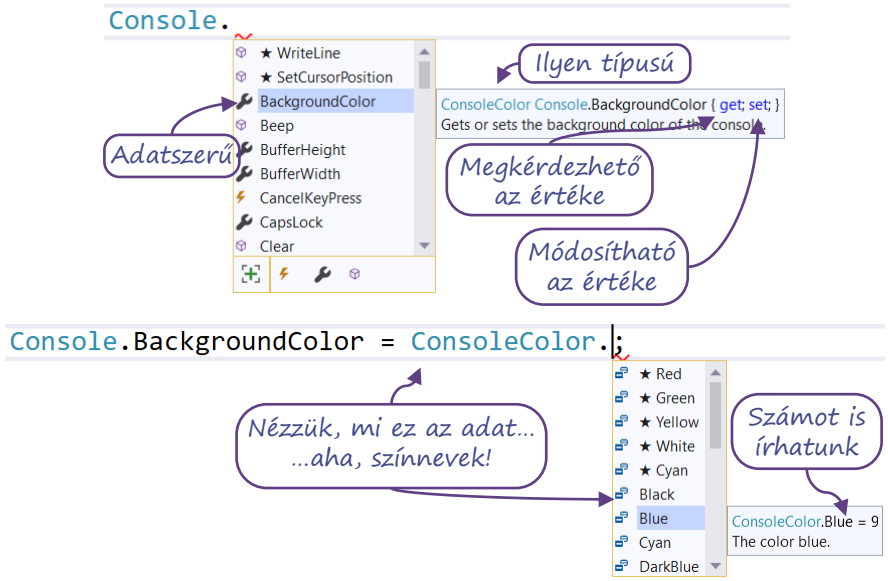
A kód megértését, hibakeresést segíti a többi szín is. A változónevek kiemelése mellett a konkrét adatoknak (literáloknak) típustól függő a színezése. Színpaletta cseréjével vagy egy másik kódszerkesztőben nézve a kódot lehet, hogy a Visual Studioban különböző színű elemek színe egyforma lesz, miközben az azonos színűekből elkülönülnek a számok, a műveleti jelek (operátorok), az utasítást lezáró pontosvesszők.

Kódkiegészítés és helyi súgó

Amikor elkezdünk írni valamit, az IDE igyekszik a segítségünkre sietni, kiírja a beírt karakterek alapján lehetséges kifejezések listáját. Célszerű erre – azaz a képernyőre – figyelni a billentyűzet helyett, mert a felajánlott lehetőségek közül nyilakkal választhatunk, a tabulátorral beírhatjuk a kívánt kifejezést, amiben valószínűleg nem lesz helyesírási, gépelési hiba.

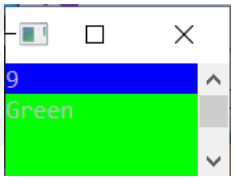
Bármelyik változó vagy objektum után beírva egy pontot, megkapjuk a benne lévő (rejlő) lehetőségek listáját. A Visual Studioban érdemes figyelni a listában a jelöléseket:

- Csillag jelöli az IDE szerint leginkább az adott helyzethez illő lehetőségeket.
- Doboz jelöli a függvényeket – aminek a beírása után kell a (), a kerek zárójel;
- „Villáskulcs” valamilyen adat- vagy objektumszerű tulajdonságot jelöl.
- A lapocska konstans értéket, feljegyzést.
- Ha esetleg sárga villámot látunk, az az esemény jele, ilyen esemény lehet egy konzolalkalmazás esetén a megszakítás, grafikus program esetén az egérrel kattintás.



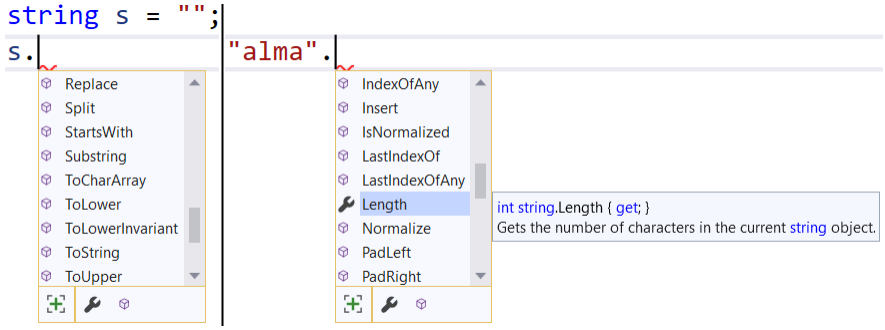
Próbáljuk is ki!

```
1.  Console.BackgroundColor = ConsoleColor.Blue;
2.  ConsoleColor hatterszin = Console.BackgroundColor;
3.  Console.WriteLine((int)hatterszin);
4.  hatterszin += 1;
5.  Console.BackgroundColor = hatterszin;
6.  Console.WriteLine(hatterszin);
```



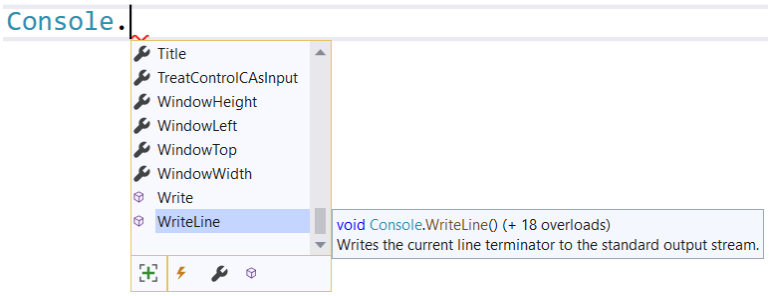
Az 1. sor a felső ábrán látható ’set’ miatt lehetséges, a 2. sorban a ’get’-et használjuk ki. A 3. sorban explicit konvertálhatjuk a `ConsoleColor` típusú adatot, mert látjuk, hogy ez valójában egy számot jelöl (Blue = 9). A 4. sorban szintén a „szám”-ságot („egész”-séget) használjuk ki, amikor eggyel növeljük az értéket. Az 5. sorban újra a ’set’ tulajdonságot használjuk ki, míg a 6. sorban a `hatterszin` láthatatlan `ToString`() függvényét vetjük be a színnév kiírásához.

Egy OOP nyelvben az adatok, változók is objektumok. Egy szöveges adatnak, egy `string` vagy `String` típusú változónak nagyon sok függvénye és néhány tulajdonsága van, érdemes ezeket is tanulmányozni:

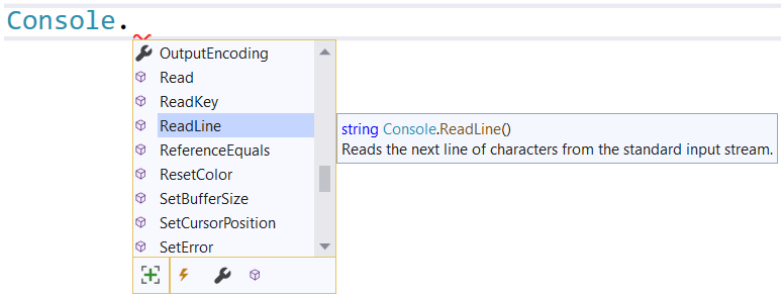


Látható, hogy a szöveg `Length` tulajdonsága egy egész szám, amit le tudunk kérdezni (`get`), azaz írhatunk olyat, hogy `int hossz = "alma".Length`, ekkor a hossz értéke 4 lesz; de nem módosíthatjuk értékadással a `Length` értékét (nincs set), hibás az `"alma".Length = 5;` utasítás. Ha belegondolunk, a szöveg hosszától pont ezt a viselkedés várjuk el.

Ha nem tulajdonságot, hanem függvényt (doboz jelűt) választunk, akkor a leírás adatai is másról szólnak.

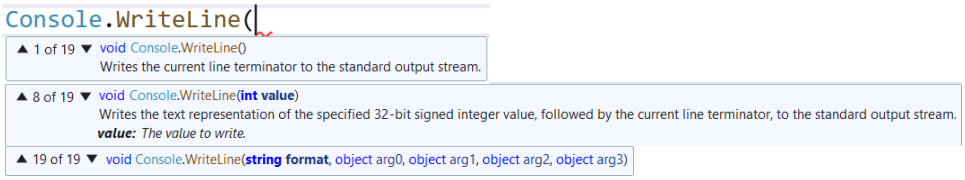


A `Console.WriteLine`-ról már az előtte látható doboz alapján tudjuk, hogy függvény, ezért a neve után kell a () kerek zárójel. Kiválasztva megtudhatjuk, hogy eljárás (`void`, azaz a semmi a visszatérési értéke). Az is kiolvasható, hogy a zárójelek közé nem kell semmit sem írni, – ilyenkor csak egy üres sort fog kiírni – de 18-féleképpen(!) lehet paramétereket megadni hozzá.



A `Console.ReadLine`-ról is láthatjuk, hogy függvény. A leírásából megtudjuk, hogy `string` típusú az eredménye (a visszatérési értéke). A zárójelek közé nem kell semmit sem írni, sőt, nem is szabad, mert nincs további lehetőség jelezve.

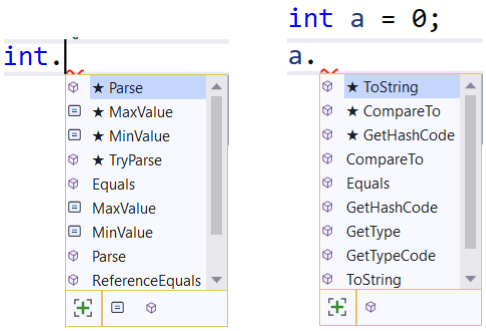
A `Console.WriteLine` utáni nyitó zárójelet beírva a beírási módokat is áttekinthetjük:



A függvény lehetséges paraméterezéseit a baloldalon látható nyilakra kattintva érjük el. Az egyes lehetőségeknél minden paraméternek láthatjuk a típusát (például `int`, `string`, `object`) és a szerepére vonatkozó kifejezést, változónevet (például: value, format). A függvény szerepének leírása alatt először az első paraméterről láthatunk részletes leírást, de ezt megadva, a vessző beírása után már a következő paraméterről tájékoztat a helyi súgó.

Kérdések, feladatok

1. Nézzük meg, mit tud a `string`, az `int`, illetve a `double` típus, mit tudnak az ilyen típusú változók! Bár nem lesz minden teljesen világos, válasszunk ki néhány hasznosnak tűnő tulajdonságot, függvényt, olvassuk el, hogy hogyan használhatók!



2. Játékprogramokban szinte nélkülözhetetlen az idő mérése. Keressük meg a dátum és az idő adattípust, az időintervallum adattípust! Írjuk ki a képernyőre a pontos időt!

Tanulást is támogató eszközök

A programkódot azért írjuk, hogy a fordítóprogram számára érthető módon megadjuk, mit csináljon a programunk. Amellett, hogy a fordítóprogramnak értenie kell a kódokat,

ugyanolyan fontos, hogy mi is értsük a kódot. Ne csak abban a percben értsük, amikor megírjuk, hanem másnap is, sőt két hónap múlva is, vagy akár évek múlva is. Egy programkód olvasottságát vizsgálva azt láthatjuk, hogy egy kódrészletet többször olvas és fordít saját nyelvére ember, mint fordítóprogram. Ezért a programkódot a szakmabeli programozók is a kód humán olvasójának szóló jegyzetekkel egészítik ki. Ez annyira természetes, hogy a programozási nyelv kétféle megjegyzésre is lehetőséget ad.

C# nyelvben a hosszabb megjegyzéseket, jegyzeteket a `/*` és `*/` jelek közé írhatjuk. Azt szokták mondani a jelölés magyarázataként: ami e jelek között van az – a fordítóprogram számára – nem oszt, nem szoroz. Ez a jelölés lehetőséget ad arra, hogy a kódunkba beírjuk a feladat szövegét, esetleg hosszabb magyarázatokat írjunk a szöveghez, emellett arra is jó, hogy a kód szavai között rövid megjegyzéseket írjunk.

A rövidebb, egysoros megjegyzések `//` után írhatók. Ezt használjuk a változónevek jelentésének kifejtésére, egy-egy összetett képlet szerepének tisztázására. Az IDE abban is segít, hogy ha olyan kódrészletet írunk, ami pillanatnyilag nem szükséges, akkor a kijelölt sorok mindegyikét ezzel a jelöléssel aktív kódból megjegyzéssé tehessük, illetve újra aktiváljuk:

A programozás egyik alapelve, hogy a feladatokat bontsuk le kisebb – és még kisebb – egységekre, majd a programunkat ezekből az alapelemekből építsük fel. Ezen elv alapján tudjuk használni a konzol kiíró és beolvasó függvényeit is, nem nekünk kell megírni a részleteket. A többszázezer soros programok kódját úgy írják a programozók, hogy egy-egy részlet – például egy függvény – általában kevesebb, mint tíz utasítást tartalmaz. Vannak olyan kódrészletek is, ami ennél jóval több utasítást tartalmaznak, de nincs értelme tovább bontani. Ilyen például egy játékban egy harcos megjelenítése (részletesen megadva, hogy hol, milyen méretben, milyen kinézettel jelenjen meg), mert sok változónak kell kezdőértéket beállítani. A kód áttekinthetőségét jelentősen növeli, ha a kódot ilyen esetekben is valamilyen módon egységekre tudjuk osztani, mert az IDE minden értelmezhető egységet képes egysorosra becsukni, illetve kibontani.

A C# nyelvben egy függvényen – mint például a `Main`() – belül a `#region` és `#endregion` jelzéspárral lehetőséget kapunk a belső egységek létrehozására. A `#region` sorába leírást is adhatunk, olyan, mintha megjegyzést írnánk. A program fordításakor ezek a bejegyzések sem kerülnek be a bináris állományba, csak számunkra segítik az kód áttekintését, értelmezését.

```
using System;
namespace BeKiFeladatok
{
  class Program
  {
    static void Main()
    {
      #region 1. feladat
      /*Keressük meg és próbáljuk ki, hogyan lehet a 3,14159265 számot
       * két tizedesjeggyel kiírni!
       */
      double kbpi = 3.14159265;
      string spi = "3,14159265";
      Console.WriteLine("{0:F2} vagy {0:0.00} vagy {1}", kbpi, spi.Substring(0, 4));
      #endregion
      2. és 3. feladat
      4. és 5. feladat
      6. feladat
      7. feladat
      #region 8. feladat
      /*Milyen szöget zár be egymással a kis és a nagy mutató adott időpontban?
        Kérjük be az óra és perc értéket, írjuk ki a mutatók közötti szöget fokban!
      */
      Console.Write("Az óra értéke: ");
      int ora = int.Parse(Console.ReadLine());
      Console.Write("A perc értéke: ");
      int perc = int.Parse(Console.ReadLine());
      int percSzog = perc * 6; //perc * 360 / 60
      itt a kihívás megoldása
      Console.WriteLine("A mutatók {0}°-os szöget zárnak be egymással.",szog);
      #endregion
    }
  }
}
```

▶ Jegyzetelési módszerek: tananyag, megjegyzések, fejezetek

Elágazások

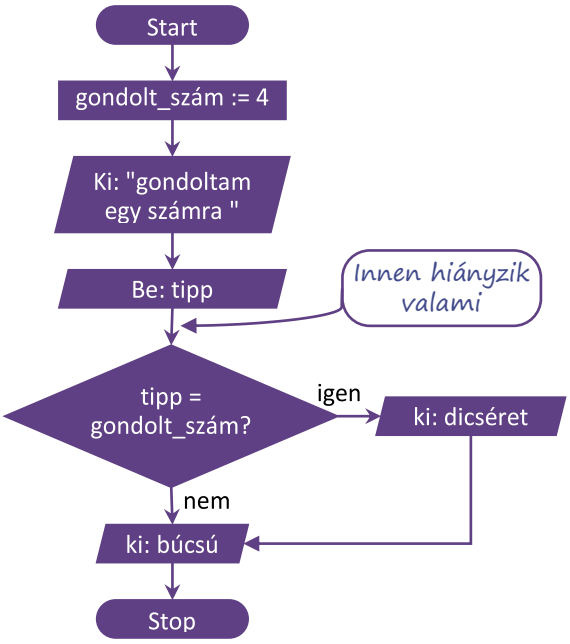
Eddig csupa olyan programot írtunk, ami elkezdődött az elején, sorban egymás után végrehajtott minden utasítást, aztán kilépett. Ebben a fejezetben ezen változtatni fogunk.

Gondoljunk egy számra

A jegyzet elején már láttunk egy olyan programot, amelyikben elágazás van. Az a program mást csinál, ha nem vagyunk még tizennégy évesek, és mást, ha már betöltöttük ezt az életkort. Hasonló működésű az ugyanott megismert „Mi a neve Mátyás királynak?” program.

Az alábbi folyamatábra is egy hasonló programot ír le. Nincsenek benne konkrét utasítások, mert fontosabb, hogy először átgondoljuk, hogy mit csinál a program, ráérünk utána azon elmélkedni, hogy miként kódoljuk.

1. Mit csinál a program?
2. Mi az a lépés, amit nem tüntettünk fel? (Ha most nem jövünk rá, nem probléma: hamarosan úgyis megírjuk a programkódot, akkor szólni fog a C#.)
3. Hogyan olvassuk ki a := jelet? Mi a megfelelője C#-ban?



Miközben a programunkat folyamatábrával ábrázoljuk, algoritmust (receptet) adunk a bennünket érdeklő probléma megoldására. A folyamatábrák elég látványosak, de hamar lelógnának a könyvlapról, így aztán a gyakorlatban gyakrabban használunk egy másik algoritmusleíró eszközt, a mondatszerű leírást. A leírás szabályaira rá fogunk érezni.

```
gondolt_szám := 4
ki: "gondoltam egy számra"
be: tipp
tipp átalakítása egésszé
elágazás
ha tipp = gondolt_szám:
    ki: dicséret
elágazás vége
ki: búcsú
```

4. Vessük össze ezt a leírást a folyamatábrával!
5. Melyik az a művelet, ami a mondatszerű leírásban már megvan, de a folyamatábrában még hiányzik?

Készítsük el a fenti két algoritmusleíró eszközzel megtervezett program kódját!

```
 8.  int gondolt_szam = 4;
 9.  Console.Write("Gondoltam egy számra. Tippeld meg! ");
10.  int tipp = int.Parse(Console.ReadLine());
11.  if (tipp == gondolt_szam)
12.      Console.WriteLine("Ügyes!");
13.  Console.WriteLine("Pápá.");
```

Ez szándékosan két egyenlőségjel

Az előző sor folytatása is lehet, de így áttekinthetőbb a kód

Teszteljük a programunkat: adjuk meg a helyes megoldást, de próbáljuk ki helytelennel is!

A program következő változatában – más szóval: verziójában – kicsit bővebben dicsérünk, illetve a hibásan tippelő felhasználókat kicsit ugratjuk. A mondatszerű leírás a következő:

```
gondolt_szám := 4
ki:"gondoltam egy számra"
be: tipp
tipp átalakítása egésszé
elágazás
ha tipp = gondolt_szám:
    ki: kétsoros dicséret
különben
    ki: ugratás
elágazás vége
ki: búcsú
```

Módosítsuk ez alapján a folyamatábrát (segítség: a rombuszból lefelé nem lesz nyíl, de balra igen, és a két nyíl a rombusz alatt összetalálkozik)!

A dicséret második sora újabb kiírás utasítást jelent az előző alatt. Írjuk be az új sort behúzva és behúzás nélkül vagy egysorban az előzővel:

```
13.    Console.WriteLine("Gratulálok!");
```

Mindegyik esetben teszteljük a programunkat! Sajnos egyik beírási mód sem jó. A C# a feltétel teljesülése esetén csak egy utasítást hajt végre. Ezért, ha több utasítást szeretnénk írni, akkor ezeket „egybe kell foglalni”, **kapcsos zárójelek** közé kell zárni. Mivel a C#-nak mindegy, hogy hol van új sor vagy tabulátor, írhatjuk a teljes utasítást egyetlen sorba, de így számunkra olvashatatlanabb lesz. A tördelésre kétféle szokás alakult ki.

- Az egyik szerint a nyitó kapcsos zárójel a feltétel sorának végén marad, utána az utasítást új sorban kezdjük egy tabulálással beljebb.

```
11.  if (tipp == gondolt_szam){
12.    Console.WriteLine("Ügyes!");
13.    Console.WriteLine("Gratulálok!");
14.  }
```

Nyitó { Utasítások Záró }

- A másik szerint a nyitó kapcsos zárójelet új sorba írjuk és egy sorral lejjebb és beljebb írjuk az első utasítást.

```
11.  if (tipp == gondolt_szam)
12.  {
13.    Console.WriteLine("Ügyes!");
14.    Console.WriteLine("Gratulálok!");
15.  }
```

Nincs vége, **nincs ;** (pontosvessző)

Nyitó { Utasítások Záró }

Akárhogy is kezdjük, a többi utasítás az első alatt, ugyanannyira behúzva kezdődik, a záró kapcsos zárójel pedig külön sorban, az `if`() i-jével egyvonalban van.

A C# nyelvre – a `namespace`, a `class` és a `Main`() függvény tördeléséhez hasonlóan – a második elrendezés a jellemző, ezért itt is ezt alkalmazzuk. Ez az elrendezés helyben átláthatóbb, de néhány sorral hosszabb kódot eredményez az első tördelési módhoz képest.

A kódolást segíti az if snippet:	Az „if” után két TAB beírásával a feltétel kerek és az igaz ág kapcsos zárójelei is bekerülnek.

Ha megvagyunk, írjuk meg az ugratós részt is. A „különben” szó angolul „`else`” – ezt kell használnunk kódoláskor. Írhatjuk a záró kapcsos zárójel után vagy külön sorba is, utána a kód tördelése az előző esetekhez hasonló.

A kódolást segíti az else snippet:	Az „el” beírását követő két TAB kiegészíti a szót és a hamis ág kapcsos zárójelei is bekerülnek.

A kész program:

```
 8.  int gondolt_szam = 4;
 9.  Console.Write("Gondoltam egy számra. Tippeld meg! ");
10.  int tipp = int.Parse(Console.ReadLine());
11.  if (tipp == gondolt_szam)
12.  {
13.    Console.WriteLine("Ügyes vagy!");
14.    Console.WriteLine("Gratulálok!");
15.  }
16.  else
17.  {
18.    Console.WriteLine("Hosszan gondolkodtál rajta? :)");
19.    Console.WriteLine("Nem érte meg. ;)");
20.  }
21.  Console.WriteLine("Pápá.");
```


A programunk tanulságai:

1. Ami az `if` után következik, az az elágazás feltétele.
2. A feltételvizsgálatban két, egymás utáni egyenlőségjel kell.
3. Ami az elágazás ágaiban van (a fenti program 13–14. és 18–19. sora), az kapcsos zárójelek között van. A C#-ban az elágazás egy ágán egy utasítás lehet önállóan, több utasítást kapcsos zárójellel „egyesítenünk” kell.

Az összehasonlítás jelölése

A programozásban általában az egy egyenlőségjel egy felszólítás (ismerjük már, értékadáskor használjuk: „legyen egyenlő”), a két egyenlőségjel kérdés: A tipp egyenlő-e a gondolt számmal?

Nem csak azt szoktuk kérdezni, hogy egyenlő-e két szám. Lehet kisebb (<), nagyobb (>), kisebb vagy egyenlő (<=), nagyobb vagy egyenlő (>=), illetve nem egyenlő (!=). Ezeket a `!=` kivételével matematikából is ismerjük.

Az informatikai jelölések nagyon gyakran a matematikai jelölést veszik át, de az egyenlőségjelnél problémát jelent, hogy a matematika kétféleképpen használja az „=” jelet. Egyrészt két kifejezés egyenlőségét vizsgálva; például egyenletek felírása során. Másrészt definíciókban, például az y = f(x) függvény felírásakor a legyen egyenlő értelemben. Informatikában háromféle megoldás van a félreértések feloldására:

1. A két egyenlőségjel a kifejezésen belüli helye miatt megkülönböztethető. Például táblázatkezelésben =HA(A1=A2;B1;C1) a képletet kezdő egyenlőségjel „legyen” értelmű, az A1=A2 között összehasonlító szerepe van. Ilyen esetekben a „nem egyenlő” jelölése a <> szokott lenni, mert az áthúzott egyenlőségjelet nem tudjuk kódolni.
2. Ahogy C#-ban is használják: Az „=” az értékadó, az „==” az összehasonlító egyenlőségjel. A „nem egyenlő” jelölése ebben az esetben mindig a „!=”.
3. Ahogy a mondatszerű leírásban használják: Az értékadó egyenlőségben a „legyen” kifejezése a „:”, így a jelölése „:=”, az összehasonlításokban használják a „=” jelet. A „nem egyenlő” jelölése „<>”, de nem okoz félreértést „!=” sem.

Az „==” és a „!=”nemcsak számok, hanem szövegek egyezésének vizsgálatára is használhatók, de az ábécé szerinti rendezettséget a relációjeleknél bonyolultabb összehasonlító függvényekkel lehet csak vizsgálni. (Érdeklődőknek: `CompareTo`()).

Összetett feltétel

Szeretnénk bővíteni a programunkat. Ha csak eggyel tippelt mellé a felhasználó, akkor ezt eláruljuk neki. Elsőként az algoritmus mondatszerű leírását módosítjuk. Az új, „különben ha” ágat megvalósító C#-utasítás az **else if**.

```
gondolt_szám := 4
ki:"gondoltam egy számra"
be: tipp
tipp átalakítása egésszé
elágazás
ha tipp = gondolt_szám:
    ki: kétsoros dicséret
különben ha csak egyet tévedett
    ki: csak egyet tévedtél
különben
    ki: ugratás
elágazás vége
ki: búcsú
```

Akár neki is foghatnánk a kódolásnak, de hogy programozandó a „ha csak egyet tévedett”? Ilyen utasítás nincs, ezért az algoritmusunkat egy-két lépéssel tovább kell finomítanunk.

Vegyük észre, hogy kétféleképp lehet egyet tévedni: eggyel nagyobbat vagy eggyel kisebbet tippelve. Az „eggyel nagyobbat tippelt” így írható le:

```
tipp = gondolt_szam + 1
```

A másik feltétel megfogalmazása nem okozhat gondot, de ezek szerint két feltétel lett az egyből. Megírhatjuk úgy az algoritmust (és a programot), hogy két „különben ha” ágat adunk meg, ugyanazzal a kiírandó üzenettel, de ez nem szerencsés – például azért, mert ha módosítani kell az üzenetet, akkor két helyen is módosítanunk kell, és az egyiket előbb-utóbb elfelejtjük megtenni. Alakítsuk inkább a két feltételünket egy összetett feltétellé:

```
tipp = gondolt_szam + 1 vagy tipp = gondolt_szam - 1
```

A két feltételből így egy lett, hiszen a „vagy” szó kapcsolja össze őket, ami igaz értéket ad, ha az egyik teljesül. Ha „és” kapcsolná őket össze, mindkettőnek teljesülnie kellene, hogy a teljes összetett feltétel igaz legyen. C# nyelven a `vagy` jelölése **||** (billentyűzeten kétszer AltGr+W); az `és` jelölése **&&** (billentyűzeten kétszer AltGr+C).

A kód most így néz ki (mindegyik zárójelezési szokást 1-1 esetben alkalmazva):

```
 8.  int gondolt_szam = 4;
 9.  Console.Write("Gondoltam egy számra. Tippeld meg! ");
10.  int tipp = int.Parse(Console.ReadLine());
11.  if (tipp == gondolt_szam)
12.  {
13.    Console.WriteLine("Ügyes!");
14.    Console.WriteLine("Gratulálok!");
15.  }
16.  else if (tipp == gondolt_szam + 1 || tipp == gondolt_szam - 1)
17.    Console.WriteLine("Ó, csak eggyel tévedtél");
18.  else {
19.    Console.WriteLine("Hosszan gondolkodtál rajta? :)");
20.    Console.WriteLine("Nem érte meg. ;)");
21.  }
22.  Console.WriteLine("Pápá.");
```

Kérdések

1. Hány **else if** ág, illetve hány **else** ág szerepelhet egy elágazásban?
2. Melyiket kell utolsóként megadni?
3. Melyiknek nincs feltétele?

4. Létezhet olyan elágazás, amelyikben nincs egyik sem?
5. Kihívást jelentő feladat: Az alábbi sor miért nem jó?

```
16.  else if (tipp == gondolt_szam + 1 || gondolt_szam - 1)
```

Összetett feltétel logikai szabályai

Két feltétel összekapcsolása a „vagy” szóval némiképp hasonlít az összeadáshoz. Ha a hamis állítást 0-val jelöljük, az igaz állítást 1-gyel, akkor a vagy eredménye – ami az igaz érték, ha az egyik teljesül – ugyanolyan értelmű, mit az összeadás eredménye: nem nulla, ha bármelyik tagja nem nulla. Az összeadás során két számot adunk meg és az eredmény egy szám lesz, azaz az összeadás számokon végzett kétoperandusú művelet. Hasonlóan, a „vagy” a logikai kifejezéseken végzett kétoperandusú művelet. A „vagy” műveleti jele C#-ban a **||**, de más nyelvekben az angolból származó **OR** jelöli, mondatszerű leírásban és folyamatábrán magyarul is írhatjuk: **VAGY**. Matematikában jelölhetjük az **∨** jellel, illetve – a hasonlóságot kiemelve, a Bool-algebrában a jele **+**.

Hasonlóan, kétoperandusú logikai művelet az „és”, műveleti jele C#-ban az &&, más programozási nyelvekben az angolból származó AND jelöli, mondatszerű leírásban és folyamatábrán magyarul is írhatjuk: **ÉS**. Matematikában jelölhetjük az **∧** jellel, illetve – a hasonlóságot kiemelve, a Bool-algebrában a jele *****. Ahogy a szorzásnál mondhatjuk, hogy egy szorzat nulla, ha bármelyik tényezője nulla, az ÉS műveletre is jellemző, hogy hamis lesz az értéke, ha az egyik feltétel hamis. (Másképp is mondhatjuk: igaz lesz az értéke, ha mindegyik igaz).

Azt gondolnánk, hogy a VAGY és az ÉS feltételei felcserélhetőek (a műveletek kommutatívak), azonban ez csak a matematikai felhasználások esetén igaz. Az informatika, a programozás erre a két műveletre a „**rövidzár**-kiértékelés” (másképp: lusta kiértékelés) szabályát alkalmazza. Ez azt jelenti, hogy az összetett feltételt csak addig értékeli ki a program, amíg az eredmény (hogy igaz-e vagy hamis) nem biztos. VAGY esetén, ha az első feltétel (operandus) igaz, akkor a másodikat nem vizsgálja. Az ÉS művelet esetén, ha az első feltétel hamis, akkor a másodikat már nem vizsgálja. Természetesen, ahogy összeadásokból vagy szorzásokból is lehet több egymás után, az ÉS és VAGY műveletekből is lehet többet megadni. Ilyenkor az eredményt addig számolja a program, amíg balról olvasva egy VAGY előtt igaz eredmény, illetve egy ÉS előtt hamis eredményt kap. Az ezt követő feltételeket, műveleteket már nem nézik.

Gondoljunk bele, mi is hasonlóképp végeznénk el a műveletet. Ha 10 számot össze kell szoroznunk, de a 3. szám a 0, akkor nem kell figyelnünk, hogy utána milyen értékek vannak…

A rövidzár kiértékelésnek a programozásban nagyon komoly gyakorlati oka van. Tipikus feladat, hogy az egyik feltétel az, hogy egy dolog létezik-e (értelmezhető-e), a másik pedig, hogy jó-e, azaz megfelelő-e. Például, ha egy számot bekérünk és ki akarjuk írni, hogy páros szám-e, akkor az összetett feltétel: a beírt adat egész szám ÉS a beírt adat páros szám. Súlyos programhibát okozhat, ha a kapott adat szöveg és a program megpróbálná kettővel elosztani. Ezért programozási szabály, hogy *összetett feltételben mindig azt a feltételt írjuk előre, amelyik a vizsgált adat létezését, érvényességét, értelmességét ellenőrzi, hogy a rövidzár-kiértékelés megállítsa a kiértékelési folyamatot egy esetleges programhiba előtt*.

Ha a feltételeket nem azonos műveletekkel kapcsoljuk össze, akkor felmerül a kérdés, hogy van-e a szorzáshoz és összeadáshoz hasonló precedencia-szabály, erősebb-e – azaz magasabb rendű-e – az egyik, mint a másik. Általában az ÉS erősebb (matekosan a * erősebb), de ez nem minden programozási nyelvben van így, ezért célszerű zárójelezéssel egyértelművé

tenni, hogy a többszörösen összetett feltételeket hogyan csoportosítjuk. Inkább legyen több zárójel, mint egy félreértett szabály.

A harmadik, informatikában jellemzően használt logikai művelet az egyoperandusú tagadás. A NEM művelet az utána szereplő feltétel eredményének ellenkezőjét adja, így, az igaz-ból hamis, a hamisból igaz értéket állít elő. Jelölése C#-ban és a C-vel rokon nyelvekben a felkiáltójel (!), más nyelvekben az angolból származó **NOT** jelöli, mondatszerű leírásban és folyamatábrán magyarul is írhatjuk: **NEM**. Matematikában jelölhetjük a ¬ jellel, illetve a komplementer képzéssel való hasonlóságot kiemelve, a Bool-algebrában a jele a felülvonás. Például az a != e így írható le az a == e tagadásaként a fenti jelölésekkel: !(a == e), NOT(a == e), NEM(a = *e*), ¬(a = e), $\overline{a=e}$.

Véletlenszám-előállítás

Meglehetősen unalmas lehet, hogy a programunk mindig a négyre gondol. A legtöbb programozási nyelvben van valamilyen módszer véletlenszám előállítására, más szóval *generálására*. A C#-ban erre – a `Console`-hoz és az `Int32`-höz hasonlóan – az előre megírt `Random` típusú objektumot (eszközt) használhatjuk. A `Console` objektum – mivel konzolalkalmazást készítünk – a program kezdetétől a végéig létezik. Ezzel szemben a `Random` eszközre nincs mindig szükségünk, azt a programon belül a változókhoz hasonlóan létre kell hozni:

1. megadjuk az objektum típusát: `Random`
2. adunk neki egy nevet, például `randgen`
3. létrehozzuk az objektumot: `= new Random();`

`new`, mert most hozzuk létre. A program futtatásakor a `Random()` a memóriában lefoglalja a megfelelő méretű helyet és még a rendszeridőt is megnézi, mert az alapján ad kezdőértéket a további véletlenszámok generálásához. Azt mondjuk, hogy a `Random()` a `Random` típus létrehozója, szaknyelven a *konstruktora*.

Ha véletlenszámra van szükségünk, akkor a `randgen` nevű `Random` objektum valamelyik generátor függvényét kell megadnunk. Jellemzően vagy a `Next()` függvénnyel kérünk egy egész számot, vagy a `NextDouble()` függvénnyel egy 0 és 1 közötti számot. A részleteket a helyi súgóban érdemes elolvasni. A `randgen` nem változó, hanem „számgyártó” eszköz. A számokat nem tárolja. A `randgen.Next()` a `Console.ReadLine()`-hoz hasonlóan működik, de az adatot nem a billentyűzetről kapja, hanem belső működésével állítja elő.

A programunk első sorai a következőképp alakulnak:

```
 8.  Random randgen = new Random();
 9.  int gondolt_szam = randgen.Next(1,6);
10.  Console.WriteLine("Súgok: {0}", gondolt_szam);
11.  Console.Write("Gondoltam egy számra. Tippeld meg! ");
12.  int tipp = int.Parse(Console.ReadLine());
```

Létrehozzuk a generátort.

A tesztelés során hasznos kiírni

ezek közül adja: 1 2 3 4 5

De, miért…?

Miért ugyanolyan színű a `Random` típus megnevezés és a `Random()` konstruktor? A `Random()` nem eljárás? Nem olyan, mint a `Console.WriteLine()`?

Nézzük, mi lenne, ha a konstruktor eljárás lenne. Valószínűleg, valahogy így adnánk meg: `Random.New()`. Ez létrehoz egy új objektumot. Ha eljárás, akkor az eredménye `void`, azaz a semmi, de itt lett valami, egy – a példában `randgen`-nek nevezett – objektum jött létre. Akkor ennek függvénynek kell lennie, aminek az eredménye egy `Random` típusú objektum. De mi az, ami a `Random.New()` függvényen belül létrejött, amit végül készen átad a függvény? Ez csak egy `Random` típusú adat lehet. És az hogyan jött létre? Jól látható, hogy ha függvény, vagy speciális esetként eljárás lenne a konstruktor, akkor a „Mi volt előbb: a tyúk, vagy a tojás?” problémába futunk. Ezért a konstruktor olyasmi, mint egy függvény vagy eljárás, de különleges szerepe miatt mégsem az.

Az eltérő szerep, az „önmaga létrehozása” még nem magyarázat arra, hogy miért nem úgy írjuk, hogy `Random.New()`. Leginkább azért, mert más programozási nyelvekben is az eléje írt `new` kulcsszóval emelik ki az új memóriaterület lefoglalását. Ez azonban nem változtatja meg – színében sem – a pont előtti részt.

Nem lehetne megspórolni az egyik `Random` leírását? De, meg lehet spórolni. Egyrészt a típust ki tudja találni a fordítóprogram a konstruktorból, ezért ezt helyettesíthetjük a `var` szóval.

```
var randgen = new Random();
```

Ez azonban megnehezíti a kód későbbi értelmezését, mivel nem a sor elején jelzi a típust. Másrészt, ha a .Net 5.x (C# 9.0) vagy ennél újabb környezetet használunk, akkor a konstruktor nevét elhagyhatjuk. Ezt a „javítást” a Visual Studio fel is ajánlja:

```
Random randgen = new ();
```

Bár tanulmányaink során a típus és a konstruktor neve sokáig meg fog egyezni, ez csak az általános eset. Az objektum orientált programozás alapelve az „öröklés”; a C# nyelvben minden adat az `object` típusból származik. Ezért írhatjuk, hogy `object id = new Int32();`. Megfelelő előkészítés után értelmes (és hasznos) lehet ez a kód is: `Ember en = new Tanulo();`

Már csak egy kérdés van: Miért kell a zárójel? Azért, mert így a konstruktorok képesek előre megadott speciális beállításokkal, paraméterekkel létrehozni az objektumot. Például a `DateTime p13 = new DateTime(2023,01,13);` nemcsak létrehoz egy `p13` nevű dátumot, de be is állítja a tulajdonságait a megadott év, hónap és nap értékeknek megfelelően.

Elágazások és véletlenek alkalmazása

Feladatok

1. Kérjünk be jelszót a felhasználótól és hasonlítsuk össze a programban tárolttal! Ha a felhasználó eltalálta a jelszót, akkor írjuk ki, hogy „Helyes jelszó.”, különben „Hozzáférés megtagadva.”.

A feltételek megfogalmazásakor használható a „>”, a „<”, a „>=” és a „<=” operátor is.

2. Kérjünk be két számot a felhasználótól! Írjuk ki a nagyobbat!
3. Állítsunk elő két véletlenszámot és kérdezzük meg a felhasználótól az összegüket! Ha helyesen válaszol, dicsérjük meg!

```
 8.  Random rnd = new Random();
 9.  int egyik = rnd.Next(1,11);
10.  int masik = rnd.Next(1,11);
11.  Console.Write("Mennyi {0} és {1} összege? ", egyik, masik);
12.  int tipp = int.Parse(Console.ReadLine());
13.  int osszeg = egyik + masik;
14.  if (tipp == osszeg)
15.    Console.WriteLine("Valóban annyi, ez igen!");
```

4. Írjunk olyan programot, amelyik bekéri két kosárlabda csapat nevét és a meccsen elért pontszámokat, majd kiírja a mérkőzés eredményét (a felhasználó válaszai vastagabbal szedve):

```
Mi az egyik csapat neve? Tóparti királyok
Hány pontot szerzett? 78
Mi a másik csapat neve? Talpasi csodatévők
Hány pontot szerzett? 54
Az összecsapás eredménye:
Tóparti királyok - Talpasi csodatévők
78 : 54
Tóparti királyok nyert.
```

5. Vegyük elő azt a programunkat, amelyik a felhasználó nevét és korát kezeli. A felhasználónak javasoljunk életkorának megfelelő olvasnivalót!
   - 0–3 év: „Totyogóknak a kettes számrendszerről”
   - 4–6 év: „Hackeljük meg az óvodát!”
   - 7–14 év: „Felhőtechnológia a menzán”
   - 15–18 év: „Big data a középiskolában”
6. Ha bonyolultabb, összetett feltételeket fogalmazunk meg, érdemes lehet zárójelek használatával egyértelműsíteni a szándékunkat. Az alábbiak közül melyik feltételmegfogalmazás biztosítja, hogy „tudjunk rajzolni”?
   a) Ha (van tollunk és van ceruzánk) vagy van egy papírlapunk: tudunk rajzolni valamit.
   b) Ha (van tollunk vagy van ceruzánk) és van egy papírlapunk: tudunk rajzolni valamit.
   c) Ha van tollunk vagy (van ceruzánk és van egy papírlapunk): tudunk rajzolni valamit.
7. Kihívást jelentő feladat: A kistesód szülinapi banános nyaflatyát csinálod. A kistesód szerint a jó banános nyaflaty jellemzője, hogy:
   a) nincs benne egyszerre vaníliás cukor és tortareszelék;
   b) ha van benne fahéj, akkor kell rá tejszínhab is;
   c) nincs benne fahéj, nem kerülhet rá tejszínhab sem.

   Írj programot a nyaflaty jóságának eldöntésére!

Ciklusok

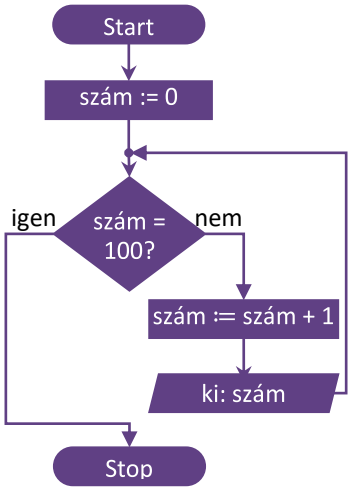
A ciklus eredetileg valamilyen egyforma időközönként, periodikusan ismétlődő dolgot jelent, gondolhatunk holdciklusra, választási ciklusra, árapályciklusra. Mi ebben a könyvben egy olyan programrészletet értünk rajta, amely valahányszor megismétlődik.

- Minden programozási nyelvben van *feltételes* ciklus, ami egy feltétel teljesülése esetén ismétli a programrészletet. A feltételt az ismétlődő rész elé írjuk (megcsinálja-e az ismétlést), ezt nevezzük *elöltesztelő* ciklusnak, vagy *while*-ciklusnak; angol neve: *while-loop*).
- Néhány nyelvben megoldható az is, hogy az ismétlődő rész végén legyen a feltétel (azt dönti el, hogy újra megcsinálja-e). Ez a *hátultesztelő* ciklus vagy *do-while*-ciklus, angolul *do-while-loop*.
- Egyes nyelvekben van *számlálós* ciklus, itt az ismétlések számát egy számlálóval adjuk meg. Például a Scratch-ben olyan számlálós ciklus van, amiben csak az ismétlések számát kell megadni. Általánosabb, hogy megadhatjuk a számláló kezdőértékét és az elérni kívánt végértéket, a program minden ismétlésnél növeli a számlálót, figyeli, hogy az elérte-e a végértéket. Napjainkra jellemző a programozási nyelvekben a számlálós ciklus továbbfejlesztett változata, amelyben a végérték helyett feltételt adhatunk meg, így sokkal általánosabban használható. A számlálás ebben az esetben nem helyettesíti a feltételek megadását, hanem azt kiegészíti. A számlálót léptető ciklus szokásos megnevezése *for*-ciklus (*for-loop)*, ami programozási nyelvekben jellemző angol *for* kulcsszóból ered.
- Néhány nyelvben a számlálós ciklus egyszerűsített változata, a *bejárós* ciklus – közhasználatú nevén *foreach* ciklus, angolul *foreach-loop* – is létezik, ami a gyakran előforduló, tipikus esetekben megkönnyíti a kódolást.

A C# mindegyik ciklusfajtát ismeri, a négyféle lehetőség a programozó kényelmét szolgálja. Van, aki mindig csak a while-ciklust használja, más mindig a for-ciklust. A lehetőségek ismeretén túl a választást a program könnyű értelmezhetősége és a szokások befolyásolják. Mind a négy változat használatára lesz példa, de a feladatok megoldása során – ha egyébként a program helyesen működik – nem hiba a mintán láthatótól eltérő megoldás használata.

A feltételes ciklus (while-ciklus)

Elsőként megvizsgáljuk az alábbi folyamatábrát. Vajon mit csinál a program?

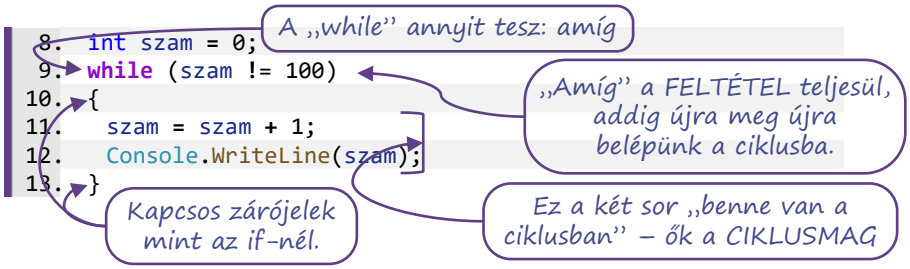


Mondatszerű leírással így néz ki az algoritmusunk:

```
szám := 0
ciklus amíg szám <> 100:
    szám := szám + 1
    ki: szám
ciklus vége
```

A <> azt jelenti, hogy „nem egyenlő!”
Itt írhatjuk így is: !=

C# nyelven pedig az alábbi formát ölti programunk:




```
 8.  int szam = 0;
 9.  while (szam != 100)
10.  {
11.    szam = szam + 1;
12.    Console.WriteLine(szam);
13.  }
```

1. Mi történik, ha a ciklusmag első sorát elhagyjuk? (Ha végtelen ciklusba kerül programunk, akkor a konzolablakot zárjuk be – bezáró gomb vagy CTRL+BREAK vagy CTRL+C – vagy az IDE-ben a Start gombtól jobbra található piros Stop gombbal – SHIFT+F5 – állítsuk le a futtatást.
2. A `!=` helyett milyen feltétellel érhetjük el ugyanezt az eredményt?
3. Hogyan változik a program kimenete, ha a ciklusmag két sorát felcseréljük? Ha a felcserélt sorokkal is szeretnénk a számokat 1-től 100-ig kiírni, mit kell még módosítani a programon?
4. Hogyan íratható ki minden második szám 100-ig? Hogyan íratható ki minden harmadik szám 100-ig?

Használjuk a while snippetet: A „while”-t két TAB a feltétel körüli kerek és a ciklusmag helyének kapcsos zárójeleivel egészíti ki.

Következő órára leírod százszor, hogy … (for-ciklus)

Bizonyára sokaknak viccekből, régi történetekből ismerős az alcímben jelzett tanári büntetés. Például le kellett írni százszor, hogy „A folyosón sétálunk.”. A fenti program egyetlen sorának módosításával megoldható ez a feladat, mi most mégis több módosítást végzünk, hogy „beszédesebb” legyen a kódunk.

- Azt, hogy hányadik kiírásnál tartunk, számláljuk, ezért a változót nevezzük „szamlalo”-nak.
- Mivel először az elsőt írjuk ki, a számláló kezdőértéke legyen 1 és addig írjuk, amíg ez az érték kisebb vagy egyenlő mint 100. Mivel azzal az értékkel kezdjük, ahányadik kiírás következik, ezért előbb kiírjuk a szöveget és utána módosítjuk a számlálót.
- A számlálót 1-gyel növeljük (nem azt mondjuk, hogy legyen önmagánál eggyel nagyobb). A „növelés”-t kódban is ki tudjuk fejezni: `szamlalo +=1`.

  Hasonló jelölést a többi alapművelettel is tudunk végezni:
  - `x -= 3` jelentése: x-et 3-mal csökkentjük;
  - `y *= 2` jelentése: y-t megduplázzuk;
  - `z /= 4` esetén z-t negyedeljük.

```
 8.  int szamlalo = 1;
 9.  while (szamlalo <= 100)
10.  {
11.    Console.WriteLine("{0}. A folyosón sétálunk!", szamlalo);
12.    szamlalo += 1;
13.  }
```

Kódunk talán jobban érthető, de a szöveg százszor leírása kézzel semmiség, ahhoz képest, hogy egy programozónak hányszor kell számlálót használnia és egyenként növelnie az értékét,

azaz számlálnia. Ezért a programozók nem szamlalo-nak hívják a számlálót, hanem az angol *iterator* szóból rövidítve „i”-vel szokták jelölni. Ha az i éppen foglalt, akkor „j” a szokásos jelölés vagy az i-t kiegészítik 1-2 betűvel, például it, ix, ai. Ez a szokás körülbelül annyira általános, mint amennyire a matematikában a változót x-szel – ha már foglalt, akkor y-nal, z-vel … – jelölik. Apró különbség, hogy míg az x általános használatának okát inkább csak találgatjuk, az iterator szó magyarul ismételgetést jelent (ez is i-vel kezdődik), de használhatjuk a magyarított megnevezést is: iterátor.

De még ez sem elég egy programozónak … A programozási nyelvek jelentős részében az egyesével számlálásra külön jelölést használnak:

- `i++`; jelentése: „i értéke ezután eggyel több”, ezt használják leggyakrabban C#-ban;
- `++i`; jelentése: „i értéke eggyel nő”, az előzőtől alig különbözik, de nehezebb beírni;
- `i--`; jelentése: i értéke ezután eggyel kevesebb” – visszafelé számláláshoz;
- `--i`; jelentése: „i értéke eggyel csökken”, szintén visszaszámláláshoz.

Ezzel a két rövidítéssel így néz ki a kódunk:

```
 8.  int i = 1;
 9.  while (i <= 100)
10.  {
11.    Console.WriteLine("{0}. A folyosón sétálunk!", i);
12.    i++;
13.  }
```

Hatsoros kódunknak egyetlen hosszabb sora van, a kiírás. Nem lehetne kevesebb sorral megúszni? De, lehet. Ahogy a fejezet elején már láthattuk, van más ciklus is: a számlálós … és mi most itt számlálóval számoljuk, hogy hányadik kiírásnál tartunk. A C#-ban – és több más nyelven – a for-ciklussal ugyanezt a programot sokkal kevesebb sorban, a lényeges dolgokat összefogva írhatjuk meg:

```
 8.  for (int i = 1; i <= 100; i++)
 9.    Console.WriteLine("{0}. A folyosón sétálunk!", i);
```

Első ránézésre brutális, hogy harmadára csökkent a sorok száma, de azért ez egy kicsit csalóka.

A **for** kerekzárójelei között – neve: *ciklusfej* – nemcsak a feltétel szerepel: *három paramétert adhatunk meg pontosvesszőkkel elválasztva*. A feltétel elé írjuk be a korábbi kódban a **while** előtt lévő sort, a feltétel után szerepel a számláló növelése, ami a *ciklusmag utolsó utasítása* volt. A növelés (változtatás) bármelyik megvalósítását beírhatjuk, lehet `i = i + 3` is, csak arra kell figyelni, hogy amit ide írunk, az a ciklusmagba írt utolsó utasítás után lesz végrehajtva.

A kapcsos zárójelek eltűnése újabb két sorral csökkentette a kódunkat, azonban a kapcsos zárójeleket itt is úgy kell használni, mint a **while** vagy az **if** esetén: elhagyható, ha csak egy utasítást kell végrehajtani. Másképp: a ciklusmag itt is egy parancssor vagy kapcsos zárójellel összefogott parancssorok.

A for snippet helyettünk kódol:	A „for”+ két TAB első hatása for (int i = 0; i < length; i++) és kiírt { }. Átnevezhető az i, TAB után átírható a length.

Ha egy while-ciklus előtt egy változónak kezdőértéket adunk, majd ezt az értéket a feltételes ciklus magjának utolsó utasításaként módosítjuk, akkor a kód egyszerű áthelyezésekkel írható

át számlálós ciklussá. A kód szerkezetének szabályosságát kihasználva a for snippettel megspórolhatjuk a gépelés jelentős részét. A kód valószínűleg kevesebb elgépelést fog tartalmazni, de a változók pontos használata, a feltétel pontos megfogalmazása továbbra is a programozó feladata marad.

A számlálós ciklus különböző nyelvekben

A C#-ban a for-ciklus a számlálós ciklus továbbfejlesztett változata. Gyakorlatilag a for-ciklus nem más, mint while-ciklus összevont megfelelője. C#-ban a két nyelvi eszköz használata között csak egy különbség van:

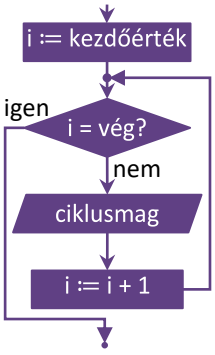
- Ha a számláló létrehozása nem a **while** előtt történik, hanem a **for** első paraméterében, azzal a „láthatóságuk” is különböző lesz.

A gyakorlatban, ha a több while-ciklust írunk egymás után, akkor a számláláshoz elegendő egy számlálót létrehozni, a következő használatkor csak a kezdőértéket állítjuk át és a két ciklus között látható, hogy éppen mennyi a számláló értéke. Ha a számlálót a for-ciklus első paraméterében hozzuk létre, akkor ez a számláló csak a cikluson belül lesz használható. Ilyenkor minden egymásutáni for-ciklushoz használhatjuk ugyanazt a nevet a számláláshoz, de ez nem ugyanaz a változó lesz, mert az egyik megszűnik, mire a következő létrejön. Ha a cikluson kívül is szükségünk van a számláló értékére, akkor a változót a ciklus előtt kell létrehozni – *deklarál*ni – és a for első paraméterében csak kezdőértéket adunk neki, azaz *inicializál*juk.

```
 8.  int i;
 9.  for (i = 1; i <= 100; i++)
10.  {
11.    Console.WriteLine("{0}. A folyosón sétálunk!", i);
12.  }
```

Így a feladat első megoldásával teljesen azonos tulajdonságú a kódunk, de csak egy sorral rövidebb. Mondhatnánk, hogy nem a sorok száma, hanem a beírt karakterek száma számít. A „for” rövidebb a „while”-nál. Nézzük meg alaposan, mi hova került: hány karaktert kell begépelni a while-ciklusba és mennyit a for-ciklusba? Az eredmény: a for-ciklusban 1 db tabulátorral van kevesebb. Látható, hogy ezért nem érdemes összevont alakot használni. A for-ciklus népszerűségének (van, aki mindig ezt használja) más oka van.

A „következő órára írd le százszor …” feladatban nemcsak az a nehéz, hogy sokat kell körmölni, hanem az is, hogy fejben tartsuk, éppen hányadiknál tartunk. A szavazatok számlálását is többször ellenőrzik, mert könnyű elrontani. Szerencsére a számítógép sokkal jobban bírja a monotonitást, nem szokta elfelejteni, hogy éppen hol tart, nem ugrik a negyven után a hetvenegyre (kivéve, ha erre utasítjuk). De a programozókról ugyanez nem mondható el. A while-ciklus kódolásakor a leggyakoribb hiba, hogy mire a ciklusmag végére ér a programozó, addigra elfelejtheti, hogy még a ciklusszámlálót is növelni kellene. Az eredmény: végtelen ciklus. Észrevenni is nehéz ezt a hibát, mert a kód a programozó fejében megszületett, csak a kivitelezés maradt el. Ezért a for-ciklus legnagyobb előnye az, hogy a számlálást ott írhatjuk le, ahol gondolunk rá. Valójában egy olyan feltételes ciklus, amely egyes helyzetekben jobban kifejezi a gondolatunkat.



Folyamatábrán a számlálós ciklusnak nincs egyedi formája. Elöltesztelő ciklusként lehet megjeleníteni, ami egyúttal jól mutatja, hogy miből származik.

A számlálós ciklus – több programozási nyelvben – speciálisabb. Azért érdemes megismerkedni ezekkel is, mert mondatszerű leírásban és blokknyelvekben használhatjuk és sok esetben egyszerűbbé teszi az algoritmus leírását.

Számláló változó nélkül:

```
ismételd 100-szor:
    ki: „…séta”
ciklus vége
```

Számláló változóval:

```
ciklus szám = 1-től 100-ig:
    ki: szám
ciklus vége
```

Számlálóval és lépésközzel:

```
ciklus szám = 1-től 100-ig lépésköz 1:
    ki: szám
ciklus vége
```

Túltenni Gausson

A kis Gauss, amikor még nem volt nagy matematikus, az anekdota szerint egész osztályával együtt azt a feladatot kapta a tanárától, hogy adja össze a számokat egytől százig. A tanár közben nekiállt valami más munkának, de a kis Gauss két perc múlva szólt, hogy készen van, és az eredmény 5050. Rájött, hogy 1 + 100 = 101, 2 + 99 = 101 és így tovább. 50-szer 101 pedig 5050. Ha már van számítógépünk, illendő a két percnél jobb eredményt hoznunk, méghozzá Gauss felismerésének kihasználása nélkül.

A megoldás ötlete, hogy a számlálás mellett szükségünk van még egy változóra, amiben az összeget folyamatosan – mindig az újabb értékeket hozzáadva az éppen aktuális értékhez – számítjuk ki a végeredményt. A kivitelezést tekintve, az eddig megismert nyelvi elemekkel, azonos változóneveket használva, nem vizsgálva, a tördelést és a kód igazítását, száznál többféle megoldás létezik. Itt ezek közül kettő látható. Próbáljunk ki több változatot, keressük meg azt a kódot, amely számunkra a legkifejezőbb!

```
 8.  int szamlalo = 0;
 9.  int osszeg = 0;
10.  while (szamlalo < 100)
11.  {
12.    szamlalo = szamlalo + 1;
13.    osszeg = osszeg + szamlalo;
14.  }
15.  Console.WriteLine("Összesen: " + osszeg);
```

```
 8.  int osszeg = 0;
 9.  for (int i = 1; i <= 100; i++)
10.    osszeg += i;
11.  Console.WriteLine("Összesen: {0}", osszeg);
```

A logikai típus

Vegyük elő a számkitalálós programunkat és csupaszítsuk le annyira, hogy csak a jó válaszra reagáljon! Tervezzük át úgy, hogy kérdezzen addig, amíg ki nem találjuk a számot! A mondatszerű leírásba nem írjuk bele az véletlengenerátort létrehozó utasítást:

```
gondolt_szám := 1 és 6 közötti véletlenszám
kitalálta = hamis
ciklus amíg ki nem találta:
    be: tipp
    ha tipp = gondolt_szám:
       kitalálta = igaz
ciklus vége
```

A mondatszerű leírásban bevezettünk egy változót, amiben igaz vagy hamis érték tárolható. A változó kezdeti értéke hamis, és akkor állítjuk át igazra, ha a tipp jó volt. A változót azért inicializáltuk hamis értékkel, hogy a ciklusba legalább egyszer belépjünk.

Azok a változók, amelyek igaz (`true`) és hamis (`false`) értéket vehetnek fel, úgynevezett logikai típusú változók (a C#-ban a típus neve `bool`, George Boole matematikus után, aki efféle problémákkal való foglalatosságáról vált híressé). Mindez kódként így néz ki:

```
 8.  Random rnd = new Random();
 9.  int gondolt_szam = rnd.Next(1,7);
10.  bool kitalalta = false;
11.  while (!kitalalta)
12.  {
13.    Console.Write("Szerinted? ");
14.    int tipp = int.Parse(Console.ReadLine());
15.    if (tipp == gondolt_szam)
16.      kitalalta = true;
17.  }
18.  Console.WriteLine("Végre!");
```

false és true kis kezdőbetűvel

Vagy: while(kitalalta == false)
*A ! olvasata **NEM**.*
Ez a szokásos írásmód.

Beljebb, mert csak az if teljesülése esetén érvényes.

Kódunkban érdemes megfigyelni a változók élettartamát, más néven hatókörét. A `tipp` változót a ciklusmagban *deklarál*tuk (hoztuk létre) és *inicializál*tuk (megadtuk, hogy mennyi legyen az értéke). Ez azt jelenti, hogy minden ciklusléspésben újra és újra létrehozzuk a `tipp` változót. Másrészt, a változóknak egyedinek kell lennie, azaz nem lehet két `tipp` nevű változónk. A kód mégis helyes, mert ami a kapcsos zárójelek között keletkezik, az a záró kapcsos zárójelnél megszűnik. A zárójelek nemcsak egybefogják az utasításokat, de programblokkot is képeznek. A belső – lokális – változók a blokk végéig léteznek. Emiatt megfontolandó, hogy hol deklarálunk egy változót, hol adjuk meg a típusát és a nevét. Általában ott célszerű megadni, ahol először használni szeretnénk, de ha – ez egy blokkon belül történne és – a blokk végrehajtása után is használnánk, akkor a deklarálásnak a blokk előtt kell megtörténnie, az értékét a blokkban csak felülírjuk. Ez a teendő akkor, ha a programunkban nemcsak azt írjuk ki, a végén, hogy „Végre!”, hanem a legutolsó beírt értéket is.

```
 8.  Random rnd = new Random();
 9.  int gondolt_szam = rnd.Next(1,7);
10.  bool kitalalta = false;
11.  int tipp;
12.  while (!kitalalta)
13.  {
14.    Console.Write("Szerinted? ");
15.    tipp = int.Parse(Console.ReadLine());
16.    if (tipp == gondolt_szam)
17.      kitalalta = true;
18.  }
19.  Console.WriteLine("Ez az: {0}!", tipp);
```

Előre deklarált tipp
Itt nem kell inicializálni, de illene:
int tipp = 0;

Nem deklarálhatjuk újra:
nem írhatjuk ki, hogy int

Összetett ciklusfeltétel

A program türelmes, így a felhasználó a végtelenségig próbálkozhat. Írjuk át a programunkat úgy, hogy csak három próbálkozást engedjen meg! Számolnunk kell a próbálkozásokat, de nem elég, ha a ciklus feltételeként csak azt adjuk meg, hogy még nem használtuk el mindegyiket. Arra is figyelnünk kell, ha közben a felhasználó kitalálta a gondolt számot. Figyelni kell a próbálkozások számát **és** a tippelt értéket is. Szerencsére a `while`, pont ugyanúgy, mint az `if`, képes összetett feltételek kezelésére.

1. Gondoskodjunk az elhasznált lehetőségek nyilvántartásáról (számlálásáról)?
2. Fogalmazzuk meg a `while` feltételét!

A teljes kód a következő:

```
 8.  Random rnd = new Random();
 9.  int gondolt_szam = rnd.Next(1,7);
10.  bool kitalalta = false;
11.  int szamlalo = 0;
12.  while (!kitalalta && szamlalo < 3)
13.  {
14.    Console.Write("Szerinted? ");
15.    int tipp = int.Parse(Console.ReadLine());
16.    if (tipp == gondolt_szam)
17.      kitalalta = true;
18.    szamlalo++;
19.  }
20.  Console.WriteLine("Vége");
```

A `szamlalo` változó utal arra, hogy talán számlálós ciklussal is megoldhatjuk a feladatot. Igazi, hagyományos értelemben vett számlálós ciklusban nem tudunk feltételt megadni, de a C# – és nagyon sok más nyelv is – a for-ciklus középső részében tetszőleges logikai kifejezést megenged, így rövidíthetjük a kódot:

```
 8.  Random rnd = new Random();
 9.  int gondolt_szam = rnd.Next(1,7);
10.  bool kitalalta = false;
11.  for (int i = 0; !kitalalta && i < 3; i++)
12.  {
13.    Console.Write("Szerinted? ");
14.    int tipp = int.Parse(Console.ReadLine());
15.    if (tipp == gondolt_szam)
16.      kitalalta = true;
17.  }
18.  Console.WriteLine("Vége.");
```

3. Helyezzünk el ismét olyan programsorokat a kódban, amelyek a felhasználó dicséretéért, ugratásáért felelnek! Több helyre is elhelyezhetőek, mindnek megvannak az előnyei és hátrányai. Hasonlítsunk össze néhány lehetséges megoldást!
4. Szeretnénk megoldani, ha a felhasználó a harmadik lehetőségre már majdnem kitalálta a megoldást (csak egyet tévedett), akkor kapjon még egy esélyt. Ahogy programozáskor mindig, ismét több helyes megoldás lehetséges – valósítsuk meg a nekünk legjobban tetszőt!

Ciklusok és véletlenek

Feladatok

1. Ciklussal meg tudunk oldani egyenleteket az egész számok halmazán – próbálgatással.
   a) Oldjuk meg a $3x + 2 = 59$ egyenletet a pozitív egészek halmazán!

```
 8.  int x = 1;
 9.  while (3*x + 2 != 59)
10.    x = x + 1;
11.  Console.WriteLine("A megoldás: {0}", x);
```

Érdekesség, hogy for-ciklussal is megoldható a feladat, de a ciklus magnélküli lesz és a sorok száma sem lesz kevesebb, mert az `x`-et előre létre kell hozni, hogy a kiíráskor is létezzen. Ilyenkor a `for` ciklusfejének első paramétereként az `x`-nek csak kezdőértéket adunk, nincs `int`.

```
 8.  int x; //Itt deklaráljuk, hogy a 11. sorban is létezzen
 9.  for (x = 1; 3*x + 2 != 59; x++)
10.  ; //Jelezzük, hogy nincs mag. Másik jelölés: {}
11.  Console.WriteLine("A megoldás: {0}", x);
```

   b) Oldjuk meg a $6x^2 + 3x + 8 = 767$ egyenletet az egész számok halmazán! Honnan érdemes indítani a próbálgatást?
   c) Diophantosz ókori görög matematikus verses sírfelirata több fordításban fellelhető az interneten. A felirat alapján fogalmazzuk meg az egyenletet, és írjunk programot, ami megoldja! Hány évesen halt meg Diophantosz?
   d) Ilyen módszerrel csak akkor oldható meg biztosan az egyenlet, ha az eredmény egész szám. Miért?
   e) Ha az egyenletnek nincs egész gyöke (például $6x^2 + 3x + 8 = 768$), akkor végtelen ciklusba kerül a programunk. Miként biztosítható, hogy ne lépjen végtelen ciklusba a program, és ha már esélytelen, hogy talál megoldást, ne próbálkozzon tovább? Gondolkozzunk összetett feltételben!
2. Írjunk pénzfeldobás-szimulátort!
   a) A gép írja ki, hogy fejet vagy írást „dobott”! A Random gépünk `NextDouble()` függvényével hamis érmét is szimulálhatunk, ha nem pontosan felezzük a [0; 1[tartományt.

```
 8.  Random dob = new Random();
 9.  int dobas = dob.Next(2);
10.  if (dobas == 1)
11.    Console.WriteLine("fej");
12.  else
13.    Console.WriteLine("írás");
```

```
 8.  Random dob = new Random();
 9.  double dobas = dob.NextDouble();
10.  if (dobas < 0.5)
11.    Console.WriteLine("fej");
12.  else
13.    Console.WriteLine("írás");
```

   b) Készítsünk statisztikát! Egymillió feldobásból mennyi lesz fej és mennyi írás?
   c) Írjuk át a programot úgy, hogy kockadobásokat számoljon (mennyi lesz 1-es… 6-os)!

d) Készítsünk cinkelt kockát! A hatos jöjjön ki kétszer akkora eséllyel, mint a többi szám!

3. Írjunk randiszimulátort!
   a) A számítógép kérdezze azt, hogy „Szeretsz?”, amíg azt nem válaszoljuk, hogy „Nagyon!”, vagy azt, hogy „Jobban, mint a kókuszgolyót!”.
   b) A számítógép legyen durcás: legfeljebb három kérdés után zavarjon el bennünket, ha nem adjuk meg a „helyes” válaszok valamelyikét!
   c) A számítógép legyen szeszélyes: zavarjon el bennünket 2–4 „rossz” válasz után véletlenszerűen!

Ciklusok oda-vissza, illetve egymásba ágyazva

Feladatok

1. Írjuk ki a számokat csökkenő sorrendben 100 és –100 között egymás alá! Oldjuk meg úgy is a feladatot, hogy a számokat egymás mellé, szóközzel elválasztva írjuk ki!
2. Írjuk ki (két) ciklussal, hogy „12345678987654321”!
   a) Hány helyen kell módosítanunk a programot, hogy ne kilencig, hanem kilencmillió-kilencszázkilencvenkilencezer-kilencszázkilencvenkilencig írja a számokat?
   b) Hogyan változik a program futásának sebessége, ha nem íratjuk ki a számokat, csak elszámoltatunk oda-vissza? (// jellel tegyük kommentté a kiírást végző sorokat.)
3. Írjunk egymás mellé 10 csillagot („*”) úgy, hogy a programkódban csak egyetlen csillag karakter legyen!
4. Csillagok több sorban
   a) Írjunk egymás alá öt csillagsort! Ötlet: tegyük az előző feladat ciklusát egy másik ciklus belsejébe. Az áttekinthetőség érdekében, ott is írjuk ki a kapcsos zárójelet, ahova nem feltétlenül szükséges.

while-ciklusokkal

```
 8.  int sorsz = 0;
 9.  while (sorsz < 5)
10.  {
11.    int csillsz = 0;
12.    while (csillsz < 10)
13.    {
14.      Console.Write('*');
15.      csillsz++;
16.    }
17.    Console.WriteLine();
18.    sorsz++;
19.  }
```

for-ciklusokkal

```
 8.  for (int si = 0; si < 5); si++)
 9.  {
10.    for (int csi = 0; csi < 10); csi++)
11.    {
12.      Console.Write('*');
13.    }
14.    Console.WriteLine();
15.  }
```

Figyeljük meg mindkét megoldásban:

- A két ciklusnak két külön számlálója van.
- Minden sor elején a csillagok számlálását újrakezdjük.

   b) Helyezzünk el breakpoint-ot (piros pontot) a csillagot kiíró sor elé! Így Debug módban futtatva ezen a ponton megáll a programunk, amit a Continue (volt Start) gombra kattintva (vagy F5) folytathatunk. Figyeljük meg a Watch ablakban, illetve egérrel az adatra mutatással, az egyes változók értékét: hogyan változnak két megállítás között!
   c) Az előző program megoldásának egyetlen helyen való megváltoztatásával alakítsuk háromszöggé a program kimenetét! (Ötlet: minden sorba annyi csillagot kell kiírni, ahányadik…)

d) Ha elkészültünk, írjuk át úgy a programot, hogy minden sorban csak az utolsó csillag jelenjen meg!
e) …minden sorban az első és az utolsó csillag jelenjen meg!

5. Írjunk szorzótáblát a kicsiknek!

Minta kimenet a 4.b feladathoz:

```
*
**
***
****
*****
```

Minta kimenet az 5. feladathoz:

```
1 * 1 = 1
1 * 2 = 2
…
6 * 6 = 36
6 * 7 = 42
…
10 * 9 = 90
10 * 10 = 100
```

Összetartozó adatok kezelése

Adatok sorozata

A programjainkban az adatokat változókban tároljuk, és eddig öt változótípust, adattípust ismerünk:

- szöveg típust (`string`);
- az egész szám típusokat (integer, például `int`);
- a valós, tizedestörtként megjelenő számokat (lebegőpontos szám, például `double`)
- karaktereket (`char`)
- és a logikai értékeket (`bool`).

Ezek közül egyik sem jó akkor, amikor több egymáshoz tartozó adatot szeretnénk tárolni, kiírni, műveletet végezni velük. Például tárolni szeretnénk a csoportunkba járók neveit, hogy ki tudjuk sorsolni, ki lesz a következő felelő. Vagy, jó lenne tárolni minden osztály létszámát, hogy meg tudjuk mondani azt is, hogy összesen hány diák jár az iskolába és azt is, hogy melyik osztályokba járnak az átlagosnál többen. A szám-kitalálós programban a végén jó lenne kiírni a tippeket. Esetleg írhatnánk egy szólánc programot, de a gépnek meg kellene jegyeznie a már korábban beírt szavakat.

Az ötletekre minden programozási nyelvben van megoldás, nem csak a szöveg (karaktersorozat) tartalmazhat több adatot. Bármilyen adatokból képezhetünk összetett adattípusokat. A lehetőségek közül most két olyan összetett adattípust ismerünk meg, amelyekben tetszőleges, de azonos típusú adatok sorozatát tárolhatjuk.

Bár a két adattípusnak eltérőek az előnyei és hátrányai, a programozással ismerkedőknek elég az egyiket ismerni. Az, hogy melyik jobb, melyik könnyebb, ízléstől és a konkrét feladattól függ. Az egyiket alaposan meg kell ismerni, meg kell érteni, hogy hogyan tudjuk használni.

A tömb és használata

Több egyforma dolog együttesének sokféle elnevezése van: (szám)sorozat, (karakter)lánc, (fekete) lista és ide tartozik az egydimenziós tömb is. Több egyforma dolog együttesét elképzelhetjük táblázatban, mátrixban, kétdimenziós tömbben. A tömb – akárhány dimenziós – az

adatokat elrendezve tartalmazza. A programunkban létrehozandó egydimenziós tömbhöz a memóriában szomszédos memóriaterületeket használunk. Ezt elképzelhetjük úgy, hogy egymás mellé vagy úgy is, hogy egymás alá/fölé tesszük az egyforma adatokat. Ha megadjuk, hogy milyen és mennyi adatból szeretnénk tömböt létrehozni, akkor ennek megfelelő méretű helyet keres a programunk, és – mintha rekeszes szekrény lenne – kijelöli az egyes adatok helyét. Mivel minden adatunknak előkészítjük a helyét, ezután bármelyiknek, akár tetszőleges sorrendben is, megadhatjuk az értékét, később tudjuk módosítani. Problémát csak az jelenthet, ha több adatunk van, mint amennyi helyet kezdetben létrehoztunk. Ilyenkor ugyanis az összes adatot át kell tenni egy új (nagyobb) hely megfelelő rekeszeibe. Toldozni, foldozni, bővíteni a már lefoglalt területet nem lehet, ha a lefoglalt részen túl szeretnénk valamit tárolni, akkor – programozási nyelvtől függően – hibajelzéssel elszáll a programunk vagy esélyes egy kékhalál, amikor az adattal egy másik program adatát írja felül a program.

Az egydimenziós tömb a C# nyelv legegyszerűbb összetett adattípusa. Ezért a létrehozása és használata nem túl bonyolult. Az egyetlen nehézséget az okozhatja, hogy mindig a szögletes zárójelet kell használni, aminek így – a környezetétől függően – háromféle szerepe van:

- amikor megadjuk, hogy milyen adattípusokról van szó, akkor a [] jelöli, hogy több adat lesz a megnevezett típusból, hogy tömb lesz.
- amikor létrehozzuk és memóriaterületet foglalunk neki, akkor általában egy speciális konstruktort használunk, [] jel között adjuk meg, hogy hány darab adatra kell a foglalás.
- használjuk, akkor a [] közötti szám azt mutatja meg, hogy a lefoglalt terület kezdetétől hány adatnyira van az éppen kiválasztott hely, azaz – 0-tól számozva – hányadik adatot használjuk.

Feladatok

1. Tároljuk kacatjainkat tömbben! Készüljünk fel arra, hogy akár 1000 db kacatunk is lehet, de csak az első három helyre írjunk be adatot!

```
 8.  string[] kacat = new string[1000];
 9.  kacat[0] = "csat";
10.  kacat[1] = "gombolyag";
11.  kacat[2] = "vonatjegy";
```

string-ek tömbje

Hely 1000 db stringnek

A 2-es indexű 3. adat, egy string

2. Tároljuk el a hét napjainak rövidített neveit egy tömbben! Most tudjuk, hogy 7 db stringet kell tárolnunk és pontosan ismerjük a megnevezéseket is. Ilyenkor – a létrehozás és egyenként értékadás helyett, – kapcsos zárójelekkel „egyesítve” megadhatjuk a szavak listáját:

```
 8.  string[] napnev = {"H", "K", "Sze", "Cs", "P", "Szo", "V"};
 9.  Console.Write(napnev[6]); // Ki: V
```

7 string lesz benne.

3. Tároljuk el az egyjegyű prímszámokat, majd írjuk ki őket szóközökkel elválasztva!

```
 8.  int[] primek1 = {2, 3, 5, 7};
 9.  for (int i = 0; i < primek1.Length; i++)
10.    Console.Write("{0} ",primek1[i]);
```

A tömb felismerte a méretét.

A ciklusszámlálót használhatjuk indexnek.

Gyakran elő fog fordulni, hogy adatokat egy sorban, szóközzel elválasztva kap a programunk. A szóközön kívül gyakori elválasztójel a vessző, a pontosvessző és a tabulátor. Ha a beolvasott egyetlen stringből az egyes adatokra külön-külön lenne szükségünk, fel kellene darabolni. Erre a `string` `Split`() függvénye használható. Nagy előnye a függvénynek, hogy az eredményének eltárolásához nem nekünk kell előzetesen lefoglalni a megfelelő méretű memóriát, mert ezt a függvény megoldja.

Adatok beírásakor különböző típusú adatok is kerülhetnek egy sorba, ezért nincs egyszerű mód arra, hogy a beírt adat részleteiből rögtön a megfelelő típusú adatok a megfelelő változókba kerüljenek. Ha azonos típusú – de nem string – adatokat tartalmaz egy adatsor, akkor a feldarabolás után, egy for-ciklust (vagy while-ciklust) bevetve a string tömb minden részletét át tudjuk konvertálni a másik típust tartalmazó tömbbe.

4. Írjunk programot, ami bekéri a heti lottószámokat egy sorban, szóközökkel elválasztva, majd ezt egész számokként eltárolja! (A feladat folytatása lehet, hogy bekéri a megjátszott számainkat is és megmondja, hány találatot értünk el. De most még csak beolvassuk és eltároljuk a számokat.) Úgy írjuk meg a programot, hogy az ötös és a hatos lottóhoz is jó legyen!

```
 8.  Console.Write("Mik a heti lottószámok? ");
 9.  string sor = Console.ReadLine();
10.  string[] adatok = sor.Split(' ');
11.  int[] lotto = new int[adatok.Length];
12.  for (int i = 0; i < adatok.Length; i++)
13.    lotto[i] = int.Parse(adatok[i]);
```

szóköz karakternél darabol: aposztrófok között szóköz

annyi szám, ahány szövegdarab

Láthattuk, hogy a tömb elemeit egyenként tudjuk használni, egyenként tudjuk kiírni. Ha a tömb minden elemének értéket adtunk, akkor a tömb mérete – a `Length` tulajdonsága – segítségével, egy for-ciklussal könnyen kiírhatók a tömbelemek. Figyelni kell a tömb elemeinek használatára – és kiíráskor is –, ha a tömb nagyobb méretű, mint ahány adatot éppen tárolunk benne. Ezért, ha a tömböt „üresen” hozzuk létre, akkor célszerű a tényleg felhasznált elemek számának tárolására egy változót mellé tenni és folyamatosan jegyezni, hogy éppen meddig tartalmaz a tömb értelmes adatot.

Ha az adatokat össze-vissza írjuk be, akkor célszerű a használat előtt az összes tömbelemnek kezdőértéket adni. Programozási nyelvenként eltérő, hogy melyik adattípusnak van automatikusan értelmes kezdőértéke, illetve melyik kapja értékül az adott memóriahelyen talált szemetet. Sokkal jobb, ha gondoskodunk a megfelelő kezdőértékekről.

5. Alakítsuk át e két szempontnak megfelelően a kacatokat tároló programunkat! Írjuk ki a tárolt kacatok listáját egymás alatti sorokba!

```
 8.  string[] kacat = new string[1000];
 9.  int kacatdb = 0;
10.  for (int i = 0; i < kacat.Length; i++)
11.    kacat[i] = "-"; //Mindegyik "-". Lehet " " vagy "" (üres) is.
12.  kacat[0] = "csat"; //Átnevezzük "-"-ről "csatt"-ra
13.  kacatdb++; //1 lett
14.  kacat[kacatdb] = "gombolyag";
15.  kacatdb++; //2 lett
16.  kacat[kacatdb] = "vonatjegy";
17.  kacatdb++; //3 lett
18.  for (int i = 0; i < kacatdb; i++)
19.    Console.WriteLine(kacat[i]);
```

6. Tegyünk a kód 8., 9., és 12–17. sora elé breakpoint-ot, futtassuk a programot Debug módban! Figyeljük a Watch ablakban és egérrel rámutatással is a változók és a tömb értékeinek módosulását!

Kiegészítés: 2D

Eddig egydimenziós tömbökről volt szó, de – főleg a legkülönbözőbb helyzetekben kapott – táblázatok kapcsán, a tömböt is „legalább” kétdimenziósnak képzeljük el. C#-ban az egydimenzióhoz képest az adatok két- vagy többdimenziós tárolása nem bonyolult. Persze a használatához – mondjuk a négydimenziós térben tájékozódáshoz – már komoly absztrakciós képesség szükséges.

A kétdimenziós tömb létrehozása és használata az egydimenzióshoz képest annyiban tér el, hogy minden [] között van egy vessző is és ahol 1 szám volt, ott kétdimenzió esetén 2 számot kell megadni.

```
int[,] jegyek = new int[13,10];
jegyek[7,5] = 5;
```

A háttérben a programozási nyelvek a többdimenziós tömböket is egydimenzióként tárolják, méghozzá sorfolytonosan, azaz a példában a `jegyek`[0,9] elem után a `jegyek`[1,0] következik. A gyorsabb memóriaműveletek érdekében célszerű az adatok írásánál, sorban olvasásánál ezt a sorrendet betartani.

Másik lehetőség a tömbök tömbje, ezzel az is megoldható, hogy a tömb sorainak elemszáma különböző legyen. Részletekről a C# dokumentációjában vagy internetes fórumokon tájékozódhatunk.

A lista és használata

Amikor több, valamilyen szempontból egyforma dolgot szeretnénk felsorolni, azt mondjuk, listát készítünk. Ilyen például a bevásárló lista, a hiányzók listája, a „todo list”. Ezeknek a felsorolásoknak a közös jellemzője, hogy nem tudjuk előre, hogy a végén hány elemet teszünk bele. Ez a mindennapok során nem szokott komoly problémát jelenteni: ha betelt a bevásárlólistánk cetlije, akkor: körbe, a szélén még elfér néhány tétel vagy megfordítjuk és a hátoldalán folytatjuk vagy keresünk egy másik cetlit. Amikor egy programozási nyelven szeretnénk listát készíteni, akkor a fordítóprogramnak hasonló problémákkal kell megküzdenie. Olyan programkódot kell készíteni, amelyik akárhány adatot el tud tárolni, eközben nem ír felül más célra lefoglalt területeket és számon tudja tartani, az egyes elemek sorrendjét. Az is előny, ha bármelyik elem gyorsan elérhető, azaz nem kell sorba vennie az elemeket ahhoz, hogy az utolsó előttit megtalálja. A feladat annyira bonyolult, hogy egyes programozási nyelveken

többféle adattípust is elkészítettek, mert a „gyorsan végrehajtható algoritmusok írhatók rá”, a „kevés memória is elég neki” és a „könnyen kódolható” hármasfeltételből szinte lehetetlen egyszerre mindet teljesíteni.

Most egy olyan adattípussal fogunk megismerkedni, ami

- egyforma adattípusokból többet tartalmaz,
- tetszőlegesen bővíthető,
- a benne lévő adatok gyorsan, könnyen elérhetők,
- (a többi hasonló típushoz képest) könnyen használható kódjaink írásához.

A második feltétel kivételével, egy fixen lefoglalt adatterület, ahol minden adatnak előre kijelölt helye van, éppen megfelelne. Ez lenne a tömb adattípus. De most többet akarunk … A tetszőleges bővíthetőséget lehet úgy biztosítani, hogy kezdetben lefoglalunk egy adott méretet (bevásárló lista egy cetlije); amikor ez betelik, akkor keresünk egy nagyobb helyet, ahova átvisszük a korábbi adatokat és ott már lehet további adatokat is írni. Ha ez is betelik, akkor újabb költözés …

Az itt leírt adattípus elméleti neve: *dinamikusan bővülő tömb*, C#-ban a `List` adattípus ilyen tulajdonságú. A leírásból következtethetünk az adattípus gyenge pontjaira is: minden átköltözéshez idő kell. A `List` kettőhatványonként bővíti a tárolási kapacitását. Utána számolhatunk: 8 adat beírása a memóriában körülbelül 15 adatírást jelent, a 9. adat beírása a már bennlévő 8 áthelyezése miatt 9 további adatírás, így ekkor már 24-szer írt adatot a programunk. Amikor nem számít a program futási ideje, akkor ez így is jó. De ha nagymennyiségű adatot szeretnénk bevinni, akkor célszerű a konstruktorban megadni a tervezett kapacitást.

Az adattípus figyelemre méltó tulajdonsága, hogy korlátlanul bővíthető futtatás közben a mérete. Vajon ez mindig jó?

A List<>

A `List<>` adattípus használata előtt, a programunk elején be kell vennünk a speciális adattípusokat tartalmazó **System.Collection.Generic** névteret is.

```
1.  using System;
2.  using System.Collection.Generic;
```

Feladatok

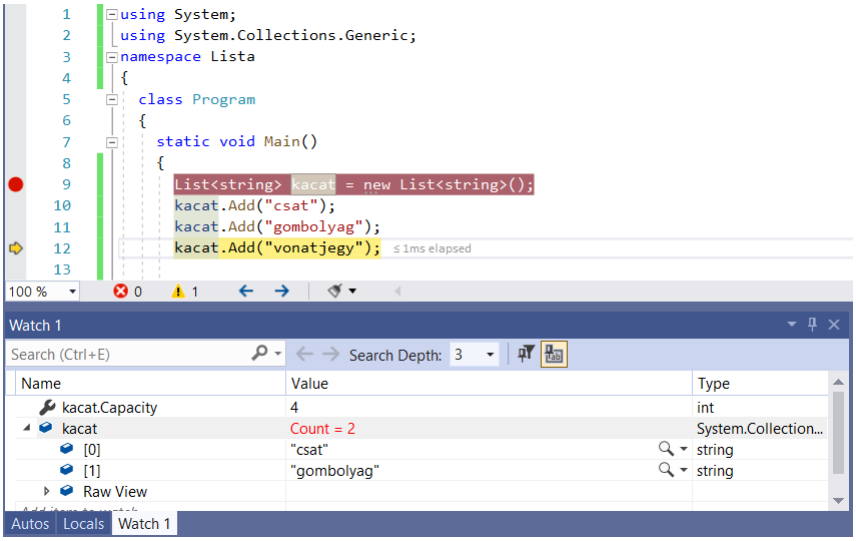
1. Tároljuk a kacatjainkat listában! Tegyünk bele három kacatot! Futtassuk lépésenként a programot, figyeljük meg a kapacitás és az adatok számának a változását!

```
 9.  List<string> kacat = new List<string>();
10.  kacat.Add("csat");
11.  kacat.Add("gombolyag");
12.  kacat.Add("vonatjegy");
```

string-ek listája

konstruktor, üres lista, mint a Random-nál

A lista végéhez hozzátesz egy adatot



2. Tároljuk el a hét napjainak rövidített neveit egy listában! Most tudjuk, hogy mit szeretnénk tárolni, ezért a lista létrehozása után közvetlenül, kapcsos zárójelekkel „egyesítve” megadhatjuk a szavak tömbjét:

```
 9.  List<string> napnev = new List<string>()
                           {"H", "K", "Sze", "Cs", "P", "Szo", "V"};
10.  Console.Write(napnev[6]); // Ki: V
```

így átláthatóbb, de nem kell új sor

3. Ha tudjuk, hogy hány eleme lesz a listánknak, akkor azt illik megadni. Ez a program számára a létrehozás pillanatában jelent információt, utána a méret igény szerint módosulhat. Próbáljunk ki többféle értéket, 7-nél kisebbet és nagyobbat is. A kiírás sorába tett breakpointtal állítsuk meg programunk futását, figyeljük meg a méret és kapacitás értékeket!

```
 9.  List<string> napnev = new List<string>(7)
                           {"H", "K", "Sze", "Cs", "P", "Szo", "V"};
10.  Console.Write(napnev[6]); // Ki: V
```

konstruktorban megadjuk a kezdeti kapacitást

4. Tároljuk el az egyjegyű prímszámokat, majd írjuk ki őket szóközökkel elválasztva!

Egészek listája

A kapacitása 5

```
 9.  List<int>  primek1 = List<int>(5) {2, 3, 5, 7};
10.  for (int i = 0; i < primek1.Count; i++)
11.    Console.Write("{0} ",primek1[i]);
```

A ciklusszámlálót használhatjuk indexnek

Az elemszám (4) a List tulajdonsága

5. A tömböknél megismert Split() függvényt használva írjunk programot, ami bekéri a heti lottószámokat egy sorban, szóközökkel elválasztva, majd egész számok listájában eltárolja! Úgy írjuk meg a programot, hogy az ötös és a hatos lottóhoz is jó legyen!

A megoldás során az adatok szöveges tárolásához is listát használunk.

```
 9.  Console.Write("Mik a heti lottószámok? ");
10.  string sor = Console.ReadLine();
11.  List<string> adatok = List<string>(sor.Split(' '));
12.  List<int> lotto = new List<int>(adatok.Count);
13.  for (int i = 0; i < adatok.Count; i++)
14.    lotto.Add(int.Parse(adatok[i]));
```

létrehozás és feltöltés egy string tömb elemeivel

annyi a kapacitása, amennyi a szöveges adat

Láthattuk, hogy a lista elemeit egyenként tudjuk használni, egyenként tudjuk kiírni. Ehhez a [] szögleteszárójelek között az adat listán belüli sorszámát adjuk meg, ahol az első elem sorszáma 0. A listában nincsenek lyukak, a lista elemszámát a `Count` tulajdonsága határozza meg. Ha a lista létrehozásakor nem adunk meg adatot, akkor a lista elemszáma 0. A lista minden elemének a sorszáma kisebb, mint az elemszám. Ennek ismeretében egy for-ciklussal könnyen kiírhatók a listaelemek. A listák használatakor figyelnünk kell arra, hogy a kapacitás és az elemszám két különböző adat. A fenti kód 12. sorában a `lotto` lista elemszáma 0, csak a kapacitása lett annyi, amennyi az `adatok` elemszáma.

6. Egészítsük ki a kacatokat tároló programunkat! Írjuk ki minden lépés után egy-egy sorba a `kacat` lista kapacitását és méretét, ezt követően pedig a tárolt kacatok listáját egymás alatti sorokba!

```
 9.  List<string> kacat = new List<string>();
10.  Console.WriteLine("Hely: {0}\tDB: {1}", kacat.Capacity, kacat.Count);
11.  kacat.Add("csat");
12.  Console.WriteLine("Hely: {0}\tDB: {1}", kacat.Capacity, kacat.Count);
13.  kacat.Add("gombolyag");
14.  Console.WriteLine("Hely: {0}\tDB: {1}", kacat.Capacity, kacat.Count);
15.  kacat.Add("vonatjegy");
16.  Console.WriteLine("Hely: {0}\tDB: {1}", kacat.Capacity, kacat.Count);
17.  for (int i = 0; i < kacat.Count; i++)
18.    Console.WriteLine(kacat[i]);
```

Kiíráskor csak a Count számít!

A szöveg karakterlánc

Miután megismertük több összetartozó adat kezelésének lehetőségét, érdemes egy kicsit átgondolni, hogy mi a helyzet a `string` adattípussal. A `string` karakterek sorozata, karakterlánc, esetleg karakterek tömbje vagy listája. Bár egy adatnak tekintettük, valójában mindvégig tudtuk, hogy több adat együtt alkot egy `string`et, azaz épp úgy összetett, mint a most tanult tömb és lista típus.

Vajon a `string` tömb, vagy lista? Vizsgáljuk meg a tulajdonságait:

- A `string`nek épp úgy `Length` a hossza, mint a tömbnek.
- A `string`nek épp úgy pont annyi karaktere van, mint amennyi a mérete, ebben a listához hasonlít.
- A `string`ben a kapacitás mindig akkora, ahány karakter van benne, ebben mindkét típustól különbözik.
- A `string` karakereit ugyanúgy szögleteszárójelek között *index*eléssel lehet elérni, az első sorszám a 0; ebben mindkét típushoz hasonlít.

- A stringet nem az Add() függvénnyel bővítjük, hanem a + operátorral fűzünk össze vagy a += operátorral fűzünk hozzá. Ráadásul nem csak karaktereket, hanem szövegeket is hozzá tudunk fűzni.
- A string karaktereit egyenként meg tudjuk nézni, ki tudjuk írni, másik szöveghez hozzá tudjuk fűzni, de nem tudjuk módosítani. A tömb és a lista elemeit, a tárolt változók értékét lehet módosítani, de a „hamut”-ból nem lesz a 0. karakter módosításával „mamut”.
- A stringnek más és sokkal több – szövegkezelő – függvénye van, ami egy tetszőleges adattípus tárolására alkalmas tömb vagy lista esetén nem értelmes.

Összességében, a string is összetett adat, olyan, mint a tömb vagy a lista, de sem nem tömb, sem nem lista.

A nagymértékű hasonlóság miatt egy string-ből a ToCharArray() függvénnyel előállítható a karaktereinek tömbje, amit már karakterenként tudunk módosítani. Szintén a hasonlóság miatt, egy string típusú adatból, a lista konstruktorában megadva, létre lehet hozni a karakterek listáját.

A karakterek tömbjének és listájának szöveggé alakításához adná magát a ToString() függvény, de ez nem a kívánt eredményt adja, hanem az összetett adat típusát írja ki. Belegondolva … a ToString() csak a karakterekből álló tömb és lista esetén lenne értelmes, minden más esetben csak bajt okozna, ha összefűzné az elemeket.

Másik irányból közelítve, a string egy összetett adattípus, ezért ennek is, mint a tömbnek és a listának van konstruktora. Ahogyan egy listát létre lehet hozni ugyanolyan adattípusokból álló tömbből, úgy egy stringet is létre lehet hozni akár karakterek tömbjéből akár karakterek listájából.

```
 9.  string szo = "ati";
10.  char[] kartomb = szo.ToCharArray();
11.  kartomb[1] = 'k';
12.  string tszo = new string(kartomb);
13.  List<char> karlist = new List<char>(szo);
14.  karlist[1] = 'l';
15.  string lszo = new string(karlist);
16.  Console.WriteLine(szo + " " + tszo + " " + lszo);
```

Csak a teljesség kedvéért: láttuk, hogy bármilyen típusú tömbből a lista konstruktora képes ugyanolyan típusokból álló listát készíteni. Visszafelé ez nem működik, mert a tömb konstruktora speciális, a szögletes zárójelek között csak a méret adható meg. Itt ugyanaz a trükk kell, mint a string esetén, a List<> típusú változónk ToArray() függvényét használhatjuk.

```
 9.  int[] primek = {2, 3, 5, 7};
10.  List<int> primlist = new List<int>(primek);
11.  int[] primtomb = primlist.ToArray();
```

Akkor is jól jöhet a tömb vagy lista elemeinek egy stringbe egyesítése, ha nem karaktereket tartalmaznak. Ilyenkor tipikusan szükség van valamire, ami az adatokat elválasztja egymástól. A feladat hasonló az adatok kiírásához, de van, hogy nem a kimeneten kellene összeállnia a teljes szövegnek, hanem egy változóban. A kiíráshoz hasonlóan a megoldás: egy ciklusban az elemi adatok ToString() függvényét használva – esetleg formázva – elvégezzük az összefűzést. Tömeges egyszerű szövegegyesítéshez ezenkívül használhatjuk a string.Join() függvényt, amiben megadható az adatokat elválasztó szöveg is.

String interpolation

Talán ijesztő a cím, de akinek sokszor kell teljes mondatokat létrehoznia, az pont ilyet szeretne. Ez egy C# nyelvi specialitás – nem kötelező tananyag, mindig van más megoldási lehetőség is –, amit csak egyszer kell látni és attól kezdve ezt használja a C#-ban programozó.

Egy korábbi feladatunkban két véletlenszám összegét kellett kitalálnia a felhasználónak. Ehhez ki kell írni a konzolra, hogy mely számokat kell összeadni.

```
11.  Console.Write("Mennyi {0} és {1} összege? ", egyik, masik);
```

Ezzel a helyőrzős (composit) módszerrel könnyen ellenőrizhető a kiírt mondat helyessége, a kiírás helyén a behelyettesítendő adat formátumát is meg tudjuk adni, de a változók felsorolásánál figyelni kell, hogy mit hova helyettesítünk. Ráadásul, ez csak kiíráskor használható, nem tudunk így `string`et létrehozni.

Tudjuk, hogy ugyanezt lehet így is írni:

```
11.  Console.Write("Mennyi "+ egyik +" és "+ masik +" összege? ");
```

Itt a kiírandó adat egyetlen string, amit összefűzéssel hozunk létre. Komoly figyelmet igényel az idézőjelek szóközök és + jelek helyes alkalmazása. Az egyes változóknak a `ToString`() függvényében lehet megadni a formátumot, de ez jelentősen hosszabbá teszi a kiírást, rontja az átláthatóságot.

A *string interpolation* módszerrel a `string`be értékeket szúrhatunk be. Ha ilyen `string`et szeretnénk beírni, akkor egy `$` jellel kell jeleznünk a fordító számára, hogy nem tisztán karaktersorozatról van szó, írunk bele változókat is. Majd jöhet a kiírandó szöveg úgy, ahogy elképzeljük:

```
11.  Console.Write($"Mennyi {egyik} és {masik} összege? ");
```

A változók formázása a helyőrzős formázással azonos, a változónév után vesszővel írhatjuk az adat igazítását és szélességét, kettősponttal megadhatjuk a számformátumot. További hasznos tulajdonsága ennek a nyelvi megoldásnak, hogy nem kötődik a kiíráshoz, értéket adhatunk egy `string` változónak ezzel a formátummal.

```
11.  string hexmeg = $"Mennyi {egyik:X} és {masik:X} összege?";
```

Miért nem ezzel kezdtük? Azért, mert az összefűzés és a helyőrzős megoldás általánosabb ismeretet. Ez a megoldás csak a C# nyelv 6. verziójától érhető el és vannak olyan fordítóprogramok, amelyek nem ismerik fel.

Bekért adatok ellenőrzése (do-while-ciklus)

A kacatos listát a felhasználó saját kacatjaival töltjük fel. Minthogy senkinek sincs csak egy kacatja, ciklust szervezünk a feladatra. A kacatokat egyesével kérdezzük meg és mindegyiket betesszük az adatsorozatunkba. De honnan fogjuk tudni, hogy befejezhetjük-e, hogy nincs több bekérendő kacat? A megoldás kétféle lehet.

1. A kacatok nevének bekérése előtt megkérdezzük a felhasználót, hogy hány kacatot szeretne megadni.
2. A kacatok nevének bekérése előtt megmondjuk a felhasználónak, hogy hogyan jelezze, hogy befejezte a beírást. Például, írja be azt, hogy „elfogyott”.

Az első esetben a megadott számnak megfelelő méretű tömböt hozunk létre, vagy létrehozunk egy, a megadott értéknek megfelelő kapacitású listát. Ezután, mivel ismerjük a beírni kívánt adatok számát, praktikusan for-ciklussal (de a while-ciklus is megfelel) be tudjuk kérni az adatokat.

A második esetben nem tudjuk, hány elemünk lesz. Ha tömböt szeretnénk használni, akkor olyan nagy méretet kell megadni a kódunkban, ami biztosan elég lenne. Mennyi legyen? Becsüljük meg: a beírt adatok a program bezárásával elvesznek, 8 órán át percenként 30 adat … Kerekítsünk felfelé, legyen 20 000. De lehet, hogy valami gép fogja beírni, nem ember … Ezért jobb megoldás, ha beírunk a program elejére egy értelmesnek tűnő értéket (pl. 10), ekkora méretű tömbbel dolgozunk. A programba beleírjuk azt is, hogy ha elérte a határt a kacatok száma, akkor fejezze be az adatok bekérését, továbbá kérjen elnézést a kapacitáshiány miatt.

Ha listával oldjuk meg a feladatot, akkor elvileg nem kell foglalkoznunk az adatok maximális számával. De elképzelhető – ha például a felhasználó egy gép –, hogy addig tölti a listánkba az adatokat, amíg el nem fogy a gépünk memóriája. Erre nem igazán tudunk felkészülni, de programunk jövője szempontjából nem is reális.

Bármelyik megoldást választjuk, minden beírt kacatot meg kell vizsgálnunk mielőtt beillesztjük az adatsorozat végére, mert ha a beírt adat az, hogy „elfogyott”, akkor nem kell beilleszteni az adatsorozatunkba, de be kell fejeznünk az adatok bekérését. Az adatbekérés ciklusának a vége eszerint egy feltételtől függ: a beírt adat elfogyott. Másként: a kacat nevét akkor kérjük, amíg az előzőleg beírt adat nem lesz az „elfogyott”. (Ezt a feltételt kell esetleg kiegészíteni azzal, hogy „és a beírt kacatok száma nem érte el a megadott határértéket”.)

Van ezzel a megoldással egy kis nehézség, rögtön az első adat beírásakor, ugyanis az első adat előtt nincs adat, amit ellenőrizhetnénk. Ezért a megoldásunkban „előre-olvasás” szükséges. Ez azt jelenti, hogy az első adatot beolvassuk és csak ezután jön a feltételes ciklus.

Az adatbekérést követően érdemes kiírni az összes eltárolt adatot, ezzel „visszaigazoljuk” az adatok rögzítését. Az adatok közé vesszőt és szóközt írjunk, a végén legyen pont.

Feladatok

1. Írjuk meg a programunk mondatszerű algoritmusát, foglaljuk össze ebben eddigi gondolatainkat!

```
kacattar létrehozása
ki: "vége: elfogyott"
ki: "Kérek egy kacatot"
be: kacat
ciklus amíg kacat != "elfogyott":
    kacattar.betesz(kacat)
    ki:"Kérek egy kacatot"
    be: kacat
ciklus vége
ki: "feljegyzett kacatok:"
ciklus 0-tól kacattar.elemszama-1-ig:
    ki: kacattar.eleme + ", "
ciklus vége
ki: kacattar.eleme + "."
```

Ha figyelnünk kell az adatok számára, akkor több helyen kiegészül az adatok bekérése:

```
maxkacat := 10
kacattar létrehozása
darab := 0
…
ciklus amíg kacat != "elfogyott" ÉS darab < maxkacat:
    kacattar.betesz(kacat)
    darab++
…
ciklus vége
ki: "feljegyzett kacatok:"
…
ki: kacattar.eleme + "."
ha darab = maxkacat:
    ki: "Sajnos, csak ennyit tudok eltárolni."
```

2. Írjuk meg a programot! Az ellenőrzéshez egy tömbös megoldás:

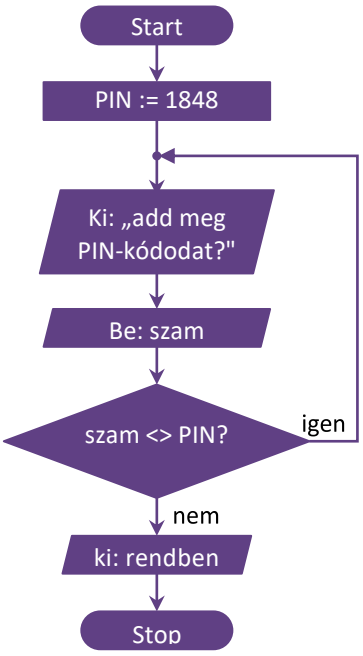
```
 8.  int maxkacat = 10;
 9.  string[] kacattar = new string[maxkacat];
10.  int darab = 0;
11.  Console.WriteLine("Vége: \"elfogyott\""); //\" kiírja az "-t
12.  Console.Write("Kérek egy kacatot");
13.  string kacat = Console.ReadLine();
14.  while (kacat != "elfogyott" && darab < maxkacat)
15.  {
16.    kacattar[darab] = kacat;
17.    darab++;
18.    Console.Write("Kérek egy kacatot");
19.    kacat = Console.ReadLine();
20.  }
21.  Console.Write("Feljegyzett kacatok: ");
22.  for (int i = 0; i < darab - 1; i++)
23.    Console.Write(kacattar[i] + ", ");
24.  if (darab == 0)
25.    Console.WriteLine(kacattar[darab - 1]+ ".");
26.  if (darab == maxkacat)
27.    Console.WriteLine("Sajnos, csak ennyit tudok eltárolni");
```

Ahhoz, hogy a feladatot a leírásnak megfelelően oldjuk meg – a kiírás végén pont legyen – a minta megoldás 22. sorában a kiírandók számát eggyel csökkenteni kellett, a ciklust követően tudjuk kiírni az utolsó adatot – de csak akkor, ha létezik – a végén a ponttal. Másik megoldás, hogy a cikluson belül feltételhez kötjük az elválasztójel kiírását. Harmadik megoldás a ciklus után `Console.WriteLine("\b\b.");`. Melyik jobb és miért?

Bármilyen módon írtuk meg a programunkat, az első kacatot kivételnek kellett tekintenünk. Emiatt egy kódrészlet kétszer szerepel, amit a programozók nem tartanak jó megoldásnak. Ezért van több programozási nyelvben lehetőség olyan ciklus írására, aminek a végén ellenőrizzük, hogy kell-e még ismételni a feladatot, azaz hátul teszteljünk. A C# – és sok más – nyelvben a do-while-ciklus elején a **do** kulcsszó jelöli azt a pontot, ahova vissza kell lépni. Ezt követi a ciklusmag, majd ennek lezárása után a **while**-nak a paramétereként azt adjuk meg, hogy mikor kell visszatérni a **do**-hoz.

Hátul-tesztelő ciklust mondatszerű leírásban is használhatunk, a programunk algoritmusa is egyszerűbb lehet így.

```
kacattar létrehozása
ki: "vége: elfogyott"
ciklus:
    ki:"Kérek egy kacatot"
    be: kacat
    ha kacat != "elfogyott"
     kacattar.betesz(kacat)
amíg kacat != "elfogyott"
ciklus vége
ki: "feljegyzett kacatok:"
ciklus 0-tól kacattar.elemszam-1-ig:
    ki: kacattar.eleme + ", "
ciklus vége
ki: kacattar.eleme + "."
```



A hátultesztelő ciklus alkalmazásával az adatbekérés egységes lett, de a cikluson belül is ellenőrizni kell a választ, vagy a bevitel végén valamit kezdeni kell azzal, hogy az „elfogyott" is bekerült a kacatok közé. A megoldás valami remove-szerű függvénye lehet a `kacattar`-nak.

A hátul-tesztelő ciklust – ha a programozási nyelvben létezik – jellemzően adatbekérésre szokták használni. Abban az esetben ideális, amikor újra kérnünk egy, a feltételeinknek nem megfelelő adatot.

Például, amikor egy PIN-kódot addig kérdezünk, amíg a felhasználó el nem találja. A folyamatábrán jól látszik, hogy a visszalépés nem közvetlenül a feltétel elé történik, hanem egy korábbi pontra, másrészt ezen az ágon nincs (és nem is lehet) utasítás.

A hátul-tesztelő ciklusnak egyes programozási nyelvekben kissé más a megvalósítása: a ciklus végén nem azt vizsgálja, hogy vissza kell-e lépnie, hanem azt, hogy mehet-e tovább.

3. Írjuk át az algoritmusnak megfelelően a kódot is!

   Az ellenőrzéshez egy listás megoldás:

```
 9.  List<string> kacattar = new List<string>();
10.  Console.WriteLine("Vége: \"elfogyott\"");
11.  string kacat;
12.  do
13.  {
14.    Console.Write("Kérek egy kacatot");
15.    kacat = Console.ReadLine();
16.    if (kacat != "elfogyott")
17.      kacattar.Add(kacat);
18.  }while (kacat != "elfogyott");
19.  Console.Write("Feljegyzett kacatok: ");
20.  for (int i = 0; i < kacattar.Count - 1; i++)
21.    Console.Write(kacattar[i] + ", ");
22.  if (kacattar.Count != 0)
23.    Console.WriteLine(kacattar[kacattar.Count - 1] + ".");
```

Használjuk a do snippetet: A „do” és két TAB kiírja a kapcsos zárójeleket és a while-t is a feltétel helyével .

Informatikus szemmel nézve nem túl szerencsés, hogy a befejezéshez egy szót adunk meg. Mi van, ha egyszer véletlenül a felhasználónak mégis lesz „elfogyott” nevű kacatja? Ezt nem tudja bevinni a programunkba. Ezért a végjeles adatbekérésnél egy megadott szó helyett az „üres karakterlánc” szokott szerepelni. Ha a felhasználó a kérdésünkre csak egy ENTER-rel válaszol, akkor az eredmény a `""` lesz.

Kiegészítés: Az adatok beírását a `ReadLine`() az ENTER leütésekor kapja meg. Előtte törölhetjük a hibásan beírt adatot, erről a program nem fog tudni. Az ENTER leütésével átadott karaktersorozat nem fogja tartalmazni az ENTER kódját. Helyette általában „lezáró nulla” jelzi a szöveg végét. A lezáró nulla valójában a 0 ASCII-kódú vezérlőjel. Ha a programunkban használnánk, akkor ez a `'\0'` karakter.

Adatsorok kezelése

Most már tudunk adatsorokat tárolni és ki is tudjuk írni képernyőre. A tárolásnak a célja, hogy mindenféle kérdéseinkre gyorsan tudjon a programunk válaszolni. Persze csak akkor érdekes a feladat, ha sok adattal tudunk dolgozni. Elvileg kacatjainkból is sok van, de beírni mindet …

Hogyan tudunk sok adatot használni egy-egy programban? Az adatokat nem a programok kódja tárolja, hanem mindig kívülről kapja a program: „internetről”, fájlból, mérőműszertől vagy konzolról. Ezekre az „adatcsatornákra” programozási nyelvekben megtalálhatók az adatok írását (kiküldését) és (be)olvasását végző eljárások és függvények. Mi csak a konzolt használjuk, de a programozási nyelv dokumentációjában megtalálhatók a többi csatorna használatára is a megoldások. A megoldások keresésének egyi kulcsszava a „*stream*”, mert a csatornában az adatok egymás után sorban haladnak, a csatornába betöltik az adatot, illetve kiveszik belőle, a byte-ok értelmezés és struktúra nélküli folyamot alkotnak. Minden adatcsatornába az egyik végén lehet betölteni az adatot, a másik végén megnézhetjük, hogy mi van, kivehetjük a szükséges bájtokat (egy integer esetén például 4 bájtot veszünk ki).

Fájlból konzolra másolás

A konzol általában három csatornát tud kezelni: egyet a beolvasáshoz, egyet a kiíráshoz és egy harmadikat a technikai részletek (error vagy log) feljegyzéséhez. Ebből most számunkra a

beolvasásra használt csatorna a lényeges, mert azon keresztül lenne jó sok adatot bejuttatni a programunkba. Az első megoldás a működés ismeretéből kézenfekvő: a program nem „látja”, hogy hogyan kerül be az adat a csatorna másik végére. Nem érzékeli a billentyűlenyomásokat … csak a bevitt adatok sorozatáról tud. Ezért a csatorna másik oldalán ügyeskedhetünk: összegyűjthetjük a beírandókat és bemásolhatjuk a bemenetre.

Sok adatot úgy lehet „begépelni”, hogy egy egyszerű szöveges fájlban eltároljuk, amit majd be szeretnénk írni. A program adatsorokat olvas, ezért a beíráskor ennek megfelelően, enterrel tördeljük sorokra az adatainkat. A program futtatásakor a szöveget kimásoljuk és a konzolra beillesztjük. Ehhez az egérrel jobb-klikk, Beillesztés lehetősége vagy a CTRL+V billentyűkombináció is alkalmas.

Nem kell félni, a programunk nem fog sokkot kapni 1000 adattól még akkor sem, ha csak egyetlen számot vár – főleg, ha a bemásolt adatok első sorában valóban egy szám van –, mert a bájtok kivételét az első enterig végzi. A többi adat a csatornában vár, amíg sorra kerül, vagy amíg befejeződik a program. A program bezárásakor a csatorna is megszűnik, így a benne maradt adatok is törlődnek.

Az adatok beírásának ilyen módja abban az esetben okozhat problémát, ha az ENTER-t nem „érti meg” a programunk. Azt tudjuk, hogy a kódban a `'\n'` escape karakter jelöli ezt. Neve LF (LineFeed), ASCII-kódja 10. Ezt a karaktert Linux operációs rendszerben az ENTER, Windows operációsrendszerben a sortörés, SHIFT+ENTER billentyűleütéssel lehet beírni. A Windowsban az ENTER leütésekor `'\r'` – CR (CarrigeReturn) ASCII kódja 13 – bájt is keletkezik. Emiatt a sorok végén valójában `\r\n` van. Hogy ez ne okozzon gondot, a Windowson használt programozási nyelvekben a `'\n'` együtt értelmezi a két karaktert, beolvasáskor a CR-nél befejezi a beolvasást de még a következő LF-et is beolvassa, kiíráskor mindkettőt kiírja. Nem is lenne gond, ha mindig ott lenne a sorok végén a CR, de – mivel Linuxon nincs – a sorok végét sokszor csak az LF jelzi. Ilyenkor a program nem veszi észre, hogy hol a sor vége. A problémát bonyolítja, hogy vannak olyan operációs rendszerek, amelyekben csak a CR jelöli a sorvéget és olyanok is, amelyekben fordított sorrendben van a két kód: `\n\r`.

Nem túl biztató a helyzet, de azért van megoldás. Nagyon sok program fel van készítve arra, hogy bármelyik változatot értse és helyesen értelmezze. Egy jellemző eset, hogy a zip tömörítésből Jegyzettömbbel megnyitott txt fájlokban csak LF van, de ha kitömörítés után nyitjuk meg, akkor már CR LF lesz benne. Ha másképp nem megy, fejlett szövegszerkesztőben van mód arra, hogy a sortörést bekezdésvégjelre cseréljük, ezt visszamásolva egy txt-be, az adott operációsrendszerben használható szöveges fájlt kapunk.

Konzol átirányítása

A konzol csatornáinak egyik végén a program van, a másik vége „szabadon”. Láttuk, hogy a gépelés helyett a „Vágólap”-ról is tudunk adatot bevinni. Az operációs rendszerek nagyon sok olyan programmal dolgoznak, amelyek egymásnak adják át az adatokat, ezért képesek arra, hogy az egyik program konzol kimenetét egy másik program konzol bemenetére irányítsák – összekössék a csatornákat. Ennek kiegészítéseként, egy programba fájlból is be tudjuk irányítani az adatokat. A lehetőséget az operációs rendszer adja, ezért nem az IDE-ből futtatjuk ilyenkor a programunkat, hanem a fordítás után, egy konzolablakban (terminál ablakban) indítjuk el a programunkat. A fájl „becsatornázása” a programba a ’<’ jellel oldható meg. Például így indítjuk a programot:

```
program.exe < adatok.txt
```

A kiírást is átirányíthatjuk, ekkor nem a képernyőre, hanem a fájlba ír a programunk:

```
program.exe > eredmeny.txt
```

Lehet egyszerre mindkettőt használni, a bemenő adatokból előállítani az eredményfájlt:

```
program.exe < adatok.txt > eredmeny.txt
```

Lényegében ezt használják ki, amikor egy programot tesztelnek. Az operációs rendszerben lehet írni olyan scriptfájlokat, amelyekben megadják a bemeneteket és a kimeneteket, majd egy ciklussal sorban mindegyikkel lefuttatják a programot. A kimenetet persze át lehet adni egy kiértékelő programnak is, amelyik elemzi az eredményt és a kimenetén megjelenik az értékelés…

Szerencsejáték esélyek

Most már meg tudnánk vizsgálni korábban eltárolt adatokat is, de az adatfájlok elkészítése is idő. Ezért tárolt adatok nélkül, kockadobással gyártunk sok adatot.

Feladatok

1. Szimuláljunk tízmillió kockadobást, és tároljuk az eredményeket!

   A feladatot tömb vagy lista használatával oldjuk meg, aminek a neve `dobasok`. Ebben az esetben – mivel pontosan tudjuk, hogy mennyi adat lesz, egyszerűsége miatt – a tömb a „szebb” megoldás. A számok előállításához egy véletlenszám-generátort használunk tízmilliószor.

2. Programunk számolja meg, hogy hányszor „dobtunk” hatost!

   Csak az adattípustól független kódrészletet nézzük:

```
15.  int hatosdb = 0;
16.  for (int i = 0; i < 10000000; i++)
17.  {
18.    if (dobasok[i] == 6)
19.    {
20.      hatosdb++;
21.    }
22.  }
23.  Console.WriteLine("Összesen {0} darab hatost dobtunk." ,hatosdb);
```

3. Számoljuk össze mind a hat lehetőség előfordulásait!

   A megoldást először egy 1 és 6 között futó külső ciklussal készítjük el:

```
25.  for (int szam = 1; szam <= 6; szam++)
26.  {
27.    int db = 0;
28.    for (int i = 0; i < 10000000; i++)
29.    {
30.      if (dobasok[i] == szam)
31.      {
32.        db++;
33.      }
34.    }
35.    Console.WriteLine("{0} esetben dobtunk {1}-t." ,db, szam);
36.  }
```

Viszonylag kevés módosítással hatszor több eredményünk van. De a megoldási idő is hatszor több, hiszen a tízmillió értéket hatszor nézzük végig. Oldjuk meg úgy a feladatot, hogy csak 1-szer nézünk meg minden dobott értéket és eldöntjük, hogy melyik csoportba tartozik!

Ehhez az dobás értékének vizsgálatát kellene hatszor beírni, természetesen 6 külön változóba. Sem a 6-szor beírni majdnem ugyanazt, sem a 6 külön változót létrehozni nem vidít fel egy programozót. Kezdjük a „6-szor beírni” szépítésével.

Beírhatunk 6 `if`-et, de akkor a programunk mindig, mind a hatot ellenőrzi, akkor is, ha már az első helyes volt (és a többi nem lehet az). Ezért, ha így oldjuk meg, akkor az `else if` sokkal praktikusabb.

Mivel itt csak néhány konkrét értékeket kell megvizsgálnunk, a feltételek elég unalmasak lesznek: egyenlő az egyikkel, egyenlő a másikkal … Ilyenkor lehet bevetni egy `switch`-et. Bár a megoldás nem lesz lényegesen rövidebb, de a switch-snippet segít és a kód átláthatóbb lesz.

```
25.  int db6, db5, db4, db3, db2, db1;
26.  db6 = db5 = db4 = db3 = db2 = db1 = 0;
27.  for (int i = 0; i < 10000000; i++)
28.  {
29.    switch (dobasok[i])
30.    {
31.      case 1: db1++; break;
32.      case 2: db2++; break;
33.      case 3: db3++; break;
34.      case 4: db4++; break;
35.      case 5: db5++; break;
36.      case 6: db6++; break;
37.      default: break;
38.    }
39.  }
```

Egész szám vagy karakter… Ez == valamelyik eset

break: kilép a switchből. ha hiányzik, akkor a következő utasítást is végrehajtja

A kód áttekinthetőbb, ezért még jobban látszik, hogy a hat változó mennyire egyforma funkciójú. Ezek bizony egy tömbbe tartoznak! Ha létrehozunk egy tömböt nekik, és az első helyen az 1-es dobások értékét gyűjtjük, akkor könnyen belezavarodhatunk, mert az a 0. index. Ezért úgy készítsünk tömböt, hogy az első adat legyen 0, a második helyen – az 1-es indexű helyen számláljuk az 1-es dobások számát!

Csak a `switch` részt kiemelve, a kódunk így néz ki az átalakítás után:

```
29.    switch (dobasok[i])
30.    {
31.      case 1: darab[1]++; break;
32.      case 2: darab[2]++; break;
33.      case 3: darab[3]++; break;
34.      case 4: darab[4]++; break;
35.      case 5: darab[5]++; break;
36.      case 6: darab[6]++; break;
37.      default: break;
38.    }
```

Egy változónévben bármilyen szám szerepelhet, a db1 lehetne `dbegy` is. De a jelenlegi formában a `darab`[1] változóban az 1 is szám, a darab tömb egyik elemének az indexe. Ha kiírjuk az adatokat, akkor erre a számra hivatkozva választjuk ki, hogy melyiket szeretnénk kiírni. A kiírás közben az index értéke változik – így lép egyik adatról a másikra. Az index helyére eszerint bármilyen egész számot eredményező kifejezést be lehet írni. A kódot elemezve

megállapítható, hogy *a darabszámokok közül mindig azt kell növelni, amelyiknek az indexe megegyezik a* `dobasok[i]` *értékével.* Ezt a gondolatot kódolva, a `switch` egyetlen sorral helyettesíthető:

```
29.    darab[dobasok[i]]++;
```

Azaz vegyük az annyiadik darabot, amennyi az `i`-edik dobás értéke és növeljük meg 1-gyel.

A kód végső állapota az eredmény kiírásával együtt:

```
25.  int[] darab = {0,0,0,0,0,0,0};
26.  for (int i = 0; i < 10000000; i++)
27.    darab[dobasok[i]]++;
28.  for (int szam = 1; szam <= 6; szam++)
29.    Console.WriteLine("{0} lett {1} dobás.", szam, darab[szam]);
```

A megoldás nem tartalmaz speciális programnyelvi eszközt, a megoldás feltétele, hogy

- a tömb vagy lista minden indexe bármikor elérhető legyen, azaz az egyes elemeket az indexeiken keresztül közvetlenül elérjük.
- értsük, hogy pontosan mit jelent egy adatsorozat elemének az indexe, illetve értéke; értsük, hogy egy adatsorozat egyik elemének az értéke megadhatja egy másik adatsorozatban egy elem indexét, amelynek az értékét így módosíthatjuk.

Programozási tanulmányaink kezdetén a feladatnak mindegyik itt látható megoldása helyes (max. pontos). Nem baj, ha az utolsó még zavaros, az első megoldás is jó. A `switch` használatát sem kell tudni. A programozás egy kicsit olyan, mint a nyelvismeret, ahol 100 szó elég, hogy megértessük magunkat, mégis, évekig tanuljuk, fejlesztjük a szókincsünket, tanuljuk a nyelv szabályait gondolataink minél pontosabb kifejezése érdekében.

Indexek használata

Az előző feladatban láthattuk, hogy az adatsorok elemeit nem csak sorban lehet elérni. A következő feladatokban az lesz a megoldás kulcsa, hogy – bár egymás után nézzük az egyes elemeket, de egy-egy lépésben az adatsor több elemét vizsgáljuk.

4. Hány helyen előzi meg a hatos dobást ötös dobás?

   A feladat nagyon hasonló a 2. feladathoz, amikor a hatos dobások számát vizsgáltuk, de módosult a számlálás – a jó eset – feltétele: hatos előtt ötös. Az egyik dobás értéke 6 és az előtte lévő dobás értéke 5.

   Írjuk át a 2. feladat megoldását ennek megfelelően:

```
32.  int otrehatdb = 0;
33.  for (int i = 1; i < 10000000; i++)
34.  {
35.    if (dobasok[i] == 6 && dobasok[i - 1] == 5)
36.    {
37.      otrehatdb++;
38.    }
39.  }
40.  Console.WriteLine("{0} esetben lett 5 után 6." ,otrehatdb);
```


5. Hány helyen van egymás után két hatos?

Oldjuk meg a feladatot először úgy, hogy amikor három hatos van egymás után, azt tekintsük két olyan esetnek, amiben két hatos van egymás után. Ezután módosítsuk úgy a programunkat, hogy csak azokat az eseteket számolja, amelyekben pontosan két darab hatos van egymás után. Ennek megoldásában praktikus néhány speciális esetet a cikluson kívül megvizsgálni.

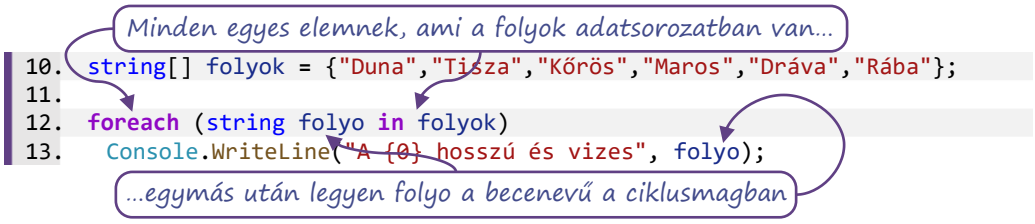
A bejárós ciklus (foreach-ciklus)

Mostanra megismerkedtünk a feltételes ciklusokkal. Először a while-ciklussal, ahol a ciklusmag előtt döntjük el, hogy végrehajtjuk-e vagy kihagyjuk. Tettünk egy rövid kitérőt a hátultesztelő do-while-ciklus felé. Ez abban az esetben könnyíti meg az algoritmus és a kód írását, amikor a ciklusmag végrehajtása után szeretnénk eldönteni, hogy szükséges-e visszamenni és újra (meg újra) végrehajtani a ciklusmagot.

A for-ciklust szinte a while-ciklussal egyidőben kezdtük használni és az adatsorozatok megjelenésével egyre inkább látható a praktikussága: a ciklusfejben adjuk meg a számláló változását, mert sokkal kisebb a valószínűsége a lefelejtésének. Most már az is látszik, hogy az adatsorozatok elemeinek egymás utáni elérését is megkönnyíti, hogy az iterátort (számlálót) fel tudjuk használni az adatsorozat elemeinek indexeléséhez, egymásutáni kiválasztásához. Az iterátor és index olyan gyakran kapcsolódik össze, hogy az i-t nem csak az iterátor szó kezdőbetűje alapján használjuk a ciklusokban. Az adatsorozatok tetszőleges elemének indexét is tipikusan i-vel jelöljük. A matematikai függvényekben – f(x), |x| – az x a függvény értelmezési tartományának egy tetszőleges eleme. Az adatsorozatok esetén hasonló értelemben használjuk az i-t, a feladatról írva: egy *A* adatsorozat tetszőleges eleme *A[i]*, azaz az *A lehetséges indexeinek egyike által meghatározott adat*.

Adatsorok elemzése, statisztikai számítások végzése során gyakori, hogy az adatsor minden elemén végig kell lépkedni és az elemzéshez, az elemen vagy elemmel elvégzendő tevékenységhez az adatsorból csak az éppen kiválasztottra van szükség. Ezeknek a feladatoknak a megoldására írták meg nagyon sok programozási nyelvben a bejárós ciklust.

A bejárós ciklus egyesével végiglépked egy bejárható objektum, például egy lista értékein. Értékein lépked, azaz a ciklusváltozó nem az index lesz, hanem az adat. A ciklus magjában használhatjuk ezt a változót: megnézhetjük, módosíthatjuk az értékét.



Az többi ciklushoz hasonlóan, a foreach-ciklus paramétereit a ciklusfejben adjuk meg, ezt követően egy utasítás a ciklusmag vagy kapcsos zárójelek között adhatunk meg több utasítást.

Ugyanezt mondatszerű leírásban is megadhatjuk:

```
folyok létrehozása
ciklus folyok minden folyo-jára:
    ki: "A " + folyo + " hosszú és vizes."
ciklus vége
```

A foreach-ciklus ciklusváltozójának a típusa mindig megegyezik az adatsorozat típusával, emiatt a példában szereplő folyo egy for-ciklus minden i-jére azonos folyok[i]-vel. Főleg kezdetben, az adattípusok összekeverésének elkerülése miatt érdemes kiírni a ciklusváltozó elé a valódi adattípust. Mivel ez mindig az adatsorozat elemeinek az adattípusa, ezt a fordítóprogram (is) ki tudja találni. Ezért – ha már jól tudjuk használni a foreach-ciklust – az adattípus kitalálását rábízhatjuk a fordítóprogramra. A C#-ban a „találd ki, milyen adattípus” jelölése: `var`.

A foreach snippet adja magát: `foreach (var item in collection) { }`
Átírandó részek között TAB-bal mozoghatunk.

A foreach-ciklust nagyon kényelmes használni. Annyira, hogy egyeseknél az addikció gyanúja is felmerül: a feladat megoldásához – akkor is, ha nem praktikus – a foreach-ciklust próbálja használni, ezért a kód – ha egyáltalán sikerül megírni – bonyolult lesz vagy hibás eredményt ad. Ezért nézzük, hogy mely esetekben **nem jó** a foreach-ciklus:

- A foreach minden adatsorozat-típusra használható, a tömbre is. DE, ha a tömbünk mérete nagyobb, mint amennyi adataink tárolásához szükséges (vagy nem tudunk válaszolni erre a kérdésre), akkor jobb, ha nem foreach-et használunk, mert az a használaton kívüli lefoglalt területet is vizsgálja.
- Ne használjunk foreach ciklust, ha van olyan adat, amelyet ki szeretnénk hagyni. A korábban megoldott feladatok közül ilyen:
  - Adatsorozat elemeinek kiírása úgy, hogy közben vessző választja el az adatokat, de az utolsó után pont van. (Kacatok kiírása.)
  - Olyan adatsor elemeinek kiírása, ahol a 0. elemet – az értelmezés miatt – nem használjuk. (Kockadobás értékeinek számlálása.)
- Olyan feladatokban, ahol egymáshoz viszonyított helyzetet kell vizsgálni. (Kockadobás szomszédos adatok.)
- Olyan feladatokban, ahol nem az adat értékét kell megadni, hanem azt, hogy melyik adatról van szó, hányadik, hol van az adatsorozaton belül. Ezeknél a feladatoknál az érték megadása után külön kell a helyet megkeresni. Ha egy feladatban az index és az érték megadása is szükséges, akkor elsősorban az index megadása a cél, mert abból keresés nélkül megadható az érték. Fordítva nem igaz, mert attól, hogy ismerjük az értéket, még nem tudjuk, hogy hol van.

Feladatok

1. Írjunk olyan programot, amely egy futóverseny résztvevőinek célba érkezés szerinti neveit kéri a felhasználótól, majd kiírja a dobogósokat és a sereghajtót! Minden újabb versenyző bekérése előtt írja ki az épp aktuális névsort!

```
List<string> versenyzok = new List<string>();
string versenyzo;
do
{
  Console.WriteLine("A versenyzők névsora:");
  foreach(string v in versenyzok)
    Console.WriteLine(v);
  versenyzo = Console.ReadLine();
  if (versenyzo != "")
    versenyzok.Add(versenyzo);
} while (versenyzo != "");
for (int i = 0; i< versenyzok.Count && i < 3; i++)
  Console.WriteLine("{0}. helyezett: {1}", i + 1, versenyzok[i]);
if (versenyzok.Count > 1)
  Console.Write("A sereghajtó: {0}", versenyzok[versenyzok.Count - 1]);
```

2. A fenti megoldás a négyféle ciklusból hármat tartalmaz, csak a while-ciklust nem használja. Másrészt, a while-ciklus a legáltalánosabb, a többi átírható erre. Írjuk meg a kódot csak while-ciklus használatával!

Adatsorozatok függvényei

Nemcsak a bejárós-ciklus alkalmazása tipikus adatsorok esetén, hanem egyes tulajdonságok, műveletek, statisztikai feladatok is. Ilyen tulajdonság az adatsorozat elemszáma (`Count` vagy `Length`), műveletként alapvető a hozzáfűzés `Add`() eljárása. Az egyes adatsorozat-típusoknak különböző függvényei vannak, érdemes tájékozódni felőlük az IDE intellisence helyi kódjavaslatai alapján.

Például egy `List<string>` típusú adatnál az `Add`() és `Count` mellett még több, mint 30 másik dolgot látunk. Keressünk olyanokat, amelyeknek érthető a helyi leírása (nem csak az, hogy mi a funkciója, hanem az is, hogy milyen paramétereket kell megadni). Esetleg ki is próbálhatunk néhányat. Néhány ígéretes függvénynév: `IndexOf`(), `Insert`(), `Remove`(), `Sort`()

Ha még nem tettük, most már ideje megnézni, hogy mit tud a `string`, milyen függvényei vannak. Néhány függvényről már volt szó, ezeket is elevenítsük fel és nézzük meg, melyik lehet még érdekes. Például a `Replace`(), a `ToUpper`().

Feladatok

1. Írjunk programot, amely egy autókölcsönző munkáját szimulálja! A kölcsönző a munkanapot egy listányi autóval kezdi, és addig kölcsönöz, amíg minden autót ki nem ad. A program írja ki az autók listáját és kérdezze meg, melyiket kölcsönzi ki a felhasználó. Írja ki, hogy mik maradtak benn, és kérdezzen újra és így tovább.

```
List<string> autok = new List<string>()
                                {"Trabant", "T-Modell", "Rolls-Royce" };
while(autok.Count > 0)
{
  Console.WriteLine("Kölcsönözhető: {0}", string.Join((", ", autok));
  Console.Write("Melyik autót kölcsönzi ki? ");
  string mit = Console.ReadLine()
  if(autok.Contains(mit))
    autok.Remove(mit);
  else
    Console.WriteLine("Ilyen autóval nem szolgálhatunk");
```

2. Írjuk meg a programot a `Contains`() és `Join`() függvény használata nélkül!
3. Írjuk meg a programot úgy, hogy az adatokat (kellően nagy méretű) tömbben tároljuk és a `Remove`() helyett a töröltet követő adatokat eggyel előre tesszük!

Adatsorozatok és ciklusok

Feladatok

A következő feladatok megoldását – amennyire tudjuk – oldjuk meg többféle ciklussal és – esetleg – tömb és lista használatával is!

1. Írjunk olyan programot, amely egy-egy listába bekéri három-három leves, főétel és desszert nevét, majd kiír három menüt, mindegyikben egy levessel, egy főétellel és egy desszerttel!
2. Írjuk ki az első 10 természetes számot és a négyzetüket!
3. Három egymásba ágyazott ciklussal rajzoljuk ki a jobb oldalon láthatóábrát!

```
 8.  for (int tegla = 0; tegla < 3; tegla++)
 9.  {
10.    for (int sor = 0; sor < 4; sor++)
11.    {
12.      for (int hely = 0; hely < 5; hely++)
13.        Console.Write("o");
14.      Console.WriteLine();
15.    }
16.    Console.WriteLine();
17.  }
```

```
ooooo
ooooo
ooooo
ooooo

ooooo
ooooo
ooooo
ooooo

ooooo
ooooo
ooooo
ooooo
```

   a) Mi a szerepe az 7. és 9. sornak?
   b) Hol kell átírni a kódot, hogy három, az itt láthatóval egyező háromszöget rajzoljon?

```
o
oo
ooo
oooo
```

4. Állítsunk elő egy tízelemű, pénzfeldobások eredményeit tartalmazó listát! Hány olyan eset van, amikor az aktuális és az előző dobás is „fej”?
5. Állítsunk elő harmincelemű, nulla és kilenc közötti véletlenszámokat tartalmazó listát! A számok egy útvonal magassági adatait jelentik. Meredek az útszakasz, ha legalább kettővel magasabb az aktuális hely, mint az előző. Hány meredek szakasz van az úton? És visszafelé?
6. Kihívást jelentő feladat: A programunk elején adjunk meg két listát:
   - az első tartalmazzon öt filmcímet,
   - a második a filmek egy-egy főszereplőjét!

   a) Az első filmhez az első szereplő tartozik, a másodikhoz a második, és így tovább.
   b) Írjuk ki a filmcímeket, majd az egyik, véletlenszerűen kiválasztott szereplőt!
   c) Kérdezzük meg a felhasználótól, hogy a kiírt szereplő melyik filmnek a főszereplője!
   d) Értékeljük a válaszát!
7. Kihívást jelentő feladat: Állítsunk elő nyolcvanelemű, –5 és 3 közötti egész számokból álló listát! A számok egy úszó palackorrú delfin magasságát jelentik. A delfin ki-kiugrál a vízből, ilyenkor pozitív a magassága. Nulla a magasság, amikor a felszínen úszik, negatív, amikor a víz alatt. Írjunk programot, ami választ ad a következő kérdésekre!

a) Az idő hányadrészét töltötte a delfin a vízben, illetve a víz alatt? A válaszok megadhatóak törtszámként és százalékként is. (Feltételezhetjük, hogy az adatok egyenlő időközönként érkeztek.)
b) A víz alatt vagy a víz felett volt többet a delfin? A vízfelszínen való utazás egyik esetbe sem számít bele.
c) Milyen hosszú volt a leghosszabb kiugrása? Az út hányadik pontjánál kezdődött?
d) Hányszor törte át a vízfelszínt, azaz hányszor követ a listában negatív számot pozitív, vagy fordítva?
e) Mély merülésnek számít, ha a delfin –4-es vagy –5-ös mélységben van. Az út során hányszor merült mélyre? Figyeljünk arra, hogy például a 4 –2, –4, –5, –5, 3 útvonal csak egy mélyre merülést jelent!

Tárgymutató

$ jel 57
bejárós ciklus 40, 66
breakpoint 11, 48, 52, 54
build 5
ciklusfej 42
ciklusmag 41, 42, 43, 59, 66
class 26
compiler 5, 11
debug 11
deklarál 45
double 22
do-while 40, 57, 59, 66
do-while-loop 40
eljárás 18, 26, 28, 37, 38
elöltesztelő ciklus 40
ÉS művelet 36
escape character 15
explicit 22
feltételes ciklus 40, 42, 43, 58, 66
float 22
for 40, 41, 42, 43, 46, 51, 55, 58, 66, 67
foreach 40, 66, 67
foreach-loop 40
for-loop 40
függvény 18, 20, 23, 24, 26, 28, 29, 38, 69
get 28
hátultesztelő ciklus 40, 60
helyőrző 19, 24, 57
implicit 22
index 64, 66, 67
inicializál 45
interpreter 5
iterator 42
konstruktor 37, 38
konzolablak 11
NEM művelet 37
object 38
objektum 26, 27, 37, 38, 66
rövidzár 36
set 28
stream 61
switch 64
számlálós ciklus 40, 43, 44, 46
szintaktika 9
terminál 4, 62
tulajdonság 24
VAGY művelet 36
validál 23
var 38, 67
visszatérési érték 26
void 26, 28, 38
while 40, 43, 51, 58, 66
while-loop 40