programozas_gyorstalpalo

August 27, 2022

1 Üdv

Ez egy interaktív-szerű tanuló füzet beépített *C#* kernellel, ami lehetővé teszi a sorok futtatását.
Próbáld ki nyugodtan, bár elsőre bonyolult lesz (itt egy kis help).
Ha PDF-ként kaptad meg, akkor másold ki a kódokat és futtasd offline/online, akár *VSCode*-ban.

A *sorbeolvasást* és a *névtérdeklarálást* leszámítva minden más működik, ami kimenetet ad.

Mivel nem teljes a sztori fejlesztés-oldalról és saját oldalról sem, ne lepődj meg
ha egy-két errort vagy bugot találsz. Előre bocs.

A leckék sorrendje a Sololearn (sololearn.com) (Mobile) platformból lett átvéve, és átdolgozva az érthetőség érdekében. Néhány infót pedig TutorialsPoint (tutorialspoint.com), Microsoft .NET (docs.microsoft.net) és C# Tutorial (csharptutorial.hu) oldalakról szedtem.
(ez utóbbi úgy tűnhet, mintha ezt másoltam volna, ám nekem is a saját könyvem írása végén szembesültem ezzel a pacek weboldallal. Ajánlom mindenkinek!)

1.0.1 1. C# avagy Csharp

- OOP nyelv = Tárgy-orientált = "Mindent generalizálni, instanciálni kell." xd
- .NET keretrendszeren működőképes appokra szánt
- Windows, web, mobil, szerver, adatbázis

Az ember-olvasható programfájl kiterjesztése `.cs`. Ezt a fordító/összerakóprogram (a "compiler") a .NET könyvtárait felhasználva lefordítja futtatható programmá (`.exe` vagy `.dll` vagy etc.).

1.0.2 2. Változók és Kommentek

Mégmielőtt tárgyalnánk a legkönnyebb dolgokat, több dolgot leszögezek: * Egy **kifejezés** (expression) akkor kifejezés, ha futás közben *egy értékre* fejezhető ki. (például `19` vagy `int kettő = 2;`) * Egy **állítás** (statement) a program alapvető része. *Sokfajta* van belőle, ezeket tárgyaljuk majd * Egy **test** (block) pedig *nulla vagy több állítás* csoportja egy `{}`-n belül

Változók *Változóknak* nevezzük azokat a tárgyakat amikben értéket tudunk tárolni.
Minden változót legalább egyszer (*legelőször*) el kell nevezni (**deklarálás**).
Próbálj **jól leíró** neveket használni (pl. `iSzam`, `sKeresztnev`, `tPerc`), és az ideiglenes változóknak adni a random neveket (pl. `x`, `y`, `i`, `temp`, `elem`).
Ha lehet kerüld az angol ábécén kívüli karaktereket, nem lehet tudni mikor lesz rossz.

A Csharp kulcsszavait (`if`, `else`, `return`, `using`, `class`) NEM lehet nevekként használni,
sem számokat legelső karakternek (pl. `123filmek`),

sem speckó karaktereket (`$:;?,%!+"'`) egyáltalán, kivétel ez alól az alsóvonás (_).

Syntax:

```
Ttípus név;           //  <-  deklarálás (elnevezés)
Ttípus név = érték;   //  <-  deklarálás és értékadás
név = érték;          //  <-  csak értékadás
név                   //  <-  érték vissza
```

Kommentek **Kommenteket** teljesen figyelmen kívül hagyja a fordítóprogram.
Ezen okból *dokumentálásra*, sorok *hatástalanítására* és az *olvashatóság* növelése érdekében megéri kommentelni.

Syntax:

```
// egysoros
      /* több-
        soros
                        komment
*/
```

[ ]:
```
int iSzám;                        // sokak által használt nevezés:    típus első␣
 ↪betűje a név elején, utána a szótagok nagyonNagyBetűsek (camelCase)
char asd;                         // :/   nem lehet tudni mi a feladata
int iSzámÉrtékkel = 16;
iSzám = 26;                       //  értékadás
// iNemLétezőVáltozó = 90;        //  <-  HIBA!    Nem volt elnevezve␣
 ↪(deklarálva)!
iSzám = 29 + 1;                   //  értékadás művelettel. Bármilyen művelettel␣
 ↪lehet értéket adni (amennyiben a típusa jó)
```

1.0.3 3. Adattípusok

Többféle adatot tudunk tárolni, és fontos hogy meg lehessen határozni őket.
Egy elnevezett változónak csak egyféle típusa lehet.

Itt egy pár beépített típus (System névtér):
- *int* = egészszám (System.Int32), range: `-2147483648 - 2147483647`

- *float* = törtszám (System.Single) lebegőponttal, range: `-3.402823e38f - 3.402823e38f` (KELL az '`f`')
  nem annyira pontos (ld. itt (docs.microsoft.com)).
- *double* = törtszám (System.Double) lebegőponttal, pontosabb, range: `-1.79769313486232e308 - 1.79769313486232e308`
- *bool* = Boolean igaz-hamis, range: `igaz`, `hamis`
- *char* = egyetlen karakter **egyes** idézőjeleken belül, pl. `'c'`
- *string* = karakterlánc avagy szöveg idézőjeleken belül, pl. `"Helló Világ!"`

[ ]:
```
int iSzámocska = 66666666;
// int iSzám = 12345678;        //  <-  HIBA!   két ugyanolyan nevű változó nem␣
 ↪létezhet
float fPi = 3.14f;              // ha float akkor kell a szám után egy 'f'!!
bool anyukádMegvolt = true;     // true = igaz; false = hamis
char cBéBetű = 'b';             // CSAK egy karaktert vesz be, különben HIBA
string sLeghosszabb =␣
 ↪"eltöredezettségmentesítőtleníttethetetlenségtelenítőtlenkedhetnétek";
```

1.0.4 4. Első C# Program

A VS előregenerál egy alap fájlt mikor új projektet kezdesz, amiben csak a program futásához szükséges sorok szerepelnek.

Ha VSCode-ot használsz, plusz egy lépés a konzolba beírni hogy `dotnet new console` (vagy amilyen típust szeretnél a *console* helyett).

> Update: Az új .NET 6.0 rendszer miatt nem *muszály* kiírni a teljes kód testét, csak a top-level eljárások elegendőek. Emiatt a default fájl is leszűkült a 'Hello World!' sorra :/

Az alap program így fog kinézni:

[ ]:
```
using System;

namespace MyApp // Note: actual namespace depends on the project name.
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Az eljárások összegyüjtött, bármennyiszer hívható kódgyűjtemények, melyek segítik a program haladását és olvashatóságát.
Egy futtatható C# program (nem könyvtár) MINDIG tartalmaz `Main()` eljárást, mivel ebben kezdődik a program **igazi** futása.

A többi részét a programnak később érdemes átvenni.

Miután futtatod, egy *parancssor* (terminál) megjelenik, amiben működik a program.

1.0.5 5. Kiírás és beolvasás, kényszerítés

A legtöbb appnak kell *bemenet* és van *kimenete*.

Kiíráshoz legeslegtöbbször a `Console.Write()` vagy `Console.WriteLine()` eljárást használjuk.
A kiírandó értéket a zárójelek belsejébe rakjuk. Az alaptípusok mindegyikét ki tudja írni. lásd:

```
[ ]: int harom = 3;
     Console.WriteLine("Hello World!");
     Console.WriteLine(6 + 5);
     Console.WriteLine(true);
     Console.WriteLine('c');
     Console.WriteLine(harom);                                  // változó értéke
     Console.WriteLine("Három, avagy {0}",harom);               // formatálás
     Console.WriteLine("Három, avagy " + harom);                // lánccsatolás␣
      ↪(concatenation)
     Console.WriteLine($"Három, avagy {harom}");                // formatálás 2.0
```

Beolvasáshoz pedig a `Console.ReadLine()` vagy `Console.ReadKey()` jut az eszünkbe.

Mivel *visszaad (return-ol)* egy értéket (és el akarjuk menteni), **változóba tároljuk**.

És vigyázat! Ez a *funkció* (eljárás) csak `string` értékként adja vissza a bemenetet.
Ezért fontos, ha számot kell varázsolni, az `int.Parse()` vagy a `Convert.ToInt()` funkció segíthet.
Viszont ha *alap* típusokkal gondolkozunk észszerűen, akkor használhatunk **típuskényszerítést**, például `double` to `int` esetében (ekkor a törtrész elveszik!).

Syntax:

```
(Túj_típus)érték            //  <-  csak akkor, ha észszerű a kényszer.
```

```
[ ]: Console.WriteLine("Mi a neved?");
     string sNév = Console.ReadLine();

     Console.WriteLine("Hány éves vagy?");
     int iKor = Convert.ToInt32(Console.ReadLine());                    // itt␣
      ↪string->int kényszer HIBA lenne

     Console.WriteLine("Szia {0}, te {1} éves vagy", sNév, iKor);
```

1.0.6 6. Operátorok és alap kulcsszavak

Egy operátor egy karakter ami programlogikai, matematikai vagy logikai feladatot lát el.
A sorrendiség követi a matematika elveit (*PEMDAS*). Továbbiakért lásd itt (docs.microsoft.com).

Aritmetikus opok Ezek adják a programozás számtani alapműveleteit.

Operátor	Karakter	Példa
Összeadás	+	x + y;
Kivonás	-	x - y;
Szorzás	*	x * y;
Osztás **(!)**	/	x / y;
Moduló/Maradék	%	x % y;
Ellentett	-	-x;

!: Az osztásnál figyelembe kell venni a két tag típusát. `int`-et `int`-tel osztva elveszik a törtrész. Ekkor érdemes `double`-be vagy `float`-ba konvertálni először.

[ ]:
```
int x = 11, y = 4;              // több változót is lehet deklarálni +␣
 ↪beállítani egy sorban!
Console.WriteLine(x+y);         // 15
Console.WriteLine(x-y);         // 7
Console.WriteLine(x*y);         // 44
Console.WriteLine(x/y);         // 2.75 DE AJJAJ csak 2-t kapunk, hol a maradék?
Console.WriteLine(x%y);         // 3    itt az egész osztásos maradék
Console.WriteLine((double)x/y); // típuskényszerítés :)
Console.WriteLine(-x);          // -11
```

Hozzárendelő opok, növelők Syntax:

```
változó = érték;        //    <-    átírás
változó += módosítás;   //    <-    módosítás
```

Az egyszerű *egyenlőségjel* (=) adja a hozzárendelést.

Az első öt számtani operátort (és a bitszintűeket is) *össze lehet vonni* a hozzárendeléssel: - += - -= - *= - /= - %=

Ezek egyenértékűek a `változó = változó X módosítás;` sorral, persze mindegyik opra külön.

A növelők pedig:

Operátor	Karakter	Példa
Növelés	++	x++; ++x;
Csökkentés	–	x–; –x;

Ezek pedig egyenértékűek a `változó = változó X 1;` sorral, persze külön-külön.

Növelésnél/Csökkentésnél egyel változik a változó értéke, viszont hogy mikor az nem mindegy.
* Ha a jel a név előtt van *(prefix)* akkor először **változik** aztán adja vissza a már változott értéket.
* Ha a jel a név mögött van *(postfix)* akkor a változó értékét **visszaadja**, majd növeli/csökkenti.

[ ]:
```
int x = 0;
x += 5;         // x = 0 + 5    = 5
x -= 1;         // x = 5 - 1    = 4
x *= 3;         // x = 4 * 3    = 12
x /= 6;         // x = 12 / 6   = 2
x %= 3;         // x = 2 % 3    = 2
Console.WriteLine(x++);     // először kiírja hogy 11; aztán növeli (x = 12)
Console.WriteLine(++x);     // először növeli (x = 13) aztán kiírja: hogy 13
Console.WriteLine(y--);     // 4        (y = 3)
Console.WriteLine(--y);     // 2        (y = 2)
```

1.0.7 7. Elágazások, logika, hurkok

Hogy a program a bemenetétől függjön, feltételes elágazásokra van szükség.
A feltétel kimenetele mindig két értékű lehet: `true` vagy `false`.

Relációs opok

Operátor	Karakter	Példa
Nagyobb	>	7 > 4 -> true
Nagyobb v. egyenlő	>=	7 >= 4 -> true
Egyenlő	==	“abc” == “cba” -> false
Kisebb v. egyenlő	<=	7 <= 4 -> false
Kisebb	<	7 < 4 -> false
Nem egyenlő	!=	“abc” != “cba” -> true

Logikai opok

Operátor	Karakter	Példa
És	&&	true && false -> false
Vagy	||	true || false -> true
Nem	!	!false -> true

Ha-más kulcsszó Syntax:

```
if (feltétel)
{
    //ha igaz...
}                   // ha van '{}' teste akkor nem kell pontosvessző
else                // ez kiegészít egy 'if' mondatot; önmagában HIBA
{
    //ha hamis...
}

if (feltétel) return 1;    // itt viszont az 'if' egysoros, test nélkül, KELL ';'

//külön operátorral:
feltétel ? /*ha igaz*/ : /*ha hamis*/;        // értéket kell visszaadnia!
    // pl.
    Console.Write(feltétel?"Jó":"Rossz");
    // u.a. mint
    if (feltétel) {Console.Write("Jó")};
    else {Console.Write("Rossz")};
```

Ha-más tagokat lehet egymás után kötni (chain-elni)

```
if (felt1) {
}
```

```
else if (felt2) {
}
else if (felt3) {
}
//...
else {
}
```

[ ]:
```
bool jólVagy = true;
if (jólVagy == true)
{
    Console.WriteLine("Egészségedre!");
}
else
{
    Console.WriteLine("Jobbulást!");
}
```

[ ]:
```
int iSzám = 16;/*<==szerkessz meg*/
if(iSzám % 2 == 0)                              // értelmezés:   ha a (szám kettes␣
 ↪maradéka) == nulla, azaz osztható kettővel
{
    Console.WriteLine("{0} osztható kettővel.", iSzám);
    if (iSzám % 4 == 0)                                          // egymásba is␣
 ↪lehet rakni elágazásokat
        Console.WriteLine("{0} néggyel is osztható.", iSzám);    // "egysoros"
}
else                                            // az else csak akkor nem dob hibát␣
 ↪egyedül, ha utána 'if' vagy '{}' van
if(iSzám == 1)
{
    Console.WriteLine("{0} == egy.", iSzám);
}
else                                            // itt '{}' van utána
{
    Console.WriteLine("Nem egy és nem osztható kettővel :(");
}
```

Mégmielőtt a hurkokat és a switchet tárgyalnánk, egy pár szót az elágazások **vég-előtti befejezéséről**:

Három kulcsszó fontos jelenleg nekünk, amivel ki lehet lépni egy elágazásból.
Ezek pedig: `continue;`, `break;`, `return x;`.

- A `continue;` (kell a ';' a végére) egy hurkon az **adott kört befejezi**, visszaugrik a hurok elejére.
- A `break;` **teljesen** kilép az adott hurokból, elágazásból (pl. if-else-ből).
- A `return érték;` pedig az adott **eljárásból lép ki**, és *visszaad* a szülőfolyamatnak egy `érték`et.

A return nem adhat a funkció visszatípusától eltérő értéket, nem lehet tudni miként értelmezi a program.

Vissza:

Ha túl sok egyenlőséget néznél meg egy változón, használd a `switch` elágazást.

Syntax:

```
switch (vált)
{                               // csak a változó kell a zárójelbe!!!
    case 1:
        // if (vált == 1)
        break;                  // FONTOS!!! A breakek nélkül átcsúszna a program nem-akart térb
    case 2:                     // amúgy lehetséges és nem hibás ha kihagyásra kerül (be lehet á
        // if (vált == 2)
        break;
    // case 1:                  // HIBA! kétszer ugyan az a case nem szerepelhet
    case 3:
        // if (vált == 3)
        break;
    //...
    case n:
        // if (vált == n)
        break;
    default:
        // else
        break;                  // ez már felesleges, de néha a compiler beszól a kihagyott brea
}
```

[ ]:
```
int x = 4;
switch (x)
{
    case 1:
        Console.WriteLine("x az egy.");
        break;
    case 2:
        Console.WriteLine("x az kettő.");
        break;
    case 3:
        Console.WriteLine("x az három.");
        break;
    case 4:
        Console.WriteLine("x az négy.");
        break;
    case 5:
        Console.WriteLine("x az öt.");
        break;
    default:
```

```
        Console.WriteLine("x valami más.");
        break;
}
```

Hurkok Hurkoknak nevezzük az ismétlődő elágazásokat, amik akár 'a végtelenségig' is futhatnak.

A `while`-hurok a legegyszerűbb fajta, a test végén visszaugrik a test elejére:

Syntax:

```
while (feltétel)
{
    // valami... amíg feltétel az igaz
}                                           // nincs ';'

while (true) Console.WriteLine("AAA");      //végtelen hurok, egysoros

do
{
    //a csináld..amíg hurok minimum egy kört lefut!
} while (feltétel);                         // pontosvessző!
```

A `for`-hurok eggyel bonyolultabb, itt három dolgot kell figyelembe venni:
- Az *init* rész a hurok rajtja előtt lefut, itt általában ideiglenes változó-elnevezés szokott lenni.
- A *feltétel* az ugyanaz, mint idáig. Igaz vagy hamis. - A *körvége* pedig minden lefutott kör végén kerül futtatásra.
Általában tömbökön keresztüli indexelt feladatokra használták, arra ma már van jobb mód.

Syntax:

```
for (/*init*/ int i = 0;feltétel;/*körvége*/ i++)        //két ';' az elválasztó
{
    //amíg a feltétel..
}
```

[ ]:
```
bool nappalVan = false;
while (nappalVan)
{
    Console.WriteLine(DateTime.Now.ToString("T"));      // ha nappal van, írjuk␣
 ↪ki az időt; a "T" itt formátumot jelöl, mint [T]ime
    if(DateTime.Now.Hour == 16) break;                  // ha délután négy van,␣
 ↪fejezzük be
}
```

[ ]:
```
string sDolog = "Hello!";
for (int i = 0;i<sDolog.Length;i++)                     // 'i' egy ideiglenes␣
 ↪iterátor (sorszám lesz), növekszik egészen sDolog hosszáig kizárólag
{
    Console.WriteLine(sDolog[i]);                       // a szöveg betűit␣
 ↪egyenként írjuk ki
```

```
}
```

1.0.8 8. Eljárások

Eljárásokat készíthetünk újra és újra kellő feladatokra.
A .NET alap könyvtáraiban ilyen *egyszerűsített*, ember-barát funkciókat találtunk már (`Console.WriteLine()`, `int.Parse()`).
Egy eljárást *bármennyiszer* hívhatunk.
A nevekre is ugyanúgy vonatkozik a *változókra vonatkozó szabályzat.*

Egy eljárás általánosan így néz ki:

```
Tvisszérték Név(T paraméter1, U paraméter2, ...)   //deklarálás
{
    //eljárás teste
}

Név(param1, param2)                                //hívás
```

ahol: - az eljárás *paraméterei* a zárójelén belül, vesszővel elválasztva vannak - az eljárás *programjai* a testén belül tartózkodnak - az eljárásnak `Tvisszérték`-et kell *visszaadnia.*

A paramétereket lehet *opcionálisnak* állítani, ha megadjuk előre a default értékét,
vagy akár lehet *névszerint* is beírni:

```
static void Duplázás(int x, int y=2, int z) {/* ... */}

// main():
int iRes = Duplázás(z: 7, x: 18);        //a paraméterek:    x = 18; y = 2; z = 7;
```

A C# három féleképpen tudja **passzolni az adatokat**: - mint *érték* - mint *referencia* - mint *kimenet*

Defaultként **értékként** adja tovább a paramétereket (nem a változókat adja, hanem az értékeiket), de a `ref` kulcsszóval lehet **referenciaként** az objektumokat bevinni az eljárásokba.
Kimenetként pedig az `out` kulcsszóval lehet, így a paraméterek inkább *kiadnak* adatot mintsem betesznek.

[ ]:
```
static void Sqr(int x) => (x*x);
static void SqrRef(ref int x) {x = x * x;}
static void GetThis(out int x, out int y) {x = 5; y = 10;}

int iEredet = 3;
Sqr(iEredet);                                    //      csak az érték kerül␣
 ↪be a funkcióba
Console.WriteLine(iEredet);                      // 3;   az eredeti␣
 ↪változóban nem módosul

SqrRef(ref iEredet);                             //      most maga az 'x'␣
 ↪változó került be referenciaként
```

```
Console.WriteLine(iEredet);                                    // 9;   az eredeti␣
 ↪változóban

int a, b;                                                      // a kimeneti paramétereket␣
 ↪nem muszály inicializálni (értéket adni neki), hiszen úgysem használjuk
GetThis(out a, out b);                                         // a = 5; b = 10 mostmár
Console.WriteLine("a = {0}, b = {1}", a, b);
```

A **rekurzió** fontos feladata lehet egy eljárásnak. Rekurzív jelentése: önmagát hívó.
Példának a faktoriálist adnám: `4! = 4 * 3 * 2 * 1 = 24`

Ezt megvalósítani nem nehéz, csak *önmagát hívatni kell*, és kell írni egy *kilépési feltételt.*
(vagy örökké fut a program, újabb és újabb forkot nyitva = amatőr fork bomb)

[ ]:
```
static int Fact(int szam)
{
    if (szam <= 1) return 1;
    return szam * Fact(szam - 1);           // vagy egysorosan:      return␣
 ↪(szam <= 1) ? 1 : szam * Fact(szam - 1);
}

int iSzám = 4;
Console.WriteLine(Fact(iSzám));
```

Túltöltésnek nevezik azt a jelenséget, ahol egy eljárásnak több paraméteres változata van. Ugyan az a név, de más paraméterekkel, ha **csak** a *visszérték* különbözik az HIBA!
Ez akkor hasznos, ha egy általános funkciót *több típusra* szeretnénk alkalmazni.

[ ]:
```
static void Kiír(int a) {Console.WriteLine("Érték: " + a);}         //szöveg␣
 ↪csatolása '+'-szal
static void Kiír(double a) {Console.WriteLine("Érték: " + a);}
static void Kiír(string cím, double a) {Console.WriteLine(cím + a);}

Kiír(15);
Kiír(7.13);
Kiír("Ez itt ", 9.9999999);
```

Egyes operátorokat *osztályokban* túl lehet tölteni, az `operator` kulcsszavat beillesztve az op elé.
Ezeket lehet túltölteni: *arithmetikai* opok (`+`, `-`, `*`, `/`), *relációs* opok (`<`, `>`, `==`, `!=`, `x^y`), bitszintű opok (`<<`, `>>`, `&`, `|`);
azaz az elsődleges és másodlagos opok nagyrésze.
Viszont egyes opokat tilos és lehetetlen túltölteni, mint: `x = y`, `x.y`, `c ? t : f`, `new`, `switch`, `delegate`, és sok más (lásd Túltölthető operátorok (docs.microsoft.com))

```
class Valami
{
    public int operator+ (int param1, int param2) {/* dolgok */} // azaz: param1 + param2
    public int operator- (int param1) {}                         // -param1
    public int operator- (int param1, int param2) {}             // param1-param2
```

```
}
```

1.0.9 9. Tömbök és Stringek

Tömbök Hogyha több, *egyfajtájú* változót akarnánk értelmezni, szükségünk lenne egy **tömbre**.

Az `Array` (tömb) osztály egytípusú értékeket csoportosít *egy név* alatt.
Például négy darab egészet:

index:	[0]	[1]	[2]	[3]
arr[]	16	2916	9999	-414

Syntax:

```
típus[] név = {,,,};                        // a típus lesz a tömb részeinek típusa is!
int[] dolog = new int {1, 2, 3, 4};         // <- `[]` jelzi a tömbösítést, pédányosítjuk (new)
                                            //    és '{}' pedig megadja a konkrét értékeit

double[] gyorsNév = {16.16, 12.12};         // <- gyors syntax (nem kell 'new int' kulcsszó)
string[] másikNév = new string[hossz];      // <- itt a hossz adja meg az üres tömb hosszát

dolog[1] = 15;                              // <- az első indexű (második) elemet módosítjuk;
                                            // az első elem nulla indexű!

// név[5]                                   // <- HIBA! az 5ös indexű (hatodik!) nem létező!
```

Tömbökkel dolgozni sokféleképpen lehet, de a legalapabb módja az **iterálás** (minden egyes elemen keresztüli feldolgozás).
Ezt `for`-looppal tudjuk egyszerűen csinálni.

```
int[] a = {1,2,3,4,5,6,7};              // a tömbünk (7 hosszú)

for (int i = 0; i < 7; i++)             // létrehozunk egy 'i' ideiglenes változót ('i'terátort)
{                                       // majd azt használjuk indexszámként, így végigmegyünk a
    Console.WriteLine(a[i]);            // pl. kiírjuk a tömb részeit egyenként
}                                       // FONTOS! ha túlmegy az iterátor a tömb hosszán, az HIB
```

Ha index-iterátort nem akarunk használni, lehet `foreach`-hurokkal is.

```
foreach (int elem in a)                 // itt NEM kell '[]'; plusz egy ideiglenes változót neve
{                                       // és abba vesszük a tömb egyenkénti értékeit is (tárgy-
    Console.WriteLine(elem);            // így elkerüljük a tömb szintaxisait, és a túlcsordulás
}
```

[ ]:
```
int[] intTömb = {10,16,21,35,47,55};
int iSzumma = 0;

foreach (var most in intTömb)
```

```
{
    iSzumma += most;                // összegszámítás
}

Console.WriteLine(iSzumma);
```

Tömb a tömbben? Egy tömb *több dimenziós* is lehet, és lehetnek a *tagjai tömbök is*, hiszen a 'tömb' nem más mint egy tárgy, egy osztály-típus.

```
int[ , ] kétdim = new int[3,4];                 // kétdimenziós tömb int-ekből
int[,] valami = { {1, 2}, {12, 9}, {5, 6} };    // u.a. csak értékmegadással
```

Ez így nézne ki:

	Oszlop1	Oszlop2	Oszlop3	Oszlop4
Sor1	[0, 0]	[0, 1]	[0, 2]	[0, 3]
Sor2	[1, 0]	[1, 1]	[1, 2]	[1, 3]
Sor3	[2, 0]	[2, 1]	[2, 2]	[2, 3]

Ne féljünk *több*, *egybeágyazott* `for` vagy `foreach` loopot használni.

Egy másik több-tömbös megoldás is létezik, ezek az **egyeletlen tömbök**.
Ezekben a fő-tömb tagjai ugyanúgy tömbök.

```
int[][] egyeletlen = new int[][]    // két '[]'-t kell rakni!
{                                   // és külön kell inicializálni (beállítani) ami itt példán
    new int[] {1,2,3,4},
    new int[] {99, 98},
    new int[] {10}
}
```

A különbség a többdimenziós tömbök és az egyelentlen tömbök közt a **memóriahasználat**.

- Egy *többdimenziós tömb* egy **megszakítatlan** memóriatér (egy *mátrix* basically) aminek *ugyanannyi oszlopa van minden sorban.*
- Egy *egyeletlen tömb* pedig **tömböknek a tömbje**, így a memória *tömbönként eltérhet.*

Fontos Tömb tulajdonságok Egypár fontos dolog az `Array` osztályból:

```
int[] arr = {2, 4, 7};
```

- arr.`Length` megmondja a tömb hosszát (elemei számát).
  - fontos lehet egy `for`-loopban!
- arr.`Rank` pedig a dimenzióinak számát.
- arr.`Max()` a legnagyobb elemet adja vissza,
- arr.`Min()` a legkisebbet,
- arr.`Sum()` az összegüket.

Egypár statikus (csak az Array osztályból hívható) eljárás: - `Array.Sort(arr)` visszaad egy új, rendezett tömböt (amit el kell még menteni!!), - `Array.Reverse(arr)` pedig egy fordított sorrendűt. - `Array.ConvertAll(arr,delegátus())` visszaad egy *delegátus*-eljárás alapján átírt tömböt.

Stringek Könnyebb egy karakterláncra úgy gondolni, hogy az egy karakterekből álló tömb. Igazából a C#-ban egy objektum már.

[ ]:
```
string asdfgh = "hello";
string qwertz = new String(new char[] {'h', 'e', 'l', 'l', 'o', '\x00'});
/*  ugyanaz a két sor  */                                    /* ↑ ez egy␣
 ↪kilépési karakter */
// a kilépési karakterek '\'-el kezdődnek, és kicserélődnek futáskor
// itt a \x00 egy null-karakterre cserélődik, ami a stringek híres lezáró␣
 ↪karaktere
Console.WriteLine(asdfgh);
Console.WriteLine(qwertz);
```

Fontos megjegyezni hogy az ember-olvasható karaktereken kívül vannak *láthatatlan*, de annál fontosabb karakterek. Ezeket **kontroll-karaktereknek** hívjuk.

Közéjük tartozik például az *újsor*, *kocsivissza*, *null*, *csengő*, *tab*, etc. Hogy ezeket *láthatóvá*, *leírhatóvá* varázsoljuk, létrejöttek az **escape kódok** (escape sequence),
használatuk: egy **visszaperjel** és utána a jellemző kód (előző példák: `\n`, `\r`, `\0`, `\a`, `\t`, …)
Ha egy literális visszaperjel kell, csak írj duplát (`\\`), így nem lesz értelmezve véletlenül.

További infó: C# Stringek (docs.microsoft.com), Escape sorozatok (wikipedia.org), és Kontroll karakter (wikipedia.org)

Amikor egy `"szöveget"` írsz, akkor végülis egy String osztályú példányt hozol létre.
Ezért a stringeknek is van pár fontos tulajdonságuk:

```
string st = "halihó";         // u.a. mint `string st = new string "halihó";`
Console.WriteLine(st[2]);     // "l"
```

- st.`Length` a hosszát adja vissza.
- st.`Split(k)` visszaad egy karakteren szétválasztott string-tömböt (!).
- st.`IndexOf(a)` a keresett érték legelső előfordulásának indexét adja vissza.
- st.`Insert(1, "be")` beilleszt egy szöveget a megadott indexnél kezdve (visszaadja csak!).
- st.`Remove(2)` kitöröl minden karaktert indextől kezdve.
- st.`Replace("régi", "új")` kicseréli a *régi* szakaszt *új*ra.
- st.`Substring(i, x)` kivág egy x hosszúságú részletet i-től.
- st.`Contains("b")` megnézi hogy benne van-e egy részlet a szövegben.

Statikus eljárások: - `String.Concat(a,b)` csinál egy összevont szöveget (u.a. mint stringek közt a '+' operátor). - `String.Equals(a,b)` ellenőrzi az azonosságot.

2 Folytatás a `programozas_gyorstalpalo_nehezebbik` című fájlban

Olyan érdekes témákkal mint:

- névterek, osztályok, tárgyak
- inheritancia és polimorfizmus (ami két fancy szó az *öröklésre* és a *sokoldalúságra*)
- struktúrák, enumok
- hibák (exceptionök)
- fájlok

- általánosítás (*fuck yes all my homies love generalizing*)
- delegátok és anonim (lambda) eljárások
- és még több érdekesség…
