

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.

Fontos: A **sorbeolvasást** és a **névtérdeklarálást** leszámítva minden más működik.

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!)

Alapszint

1. C# avagy Csharp

- OOP nyelv = Tárgy-orientált = "Mindent generalizálni, példányosítani 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.).

2. Változók, Kommentek és Whitespaces

Mégmielőtt tárgyalnánk a legkönnyebb dolgokat, több dolgot leszögeznek:

- 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 **blokk** (block) pedig *nulla vagy több állítás* csoportja egy `{}` -n belül

Fontos: C#-ban minden állítás helyére lehet blokkot írni (és fordítva)

Változók

Változóknak nevezzük azokat a tárgyakat amikben értéket tudunk tárolni.

Minden változót legalább egyszer (*legelőször*) el kell nevezni (**deklarálás**) használatuk előtt.

Próbálj **jól leíró** neveket használni (pl. `iSzám`, `sKeresztnev`, `tPerc`), és az ideiglenes változóknak adni a random neveket (pl. `x`, `y`, `i`, `temp`, `elem`).

Ha lehet kerülj az angol ábécén kívüli karaktereket, nem lehet tudni mikor lesz rossz.

A Csharp kulcsszavait (`if`, `else`, `return`, `using`, `class`) NEM lehet névként 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 lekérés
```

```
In [ ]: int iSzám;           // sokak által használt nevezés: típus első betűje a név elején, utána a szótagok nagyonNag
        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)
```

Kommentek és whitespace

A **kommenteket** és a **whitespace** teljesen figyelmen kívül hagyja a fordítóprogram, nem avatkoznak bele a futásba.

A **whitespace** egy gyűjtőfogalom az összes, *szemmel láthatatlan* szöveget alkotó **térelemre**.

Ide tartozik a **szóköz** (*space, innen a név*), **újsor** és a **tabulátor**.

A C# NEM **whitespace-érzékeny**, tehát a program működése **nem változik**, ha a kifejezések között *whitespace* van.

Ezen okból a **kommentek** és a **whitespace** dokumentálásra, sorok *hatástalanítására* és az *olvashatóság* növelésére alkalmas.

Syntax:

```
// egysoros komment
```

```
/* több-  
    soros  
    komment  
eleje: /*      vége: */
```

```
static      void      Main()  
// ↓ (felesleges) whitespace ↑  
{  
    Console.WriteLine(    // <- whitespace-el olvashatóbb  
        "Helló világ!"  
    );  
}
```

3. Adattípusok







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

Egyes primitív értékeket mi **szószerint** is be tudunk gépelni a programba,
ezeket nevezzük **literáloknak** (szó jelentéséből fakadóan).

A következő listában alaptípusok lesznek, ezek értékeit literálként meg tudjuk adni:

( : megjegyzés;  : értékei)

- *int* = egészszám (*System.Int32*)
 Range: -2147483648 -tól 2147483647 -ig
- *decimal* = tizedestört gyűjtőfogalom (*System.Decimal*)
- *float* = törtszám lebegőponttal (*System.Single*)
 Range: -3.402823e38f -tól 3.402823e38f -ig (KELL az ' f ')
 kevésbé pontos (ld. [itt \(docs.microsoft.com\)](https://docs.microsoft.com)).
- *double* = törtszám lebegőponttal, pontosabb (*System.Double*)
 Range: -1.79769313486232e308 -tól 1.79769313486232e308 -ig ( .eX a tizes hatványt jelenti)
- *bool* = Boolean igaz-hamis
 Csak `true` (igaz) vagy `false` (hamis)
- *char* = egyetlen karakter **felsővesszőkön** belül
 pl. `'c'`
- *string* = karakterlánc avagy szöveg **idézőjeleken** belül
 pl. `"Helló Világ!"`
- *var* = futáskor értelmezett típus
 CSAK akkor használható, ha egyértelműen **egyfajta típus** lesz belőle!
- *void* = értéktelen változó
 CSAK egy variábilis lehet ilyen típusú (azaz nem ad vissza semmit)

```
In [ ]: int iSzámocska = 66666666;  
        // int iSzámocska = 12345678;    // <- HIBA! kétszer ne deklaráljunk  
        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 = "eltöredezettségmentesítőtlenítettethetetlenítőtlenkedhetnétek";
```

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!");  
        }  
    }  
}
```

Futtatás

Az **eljárások** összegyűjtött, bármennyiszer hívható kódgyűjtemények, melyek segítik a program haladását és olvashatóságát.

Egy futtatható C# programnak (nem könyvtár) **tartalmaznia kell** `Main()` eljárást, mivel ebben kezdődik a program **igazi** futása. (ezt nevezik *entry point*-nak)

A többi részét a programnak később érdemes átvenni.

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

5. Kiírás és beolvasás, kényszerítés

Kiírás

A legtöbb appnak van **bemenete** és **kimenete**.

Kiíráshoz legeslegtöbbször a `Console.Write()` vagy `Console.WriteLine()` eljárást használjuk.

A kiírandó értéket a **zárójelek belsejébe** rakjuk.

```
In [ ]: int három = 3;           // egy példa változó, deklarálva
Console.WriteLine("Hello World!"); // string literal
Console.WriteLine(6 + 5);         // állítás értékét a program kiszámolja
Console.WriteLine(true);         // bool
Console.WriteLine('c');          // char literal
Console.WriteLine(három);        // változó értéke
Console.WriteLine("Három, avagy {0}", három); // formátálás
Console.WriteLine("Három, avagy " + három);   // láncsatolás (concatenation)
Console.WriteLine($"Három, avagy {három}");    // formátálás 2.0
```

Beolvasás

Beolvasáshoz pedig a `Console.ReadLine()` -t vagy `Console.ReadKey()` -t használjuk.

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

És vigyázat! Ez a *funkció* (eljárás) csak `string` értékként adja vissza a bemenetet.

Ezért fontos, ha számot kell belőle varázsolni, az `int.Parse()` eljárás segíthet.

Típuskényszerítés

Ha *alap* típusokkal gondolkozunk észszerűen, akkor használhatunk **típuskényszerítést**, például `double` → `int` esetében (ekkor a törtrész elveszik!).

Syntax:

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

```
In [ ]: Console.WriteLine("Mi a neved?");
string sNév = Console.ReadLine(); // Az interaktív notebookban a ReadLine() nem működik!!!

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);

In [ ]: Console.WriteLine("A 'c' karakter kódja: {0}", (int)'c'); // char->int; ASCII kód
Console.WriteLine("A '3.1415' egész számként: {0}", (int)3.1415); // decimal->int; a törtrész elvész
```

6. Operátorok

Egy operátor egy (vagy több) **karakter** ami *programlogikai, matematikai* vagy *logikai* feladatot lát el.

A **sorrendiség** követi a matematika elveit (*PEMDAS*). Továbbiakért lásd: (docs.microsoft.com).

A sorrend felülírható `()` zárójelekkel.

Aritmetikus opok

Ezek adják a programozás számtani alapl műveleteit.

Megnevezés	Operátor	Példa
Összeadás	+	<code>x + y;</code>
Kivonás	-	<code>x - y;</code>
Szorzás	*	<code>x * y;</code>

Megnevezés	Operátor	Példa
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.

```
In [ ]: 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);   // 2.75    típuskényszerítéssel
Console.WriteLine(-x);            // -11
```

Hozzárendelő opok

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 Δ módosítás;` utasítással, persze mindegyik opra külön.

Iteratív opok

Az iteratív operátorok csak eggyel változtatják az értéket.

Megnevezés	Operátor	Példa
Növelés	++	<code>x++;</code> <code>++x;</code>
Csökkentés	--	<code>x--;</code> <code>--x;</code>

Ezek pedig egyenértékűek a `változó = változó Δ 1;` utasítással, persze külön-külön.

Nem mindegy, hogy *milyen sorrendben* vannak az iteratív opok:

- Ha a jel a név **előtt** van (*prefix*), akkor először **változtat** aztán adja vissza a már változott értéket.
- Ha a jel a név **mögött** van (*postfix*), akkor először a változó értékét visszaadja, majd **változtatja**.

```
In [ ]: 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)
```

Relációs opok

Két érték **viszonyát** (*relációját*) fejezik ki, pont mint a matematikában.

Egy `bool` -értéket adnak vissza (*igaz-hamisat*), ezért lehet **feltételként** használni őket.

Megnevezés	Operátor	Példa
Nagyobb	>	7 > 4 → true
Nagyobb v. egyenlő	>=	7 >= 4 → true
Egyenlő	==	"abc" == "def" → false
Kisebb v. egyenlő	<=	7 <= 4 → false
Kisebb	<	7 < 4 → false
Nem egyenlő	!=	"abc" != "def" → true

Logikai opok

Két `bool` -értéket (*igaz-hamisat*) értéket **kötnek össze** logikailag.

Funkciójuk megegyezik hétköznapi szóhasználatukkal.

Megnevezés	Operátor	Példa
És	&&	true && false → false
Vagy		true false → true
Nem	!	!false → true

7. Logika, elágazások, ciklusok

Feltételes elágazásokra van szükség, hogy a program egyes értékektől függjön.

Egy feltétel értéke biztosan `true` vagy `false`, tehát biztosan `bool` értékű.

`if-else` (Ha-Más)

Az elágazások legalapvetőbb állítása. Hogyha a feltétel teljesül, a blokk lefut, ellenkező esetben nem.

Syntax:

```
if (feltétel)
{
    // ha van '{}' teste akkor nem kell pontosvessző
    //ha igaz...
}

else if(feltétel)    // ez kiegészít egy 'if' mondatot; önmagában HIBA
{
    //ha az előző(k) hamis(ak), de ez igaz...
}

else                // ez kiegészít egy 'if' mondatot; önmagában HIBA
{
    //ha hamis (és az ezelőttiek is mind hamisak)...
}

if (feltétel) return 1;    // itt az 'if' egysoros blokk-nélküli, KELL ';'
Külön operátorral tudunk egy feltételes kifejezést csinálni:
```

Fun fact: Ez az egyetlen három-paraméteres operátor, ezért *ternary* op-nak is hívják.

```
feltétel ? /*ha igaz*/ : /*ha hamis*/ ;    // ez egy KIFEJEZÉS azaz ÉRTÉKET AD VISSZA!
// pl.
Console.Write(feltétel?"Jó":"Rossz");
// u.a. mint
if (feltétel) Console.Write("Jó");
else {Console.Write("Rossz");}
```

A Ha-Más logikákat lehet egymás után kötni (chain-elni)

```
if (felt1) {
}
else if (felt2) {
}
else if (felt3) {
}
//...
```

```
else {  
}
```

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

```
In [ ]: 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} négygyel is osztható.", iSzám); // "egysoros"  
}  
else if(iSzám == 1)    // az else csak akkor nem dob hibát egyedül, ha utána 'if' vagy '{}' van  
{  
    Console.WriteLine("{0} == egy.", iSzám);  
}  
else                  // itt '{}' van utána  
{  
    Console.WriteLine("Nem egy és nem osztható kettővel");  
}
```

Elágazások megszakítása

Mégmielőtt a **switchet** és a **hurkokat** tárgyalnánk, egy pár szót az elágazások **megszakításairól**:

Három kulcsszó létezik C#-ban, amivel **ki lehet lépni** egy *elágazásból*.

Ezek pedig: `continue;` , `break;` , `return x;` .

- A `continue;` egy ciklusban az **adott kört nem fejezi be**, azonnal visszaugrik a ciklus **elejére**.
- A `break;` **teljesen kilép** az adott ciklusból, elágazásból.

- A `return` érték; pedig az adott **eljárásból lép ki**, és *visszaad* a szülőfolyamatnak egy `érték` et.
- A `return` CSAK az eljárás **visszatérési érték** (*visszérték*) típusban adhat vissza értéket!

Switch

Ha túl sok **egyenlőséget** néznél meg egy változóra nézve, használd a `switch` elágazást.

A `switch` egy kifejezést **különbféle értékekkel** vet össze, és aszerint végez műveleteket.

Syntax:

```
switch (vált)
{
    case 1:
        // if (vált == 1)
        break;           // FONTOS!!!
    case 2:
        // else if (vált == 2)
        break;
    case 3:
        // else if (vált == 3)
        break;
    //...
    case n:
        // else if (vált == n)
        break;
    default:
        // else
        break;
}
```

Tehát: sok, egy kifejezésre értett `if-else` logikát könnyebben felírhatunk egy `switch` sok esetével (`case x:`).

A `default` eset akkor megy végbe, ha egyik eset sem volt igaz.

Fontos: A `break;` -ek nélkül átcsúszna a program **másik esetekbe** (a compiler ezt hibaként jelzi), de elfogadható és működőképes megközelítés.

```
In [ ]: 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;
}
```

Ciklusok

Ciklusoknak nevezzük az ismétlődő elágazásokat, amik *egy feltételen* alapszanak.

Ciklusokból háromféle van: `while`, `for`, és `foreach`.

`while` (amíg) és `do..while` (csináld..amíg)

A `while` -ciklus a legegyszerűbb fajta: **ellenőrzi** a feltételt; ha igaz, akkor lefut a blokk, majd *kezdi előlről*.

Hogyha *feltétel-ellenőrzést* a **végén** szeretnénk tenni, használjuk a `do..while` formát

Syntax:

```
while (feltétel)
{
```

```

    // valami... amíg feltétel az igaz
}                                     // nincs ';'

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

```

In []:

```

int max = 10;
int a = 0;
while (a <= max)
{
    Console.WriteLine(a);
    a = a + 1;
}

do
{
    a = a - 1;
    Console.WriteLine(a);
} while (a >= 0);

```

for (iteratív ciklus)

A **for** -ciklus eggyel bonyolultabb, itt két extra dolgot kell figyelembe venni:

- Az **init** rész a *ciklus kezdete* **előtt** lefut, itt általában *ideiglenes változó-deklarálás* szokott lenni.
- A **körvége** pedig minden lefutott kör végén kerül futtatásra.

Általában tömbökön keresztüli **indexelt** (sorszám-szükséges) feladatokra használjuk.

Syntax:

```

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

```

```
In [ ]: string sDolog = "Hello!";
        for (int i = 0; i < sDolog.Length; i++)           // 'i' egy ideiglenes iterátor (sorszám lesz), növekszik egészen sDolog.Length-1-ig
        {
            Console.WriteLine(sDolog[i]);               // a szöveg betűit egyenként írjuk ki
        }
```

foreach (tárgyiteratív ciklus)

A `foreach` a `for`-ciklus továbbfejlesztett, általánosabb verziója.

Ezt akkor használjuk, ha egy tömb összes elemét akarjuk hasznosítani (sorszám nélkül).

A "két" paramétere, `in` kulcsszóval elválasztva:

- Egy ideiglenes változó deklarálása (`T típus ideigl`)

 Típusra használhatjuk a `var` szót (a program eldönti futás közben, melyik típus legyen)

- A tömb, aminek az elemeit vizsgáljuk

Syntax:

```
foreach(var ideigl in tömb)
{
    // utasítások az 'ideigl' felhasználásával
}
```

```
In [ ]: string[] sorok = {"Az első sor", "A második sor"};
        foreach(var elem in sorok)                     // értsd: a 'sorok' tömb összes elemére...
        {
            Console.WriteLine(elem);                   // egyik elem kiírása
        }
```

8. Eljárások

Eljárásokat készíthetünk újra és újra szükséges feladatokra.

A .NET alap könyvtáraiban ilyen egyszerűsített, ember-barát funkciókat találtunk már (pl: `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 *paramétere*i a zárójelén belül, vesszővel elválasztva vannak
- az eljárás *utasításai* a blokkján belül tartózkodnak
- az eljárásnak **Tvisszérték** -et kell *visszaadnia* visszatérésekor.

Hogyha a *visszérték típusa* **void**, akkor nem kell visszaadnia semmit (**nem kell** `return;`).

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;
```

Pass by value, reference, output

A C# három féleképpen tudja **átadni az adatokat**:

- mint *érték*
- mint *referencia*
- mint *kimenet*

Alapból **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.

```
In [ ]: 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),
GetThis(out a, out b); // a = 5; b = 10 mostmár
Console.WriteLine("a = {0}, b = {1}", a, b);
```

Rekurzió

A **rekurzió** fontos feladata lehet egy eljárásnak. Rekurzív jelentése: önmagát hívó.

Példának a faktoriális definíciója: $4! = 4 * 3 * 2 * 1 = 24$

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)

```
In [ ]: 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));
```

Túltöltés

Túltöltésnek nevezik azt a jelenséget, ahol egy eljárásnak több paraméteres változata van.

Ugyan az a név, de **különböző** paraméterekkel!

Ez akkor hasznos, ha egy általános funkciót *több típusra* szeretnénk alkalmazni.

Egyes **operátorokat** osztályokban túl lehet tölteni, az `operator` kulcsszavat beillesztve az op elé.

Ezeket lehet túltölteni: *aritmetikai* 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 \(docs.microsoft.com\)](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/operators-and-operators-overloading))

```
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
}
```

```
In [ ]: 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);
```

9. Tömbök és Stringek

Tömbök

Hogyha több, *egyfajtajú* változót akarnánk csoportosítani, szükségünk van **tömbökre**.

Az `Array` (tömb) osztály **egytípusú** értékeket csoportosít *egy név* alatt.

Az adatok egymás után, **sorban lesznek**, így mindegyik adathoz tartozik *egy sorszám (index)* **nullától kezdve!** Ebből kifolyólag a sorszámok **soha nem érik el** a tömb elemszámát (*hosszát*).

Például négy darab egészest csoportosítva:

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

Deklarálhatjuk kétféleképpen: **adatokkal**, vagy **hosszal**.

Syntax:

```
típus[] név = { /*adatok*/ };           // adatokkal deklarálás
típus[] név = new típus[n];           // 'n' hosszú üres tömb deklarálás
```

```
név[x]                                     // előhívása:
                                           // x indexű elem a név tömbből
```

```
név.Length                               // tömb hosszának lekérése
```

Tömbökkel dolgozni sokféleképpen lehet, de a legalapabb módja az **iterálás** (minden egyes elemen keresztüli feldolgozás).

Ezt **for**-looppal tudjuk egyszerűen csinálni, ahol feltétel, hogy

az **ideiglenes** változónk (itt: "iterátor") kevesebb legyen, mint a tömb hossza (**tömb.Length**).

Fontos: az iterátor változó értéke ne menjen túl a tömb hosszán (mivel nem létező elemeket kérne le) ezt nevezik túlcsoordulásnak, ekkor HIBA

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

for (int i = 0; i < a.Length; i++)      // létrehozunk egy 'i' ideiglenes változót ('i' terátort)
{                                         // majd azt használjuk indexszámként,
    Console.WriteLine(a[i]);           // így végigmegyünk a tömb minden elemén
}
```

Ha index-iterátort nem akarunk használni, lehet **foreach**-ciklussal is.

```
foreach (int elem in a)                 // itt NEM kell '[';
{
    Console.WriteLine(elem);
}
```

```
In [ ]: 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' általánosított; bármilyen típust be tud fogadni.

```
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 az **értelmezésükre**.

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 példányosítással történik
    new int[] {1,2,3,4},
    new int[] {99, 98},
    new int[] {10}
}
```

A különbség a többdimenziós tömbök és az egyeletlen 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* (pl. eltérő hossz).

Fontos Array tulajdonságok

Egypár fontos dolog az `Array` osztályból:

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

- `arr.Length` megmondja a tömb hosszát (elemei számát).
 - fontos 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 , elj)` visszaad egy eljárás alapján átírt tömböt, ahol az elemek az eljárás első paraméterébe illesztődnek be.

Stringek

A megértés kedvéért könnyebb egy **stringre** (*karakterláncra*) úgy gondolni, hogy az csupán egy *karakterekből álló tömb*; bár C#-ban már egy *általánosított osztály*.

```
In [ ]: 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);
```

Kontroll karakterek, Escape kódok

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 *láthatóak*, *leírhatóak* legyenek, 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 írd duplát (`\\`), így nem lesz értelmezve véletlenül.

További infó: [C# Stringek \(docs.microsoft.com\)](https://docs.microsoft.com/en-us/dotnet/csharp/string), [Escape sorozatok \(wikipedia.org\)](https://en.wikipedia.org/wiki/Escape_sequences), és [Kontroll karakter \(wikipedia.org\)](https://en.wikipedia.org/wiki/Control_characters)

Fontos String tulajdonságok

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

Statikus eljárások:

- `String.Concat(a , b)` csinál egy összevont szöveget (u.a. mint stringek közt a '+' operátor). (*string*)

- `String.Equals(a , b)` ellenőrzi az azonosságot. (*bool*)

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...

Licensz

© Daniel Adam Farkas 2022



Dieses Werk ist lizenziert unter einer [Creative Commons Namensnennung - Weitergabe unter gleichen Bedingungen 4.0 International Lizenz](#).



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).