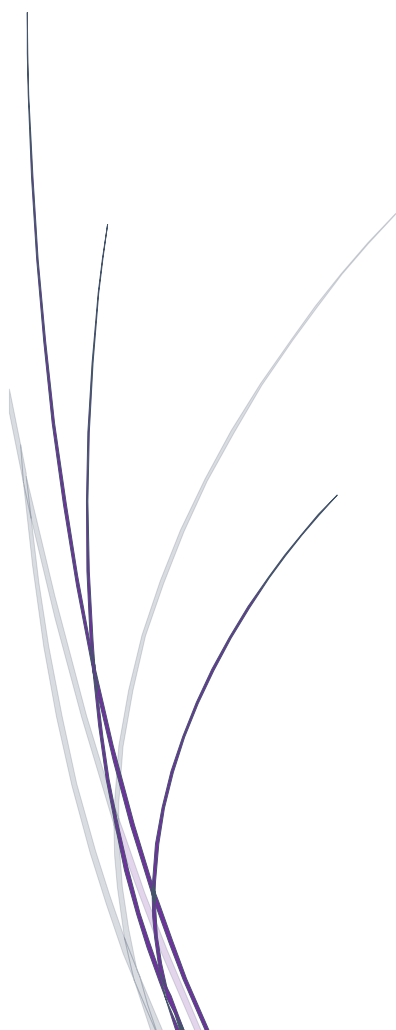




C# Programozás - Jegyzet

Karsa Zoltán



*A könyv ingyenesen terjeszthető, szabadon
felhasználható, ha feltüntetik a szerzőt!*

Tartalomjegyzék

I. Alapok

oldal

➤ Alapok, első programunk: Hello World!	2.
➤ Alapszintű adatserkezetek: változók, típusok	3.
➤ Precedenciai rangsor	5.
➤ Algoritmust leíró eszközök: folyamatábra, struktogram, mondatszerű leírás ..	5.
➤ Adatbekérés a konzolról	6.
➤ Konvertálás, castolás	7.
➤ Szelekciók – elágazások:	
▪ If-else, && és az 	7.
▪ Switch	9.
➤ Iterációk – ciklusok:	
▪ While – előtesztelő ciklus	10.
▪ Do-while – hátulatesztelő ciklus	10.
▪ For - számlálóvezérelt ciklus	11.
➤ Alap C# függvények és metódusok, Math osztály függvényei	12-14.
➤ Randomolás	14.
➤ Gyakorlás	15.
➤ Összetett adatszerkezetek: tömbök, konstansok	
1, 2 dimenziós, mutatóvektorok, foreach	16-18.
➤ String, mint tömb, metódusai, escape karakterek	18.
➤ Példaprogramok a tömbökre	20.
➤ TryParse, avagy ellenőrzött bekérés	20.
➤ Billentyűzetkezelés, ReadKey	21.
➤ Alprogramok: eljárások, függvények, paraméterátadási módok	
túlterhelés, dokumentálás, rekurzió	22-26.
➤ Felsorolás típus – Enum	27.
➤ Hibakezelés	28.
➤ Generikusok, kollekciók	
▪ Listák – List, listák metódusai	30.
▪ Referencia másolás, összetett listák	32.
▪ Láncolt listák – LinkedList, csomópontok, metódusaik	33.
▪ Szótár – Dictionary	34.
▪ Verem – Stack	35.
▪ Sor – Queue	35.
▪ Halmazok – HashSet metódusok	36.
▪ ArrayList – Sorlista	37.
➤ Alapszintű fájlkezelés: írás (StreamWriter) és olvasás (StreamReader)	37.
➤ Struktúrák: mezők, konstruktor, metódusok	38-39.
➤ Egyéb operátorok, bitműveletek, var és a goto	40-41.

II. Objektum orientált programozás

➤ Object Oriented Programming	42.
➤ Láthatósági, hozzáférési szintek (public, protected, private)	42.
➤ Konstruktor, Destruktor	43.
➤ Példány és osztálymetódus, statikusmezők	43-44.
➤ Jellemzők, Tulajdonságok	44-45.
➤ Öröklődés	
▪ Alapok	46-48.
▪ Az ős metódusainak elfeledése, újrainplementálása (virtual, new...) ..	48-49.
▪ Polimorfizmus, késői- és korai-kötés	50.
➤ Beágyazott osztályok	50.
➤ Parciális osztályok	51.

C# Programozás

I. rész - Alapok

Alapok

compiler, szintaktikai-, szemantikai hibák, program.cs

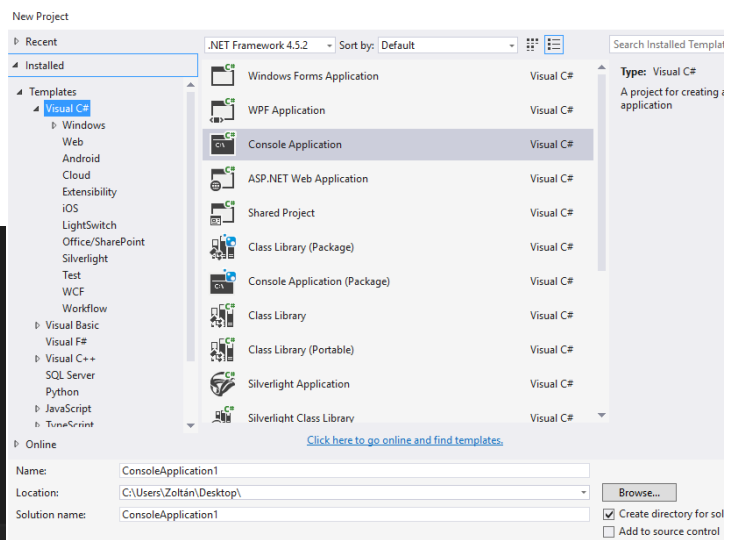
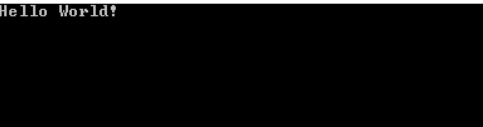
Aki még nem tanult semmiféle programozási nyelvet annak ajánlom a most következő sorokat: Régen a számítógépeknél a programokat gépi kódban írták, ami azt jelentette, hogy a programozóknak 16-os számrendszerben kellett a programokat megírni, ami így elgondolva nem valami egyszerű. De később bevezették a programozási nyelveket és egy u. fordító (compiler) programot hoztak létre, ami a nyelvet lefordította gépi byte kódokra. Most mi a Visual Studiot fogjuk használni (de lehet mást is, pl.: sima jegyzettömb is megfelelő, aminek a programnyelvét lefordítjuk egy compilerrel). C#-ban egy program.cs nevezetű fájlal kezdődik az egész, amit később lefordítunk. Vannak szintaktikai és szemantikai hibák. Az esetben, ha a programozási nyelv "nyelvtanában" hibáztunk valamit (pl. sor végén ; kell) szintaktikai hibáról beszélünk, és a compiler nem fordítja le programkódot, hanem valamilyen hibát dob. Akkor beszélünk szemantikai hibáról, ha a programunk hiba nélkül lefordul, de működése közben valami nem stimmel vele (pl.: prímszámokat kiíró program esetén kiírja az 1-et is), erre NEM figyelmeztet a fejlesztőeszközünk, ezért a program futása közben ellenőrizzük a kimenet valóságát!

Első programunk: Hello World!

modul, névtér, kiíratás

Minden programozási nyelv tanulásakor általában a Hello World programot készítik el elsőnek, hát

```
1 using System;
2
3 namespace elsoprogramunk
4 {
5     0 references
6     class Program
7     {
8         0 references
9         static void Main()
10        {
11            Console.WriteLine("Hello World!");
12            Console.ReadKey();
13        }
14    }
```



tegyünk mi is így: Nyissuk meg a visual studiot és hozzunk létre egy új projektet, a Visual C# fülnél a Console Application-t választjuk, adjunk neki nevet és helyet! Az első sorba/sorokba kerül az egyes modulok közti hivatkozás, ezzel gyakorlatilag

"lerövidíthetjük" a programkód írását. Használata: `using modul;` Pl.: **using System;** Ezzel beemeltük a System osztályt, ami később kell nekünk. A következő a névtér létrehozása, ez nem muszáj, de használjuk azért! A névtér (namespace) az a logikai egység, melyben az azonosítónak egyedinek kell lennie a többi névtértől. Ezután következik egy tetszőleges nevű osztály/ok (class), amelynek egyikében kell lennie egy Main függvénynek. Ez a rész most még nem fontos nekünk, hiszen a Mainen kívül mi nem fogunk egy darabig semmit se írni. Ezekkel még ne nagyon törődjünk a Visual Studio automatikusan kiegészíti ezeket a részeket, majd később bővebben is szó esik róluk! A mainen belül jöhet a **(System).Console.WriteLine("Hello World");** ebben az esetben nem kell a System, mert már usingoltuk. A **Console.WriteLine("Hello World");** is ugyanezt csinálja, csak a kiírás után a következő sorba ugrik a kurzor. Miután lefordítottuk és lefuttattuk (START gomb) a programot, csak néhány másodpercig ugrik elő az ablak, utána bezárul, ezért a kiíratás után használjuk a **Console.ReadKey();** ami egy darab billentyűre lenyomására vár, ha lenyomtunk egy billentyűt, akkor bezárul a parancssoros ablak. **Fontos, hogy alkalmazzunk ; a sorok végén!!** Baloldalon láthatjuk az első működő programunkat, mint már megnéztük az első sorban **using System;** modul beemelése, a 3. sorban a névtér létrehozása és elnevezése, a Visual Studio által hozzáadott „0 references” ne

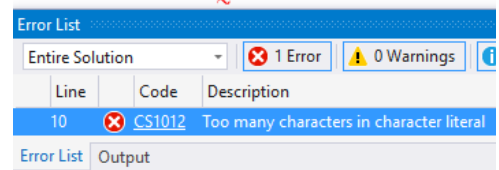
foglalkozunk. A 7. sorban a **main** metódus, aminek **minden programban lennie kell!** A **main** minden **c# kód belépési pontja**, mikor elindul a program az itt végzett utasítások, vagy alprogramok meghívása esetén az alprogramokban lévő utasítások futnak le! A 9. sorban a kiíratás és a 10-es sorban a billentyűleütésre váró függvény. Ajánlott a {} jelek használata, de nem minden esetben kötelező, a kapcsos zárójelek segítségével átláthatóbb lesz a programunk! Miután begépeljük a kódot nyomjuk meg a Start gombot, és ha nem ejtettünk **szintaktikai hibát**, akkor lefut a programunk és a képen látható eredményt adja. A szintaktikai hibákat piros recés aláhúzással jelzi a program és addig nem engedi futtatni a kódot. A nyelv lehetővé teszi kommentek beszúrását a fájlba a // és a /**/ jelek használatával, a // csak az abban a sorban levő szöveget kommentezi ki, így a fordítás utáni kimenetre az itt lévő szöveg nincs hatással, még akkor se, ha megfelelő kódot írtunk ide ugyanúgy, mint a /**/ csak hogy ez több soros is lehet.

Megjegyzés: Én a Visual Studio 2015 Enterprise verziót használom, de megteszi régebbi verzió is.

☺ **Szorgalmi feladat:** Milyen kimenet jelenik meg, ha begépeljük az alábbi kódot a mainbe?

```
Console.WriteLine("Milyen");
Console.WriteLine(" lesz");
Console.WriteLine(" a kimenete");
Console.WriteLine(" az alábbi");
Console.WriteLine(" kódnak?");
Console.ReadKey();
//Console.WriteLine("Hahaa");
//ez itt egy egysoros komment
```

```
//rosszpelda
string szo = 'nem jó';
```



Ha nem tudjuk, próbáljuk ki a visual studio-ban a programot!

Adatszerkezetek

változók, típusok

Mi a változó? A változó a memóriában lefoglalt bitek azonosítója és tartalma. deklarálás - egy változónévvel ellátott típust hozunk létre; definiálás - a változónévvel ellátott típusnak értéket is adunk. A típusok egy-egy algoritmusban a megadott típus értékének tárolására használt memória tartományának elnevezése. Mérete korlátokhoz kötött. A nyelvben az alábbi módon definiálunk és deklarálunk egy változót:

Dek.: típus változónév; Pl.: int szam1;

Def.: típus változónév = érték; Pl.: string szo = "vár";

A c# nyelvben a szám típusokat csak simán a változónév után egy = jellel és utána egy számmal deklaráljuk, míg a szöveges (char, string) típusokat char (1db karakter) esetében " jelek közé string-nél (karakterlánc - gyakorlatilag végtelen karakter) "" jelek közé írjuk, és úgy definiáljuk.

A változók névadásakor törekedjünk arra, hogy a változó neve tükrözze a változó tartalmát. A változók neve betűkkel vagy _ jellel kell kezdődnie, ezután az összes karakter megengedett (a magyar ékezetes betűk is használhatók), hossza általában 32 karakter lehet.

Fontosabb adattípusok: szám típusok: egész (sbyte, byte, short, ushort, int, uint, long, ulong) illetve a valós-lebegőpontoszámok (float, double, decimal). Az s-signed előjeles az u-unsigned előjeltelent jelöli, így például a sbyte (signed byte-előjeles byte) képes negatív számok tárolására is. Logikai típus: bool (true-igaz, false-hamis) Karakteres típus: char-karakter, string-karakterlánc

Fontos: ez nem az összes típust tartalmazza, ezek az alapvető típusok, ezeken kívül vannak osztályok, konstruktorok, objektumok, struktúrák, listák, tömbök...!

A változók nem képesek a végtelenségig a számok eltárolására, az alábbi táblázat tartalmazza az alapvető típusok értékhatárát, az értékhatáron kívül eső szám nem tárolható el benne! Törekedjünk mindig a megfelelő típust választani az adott változónak, hogy csökkentjük az adott memória igényt (pl.: tudjuk, hogy egy változó 0 és 100 közötti értéket vesz majd csak fel, ezért byte típust használunk).

Név	Értékhatár	Helyfoglalás (bit)
bool – logikai	true vagy false	1
char – karakter	1db karakter	16
string - karakterlánc	U	0 – 2 ³² db karakter
byte, sbyte	0-255, -128-127	8
ushort, short	0 – 65535; -32768 - 32767	16
uint, int	0 – 4,2 * 10 ⁹ , -2,1*10 ⁹ – 2,1 * 10 ⁹	32
ulong, long	0 – 18 * 10 ¹⁸ , -9 * 10 ¹⁸ – 9 * 10 ¹⁸	64
Lebegőpontos számok		32 - 128

Műveletek a típusokkal: Lássuk be nem sokat ér a változó, ha nem tudjuk módosítani. A matematikában megszokott műveletekkel adhatunk össze, vonhatunk ki, oszthatunk... Csak azonos típusokat adhatunk össze konvertálás nélkül (később szó lesz róla). Példák:

```
int a = 7; //változó létrehozása, értékadás
int b = 3; //változó létrehozása, értékadás
//az ab változóba lesz eltárolva a + b eredménye
int ab = a + b;
int aszorb = a * b; //21
//a b változót egyenlővé tesszük b + 3, tehát növeljük a b értékét 3-mal
b = b + 3; //ugyan azt az eredményt kapjuk, csak rövidítve
b += 3; //Például az alábbi képlet megegyezik:
a -= 2;
a = a - 2; //<--ezzel
//stringek összeadása/összefűzése
string str1 = "alma";
string str2 = "fa";
string ossz = str1 + str2; //az eredmény: almafa
int a = 2, b = 4, c; //több változó létrehozása egy sorban
```

Változók kiírása: Írassuk ki a képernyőre a változók értékét a + operátorral, stringeknél összefűzést jelent, így a + operátorral írathatjuk ki a változónkat. Példa:

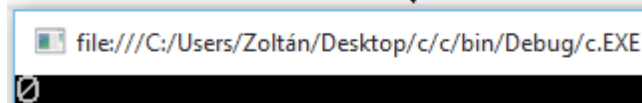
```
int eredmeny = 2;
Console.WriteLine("Az eredmény " + eredmeny + " lett.");
```

☺ *Szorgalmi feladat:* Gyakoroljunk: hozzunk létre két változót és adjunk nekik értéket, ezután írassuk ki a változók összegét és különbségét, el ne felejtjük a Console.ReadKey()-t a végéről!

Prefix és Postfix forma A programozásban nagyon sokszor kell használni az eggyel való növelést (pl.: `a = a + 1`), az egyszerűbb írásmódra létrehozták a `++a` (prefix) és az `a++` (postfix) változatot. A prefix változatnál először növeli az a változó értékét majd a többi részt, viszont a postfix alaknál csak utólag adódik hozzá az a változóhoz az 1.

Túlcsordulás: Akkor beszélünk túlcsordulásról, ha az adott érték már nem fér el a változónak fenntartott tárterületen (értékhalmoz, lásd: előző oldal). Például byte (0-255; 8bit) típusnál még a 255 belefér a tárterületbe, de ha eggyel (vagy többel) növeljük, akkor túlcsordulás következik be: a 255 kettes számrendszerben: 11111111 (8bit), ha 1-el növeljük, akkor már nem fér bele a 8bitbe így (1)00000000 lesz tehát 0 tízes számrendszerben. C#-ban az adattípus.MaxValue/MinValue megtudhatjuk az adott típus legnagyobb/legkisebb MÉG eltárolható értékét, példa: `Console.WriteLine(byte.MaxValue)`; Ami a konzolra annyit ír ki, hogy 255.

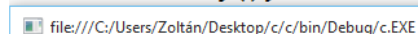
```
byte tulcsordulas = byte.MaxValue;
tulcsordulas++;
Console.WriteLine(tulcsordulas);
```



```
int j, i = 2;
j = ++i; //prefix
Console.WriteLine("j = " + j + "; i = " + i);
Console.ReadKey();
```



```
int j, i = 2;
j = i++; //postfix
Console.WriteLine("j = " + j + "; i = " + i);
Console.ReadKey();
```



Mennyit ír ki a konzolra az alábbi programkód?

```
int j, i = 5;
j = i;
j++;
Console.WriteLine(i);
```

Precedenciai rangsor

operátorok és művelettípusok rangsora

A precedenciai rangsor egy olyan rangsor, amelyet a nyelv követ és olyan sorrendben értelmezi a műveleteket, amilyen sorrendben le van írva ebben a táblázatban. Például a matematikában is a () jeles részek után a szorzást, majd az egyéb műveleteket vesszük figyelembe.

Művelettípus

Operátorok

1.	Elsődleges	(), [], x++, x--, new, typeof, sizeof, checked, unchecked
2.	Egyoperandusú	+, -, !, ~, ++x, --x
3.	Multiplikatív	*, /, %
4.	Additív	+, -
5.	Eltolási	<<, >>
6.	Relációs	<, >, <=, >=,
7.	Egyenlőségi	==, !=
8.	AND	&
9.	XOR	^
10.	OR	
11.	Feltételes AND	&&
12.	Felt. OR	
13.	Feltételes művelet	?:
14.	Értékadás	az összes értékadó operátor

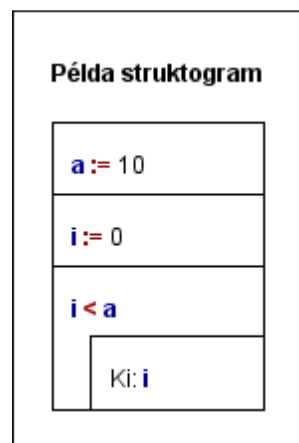
Ne ijedjünk meg a táblázat nagyságától, a nyelv megismerése során magunktól megtanuljuk ezeket, néhány operátor még nem is fontos az alapoknál.

Algoritmust leíró eszközök

folyamatábrák, struktogramok, mondatyszerű leírás

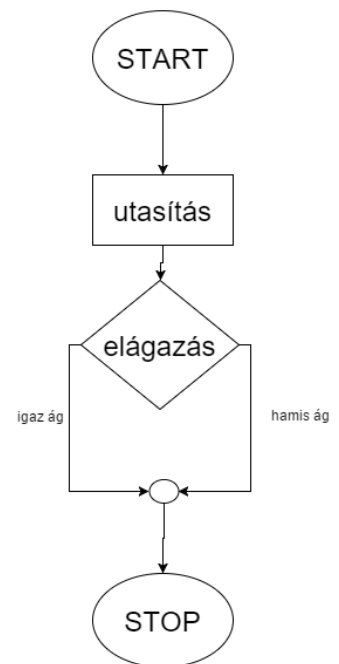
Az algoritmust leíró eszközök Pascal nyelvből származnak, ezért alapvető pascal tudás kell hozzá. Az értékadás a megszokott = jel helyett, itt := jelet használják. Nem kell típusokat adni a változókhhoz elég, ha pl.: **a := 2** használjuk. **Pontosvessző nem kell!** Még van egyéb különbség a C# és a Pascal között, ezeket később

ismertetem. Folyamatábra szerkesztőt a draw.io weboldalon találunk. **A folyamatábra**, olyan algoritmust leíró eszköz, mely a programot gráfként írja le. Ez egy olyan irányított gráf, amely csomópontokból áll, egyetlen induló és befejező része van. A következő kép megmutatja ezeket a csomópontokat, utasításokat..., és talán így már könnyebben megérthető lesz később a ciklusok használata. Minden egyes program egy Start ponttal indul és egy Stop ponttal fejeződik be. A nyilak a program útját mutatják, az utasítást egy téglalap írja le, a csomópontokat a kör jelöli. Később megismerkedünk a feltétellel (rombusz), ami szinte kihagyhatatlan lesz a programunkból, mert még a ciklusok is tartalmazzák! **Struktogram** jóval könnyebben rajzolható, viszont



(szerintem) nehezebben áttekinthető, mint a folyamatábra. Itt egy téglalap írja le az egész programot, amelyet további téglalapokra, háromszögekre osztanak. Balra egy példa stuktogram látható.

Stuktogramok készítésére a structorizer programot ajánlom **Mondatszerű leírás**, ez esetben a programot mondatokkal írjuk le a Program szóval és a Program vége szavakkal ér véget. Példa:



Program
Be: n
ciklus, amíg i := 0-tól, i < n, i = i + 1 lépéssel
Ki: i
ciklus vége
Program Vége

Adatbekérés a konzolról

Console.ReadLine(), konvertálás

Mit kell tennünk akkor, hogy ha a felhasználtól akarjuk elkérni az adatait, megkérjük, hogy adjon meg valami információt? Erre szolgálnak a **Console.ReadLine()** és a **Console.ReadKey()** függvények, a ReadLine() enter lenyomásra vár és az enter lenyomása után beolvassa a beírt szöveget, ezért stringet ad vissza, viszont ha valamilyen matematikai műveletet szeretnénk rajta elvégezni, akkor szintaktikailag hibás lesz. Ennek elkerülése érdekében át kell alakítani szám típusú string-et, hogy működő képes legyen, két lehetőségünk van a string számmá való átalakítására:

```
string beker = Console.ReadLine();//szöveg bekérése és eltárolása a beker változóban
int szam1 = int.Parse(beker); //változó = típus.Parse(string típus)
int szam2 = Convert.ToInt32(beker); //változó = Convert.To+típus+(valamilyen típus)
```

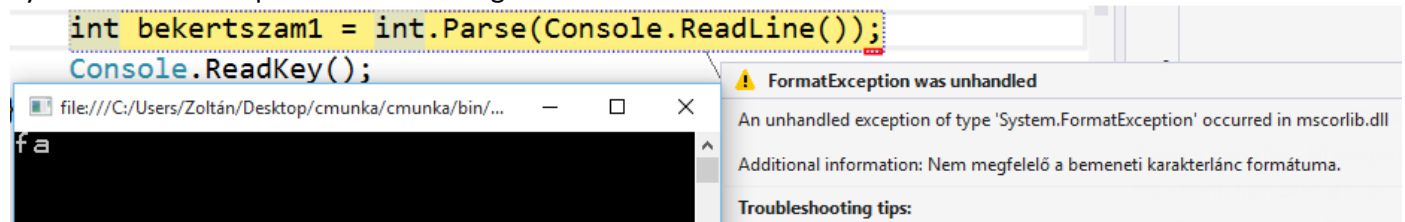
A bekérés nevű string változóba eltároljuk a bekért szöveget (mert hiszen string, hiába az értéke például 5), majd a szam1 és szam2-nél lévő Parse és Convert függvény alakítja át számmá. A Parse függvény csak string-ből képes valamilyen egyéb változóba konvertálni, viszont a Convert megbirkózik az összes változóval, így képes bármelyikből bármelyikbe konvertálni. Ha nem int-be szeretnénk konvertálni, cseréljük le az int-eket a megfelelő típusra pl.:

```
string beker = Console.ReadLine();
byte szam1 = byte.Parse(beker);
byte szam2 = Convert.ToByte(beker);
```

A konvertálást lehet egy sorba is írni az alábbi alakban, ilyenkor nem tárolunk el stringet:

```
int bekertszam1 = int.Parse(Console.ReadLine()); //vagy
int bekertszam2 = Convert.ToInt32(Console.ReadLine());
```

Ha nem valamilyen számot adunk meg, akkor kifagy a programunk és kivételt dob, ezért figyelmeztessük a felhasznált, hogy csak számot adjon meg, később megtanuljuk a kivételkezelést is. A kivétel: System.FormatException azaz nem megfelelő formátum kivétel.



Bekérésnél a Parse függvényt ajánlott használni! A ReadKey egy darab billentyűt olvas be és ConsoleKey típust ad vissza ennek átalakítását itt nem tárgyalom, egy darabig csak a programok végén fogjuk használni, hogy ne záródjon be az ablak.

Egy példaprogram az eddig tanultakról:

```
//névtér, main ...
Console.WriteLine("Add meg a téglalap egyik oldalát: ");
int a = byte.Parse(Console.ReadLine());
Console.WriteLine("Add meg a téglalap másik oldalát: ");
int b = byte.Parse(Console.ReadLine());
int ker = 2 * (a + b); int ter = a * b;
Console.WriteLine("A téglalap kerülete " + ker + " cm, területe " + ter + " cm2");
Console.ReadKey();
```

📌 **Szorgalmi feladat:** Csináljuk meg az alábbi programokat:

kérjük be egy négyzet oldalának hosszát, majd írassuk ki a képernyőre a területét és kerületét!

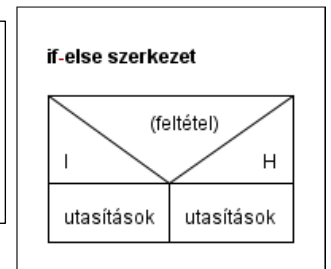
Kérjük be egy háromszög oldalának és az arra eső magasságvonal hosszát, majd számítsuk ki a területét!

Konvertálás, castolás

ConverTo, castolás

Mit kell tennünk, ha két különböző típusú számot szeretnénk összeadni, össze adni össze tudjuk, csak a típus átalakítással elveszhetnek fontos dolgok (próbáljunk meg két egymással maradék nélkül nem osztható int típust egymással elosztani, az eredmény egész?). Konvertálással viszont már a megfelelő típusúvá konvertálhatjuk a számunkat. A castolás is ugyan az csak egyszerűbb írásmódban. Példa:

```
int szam1 = 5; int szam2 = 120;
double maradeknincs = szam1 / szam2; //ha kiíratjuk 0-t kapunk
double vanmaradek = (double)szam1 / szam2; //megoldás: castolás
double vanmaradek2 = Convert.ToDouble(szam1) / szam2; //vagy konvertálással
Console.WriteLine("Nincs maradék:" + maradeknincs + " Van " + vanmaradek);
Console.ReadKey();
```



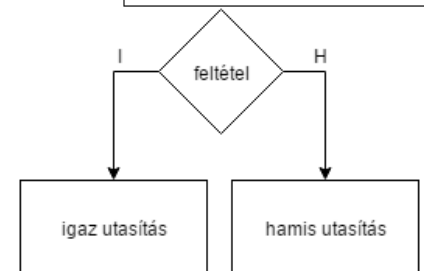
Szelekciók – elágazások: if-else

if-else szerkezet

Innentől belevágunk az igazi programozásba. Nagyon sok esetben szükségünk van egy elágazásra (szelekció), az elágazásoknak min. 1 vagy több ága is lehet, ha több ága van mint kettő, akkor már összetett elágazásról

```
if (feltétel){
    //utasítás
} else {
    //utasítás
}
```

beszélünk. A jobb oldali képen a folyamatábrája, fölötté pedig a struktogramja látható. Mondatszerű leírásban a következő: Ha (feltétel) igaz, akkor (utasítás), ha hamis, akkor (utasítás2). Bal oldalon a C# formája. A feltételhez a **relációs és egyenlőségi operátorok** használhatók (precedenciai rangsor): Az <> egyértelmű, a <= és >= jelek pedig a kisebb egyenlő és nagyobb-egyenlőt



jelölik. A == az egyenlőség vizsgálat (fontos, hogy ne csak egy db = jelt tegyünk, mert azt deklaráláskor használjuk!!), a != a nem egyenlő, ez esetben akkor kapunk igaz értéket ha a két szám NEM egyezik meg, ha a két szám egyenlő az ifnek az else ága fog érvényesülni tehát a false! Van még a **% operátor**, ami két szám maradékát adja meg. Például 7 % 3, ennek a maradékot osztásnak az eredménye 1 (mert 2 * 3 + 1 = 7). Ez használható arra, hogy eldöntsük egy számról, hogy páros vagy páratlan. Mindenki tudja, hogy egy szám akkor páros, ha osztható 2-vel (páros szám=2x, ha x∈Z, tehát x eleme a természetes számok halmazának). Ebből következik a feltételünk: ha egy x∈Z megfelelő x számot maradékosan osztva 2-vel 0 kapunk (nincs maradék), akkor az a szám páros. C#-ban a feltétel: **x % 2 == 0**. Mivel a mondatszerű leírás, a folyamatábra és stuktogram is Pascal alapú, ezért a % helyett a **mod** szót használjuk, pl.: **x mod 2** és az == jel helyett csak egy darab = jel et írunk feltételeknél: **x mod 2 = 0** Csináljunk egy példaprogramot: Kérjünk be egy egész számot, majd mondjuk meg mennyi a szám abszolút értéke! Az esetben, ha a szám nagyobb mint 0, akkor a szám abszolút értéke önmaga, viszont ha 0-nál kevesebb, a bekért szám akkor a -1 szerese. Elég egy egyágú elágazás, mert ha a szám pozitív vagy 0, akkor fölösleges bármit is csinálni vele! *Megjegyzés: kerüljük a fölösleges if ágakat, próbáljunk a lehető legkevesebb if-fel megcsinálni az adott programot, mert a későbbiekben ezek az apróságok csökkenteni fogják programunk gyorsaságát, természetesen nem a 8 soros kódoknál, mert ilyenkor észrevehetően a különbség!*

```
int bekertszam = int.Parse(Console.ReadLine());
if (bekertszam < 0) {
    //ahogy már tanultuk bekertszam = bekertszam * -1 rövidített verziója:
    bekertszam *= -1;
} //ha a szám pozitív nem kell mást tennünk, mint kiírni ugyan azt a számot mint, amit megadtunk
Console.WriteLine("A megadott szám abszolút értéke: " + bekertszam);
Console.ReadKey();
```


Feltételek egymásba ágazása Gyakran szükségünk van összetettebb elágazásokra is. Ekkor az ifen vagy az elsen belül létre kell hozni még egy elágazást. A program fentről lefelé halad így ha a feltétel igaz akkor lép be az ifen belülre és csak utána veszi

```
if (feltétel){
    //utasítás1
}
else if (feltétel){
    //utasítás2
}
```

figyelembe a következő feltételt, ha nem teljesül az első szintű feltétel, akkor a második szintű már nem hajtódik végre, ha az if ágban van! Ugyanígy, ha a második feltételt az else ágba tesszük és az első feltétel igaz lesz, akkor már nem lépbe

```
if (feltétel){
    //utasítás1
}
else {
    if (feltétel){
        //utasítás2
    }
}
```

az else ágba így nem is fogja megvizsgálni a második elágazást. Jegyezzük meg, hogy egy if-elágazásnak csak EGY kimenetele lehet, nem lehet olyan hogy mindkét ág, tehát a hamis és a negatív ágban lévő utasítások is lefutnak!! Ha egy összetettebb feltételt szeretnénk leprogramozni szükségünk van akár több if-re is, ez egy kicsit bonyolultabbá teszi a kód megértését. Ha megfelelően használjuk a tabulátort és a kapcsos zárójeleket, akkor olvashatóbbá, érthetőbbé válik a kód. Nézzünk erre

```
Console.WriteLine("Add meg a dolgozat maximális pontszámát: ");
int dogamaxp = int.Parse(Console.ReadLine());
Console.WriteLine("Add meg a dolgozatban elért pontszámot: ");
int dogaelertp = int.Parse(Console.ReadLine());
double szazalek = (double)dogaelertp / dogamaxp * 100;
if (szazalek > 90) {
    Console.WriteLine("A dolgozat jegye: 5");
} else {
    if (szazalek > 80) {
        Console.WriteLine("A dolgozat jegye: 4");
    } else if (szazalek > 70) {
        Console.WriteLine("A dolgozat jegye: 3");
    } else if (szazalek > 60) {
        Console.WriteLine("A dolgozat jegye: 2");
    } else {
        Console.WriteLine("A dolgozat jegye: 1");
    }
}
Console.ReadKey();
```

is egy példafeladatot: Csináljunk egy olyan programot, amely bekéri egy dolgozat maximális pontszámát, majd az abból elért pontszámot és megadja a jegyet. Az egyszerűség kedvéért 90%-tól 5, 80%-tól 4, 70%-tól 3, 60%-tól 2 és 60% alatt elégtelen. Gondolkozzunk el mit kell csinálni, először be kell kérni 2db adatot majd a megfelelő feltételeket alkalmazva ki kell íratnunk a jegyet. Ki kell számolni a százalékos értékét is a dolgozatoknak majd ez összehasonlítjuk a megadott százalékokkal. Az első if megvizsgálja, hogy a megkapott "szazalek" nagyobb-e 90-nél, ha nagyobb a dolgozat ötös és nem vizsgálja tovább a feltételeket (hiszen az else ágban van). Ha nem akkor tovább vizsgál ... Az else if () egy rövidített változat, gyakorlatilag azt rövidíti le, hogy ha az előző if az else ágba lép és újra követné egy if feltétel pl:

Bővebben a feltételekről: az AND(&&) és az OR(||) Gyakran szükségünk van még összetettebb feltételekre, ezeket az and és az or operátorokkal tudjuk megtenni. Ezeket az operátorokat, mint az egyenlőség jelet is duplázni kell, tehát a megfelelő alakjuk egy feltételben a következő: && és || . Mindegyikkel kettő vagy több feltételt tudunk alkotni: az and-nél(&&) csak akkor lépünk az igaz ágba, ha mind a kettő(vagy több, lényeg hogy az összes) feltétel igaz, ha csak egy is hamis értékkel tér vissza, akkor már a kimenet is hamis lesz, or-nál(||) elég csak egynek igaznak lennie, és a kimenet is igaz, viszont ha mindegyik false akkor az eredmény is false. Példa: if(a < 3 && b < 6) ez akkor lesz igaz, ha a kisebb mint 3 ÉS b kisebb mint 6 ha bármelyik is hamis akkor már az eredmény is hamis lesz!! Példa or-ra: if(a > 11 || a == 3) ez esetben akkor lépünk be az igaz ágba ha legalább az egyik feltétel igaz, VAGY a > 11 kell igaznak lennie VAGY a == 3 kell igaznak lennie. Még összetettebb feltételek: a feltételeknél is használható a zárójel, így először a zárójelen belüli rész fog kiértékelődni majd utána a többi feltétel. Példa: if(a < 3 || (a < c && b > 10)) Először a zárójel kerül kiértékelésre, azon belül van két feltétel && elválasztva tehát ebből következik, hogy mindegyiknek igaznak kell lennie, hogy igaz lehessen a "zárójeles rész",

a zárójelen kívül van egy or, ebből pedig az következik, hogy vagy az a < 3 nak kell igaznak lennie, vagy a zárójeles résznek, vagy mindkettőnek.

☺ *Szorgalmi feladat:* Gyakoroljuk az if szerkezetet! Csináljunk egy olyan programot, amely bekér egy magasságot (cm) és kiírja a képernyőre a következőket: Ha 140cm kisebb akkor írja ki, hogy alacsony, ha 140-170 átlagos és 170cm nagyobb, akkor írja ki, hogy magas. Segítség: a feladatot 2 if-else szerkezettel kell megoldani!

Szelekciók – elágazások: switch

kapcsoló

A switch egy olyan elágazás, amelyben szinte végtelen elágazás lehetséges. A folyamatábrája megtekinthető a jobb oldalon, mondatszerű leírásban sok feltételekkel kell leírni, ahol a feltétel egy része megegyezik a többi résszel. Nézzük meg a c#-os formáját: Ha megnézzük közelebbről, akkor minden egyes feltételnél az egyenlőséget vizsgáljuk (pl.: bekeres == "autó"), ha az érték hamis, akkor átugrik a következő 'összehasonlításra' és így tovább, ha nem talált egyetlen egy egyezőt sem, akkor a default ágnál lévő utasítás fog lejátszódni. Fontos hogy csak és kizárólag == lehet vizsgálni más operátorokat (<, >, != ...) nem lehet használni. A case ággal új összehasonlítást hívunk és mögé írjuk az összehasonlítást, majd egy kettősponttal nyitjuk, nem ; -vel, utána kapcsos zárójel és még az utolsó zárójel előtt egy break; utasítás, ami később a ciklusból való kilépésre fog szolgálni. A default egy végső megoldás, akkor fog lefutni a default ágnál lévő utasítás, ha semelyik case ág nem volt megfelelő. Mint az iffet is, a switchet is egymásba lehet tenni. Switchet akkor használunk, ha az értékek kötöttek. Például egy dobókockánál, ahol 6 lehetőség van, ezért könnyen lehet alkalmazni.

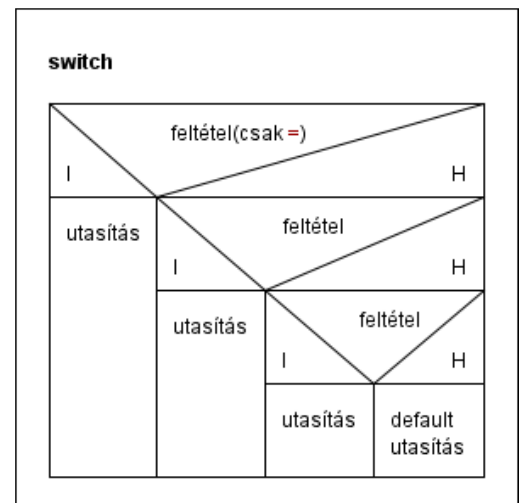
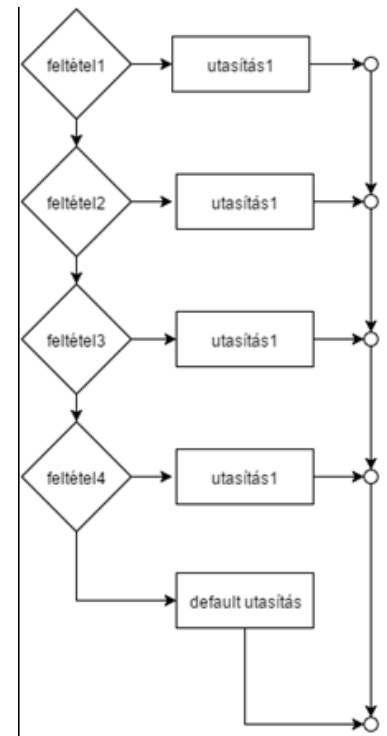
Nézzünk egy egyszerű programot a switch szemléltetésére: A felhasználó megadja, hogy hányast dobott a dobókockával, és a programunk írja ki a dobott számot betűvel *Megjegyzés: van egyszerűbb mód is erre, lásd: tömbök, de a bemutatás miatt switchel oldom meg!* Először nézzük meg hogy, hogy is nézne ki ifekkel:

```
int dobottszam = int.Parse(Console.ReadLine());
if (dobottszam == 1) {
    Console.WriteLine("Egy");
} else if (dobottszam == 2) {
    Console.WriteLine("Kettő");
} else if (dobottszam == 3) {
    Console.WriteLine("Három");
} //így tovább, nagyon fárasztó így vizsgálni
```

Nézzük meg ugyanezt switchel:

Fontos hogy csak a == feltételek helyettesítésére használható! Mondatszerű leírásban sok feltétellel lehet leírni a switch szerkezetét.

☺ *Szorgalmi feladat:* csináljuk meg a fenti kódot folyamatábrában és stuktogramban!



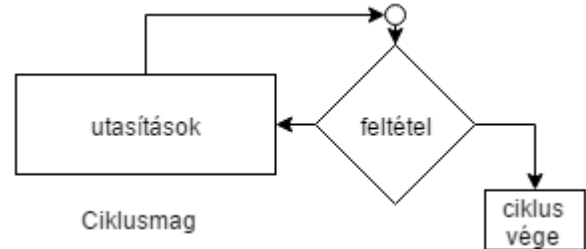
```
int dobottszam = int.Parse(Console.ReadLine());
switch (dobottszam)
{
    case 1:
        Console.WriteLine("Egy"); break;
    case 2:
        Console.WriteLine("Kettő"); break;
    case 3:
        Console.WriteLine("Három"); break;
    //...
}
```

Iterációk – ciklusok: előtesztelő ciklus (while)

while, páros számok kiírása, végtelenciklus, break, continue

Az előtesztelő ciklus "először vizsgál, és utólag végez", tehát mielőtt belépünk a ciklusmagba, a ciklus megvizsgálja a feltételt és utána végzi el az utasításokat, majd újra vizsgál és végez, újra, újra és újra, ameddig a feltétel hamis nem lesz. Szélsőséges esetben 0-szor hajtódik végre. Általában létre szoktunk hozni még egy változót, amit majd vizsgálhatunk ez esetben a parosszam az és a ciklus addig fut (addig ismétli a ciklusmagot)

```
int meddig = int.Parse(Console.ReadLine());
int parosszam = 0;
while (parosszam < meddig) {
    parosszam += 2;
    Console.Write(parosszam + ", ");
}
Console.ReadKey();
```



amíg a parosszam nem éri el a felhasználótól bekért

meddig változó értékét. A parosszam nevezetű változót 0 értékkel deklaráljuk és utána a

while(feltétel){ciklusmag} következik. Amíg a parosszam kisebb, mint a meddig változó értéke (true – igaz a feltétel), addig ismételjen és miután már nem nagyobb lépjen ki a ciklusból. Jobbra egy while-ciklus általános folyamatábrája és alatta a stuktogramja látható. Mondatszerű leírásban: **Ciklus, amíg feltétel – ciklusmag – Ciklus vége** formát veszi fel.

Néhány fontos tudnivaló a ciklusokkal kapcsolatban:

Végtelenciklus: akkor beszélünk végtelenciklusról, ha az adott ciklusból a program nem tud kilépni, így folyton ismételteti önmagát, gyakran kifagyáshoz is vezethet, lehet hogy a programozó direkt hoz létre egy végtelenciklust. Nagyon egyszerű végtelenciklust létrehozni: a while feltételébe írjuk, hogy true és már kész is.

Break: a ciklusból való kilépést, és a ciklus futásának felfüggesztését eredményezi.

Continue: a ciklusfutását úgy változtatja meg, hogy ha elér a continue a ciklusmag az utána következő ciklusutasításokat nem fogja figyelembe venni, hanem visszatér a ciklus elejére és folytatja azt, átugorja az adott ciklusmagot és a következőnél folytatja.

CS0103 The name 'a' does not exist in the current context

```
while (true)
{
    int a = 8;
    a += 2;
}
Console.WriteLine(a);
```

A break és a continue használata nem ajánlott, mert nem része a strukturált programozásnak!

Fontos: egy blokkon, cikluson belül deklarált változó csak a blokkon belül használható, a ciklusmagon kívül nem létezik (bal ábra)! Ha szeretnénk a cikluson kívül is használni, akkor még a ciklus előtt

hozzuk létre a változót, de a cikluson belül lévő változások nem vesznek el!

☺ **Szorgalmi feladat:** csináljunk egy olyan programot, ami összeadja 1-től egy tetszőleges, a felhasználótól bekért számig terjedő egész számok összegét!

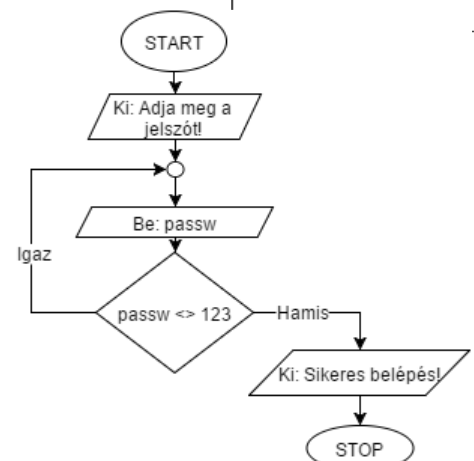
Iterációk – ciklusok: hátultesztelő ciklus

do-while, jelszavas beléptető

"először végez, utána vizsgál", hátultesztelő ciklus a nevéből adódóan először a ciklusmagban lévő utasítást hajtja végre, majd utána vizsgálja meg a feltételt, ha a feltétel igaz, akkor újra végrehajtja az utasítást... **Szélsőséges esetben legalább 1-szer lefut a ciklusmag.** Ebben a programban a do-whilet arra használjuk, hogy a felhasználót beléptessük a programba. A program belép a ciklusba és kér egy jelszót, ha a jelszó nem megfelelő, addig folytatja az újra kérést ameddig jó jelszót ír be a felhasználó. C#-ban:

while-ciklus

feltétel
ciklusmag



Mondatszerű leírással:

Ciklus – ciklusmag – amíg feltétel – Ciklus vége Pl.:

Ki: Add meg a jelszót!

Ciklus,

Be: passw

amíg passw <> 123

Ciklus vége

Ki: Sikeres belépés!

```

Console.WriteLine("Add meg a jelszót!");
string passw;
do {
    passw = Console.ReadLine();
} while (passw != "123");
Console.WriteLine("Sikeres belépés!");
Console.ReadKey();

```

Folyamatábrában az a különbség az előtesztelő ciklussal szemben, hogy először a csomópontba érkezik be, utána következik a bekérés és azután a vizsgálat, a <> Pascal nyelven a nem egyenlőt jelenti, c#-ban ez a !=. A stuktogramban először szerepel a ciklusmag és utána a feltétel. *Megjegyzés: Pascalban a hátultesztelő ciklusnál, akkor kezd el újra a ciklusmag futtatását, ha a vizsgálat hamis eredményt adott,*

de az itt látható ábrákban nem foglalkoztam vele és a továbbiakban sem használok ezt a szemléletet!

Iterációk – ciklusok: számlálóvezérelt ciklus (for)

for, prímszámok

A számlálóvezérelt ciklus (for-ciklus) teljesen úgy működik, mint egy while azzal a különbséggel, hogy itt létrehoztunk egy számlálót (i) és ennek változtatásával tudjuk a feltételt folyton vizsgálni. az első része int i = 0; létrehozuk a segédváltozót a második része a feltétel, és végül az i++ az i értékét növeli 1-el. For-nál mindig az i++ formát használjuk!

Érdekesség: A for-ciklusnak van egy "testvére" a foreach ciklus, ezt majd a tömbök körbejárására fogjuk használni, egyelőre nem lényeges. Mondatszerű leírásban a for-ciklus általános formája: **Ciklus, i := 1-től, feltétel (pl.: N-ig), x lépéssel (pl.: i := i + 1 lépéssel) – ciklusmag – Ciklus vége.** Struktogramoknál és folyamatábráknál külön utasításokkal kell növelnünk a ciklus változót és létrehozni is kell! Példafeladatként számítsuk ki 10! -át (faktoriális: egy n nem negatív egész szám faktoriálisának az n-től 1-ig terjedő sorozat elemeinek szorzata, pl.: 3! = 3*2*1 = 6) egy for-ciklussal! Mivel a szorzás tényezői felcserélhetők, ezért kezdhethetjük az i segédváltozót 10-től – 1-ig vagy 1-től növelhetjük 10-ig. Ennek megfelelően az i++ vagy az i– alakot kell használni. A feltétel növelés esetén i <= 10 vagy i < 11 ennek megfelelően csökkentés esetén i >= 1 vagy i > 0. Létre kell hozni egy fakt változót, amibe eltároljuk majd a szorzatok eredményét.

```

// a fakt változót 1-ről indítsuk, mivel ez nem foglya befolyásolni a megoldást
int fakt = 1; // elég lenne az i-t 2-vel indítani mivel az eggyel való szorzásnak nincs jelentősége
for (int i = 1; i <= 10; i++) {
    //fakt = fakt * i; rövidített alakja;
    fakt *= i;
}
Console.WriteLine(fakt);
Console.ReadKey();

```

Prímszámok kiírása 1000-ig, avagy ciklusok egymásba ágyazása

Egyszer minden programozó szembe ütközik azzal a gondolattal, hogy írassuk ki a prímszámokat (azok a természetes számok, amelyeknek csak kettő osztójuk van: 1 és önmaga): Haladjunk kívülről befelé, az első for arra szolgál, hogy léptesse nekünk azokat a számokat, ami később az osztandó szerepét fogja betölteni. Utána létrehoztam egy bool(logikai) típust ami arra szolgál, hogy ha az értéke true marad akkor ki lehet írni az aktuális i számot, ami prímszám. A következő for arra szolgál, hogy léptesse nekünk az osztót. Nem lehet ezen belül még egy i ciklusváltozó, mert már van egy. Fontos, hogy j nem lehet egyenlő nullával, mert nullával való osztás nincs és ne legyen 1-se, mert az 1 minden egész szám osztója, ezért indítjuk a j-t 2-ről. A for-ciklus feltétele j addig menjen amíg nem éri el az i-t (hiszen főleg tovább vizsgálnia), szintén a j < i feltétel nem lehet j <= i, mert akkor a szám önmagával oszthatóságát is megvizsgáljuk (mert ugye minden szám önmagával osztva 1 ad, és nem ad maradékot) és ott van egy && is mivel ha már találtunk egy olyan számot ami osztható az adott számmal, tehát a szám már biztosan nem lesz prímszám, emiatt már nem kell tovább vizsgálnunk. Az if arra kell, hogy megnézzük, hogy osztható-e a szám maradék nélkül, a % operátor visszaadja két szám maradékát. Ha a maradék 0 (tehát osztható maradék nélkül) akkor átállítjuk a primszam logikai típust false-ra, mert találtunk neki osztót, ergo nem lehet prímszám. A következő ifben, ha a primszam logikai típus igaz, azaz az aktuális i prímszám kiírja, ha hamis akkor nem.

```

for (int i = 2; i < 1000; i++) {
    bool primszam = true;
    for (int j = 2; j < i && primszam; j++) {
        if (i % j == 0) primszam = false;
    }
    if (primszam) Console.WriteLine(i);}

```

Tipp: Ha Visual Studioban F11-el elindítjuk ezt a programot, ott lépcsőről lépésre lehet haladni a programban, és közben mutatja a változók értékeit és talán úgy könnyebben is megérthető.

Alap C# függvények és metódusok

Title, SetWindowSize, WindowHeight/Width, SetCursorPosition, Math osztály

Console.BackgroundColor

A **Console.BackgroundColor** a konzolon kiírt betűk háttérszínét állítja át egy beállított színre. Használata a ConsoleColor.szín -nel

```
Console.BackgroundColor = ConsoleColor.Blue;  
Console.WriteLine("Ez kék háttérű szöveg");
```



Console.ForegroundColor

A **Console.ForegroundColor** a konzolon kiírt betűk színét állítja át egy beállított színre.

```
Console.ForegroundColor = ConsoleColor.Green;  
Console.WriteLine("Ez zöld színű szöveg");
```



Console.ResetColor()

A **Console.ResetColor()** visszaállítja alapértelmezett értékre a háttér és a szöveg színét.

```
Console.ForegroundColor = ConsoleColor.Green;  
Console.BackgroundColor =  
ConsoleColor.DarkBlue;  
Console.WriteLine("Színes");  
Console.ResetColor();  
Console.WriteLine("Alapértelmezett");
```



Console.Clear();

A **Console.Clear()** törli az ablak tartalmát, vagy ha közvetlenül a háttérszín beállítása után használjuk, akkor az egész hátteret az adott színre állítja be.

```
Console.BackgroundColor = ConsoleColor.DarkBlue;  
Console.Clear();
```

file:///C:/Users/Zoltán/Desktop/cmunka/cmu



Console.Beep();

A **Console.Beep(frekvencia, idő tartalom ms-ban)** sípolást eredményez.

```
Console.Beep(400, 1000);
```

Console.Title

A **Console.Title** a konzol ablak címének elnevezésére szolgál, ha nem állítjuk be ezt az értéket, akkor az adott program elérési útja lesz az ablak neve. A Console.Title-nek egy string típust kell adnunk = jellel, példa:

```
Console.Title = "ablak";
```

Console.WindowHeight és WindowWidth

A **Console.WindowHeight** és **WindowWidth** az ablak aktuális magasságát és szélességét mondja meg, melyet el is tárolhatunk egy változóban.

```
int m = Console.WindowHeight, sz = Console.WindowWidth;  
Console.WriteLine(m + "*" + sz);
```

Console.SetWindowSize(,)

A **Console.SetWindowSize(int szélesség, int magasság)** metódus a konzol ablak méretének beállítására szolgál, ha nem állítjuk be alapértelmezetten 80 szélességű és 25 karakterterület magasságú. Példa:

```
Console.SetWindowSize(50, 40);
```


Console.SetCursorPosition(,)

A **Console.SetCursorPosition(int oszlop ,int sor)** a kurzor helyének beállítására szolgál. Képzeljük el a konzol ablakot egy koordináta rendszerként melyben az x tengely balról jobbra 0-tól az ablak szélességéig tart, valamint az y tengely, ami fentről lefelé, 0-tól az ablak magasságáig tart. Az ablak bal felső sarka a 0, 0 koordináta és jobb alsó sarka 79, 24 koordináta (80 szélességű és 25 magasságú ablak esetén). Ha az ablakon kívülre eső koordinátát adunk meg, amit a program nem képes elérni, akkor a **System.ArgumentOutOfRangeException** kivétel adódik, ami a következő hibát jelenti: Az értéknek nullával egyenlőnek vagy annál nagyobbabbnak és a konzol pufferméreténél kisebbnek kell lennie az adott dimenzióban. A példa programban a konzol ablak közepére fog kerülni a kurzor:

```
int m = Console.WindowHeight;
int sz = Console.WindowWidth;
Console.SetCursorPosition(sz / 2, m / 2);
```

A Math osztály függvényei

A Math osztályban található függvények a matematikában megszokott képletek helyettesítésére szolgálnak. Általános alakja a **Math.Függvénynév(érték/ek)**; Például **Math.Abs(x)**, ami megmondja az x változó abszolút értékét, hogy megmaradjon az x abszolút értéke el kell tárolnunk egy változóban, pl.:

A következő pontokban a Math osztály alap függvényei lesznek bemutatva:

- **Math.Abs(double)** Mint már fent említettem, ez az abszolút értékét fogja visszaadni egy double típusú változónak.

```
double x = -20.5;
x = Math.Abs(x); //Megoldás: 20.5
```

- **Math.Ceiling(double)** Azt a legkisebb egész számot adja vissza, ami nem kisebb a megadott számnál.

```
double x = 12.6;
x = Math.Ceiling(x); //Megoldás: 13
```

- **Math.Cos(double)** A megadott értékünk koszinuszát adja vissza.

```
double x = 15;
x = Math.Cos(x); //Megoldás: -0,7596
```

- **Math.Floor(double)** Azt a legnagyobb egész számot adja vissza, ami nem nagyobb a megadott számnál.

```
double x = 15.3;
x = Math.Floor(x); //Megoldás: 15
```

- **Math.Log(double)** A megadott értékünk természetes logaritmusát adja vissza.

```
double x = 1000;
x = Math.Log(x); //Megoldás: 6,9
```

- **Math.Log10(double)** A megadott értékünk 10-es alapú logaritmusát adja vissza.

```
double x = 1000;
x = Math.Log10(x); //Megoldás: 3
```

- **Math.Max(byte_1, byte_2)** A két megadott byte típusú változók közül vissza adja a legnagyobb értékét.

```
byte sz0 = 10, sz1 = 15;
byte max = Math.Max(sz0, sz1); //Megoldás: 15
```


- **Math.Min(byte_1, byte_2)** A két megadott byte típusú változók közül vissza adja a legkisebb értékét.

```
byte sz0 = 10, sz1 = 15;
byte min = Math.Min(sz0, sz1); //Megoldás: 10
```

- **Math.Pow(double_1, double_2)** Hatványozás, az első tag a hatványalap, a második a hatványkitevő.

```
double alap = 3, kitevo = 4;
double pow = Math.Pow(alap, kitevo); //Megoldás: 81
```

- **Math.Round(double, mérték)** A megadott számunk kerekítve, a mértékét megadhatjuk magunk is.

```
double x = 7.8;
double ker = Math.Round(x); //Megoldás: 8
```

- **Math.Sin(double)** A megadott értékünk szinuszát adja vissza.

```
double x = 120;
x = Math.Sin(x); //Megoldás: 0,58
```

- **Math.Sqrt(double)** A megadott értékünk négyzetgyökét adja vissza.

```
double x = 25;
x = Math.Sqrt(x); //Megoldás: 5
```

- **Math.Tan(double)** A megadott értékünk tangensét adja vissza.

```
double x = 160;
x = Math.Tan(x); //Megoldás: -0,22
```

Matematikai állandók:

- Math.PI – a π konstanst adja vissza
- Math.E – az e konstanst adja vissza

```
Console.WriteLine("A PI értéke: " + Math.PI);
```

Randomolás

Random r, new, Next()

Sokszor előfordul, hogy egy értéket randomoljunk, magyarul véletlenszerűen generáljunk egy számot. Hogy randomolni tudjunk a System.Random osztályt kell használnunk és létre kell hoznunk egy Random objektumot (még nem lényeges, hogy számunkra mi is ez, csak használjuk) névvel ellátva: **Random név = new Random();** A new operátor egy objektum létrehozására szolgál. Ezután a megadott objektumnév segítségével tudunk majd olyan függvényekre hivatkozni, amelyek segítenek nekünk a véletlen szám generálásban. Ahogy a típusoknál is, itt is az egész illetve a lebegőpontos számokra bontjuk a használandó függvényeket: **Egész** számok generálásakor a **név.Next()** függvényt kell használni (int nagyságú), melyet, ha üresen hagyunk, akkor egy tetszőleges int szám fog generálódni. Ha a zárójelen belül adunk egy intervallumot például **név.Next(0, 11)**, akkor az első szám mondja meg a minimális **még generálható** értéket, az utolsó pedig a **már nem generálható** értéket, ez esetben 0-t még generálhat a gép, de 11-et nem, csak maximum 10-et. Hogy megmaradjon ez a generált szám tároljuk el egy változóban! A következő példa egy dobókockát fog szemléltetni:

```
//Random objektum létrehozása:
Random r = new Random();
//A generált szám létrehozása és eltárolása:
int dobott = r.Next(1, 7);
Console.WriteLine("A dobott szám " + dobott);
Console.ReadKey();
```

Lebegőpontos, valós számok generálására a **név.NextDouble()** függvényt kell használni. Itt már nem adható meg határérték alapértelmezetten 0.0 és 1.0 közé fogja generálni a számokat, ami azért baj, mert így nem

tudunk pl. 0 és 100 közé eső valós számokat generálni. De egy kicsit megpiszkálva a számot tudunk, szorozzuk meg 100-al a generált számot, és így már sikerülni fog 0 és 100 közé randomolni. Példa:

```
Random r = new Random();
double randzam = r.NextDouble() * 100;
Console.WriteLine("Generált szám: " + randzam);
Console.ReadKey();
```

Érdekesség: a `NextBytes()` byte tömbök feltöltésére szolgál.

Kérdések

1. Milyen egy számlálóvezérelt ciklus?
2. Mire használható a `Console.Clear()` és a `SetCursorPosition()`?
3. Mire használható a `Math.Abs` és a `Sqrt`?
4. Mire használható a `break` és a `continue`?
5. Miért mondja a program, hogy nem létezik az 'a' változó (ábra), indok?
6. Csináljuk meg az alábbi programnak a folyamatábráját és struktogramját: írassuk ki az első 10db 3-mal osztható számot!

CS0103 The name 'a' does not exist in the current context

```
while (true)
{
    int a = 8;
    a += 2;
}
Console.WriteLine(a);
```

Gyakorlás

Barkóba játék: Csináljunk egy olyan játékot, ami 0 és 100 között generál egy számot, majd a felhasználónak tippelnie kell, és a program megmondja, hogy a generált szám kisebb vagy nagyobb a tippelt számnál.

```
Random r = new Random();
/*bár lehetne int is, de mivel tudjuk, hogy 100-nál nem lesz nagyobb, ezért byte típust
alkalmazzunk, így csökkentve a programunk memória igényét! A Next függvény, mivel intet ad vissza
ezért castolnunk kell.*/
byte randzam = (byte) r.Next(0, 101);
byte tipp = 0;
do {
    Console.Write("Tipp: ");
    tipp = byte.Parse(Console.ReadLine());
    //Bekérjük és átalakítjuk a stringet byte-ra, majd utána megvizsgáljuk a 'tipp'-et, hogy nagyobb,
    kisebb vagy egyenlő-e a randomolt számmal
    if (tipp < randzam)
        Console.WriteLine("Nagyobb számra gondoltam!");
    else if (tipp > randzam)
        Console.WriteLine("Kisebb számra gondoltam!");
    else
        Console.WriteLine("Eltaláltad!");
} while (tipp != randzam);
//a ciklus addig fog futni, míg a tipp nem lesz egyenlő, mint a generált szám, hátul tesztelő ciklus
kell.
Console.ReadKey();
```

Lépcső: Rajzoljunk az ábrának megfelelő módon a console ablakban!

```
for (int i = 1; i <= 5; i++) {
    int j = 1;
    while (j <= i) {
        Console.Write("x");
        j++;
    }
    Console.WriteLine();
} Console.ReadKey();
```

```
X
XX
XXX
XXXX
XXXXX
```

A külső for-ciklus felel az egyes sorokért, a belső while ciklus pedig az aktuális sorban rajzolja ki a megfelelő mennyiségű X karakter(ek)e)t. Ha megtörtént az adott sorban a karakterek kiírása, akkor a következő sorba ugrik a kurzor!

Vonal: rajzoljunk a képernyő közepére egy kék csíkot az ablak fejlécével merőlegesen!

```
Console.BackgroundColor = ConsoleColor.Blue;
for (int i = 0; i < Console.WindowHeight - 1; i++)
{
    Console.SetCursorPosition(Console.WindowWidth / 2, i);
    Console.Write(" ");
}
Console.ReadKey();
```

/bin/Debug/cmunka2.EXE



Összetett adatszerkezetek: Tömbök

int[], new, vektor, mátrix, mutatóvektorok, konstans

A tömbök is adatok tárolására alkalmas, viszont a sima adatszerkezetekkel ellentétben ez már egy összetett adatszerkezet, ami képes egy bizonyos változónév alatt több elemet is tárolni egy index segítségével. Képzeljünk el egy szekrényt, aminek a neve „ruhák”. A szekrény fiókjai pedig meg vannak számozva, és minden számhoz tartozik egy ruhanemű, például az 1-es fiókban vannak a zoknik, a 2-esben a nadrágok, a 3-asban a pólók... C#-ban így lehetne valahogy létrehozni ezt a „ruhás” szekrényt:

```
string[] szekreny = new string[] { "zokni", "nadrág", "póló", "pulcsi", "sapka" };
```

Ha a tömbnek nem adunk értéket alapértelmezetten 0-val fog feltöltődni szám típusok esetében, egyéb típusoknál a *null* értéket kapja. A tömbök indexelése 0-tól kezdődik, a 0. elem az első, az 1. elem a második..., ezért az utolsó elemet mindig az eltárolható elemek n-1 indexe mutatja meg. Tömböket többféle módon is létre lehet hozni:

```
//megadjuk a tömb méretét, ez esetben 10db string tárolható el (vektor)
string[] pelda1 = new string[10];
/*megadjuk felsorolásban az elemeket egy kapcsos zárójelben, ez esetben a kapcsos zárójelen belüli
elemek mennyisége határozza meg a tömb méretét, tehát 5db int típus tárolható el ebben a tömbben:*/
int[] palda2 = new int[] { 5, 31, 78, 124, 4048};
```

A tömböket az indexük segítségével tudjuk használni, általános formája: **tömbnév[index]** Ugyanúgy használhatók, mint a változók.

```
//tömb létrehozása (vektor)
int[] tomb = new int[10];
Console.WriteLine(tomb[2]); //ha nem adunk meg semmit akkor 0
//érték adás
tomb[2] = 46; //változtatás után:
Console.WriteLine(tomb[2]);
```

Ha a tömb egy dimenziós (egy indexe van) **vektornak**, ha 2, akkor már **mátrixnak** nevezzük.

Valahogyan így lehetne elképzelni egy vektort és egy két dimenziós mátrixot:

Vektor						
Index	0	1	2	3	4	5
Adat	342	43	45	12	422	52

C#-ban a tömbök tudják a méretüket, így azt le lehet kérni, vektor esetében a **tömbnév.Length**-el. Mátrixok esetén a **mátrixnév.GetLength(dimenzió száma)**-val lehet

lekérdezni a méretét.

A már deklarált tömb méretét már nem tudjuk

befolyásolni, ha olyan indexre mutatunk, ami kívül esik a tömb méretén akkor **System.IndexOutOfRangeException** kivételt kapunk, azaz az index a tömb határain kívülre mutatott. Pl.: Van egy 10 egység méretű tömbünk és mi le akarjuk kérni a tömb 10. elemét (tömbnév[10]), akkor ilyen kivételt kapunk, mert ez a tömb 0-9-ig indexelhető, hiszen a 0 is index, ezért lesz az utolsó még jó indexünk a 10-1 (0 miatt) a 9. A tömböket legegyszerűbben for-ciklussal tudjuk bejárni, a for-ciklus fejléce általánosan az alábbi alak: **for (int i = 0; i < tömbnév.Length; i++)** így biztos, hogy nem akad ki a programunk. A for-ciklus írásra és olvasásra is feltudjuk használni, viszont van a tömbökhöz készített foreach-ciklus, ami végig járja a tömböt és mindig az aktuális elemet belerakja egy ideiglenes (pl.: item) változóba, ez viszont csak olvasásra jó,

Mátrix						
Index	0	1	2	3	4	5
0	32	32	45	1	22	5
1	324	5	3	5	35	64
2	43	5234	423	45	234	876
3	42	23	34	234	4	2
4	5467	64	567	354	13	56
5	86	34	67	52	76	432

így elemeket nem tudunk változtatni vele. Általános alakja: `foreach` (típus változónév `in` tömbnév). A következő kódban feltöltünk egy tömböt randomolt számokkal és kiíratjuk ezután egy `foreach`-ciklussal.

```
int[] randomoltszamok_Vektor = new int[10];
Random r = new Random();
for (int i = 0; i < randomoltszamok_Vektor.Length; i++) {
    //for ciklussal írni és olvasni is tudunk
    randomoltszamok_Vektor[i] = r.Next(0, 100);
} foreach (int item in randomoltszamok_Vektor) {
    //csak olvasásra jó
    Console.WriteLine(item);
}
Console.ReadKey();
```

Bővebben a mátrixról: a mátrix egy olyan kétdimenziós tömb melynek van szélessége és magassága is, míg a vektor csak 1 magas volt és bármekkora széles addig ez már legegyszerűbben egy táblázatként fogható fel, használata majdnem ugyan az, mint a vektornál csak itt már két indexet kell megadnunk, hogy elérjünk egy cellát. Deklarálása hasonló, mint a vektorok esetében, csak egy vesszőt kell beszúrunk a négyzetes zárójelek közé. Pl.: `int[,] mátrix = new int[4, 5];` Ilyenkor egy olyan `int` típusú mátrix jön létre, melynek szélessége 4 cella, magassága pedig 5. Mint már mondtam a több dimenziós tömbök esetében a **tömbnév.GetLength(dimenzió szám)** tudjuk lekérni az aktuális dimenzió méretét, a dimenziók számozása is 0-tól kezdődik és a megadott dimenzióig tart. Szintén tudunk kapcsos zárójelekkel létrehozni egy már 0-tól eltérő adatokat tartalmazó tömböt, ilyenkor szintén nem kell megadnunk a tömb méretét. Pl.:

```
int[,] mátrix = new int[,] {
    { 421, 23, 42, 1 },
    { 45, 674, 341, 52 },
    { 56, 12, 343, 54 }
};
Console.WriteLine("Magassága: " + mátrix.GetLength(0)); //3
Console.WriteLine("Szélessége: " + mátrix.GetLength(1)); //4
```

Ha ilyen módon adunk meg tömböket, akkor az egyes sorokban lévő elemek számának meg kell egyeznie, ne felejtjük el a zárójelek után vesszőt tenni és a főzárójel után egy pontosvesszőt! Mátrixok körbejárására 2 db `for`-ciklus kell, általános alakja:

```
//Mátrix feltöltése randomolt számokkal !!-a mátrix nevű tömböt már létrehoztuk-!!
Random r = new Random();
for (int i = 0; i < mátrix.GetLength(0); i++) {
    for (int j = 0; j < mátrix.GetLength(1); j++) {
        mátrix[i, j] = r.Next();
    }
}
```

A mátrix elképzelhető egy olyan vektorként melynek celláiban újabb vektorok vannak, viszont ezek a vektoroknak meg kell egyezniük a többi cellában lévő vektorok méretével, ezek a tömböket szabályos tömböknek, négyzetes mátrixoknak, **multidimenziós tömböknek** nevezzük. Léteznek olyan többdimenziós tömbök, melyeknek a mutató vektoron (fő vektor) belül lévő vektorok mérete nem azonos, ezek a **mutatóvektorok**. Példa mutatóvektorok létrehozására:

```
int[][] mutatovektor = new int[][] {
    new int[] { 6, 2, 2, 7, 3 },
    new int[] { 3, 1 },
    new int[] { 12, 31, 56 } };
Console.WriteLine("For-ciklussal:"); //végigjárása for-ciklussal, olvasásra és írásra
for (int i = 0; i < mutatovektor.Length; i++) {
    for (int j = 0; j < mutatovektor[i].Length; j++) {
        Console.Write(mutatovektor[i][j] + ", ");
    }
    Console.WriteLine();
}
Console.WriteLine("Foreach-ciklussal:"); //végigjárása foreach-ciklussal, csak olvasásra
```

```
foreach (int[] belsővektor in mutatóvektor) {
    foreach (int elem in belsővektor) {
        Console.Write(elem + ", ");
    }
    Console.WriteLine();
}
```

Konstans: létrehozhatunk c# nyelvben olyan változókat melyeknek az értéke állandó, tehát biztos, hogy nem fog változni, így a program csak a változó értékét foglalja le a memóriában. Létrehozhatunk konstans tömböt is (pl.: a magyar abc-t tartalmazó betűkészlet tömböt), általános létrehozási alakja, amit általában a deklaráláskor kell megadnunk: **const változótípus változónév = érték;** Pl.:

```
//Bár van a Pi-re függvény, használhatunk egy állandó, konstans változót is erre, hiszen tudjuk,
hogy a pi értéke nem fog változni
const double pi = 3.14159265359;
//erre hibát kapunk:
pi = 5;
```

Példa konstans tömb létrehozására:

```
const char[] magyar_abc = new char[] { 'a', 'á', 'b', 'c', ...};
```

Ugyanígy használhatjuk a **const** szót minden változótípus előtt.

A string egy tömb, bővebben a stringekről

char[], escape karakterek, string függvények,

A string egy olyan char tömb, ami a többi tömbbel ellentétben a mérete változtatható, egy string így nézne ki tömb formájában:

```
string alma = "alma"; //egyenlő ezzel:
char[] almaT = new char[] { 'a', 'l', 'm', 'a'};
```

Ugyan úgy, mint a tömböket a string-eknél is a cellák (ez esetben karakterek) tartalmát az indexük segítségével tudjuk elérni, így például az **alma[1]** az l karakter, ugyan úgy lekérdezhető a hossza is a **string.Length** -el. Csináljunk egy olyan programot, ami bekér a felhasználótól egy karakterláncot, majd megmondja, hogy hány db szóköz van az adott string-ben.

```
Console.WriteLine("Adja meg a karakterláncot: ");
string str = Console.ReadLine(); //mivel tudjuk, hogy a szamlalo sohasem lesz negatív, ezért
állítsuk uint-re a típusát és 0 kezdőértékkel:
uint szamlalo = 0; for (int i = 0; i < str.Length; i++) {
    if (str[i] == ' ') szamlalo++;
}
Console.WriteLine("Szóközők száma: " + szamlalo);
Console.ReadKey();
```

String metódusok:

- **string.Substring(honnan, hány darabot)** Kimásol az eredeti stringből az első paraméterül kapott indextől a 2. paramétert jelentő hosszúságig, a kapott index is már másolva lesz.

```
string str = "Szoftverfejlesztés";
string str2 = str.Substring(8, 3); //fej
```

- **string.IndexOf(keresett string, mettől keressem)** A paraméterül kapott stringet keresi, ha van benne, akkor visszatér a string első karakterének indexével, ha nem -1-et ad vissza (ha több van mindig az első indexét), ha 2. paraméter is adunk meg, akkor az adott indextől fogja keresni.

```
string str = "Réges rég várva várt engem";
int index = str.IndexOf("vár"); //10
```

- **string.Replace(régi string, új string)** A kapott string-ben a megadott stringet, ha van, lecseréli a 2. paraméterre. Pl.:

```
string str = "Réges rég várva várt";
string str2 = str.Replace("vár", "árt"); //Réges rég ártva ártt
```

- **string.ToUpper()/ToLower** Az adott stringet nagybetűssé/kisbetűssé alakítja. Pl.:

```
string str = "kisbetű";
string str2 = str.ToUpper(); //KISBETŰ
```

- **string.Split(határoló karakter)** A megadott határoló karakter mentén szét darabolja a stringet, és egy szöveges tömböt hoz létre, ahol a cellákban a határoló karakter mentén feldarabolt karakterlánc lesz.

```
string str = "C# programozás könyv";
string[] strT = str.Split(' ');
foreach (string item in strT) {
    Console.WriteLine(item); }
```

- **string.Trim()** Láthatatlan karaktereket(escape karakterek, szóköz) töröl le a string elejéről és végéről.

```
string str = "kisbetű";
string str2 = str.ToUpper(); //KISBETŰ
```

- **string.Remove(honnan, mennyi)** A megadott indextől töröl ki karaktereket a megadott darabszámig.

```
string str = "kisbetű";
string str2 = str.ToUpper(); //KISBETŰ
```

- **string.Contains(string)** Tartalmazás vizsgálat, logikai értékkel tér vissza.

```
string str = "kisbetű";
string str2 = str.ToUpper(); //KISBETŰ
```

- **string.Insert(hova, mit)** A megadott helytől beszúr egy stringet, és a kiegészített stringet adja vissza

```
string str = "fejlesztő";
string str2 = str.Insert(0, "C# "); //C# fejlesztő
```

- **string.PadLeft/PadRight(hosszúság, mit)** Kiegészíti az adott stringet a megadott hosszúságig a megadott karakterrel balról vagy jobbról feltöltve.

```
string str = "alma";
str = str.PadRight(6, '?'); // alma??
```

Escape karakterek:

Escape karakterek, olyan vezérlőkarakterek, amelyek a console ablakban megjelenő karakterek kimeneti módját határozza meg. Minden vezérlő karakter egy \ jellel kezdődik.

Használata: Console.WriteLine("Tabulátor:\t Helló"); Ha a karakterlánc elé egy @jelet teszünk, akkor a karakterlánc szó szerinti stringként értelmeződik, így az escape karakterek úgy lesznek kiírva, ahogy vannak.

Tabulátor: Helló

\a	alert hang
\b	törlés, backspace
\n	új sorba ugrik a kurzor
\r	vissza ugrik egyet a kurzor
\t	tabulátor vízszintesen
\v	tabulátor függőlegesen
\'	' karakter illeszt be
\"	" karaktert illeszt be
\\	\ karakter illeszt be

Kiírás + nélkül

Nehéz már + segítségével sok változót kiírni, főleg akkor, ha terjedelmes szövegbe kellene beleszúrni, erre találták ki a {index} formátumot ami megkönnyíti a kiíratást. A string után egy vesszőt teszünk, és ide írjuk a változókat vesszővel elválasztva, balról jobbra haladva, az itt lévő

változók beleszűrődnek az indexüknek megfelelően a {index} helyre, szintén 0-tól számozzuk az indexet n-1-ig, hasonlít egy vektorra, vannak formátumkódok is, de azt később vesszük. Pl.:

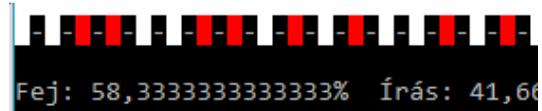
```
Console.WriteLine("{0}db pozitív szám, legnagyobb a {1}, legkisebb a {2}", db, legnagyobb, legkisebb);
```

Példa programok a tömbök, tételek használatára

A példa programok megoldása letölthető a honlapról!

Vektorok

1. Kérjünk be a felhasználótól 20db -100 és +100 közé eső számot, és tároljuk el ezeket egy vektorban, számoljuk meg mennyi osztható 3-mal, melyik a legkisebb, illetve a legnagyobb és határozzuk meg a számok átlagát! A megoldás leírással letölthető a honlapról a programozás fül alatt: karsazoltan.ddns.net
2. Csináljunk 3 darab 365 méretű vektort, az egyikbe randomoltassunk 4000 és 12000 között lévő számokat, ez a tömb lesz egy üzlet napi bevétele, a másikba randomoltassunk 2000 és 4000 között számokat ez lesz a kiadás a 3. tömbbe számítsuk ki az egyes napokra eső tiszta hasznót (bevétel - kiadás), határozzuk meg mennyi volt a legjobb nap!
3. Csináljunk egy bool típusú 60 elemű vektort, amibe egy pénzérme dobás eredményét fogjuk eltárolni (fej-true, írás-false jelenti), randomoljunk 0 vagy 1-et, ha a random 0, akkor false írjunk a tömbbe, ha 1, akkor true-t. Jelenítsük meg a kapott eredményt a képen látható módon (fej-fehér, írás-piros), valamint írjuk ki, hogy hány %-ban volt fej és hány % írás!
4. Állítsuk elő egy 50 elemű int típusú vektorba az első 40 Fibonacci számot (egy olyan sorozat, melynek az első két eleme a 0 és az 1, az azt követő számok az előző 2 szám összege). Írassuk ki egy foreach ciklussal az így kapott vektor elemeit! Ha megvagy, milyen szemantikai hiba lépett fel a programban, mi az oka?



Mátrixok

1. Hozzunk létre egy olyan tömböt, melybe hőmérsékletet fogjuk lekérdezni minden órában 30 napig keresztül, használjunk egy 30*24-es mátrixot, randomoljunk bele -5 és +15 között értékeket, ezután határozzuk meg melyik órában volt az egész hónapot nézve a leghűvösebb, és a legmelegebb átlaghőmérséklet, valamint melyik napon volt a legmagasabb átlaghőmérséklet a 30 napban.
2. Csináljunk egy 30*20 logikai mátrixot melybe egy színház foglalt illetve szabad helyeit fogjuk ábrázolni (true-szabad, false-foglalt). Randomoljunk bele true vagy false értékeket! Kérjük be a felhasználótól hány db széket szeretne lefoglalni egymás mellett, ha van ilyen, akkor írassuk ki a sor és az ülés számot, ha nincs, írjuk ki, hogy nem található ennyi üres szék egymás mellett.

TryParse használata

TryParse, out, kimeneti változó

Parse használatával eddig, abban az esetben, ha egy betű került a számunk közé, kifagyott, tehát System.FormatException kivétel adódott, ennek elkerülése érdekében a TryParse átalakító függvényt kell használnunk, fontos megjegyezni azt, hogy ugyanúgy, mint a Parse a TryParse is csak stringből képes egy bizonyos formátumba konvertálni, más bemenet nem adható meg neki! A TryParse bool, logikai típust ad vissza, általános alakja:

típus.TryParse(átalakítandó **string**, **out** a típusnak megfelelő kimeneti változó);

Ebben az esetben egy int típusú változóba (szam) szeretnénk eltárolni a Console.ReadLine()-al bekért stringet, az out szó után lévő változóba fog eltárolódni sikeres parsolás (konvertálás) esetén a bekért string. A logikai változóba (tryparse) fog eltárolódni egy olyan logikai érték, amely a sikeres vagy sikertelen konvertálásra utal, true esetén sikerült a konvertálás, tehát bele rakta a kimeneti változóba az átalakított stringet, a false esetén viszont nem sikerült átalakítani. Természetesen használható más típusok átalakítására is, mint például a long, ulong, short, byte...

```
int szam = 0;
bool tryparse = int.TryParse(Console.ReadLine(), out szam);
if (tryparse) Console.WriteLine(szam);
else Console.WriteLine("Rossz bemenet, hiba!");
Console.ReadKey();
```

Ha egy ilyen megoldást csinálunk, akkor már nem fagy ki a programunk, de a felhasználónak újra el kellene indítani a programot, hogy újra megtudjon adni értékeket. Ezért a TryParse-t foglaljuk egy do-while ciklusba, és abban az esetben fusson le újra a ciklus, ha nem sikerült a konvertálás, tehát nem került semmire a kimeneti változóba, ergo a TryParse által visszaadott logikai érték false, ezért kell még egy felkiáltó jelet odaszúrni elé a ciklus feltételében.

El ne felejtjük az eltárolandó szám változóját a ciklusmagon kívül deklarálni, mivel a ciklusmagon belül létrehozott változó nem látszik a ciklusmagon kívül!

```
long szam = 0;
do {
    Console.Write("Adja meg a számot: ");
} while (!long.TryParse(Console.ReadLine(), out szam));
Console.WriteLine(szam);
Console.ReadKey();
```

Billentyűzet kezelés

Console.ReadLine, Read, ReadKey, ConsoleKeyInfo

Mint már megtanultuk a Console.ReadLine() egy karakterláncot olvas be a bemenetről enter lenyomás után, pedig visszaadja ezt a stringet. Ha nem adunk meg karaktereket, üres mező után nyomjuk le az enter, akkor null értéket ad vissza. Ellenben a ReadLine a sima Read() egy darab karaktert olvas be enter lenyomás után és a bevitt karakter ASCII kódját adja vissza egy int típusban, ha ezt a számot karakterre parsoljuk visszakapjuk a bekért karaktert.

```
int ascii = Console.Read();
Console.WriteLine("Bekért betű " + (char)ascii + " - ascii kódja: " + ascii);
```

Példa: Be: r

Ki: Bekért betű r - ascii kódja: 114

A Console.ReadKey() egy billentyűt olvas be a bemenetről és nem vár enter lenyomásra. ConsoleKeyInfo típusú struktúrát (majd később lesz róla szó) ad vissza, amelyben el van tárolva az adott karakter unicode-ja, valamint a módosító billentyűk (Ctrl, Alt...) aktivitását is, képes eltárolni speciális karaktereket is (escape, lefele nyíl ...). Ha megadunk egy true értéket paraméternek, akkor nem fog megjelenni a leütött billentyű karaktere. A .KeyChar-ral elérhetjük a megadott karaktert, char típust ad vissza.

```
ConsoleKeyInfo cki = Console.ReadKey();
Console.WriteLine(cki.KeyChar);
```

A .Key-vel az általános billentyűk mellett lekérdezhethetünk speciális karaktereket, a Console.ReadKey().Key felsorolás típust (enum, szintén szó lesz róla később), ConsoleKey-t ad vissza.

```
ConsoleKey ck = Console.ReadKey().Key;
Console.WriteLine(ck);
```

Példa az alkalmazásra:

```
ConsoleKey ck = Console.ReadKey().Key;
switch (ck)
{
    case ConsoleKey.UpArrow:
        Console.WriteLine("A felfele nyilat nyomtad meg!"); break;
    case ConsoleKey.DownArrow:
        Console.WriteLine("A lefele nyilat nyomtad meg!"); break;
    //.....}
```

Billentyűzet puffer: amikor lenyomunk egy billentyűt a lenyomott billentyű bekerül egy puffer-ba. A `Console.ReadKey()` metódus ebből a pufferből vesz ki mindig egy értéket, ha a puffer-ben még található billentyű, akkor a `Console.KeyAvailable` true-t ad vissza, ellenkező esetben false-t. Jelszó bevétel példa:

```
ConsoleKeyInfo cki;
string jelszo = "";
do
{
    cki = Console.ReadKey(true);
    if (cki.Key != ConsoleKey.Enter)
    {
        Console.Write("x");
        jelszo += cki.KeyChar;
    }
} while (cki.Key != ConsoleKey.Enter);
if (jelszo == "1234") Console.WriteLine("\nJó jelszó");
else Console.WriteLine("\nHelytelen jelszó");
```

Alprogramok

Eljárások, függvények, paraméterátadási módok

A programok felépítése alapvetően szekvenciális, a főprogram a `Main` a programunk belépési pontja, innen kezdődik a futás, az itt meghívott függvények hajtódnak végre. Az alprogramok a programkód lerövidítésére és átláthatóságára szolgálnak, minden egyes saját eljárásnak vagy függvénynek meg van a maga szerepe és a szerepére utaló neve, így ha valamilyen javítandó probléma lenne a programunkban, akkor könnyen nyomon követhető a program felépítése és jobban korrigálható. Az alprogramok általános alakja:

static típus Név(paraméter1, p2,...){utasítás1; utasítás2... (ha függvény return valami;)} 2 fő alprogram típust különböztetünk meg, minden alprogram a `static` kulcsszóval kezdődik, ennek lényegéről majd szó esik az objektum orientált programozás résznél, a paraméterezésről majd később bővebben:

1. **Eljárások:** nincs visszatérési értéke, ezért **void** (üres) típusú

```
static void Main(string[] args){
    Kiir(); //Meghívás
}
static void Kiir() { //Maga az alprogram
    Console.WriteLine("Üdvözöllek a programban! ");
} //visszatérés
```

2. **Függvények:** van visszatérési értéke, a visszaadott értéktől függ, a **return** utasítással adjuk meg a visszaadott értéket, ami minden esetben kötelező a megadása.

```
static void Main(string[] args){
    int i = Osszeg(); //Meghívás
    //az i tartalma már 12
}
static int Osszeg(){
    return 5 + 7;
} // visszatérés int 12-vel
```

Miután meghívtunk egy eljárást és az lefut, visszatér a meghívás helyére, a függvényeknél a `return` utasítást követően tér vissza a függvény a meghívás helyére. A **return utasítás után lévő parancsok nem futnak le**, például az alábbi függvény 5 értéket ad vissza, hiába futna tovább a `for`-ciklus és lenne még egy `return` utasítás:

```
static void Main(string[] args){
    //Meghívás
    int i = Példa();
}
static int Példa(){
    for (int i = 0; i < 10; i++) {
        if (i == 5) return i;
    }
    return 0;
}
```

Az alprogramok egymást is meghívhatják, egy metódus meghívhat más függvényeket, eljárásokat, sőt akár **önmagukat is meghívhatják (rekurzió)**, viszont ilyenkor le kell kezelni azt is, hogy ne hívja meg magát mindig az alprogram, különben **System.StackOverflowException** kivétel dobódik.

Az alprogramok paraméterezése:

A metódus deklarálásánál megadott paramétereket formális, a metódus meghívásakor megadott paramétereket pedig aktuális paraméternek hívjuk. Az eljárásokat és függvényeket 3-féle módon paraméterezhetjük: **érték, cím és egy u. kimeneti paraméterként**, a paramétereket **vesszővel választjuk el** egymástól, ha értéket is adunk az alprogramunk fejlécében a paramétereknek, akkor a meghíváskor nem lesz kötelező megadni az adott paramétert.

```
static void Main(string[] args){
    Udvozlet(); //nézzük meg mi lesz a kimenet ez és
    Udvozlet("Helló!"); //ez esetén
}
static void Udvozlet(string s = "Köszöntelek a programban!"){ Console.WriteLine(s); }
```

1. **Érték sz. paraméterátadás:** Ha nem adunk meg semmilyen kulcsszót, akkor ilyen módon adódnak át a változók (kivéve tömbök, objektumok...). Ilyenkor az alprogram meghívásakor megadott aktuális paraméterek **értéke átmásolódik** az alprogram egy-egy megfelelő formális paraméterbe, ezután a formális paramétert ugyanúgy használhatjuk, mintha egy változó lenne.

```
static void Main(string[] args){
    int a = 6, b = 4, c;
    c = KétszeresétÖsszeadóFüggvény(a, b);
    Console.WriteLine("'a\' értéke:{0}\n'b\' értéke:{1}\n'c\' értéke:{2}", a, b, c);
    //a: 6, b: 4, c: 20
    Console.ReadKey();
}
static int KétszeresétÖsszeadóFüggvény(int szam1, int szam2) {
    szam1 = szam1 * 2; szam2 = szam2 * 2;
    return szam1 + szam2;
}
```

2. **Cím (Referencia) sz. paraméterátadás:** A **ref** kulcsszóval érhető el a cím sz. p., a **ref** kulcsszót az aktuális és a formális paraméter előtt is kell használni! Ebben az esetben az átadott aktuális paraméterre, változóra egy **hivatkozás jön létre a memória tartományára, csak a memóriaterület címe/azonosítója adódik át, így az alprogramon belül végzett változtatások érvénybe lépnek az átadott változóban is!** Fontos: az összetett adatszerkezetek, tömbök, objektumok (...) minden esetben referencia, tehát cím szerint adódnak át!

```
static void Main(string[] args){
    int a = 6, b = 4, c;
    c = KétszeresétÖsszeadóFüggvény(ref a, ref b);
    Console.WriteLine("'a\' értéke:{0}\n'b\' értéke:{1}\n'c\' értéke:{2}", a, b, c);
    //a: 12, b: 8, c: 20 ←itt már megváltozik az átadott változó értéke!!!
    Console.ReadKey();
}
static int KétszeresétÖsszeadóFüggvény(ref int szam1, ref int szam2) {
    szam1 = szam1 * 2; szam2 = szam2 * 2;
    return szam1 + szam2;
}
```

3. **Kimeneti változó:** Az **out** kulcsszóval érhető el ez a paraméter átadási mód, szintén a meghíváskor és az alprogram deklarálásakor is meg kell adni! Hasonlít a cím sz. p. -ra viszont az átadott paraméternek kezdetben nincs értéke a függvényünk hívott ennek értéket adni, de **kötelező az értékadás!**

```
static void Main(string[] args){
    bool paros;
    Paros(10, out paros);
}
static void Paros(int a, out bool log)
{
    if (a % 2 == 0) log = true;
    else log = false;
}
```

A következő feladatban egy ellenőrzött bekérés függvényt csinálunk, aminek kötelezően meg kell adni egy üzenetet, valamint lehetősége adódik a függvény meghívójának arra, hogy megadhatja azt, hogy mely számok között fogadjon el értékeket a függvény, a függvény visszaadja a feltételnek megfelelő bekért számot.

```
static int EllBekerf(string bekeresuzi, int mettol = int.MinValue, int meddig = int.MaxValue) {
    int szam;
    do {
        Console.WriteLine(bekeresuzi);
    } while (!int.TryParse(Console.ReadLine(), out szam) || !(szam > mettol && szam < meddig));
    return szam;
}
```

A következőben nézzünk egy olyan függvényt, ami összegzi egy tömb tartalmát:

```
static int Osszegzo(int[] tomb){
    int ossz = 0;
    foreach (int item in tomb){
        ossz += item;
    }
    return ossz;
}
```

Csináljunk egy olyan függvényt, ami egy tetszőleges int tömböt tölt fel véletlen értékekkel a megadott értékeken belül, mivel a tömbök minden esetben referencia, tehát cím szerint adódnak át ezért nem kell a ref kulcsszó, bár oda lehet tenni:

```
static void RandTomb(int[] tomb, int mettol, int meddig){
    for (int i = 0; i < tomb.Length; i++){
        tomb[i] = r.Next(mettol, meddig);
    }
}
```

A következőben csináljunk egy függvényt erre a módszerre, ami vissza ad egy int tömböt a megadott értékeken belül, randomolt számokkal, a tömb méretét is paraméterként kérjük:

```
static Random r = new Random();
static int[] RandT(int Tméret, int mettol, int meddig){
    int[] visszaT = new int[Tméret];
    for (int i = 0; i < visszaT.Length; i++){
        visszaT[i] = r.Next(mettol, meddig);
    }
    return visszaT;
}
```

Csináljunk egy olyan függvényt, ami kiírja egy tömb legkisebb elemét:

```
static void Min(int[] tomb){
    int legk = tomb[0];
    foreach (int item in tomb){
        if (item < legk) legk = item;
    }
    Console.WriteLine(legk);
}
```

Saját alprogramunk túlterhelése (alap sz.)

Csináljunk egy Osszegzo függvényt, ami paraméterül kapott 2db int egész szám összegét visszaadja.

```
static void Main(string[] args){
    Console.WriteLine(Osszegzo(10, 13));
    Console.ReadKey();
}
static int Osszegzo(int szam1, int szam2){
    return szam1 + szam2;
}
```

Ha viszont ezt a függvényt lebegőpontos számokra hívjuk meg, akkor a fordító szintaktikai hibát jelez, mit kell akkor csinálnunk, még egy függvényt írni, melynek neve különbözik az előzőtől? Például egy Összeadint és egy

ÖsszeadóDouble nevű függvény? A válasz nem. Lehet azonos név alatt különböző szignatúrájú (más az átadott típus, valamint a visszaadott érték) függvényeket, eljárásokat létrehozni. Tehát ha szeretnénk csinálni, olyan azonos nevű függvényeket, amelyek képesek az int+int-re, a double+double-re, az int+double-re, valamint a double+int-re visszaadni a két paraméter összegét, akkor elég 4db azonos nevű, de eltérő szignatúrájú függvényt csinálni, ezt nevezzük túlterhelésnek (overloads), például a Console.WriteLine() -nak 19 db túlterhelése van. A felkínálkozó túlterhelések között a le-fel nyilakkal váltogathatunk

Console.WriteLine()

▲ 8 of 19 ▼ void Console.WriteLine(int value)

Writes the text representation of the specified 32-bit signed integer value, followed by the current line terminator, to the standard output stream.

value: The value to write.

```
static void Main(string[] args) {
    int a = Osszegzo(10, 20);
    double b = Osszegzo(11.5, 13.2);
    double c = Osszegzo(13.1, 10);
    Console.ReadKey();
}
static int Osszegzo(int szam1, int szam2) {
    return szam1 + szam2;
}
static double Osszegzo(double szam1, double szam2) {
    return szam1 + szam2;
}
static double Osszegzo(double szam1, int szam2) {
    return szam1 + szam2;
}
static double Osszegzo(int szam1, double szam2) {
    return szam1 + szam2;
}
}
```

int Program.Osszegzo(int szam1, int szam2) (+ 3 overloads)

Lehet egy metódusban is kiváltani a fenti metódusokat (összetudjunk adni int-et is, meg double-t is) de azt itt nem tárgyalom!

Alprogramok tulajdonságainak dokumentálása

Nagyobb függvényeknél szükség lehet az alprogram tulajdonságainak megadására úgy, hogy a meghívás helyén már kiderüljön a függvény funkciója. Erre nyújt megoldást az xml dokumentáció, amit a /// taggel tudunk elérni, ugyanúgy, mint például a kommentek esetén. Ezeket az xml leírásokat a függvényünk előtt kell alkalmazni. A <summary>Összeadást végző függvény</summary> tagek közé az eljárásunk, függvényünk általános leírását adhatjuk meg. A <param name="paraméter neve">Egyik szám</param> az alprogramunkban szereplő paraméter szerepére mutat rá. A <returns>Eredmény</returns> megadhatjuk, hogy mit ad vissza a függvény. Valamint a <exception cref="Hiba"/> tagek figyelmeztetik a metódus meghívóját, hogy milyen hibát dobhat az alprogram, ezek a már korábban említett System.Exceptionok lehetnek. Ezek a beállítások megjelennek a meghíváskor is a visual studioban. A lenti példában felhívjuk a figyelmet arra, hogy a függvény System.OverflowException dobhat, ami a túlcsordulást jelenti.

```
/// <summary> Ez a függvény átkonvertálja a kapott string bináris számot decimális számmá </summary>
/// <param name="bin">Bináris szám</param>
/// <param name="dec">Ha sikerül a konvertálás, ide kerül az átalakított szám</param>
/// <returns>Ha sikerült a konvertálás true, ha nem false értékkel tér vissza</returns>
/// <exception cref="System.OverflowException"/>
static bool KonvBinDec(string bin, out int dec){
    dec = 0;
    for (int i = 0; i < bin.Length; i++){
        if (bin[i] == '1') dec += (int)Math.Pow(2, bin.Length - 1 - i);
        else if (bin[i] != '0')dec = 0; return false;
    }
    return true;
}
```

bool Program.KonvBinDec(string bin, out int dec)

Ez a függvény átkonvertálja a paraméterül kapott string bináris számot decimális számmá

Exceptions:

OverflowException

Rekurzió: Az olyan műveletet, melynek végrehajtásakor a saját műveletsorait hajtja végre, tehát önmagát ismétli, rekurziónak nevezzük. Programozás esetén a függvények saját magukat hívják meg. Amikor egy függvény nem önmagát, hanem egy másik függvényt hív meg, majd ez a függvény pedig meghívja azt a függvényt, amiből meghívták (...), akkor azt kölcsönös rekurzióknak nevezzük (pl. A() meghívja B()-t, majd B() meghívja A()-t...). *Figyelem: A rekurzió nagyon erőforrás igényes, hiszen akár több százszor is létrejönnek a függvény változói és ezzel memóriát foglalnak el.*

```
static ulong Faktorialis(ulong n){
    if (n <= 1) return n;
    return n * Faktorialis(n - 1);
}
static void Main(string[] args){
    Console.WriteLine(Faktorialis(3));
}
```

```
static ulong Faktorialis(ulong 3) {
    if (3 <= 1) return 3;
    return 3 * Faktorialis(3 - 1);
}
```

```
static ulong Faktorialis(ulong 2) {
    if (2 <= 1) return 2;
    return 2 * Faktorialis(2 - 1);
}
```

```
static ulong Faktorialis(ulong 1) {
    if (1 <= 1) return 1;
    return 1 * Faktorialis(1 - 1);
}
```

Faktoriális meghatározása: mivel a szorzás tagjai felcserélhetőek, ezért a paraméterként kapott számtól indulunk, nézzünk egy példát: az alábbi függvényt 3-mal hívjuk meg, a $3 \neq 1$ tehát újra meghívódik a függvény viszont most már 2-vel, de továbbra sem egyenlő 1-gyel tehát megint meghívódik, ez esetben az n az 1 így nem hívja meg továbbra is magát.

Nemes Tihamér programozás verseny (2015/2016) 2. korcsoport feladatának megoldása rekurzív módon (a verseny

kikötése, hogy 0,5s alatt kapjunk eredményt, ez a program kb. 20 fölött már lassabb). Feladat:

Egy N szintes épület szintjeit fehér (F), piros (P) és zöld (Z) színnel festhetjük ki. Piros emeletet csak fehér emelet követhet, továbbá nem lehet egymás mellett két zöld emelet! Készíts programot, amely megadja, hogy az épület hányféleképpen színezhető ki! A *standard bemenet* első sorában az emeletek száma van ($1 \leq N \leq 1000$). A *standard kimenet* egyetlen sorába a színezések lehetséges legnagyobb számát kell kiírni!

```
static int Variaciok_szama = 0;
static void Main(string[] args) {
    int emeletekszama = int.Parse(Console.ReadLine());
    EmeletSzine("F", 1, ref emeletekszama);
    EmeletSzine("P", 1, ref emeletekszama);
    EmeletSzine("Z", 1, ref emeletekszama);
    Console.WriteLine(Variaciok_szama);
    Console.ReadKey();
}
static void EmeletSzine(string s, int emeletizam, ref int emeletizam_meddig) {
    if (emeletizam < emeletizam_meddig) {
        switch (s)
        {
            case "P":
                EmeletSzine("F", emeletizam + 1, ref emeletizam_meddig);
                break;
            case "F":
                EmeletSzine("F", emeletizam + 1, ref emeletizam_meddig);
                EmeletSzine("P", emeletizam + 1, ref emeletizam_meddig);
                EmeletSzine("Z", emeletizam + 1, ref emeletizam_meddig);
                break;
            case "Z":
                EmeletSzine("F", emeletizam + 1, ref emeletizam_meddig);
                EmeletSzine("P", emeletizam + 1, ref emeletizam_meddig);
                break;
        }
    }
    else Variaciok_szama++;
}
```

A felsorolás típus - Enum

enum, Enum.TryParse, enum castolás

A felsorolás egy olyan adatszerkezet, amelyben egy meghatározott konstans (nem változhat) értékek adategységét jelenti. Enum típust **csak eljárásokon kívül, de még osztályon belül** szabad csak deklarálni, különben szintaktikai hibát kapunk. Általános alakja:

```
enum Állatok {Kutya, Tigris, Zsiráf, Oroszlán }
```

Miután létrehoztuk a felsorolást az egyik metódusunkból, már használhatjuk is az alábbi módon:

```
Állatok állat = Állatok.Tigris;  
Console.WriteLine(állatok);
```

A felsorolás elemeihez alapértelmezetten egy index társul 0-n számozással, ezt arra tudjuk használni, hogy az egyes számoknak megfeleltessünk egy-egy Állatok felsorolás egy tagját, ebben az esetben a 0-Kutya, 1-Tigris ... Ugyanígy a számokat is áttudjuk alakítani a megfelelő felsorolás értékre. Ha nem megfelelő értéket adtunk meg, akkor nem keletkezik kivétel. Példa:

```
Console.WriteLine("Az 1-esnek megfelelő a(z): {0}", (Állatok)1);  
Console.WriteLine("A(z) {0} -nak megfelelő szám a(z) {1}", Állatok.Oroszlán, (int)Állatok.Oroszlán);
```

Megadhatjuk akár mi is a felsorolás elemeinek értékét, ha az első elem értékét adjuk meg, akkor a rákövetkezők mindig az előző +1 lesz, különben az összes elemnek adnunk kell egy értéket.

```
enum Jegyek {Elégtelen = 1, Elégséges, Közepes, Jó, Kiváló}  
static void Main(string[] args)  
{  
    Console.WriteLine((Jegyek)2);  
    Console.ReadKey();  
}
```

Az elemekhez alapértelmezetten int típus társul, de a : használatával megadhatjuk a típusát.

```
enum RégiSúlymértékek : uint {Talentum = 30000, Mina = 500, Sékel = 11, Beka = 6}
```

Enum.GetValues/GetNames

Az Enum.GetValues egy tömböt ad vissza, melyben a megadott felsorolás elemei szerepelnek, ezt a tömböt át kell alakítanunk a megadott felsorolás tömbre (Felsorolás[]), a GetNames is ugyan ez az elven működik csak string tömböt ad vissza. Az Enum.GetName egy karakterláncot ad vissza a megadott felsorolásban.

```
RégiSúlymértékek[] rsmT = (RégiSúlymértékek[])Enum.GetValues(typeof(RégiSúlymértékek));  
foreach (RégiSúlymértékek item in rsmT){  
    Console.WriteLine(item);  
}  
string[] rsmTstr = Enum.GetNames(typeof(RégiSúlymértékek));  
foreach (string item in rsmTstr){  
    Console.WriteLine(item);  
}  
string str = Enum.GetName(typeof(RégiSúlymértékek), 500);  
Console.WriteLine(str);
```

Talán most már mindenki rájött, hogy a c#-ban vannak beépített felsorolás típusok, ilyen például a ConsoleColor és a ConsoleKey. Ahogy az előbb is ezekből a felsorolásokból is készíthetünk tömböt. Csináljunk egy olyan programot, ami váltogatja a háttér színét folyamatosan!

```
ConsoleColor[] consolecolors = (ConsoleColor[])Enum.GetValues(typeof(ConsoleColor));  
int i = 0;  
while (true)  
{  
    Console.BackgroundColor = consolecolors[i];  
    Console.Clear();  
    if (i++ == consolecolors.Length - 1) i = 0;  
}
```

Enum.TryParse() Képes egy karakterláncból adott felsorolás típusra átalakítani, ha sikerül a konverzió true, ha nem akkor false értékkel tér vissza.

```
enum Színek {piros, narancs, zöld, kék, barna, szürke, fekete, fehér};
static void Main(string[] args){
    Színek szín;
    if(Enum.TryParse(Console.ReadLine(), out szín))Console.WriteLine("Sikeres konvertálás " + szín);
    else Console.WriteLine("Sikertelen konvertálás");
    Console.ReadKey();
}
```

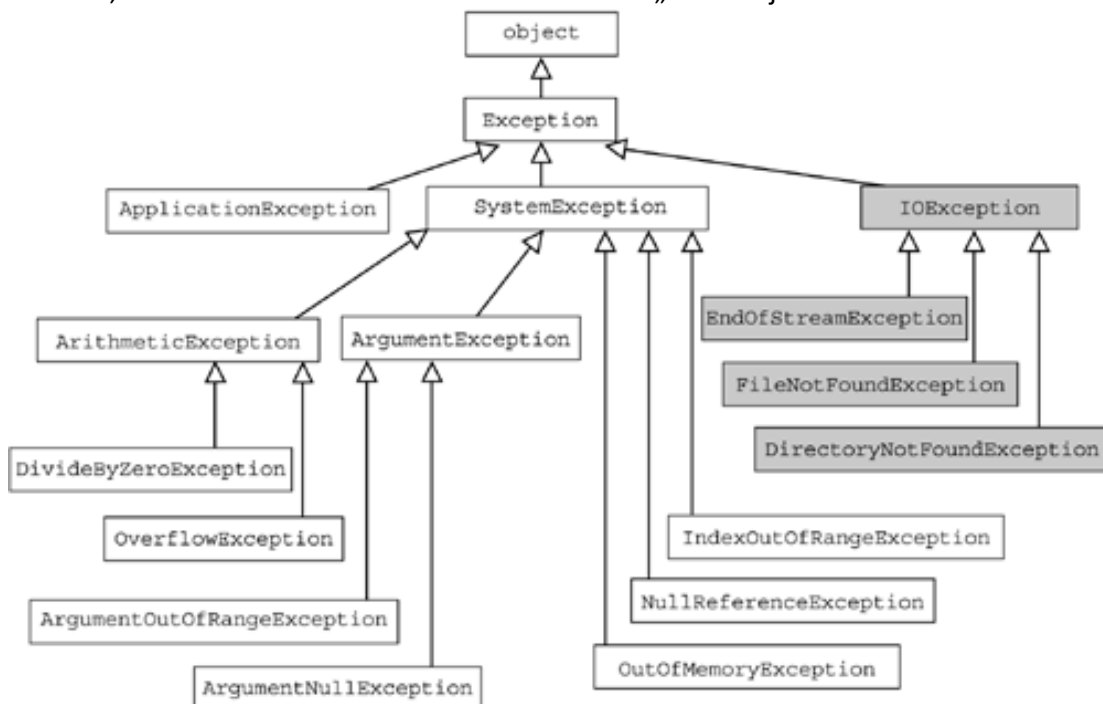
Hibakezelés

try, throw, catch, Exceptions

Abban az esetben eddig, ha egy nem kezelt kivétel (Exception) keletkezett be, akkor kifagyott a programunk és a hibát vagy a visual studio, vagy az operációs rendszer kezelte le, és egy hibaüzenetet dobott. A programunk, ekkor kifagyott. Például a string int-re való parsolásánál, ha nem egész számot adtunk meg, vagy karaktereket is megadtunk a string literálba, akkor FormatException hiba adódott, bár a TryParse erre megoldást ad, de nem minden kivételnek van egy-egy megfelelő, a hibát kezelni képes függvénye. C#-ban a kivétel egy objektum, amit akkor indít útjára egy függvény, ha valamilyen hiba történt, ekkor a programunk hibakezelés nélkül elszáll. A kivételek elkapását és lekezelését a **try-catch blokk** végzi, a **try blokkot védett blokknak** nevezik, itt keletkezhet a kivétel, melyet majd a catch blokk fog majd lekezelni, a catch-nak meg kell adni egy kivétel objektumot, ami utal a hiba típusára, egy try-blokk után több catch-blokk is következhet, hiszen érdemes a különböző kivételekre, különböző módon reagálni, nem csak azt kiírni, hogy Hiba! *Amint egy kivétel keletkezik a try blokkban az utána következő utasítások nem hajtódnak végre!* Nézzünk egy példát: Kérjünk be egy int számot, és kezeljük le az esetleges FormatException-t, mivel a try is egy blokk, el ne felejtjük a bekért változót már deklarálni a try előtt, különben nem tudjuk később használni!

```
int a = 0;
try{
    a = int.Parse(Console.ReadLine());
}
catch (FormatException){
    Console.WriteLine("Hiba - Nem sikerült átalakítani a bevitt karakterláncot!");
}
Console.WriteLine(a);
```

Minden kivétel őse (eredete) az Exception osztály, ha ezt állítjuk be a catch blokkba, akkor bármilyen kivétel is dobódik, minden hiba lekezelésre kerül. A kivételek „családfája”:



Például az `IndexOutOfRangeException` kivétel őse a `SystemException`, így egy `catch(SystemException){ }` blokkal elkaphatjuk ezt a kivételt is.

Ha jobban átgondoljuk nem kezeltük le az `OverflowException` (túlcordulás) kivételt, ha megadunk egy nevet a kivétel objektumunknak, akkor az objektum tartalma lekérhető a **név.Message** metódussal. Csináljuk meg akkor a kiegészített programot, biztosításként csináljunk még egy sima `Exception` ágat.

```
int a = 0;
try{
    a = int.Parse(Console.ReadLine());
}
catch (FormatException){
    Console.WriteLine("Hiba - Nem sikerült átalakítani a bevitt karakterláncot!");
}
catch (OverflowException ofe){
    Console.WriteLine(ofe.Message);
}
catch (Exception e){
    Console.WriteLine("Ismeretlen hiba: " + e.Message);
}
```

Dobhatunk akár saját magunk is egy kivételt, ezzel tudunk reagálni arra az esetekre is, amikor eredetileg nem keletkezne kivétel, de a későbbiek folyamán fontos lenne. Kivételt a `throw`-val tudunk dobni. Például amikor a felhasználótól egy karakterláncot kérnénk be, de csak üres stringet kapunk, ezt az üres karakterláncot nem tudjuk mire felhasználni, ezért egy saját kivételt dobunk, vagy bekérjük a felhasználó keresztnévét, de megad számokat is... Ugyanúgy, mint a rendszerszintű kivételeknél is a `throw` esetén is a rákövetkező utasítások nem hajtódnak végre és a megfelelő `catch` ágra ugrik a vezérlés. Természetesen, ha azt szeretnénk, hogy a kivétel után újra megtudja adni a felhasználó a szöveget, akkor kell egy logikai változó valamint egy `do-while` ciklus. Kivétel dobás általános alakja: **`throw new Kivételneve("hiba üzenet")`**.

```
bool hiba = true;
do{
    string s = "";
    try {
        Console.Write("Kérem a szöveget: ");
        s = Console.ReadLine();
        if (s.Length == 0) throw new Exception("Nem adott meg szöveget");
        hiba = false; //ide csak akkor jut el a vezérlés, ha nem lesz kivétel
    }
    catch (Exception e) {
        Console.WriteLine(e.Message);
    }
} while (hiba);
```

Finally blokk: a `finally` blokkot minden esetben az utolsó blokk-ként kell használni, az itt megadott utasítások **minden esetben** lefutnak, ha keletkezett kivétel, ha nem! Példa:

```
try {
    int a = 10;
    int b = 0;
    Console.WriteLine(a / b);
}
catch (DivideByZeroException) { Console.WriteLine("Nullával osztott!"); }
finally{ //ez az ág mindenképpen lefut!
    Console.WriteLine("Vége");
}
```

Feladat: Csináljunk egy ellenőrzött bekérés függvényt (`TryParse` nélkül) kezeljük le az esetleges kivételeket: `FormatException`, `OverflowException` majd ezt a függvény által visszaadott számnak vegyük a négyzetgyökét (`Math.Sqrt`) vigyázzunk, mert negatív számnak nincs négyzetgyöke így dobjunk saját kivételt a `NotFiniteNumberException` kivétellel és kezeljük le!

Generikusok

lista, láncolt lista, szótár, verem, sor, halmaz

A generikusokat (**using**) `System.Collections.Generic` névtérben találhatjuk, el ne felejtjük beemelni! Kell a new kulcsszó, referencia típusok! A fölöslegessé vált generikus változók felszabadítását a szemétyűjtő (Garbage Collector) végzi. Értékátadáskor referencia másolás történik, lásd: Listák- 31. o.

Listák - List -Generikusok

Remélem, még emlékszünk a tömbökre! Most a lista ismertetése után teljesen elfeledhetjük ezeket (de azért emlékezzünk rá) a buta adatszerkezetet, a tömbnek az volt a legnagyobb hátránya, hogy kötött volt a mérete, miután létrehoztunk egy tömböt és megadtunk neki egy méretet, akkor a méretnél több elemet nem tárolhattunk el benne, hiszen `IndexOutOfRangeException` kivételt dobott, ha olyan cella indexére mutattunk, ami nincs. A lista ezzel szemben egy dinamikus tömb, méretét automatikusan változtatja. A lista egy generikus típus, tehát bármilyen típusból (akár tömbből, osztályból, struktúrából, listából [ennek bemutatása később]...)

létrehozhatunk saját listát. A lista általános alakja: `List<T>` lista = `new List<T>()`, ahol T egy típus (nyilvánvalóan ennek megfelelő elemeket lehet majd a listába felvenni). Kattintsunk bele a `List` szövegbe, majd Visual Studio esetén nyomjuk meg az F12-t, megnyílt maga a beépített List (bal ábra) és benne található a használható metódusokat, a List előtt felfedezhetünk egy class kulcsszót, ez jelenti azt, hogy osztály, ezért kell a

```
namespace System.Collections.Generic
{
    ...public class List<T> : IList<T>, ICollection<T>
    {
        ...public List();
        ...public List(int capacity);
        ...public List(IEnumerable<T> collection);

        ...public T this[int index] { get; set; }

        ...public int Capacity { get; set; }
        ...public int Count { get; }
    }
}
```

```
List<int> lista = new List<int>();
foreach (int item in lista){
    Console.WriteLine(item + ", ");
}
```

példányosításnál a végére () jelek. Ebből rá is jöhetünk arra, hogy a List egy referenciatípus, tehát paraméterátadás esetén cím szerint adódik át (22. o.), ugyanúgy, mint a tömbök. Hozzunk létre egy int típusú

listát, és járjuk végig egy foreach ciklussal, futassuk le, mit kaptunk? Mindenkinek gondolom semmit nem írt ki a képernyőre, tömbök esetében viszont 0-kat írta ki, de ez esetben, ebben a listában valóban nincs semmi, próbáljuk meg pl. a 2. indexű elemét kiírni (`lista[2]`), ez esetben `ArgumentOutOfRangeException` kivételt kaptunk. A lista alapvetően egy tömb háttérű összetett adattípus, alapvetően egy 4db kapacitású tömb jön létre, és ebbe a tömbbe kerül feltöltésre a megadott értékek, ha a kapacitás elfogy, nagyobb lesz az eltárolandó értékek száma, mint a kapacitás, akkor egy új tömb jön létre, kapacitása már az előző duplája lesz és az értékek átmásolódnak ebbe a tömbbe. Ha zárójelben megadunk egy értéket, akkor az a kezdő kapacitás lesz, a kapacitást a `lista.Capacity` metódussal tudjuk lekérni, méretét pedig a `lista.Count` -tal. Új elemet a `lista.Add(érték)` metódussal tudunk felvenni, ami mindig az utolsó helyre szúrja be a megadott értéket, természetesen az értéknek meg kell egyeznie a lista típusával. Nézzünk egy példát: A lista kapacitását 8-ról indítjuk és felveszünk 3db elemet.

```
List<int> lista = new List<int>(8);
lista.Add(12);
lista.Add(43);
lista.Add(2);
Console.WriteLine("Kapacitása: " + lista.Capacity);
Console.WriteLine("Mérete: " + lista.Count);
```

A for-ciklus és a foreach-ciklus is használható az összes elem kiírására, az egyes elemeket ugyanúgy tudjuk elérni, mint tömb esetén (pl.: `lista[1]`, ha min. 2db elem van benne), for esetén a `lista.Count`-ig fut a ciklus. Később a feltöltött listát tudjuk módosítani is, pl. `lista[0] = 14;`

```
List<int> lista = new List<int>();
lista.Add(12);
lista.Add(43);
lista.Add(2);
for (int i = 0; i < lista.Count; i++){
    Console.WriteLine(lista[i]);
}
```

A listákat már kezdő értékekkel is deklarálhatjuk, pl.: `List<int> lista = new List<int>() { 10, 23, 4};`

A Listák metódusai

Nézzünk meg pár hasznos metódust a listák kezelésére, az Add, Capacity és a Count-on kívül:

- **List.Insert(hova, mit)**

A megadott helyre beszúrja a megadott értéket, az ezt követő elemek eggyel jobbra csúsznak, ezért időigényes adatfelvételi mód. Példa: Először hozzáadunk 3db elemet (12, 43, 2) a listához, majd a 1.(43) indexre szúrjuk be a 31-et, ezután a lista így néz ki: 12, 31, 43, 2

```
List<int> lista = new List<int>();  
lista.Add(12); lista.Add(43); lista.Add(2);  
lista.Insert(1, 31);
```

- **List.Remove(mit)**

A megadott értéket kitörli az adott listában, a kitörölt érték valóban eltűnik, így az utána lévő elemek eggyel előrébb csúsznak, szinten időigényes a sok mozgatás miatt. Ha olyan értéket próbálunk kitörölni, ami nincs a listánkban, **nem keletkezik kivétel**. Példa: 10, 23, 4 -> 10, 4

```
List<int> lista = new List<int>() { 10, 23, 4};  
lista.Remove(23);
```

- **List.RemoveAt(index)**

A megadott indexű elemet kitörli a listából, a kitörölt érték valóban eltűnik, így az utána lévő elemek eggyel előrébb csúsznak, szinten időigényes a sok mozgatás miatt. Ha olyan indexű elemet próbál kitörölni, ami nincs, nagyobb az index, mint a lista mérete, akkor **ArgumentOutOfRangeException** kivétel keletkezik. Példánkban a 2. indexű (4) elemet töröljük ki. 10, 23, 4, 30 -> 10, 23, 30

```
List<int> lista = new List<int>() { 10, 23, 4, 30};  
lista.RemoveAt(2);
```

- **List.Clear()**

Törli az adott lista teljes tartalmát és egy teljesen üres listát ad vissza, a lista kapacitását nem befolyásolja, ezért előfordulhat, hogy hiába nincs semmisem a tömbben, mégis nagy helyet foglal a kiürített listánk, a háttértömb mérete miatt!

```
List<int> lista = new List<int>() { 10, 23, 4, 30, 10, 16, 19, 20, 34, 10};  
lista.Clear();  
Console.WriteLine("Törlés után a kapacitása: {0}, mérete:{1}", lista.Capacity, lista.Count);
```

- **List.AddRange(kollekció)**

Képes egy tetszőleges, megfelelő típusú kollekció(tömbök(1dimeziós), halmazok, más listák...) összes tartalmát átmásolni az adott lista végére.

```
int[] tömb = new int[] { 20, 30, 12 };  
List<int> lista = new List<int>();  
lista.AddRange(tömb);
```

- **List.InsertRange(index, kollekció)**

Hasonlóan, mint az AddRange, ez is képes kollekciókat hozzáadni a listához, de a megadott indexű elemtől kerülnek beszúrásra a kollekció elemei.

```
int[] tömb = new int[] { 20, 30, 12 };  
List<int> lista = new List<int>(){ 10, 24, 32};  
lista.InsertRange(2, tömb);
```

- **List.Sort()**

Az adott listát növekvő sorrendbe rendezi, stringek esetében az abc rend szerint rendezi az elemeket, egyes rendszereken a magyar abc-t is támogatja, egyszerűbb típusoknál működik.

```
List<string> lista = new List<string>() { "banán", "alma", "körte", "narancs", "kiwi"};  
lista.Sort();  
foreach (string item in lista){  
    Console.WriteLine(item);  
}
```


- **List.Contains(elem)**

Az adott listában, ha a megadott érték szerepel, akkor true, ha nem akkor false értékkel tér vissza.

```
List<string> lista = new List<string>() { "banán", "alma", "körte", "narancs", "kiwi"};
if (lista.Contains("körte")) Console.WriteLine("Van körte");
```

- **List.IndexOf(keresett elem) /LastIndexOf**

Az adott listában megkeres egy elemet és visszaadja az indexét, ha nincs ilyen elem, akkor -1 ad vissza, a LastIndexOf jobbról kezdi el a keresést.

```
List<string> lista = new List<string>() { "banán", "alma", "körte", "narancs", "kiwi"};
int index = lista.IndexOf("körte");
if (index != -1) Console.WriteLine("Van körte, indexe:" + index);
```

- **T[] = List.ToArray()**

Az adott listát egy tömbbé konvertálja, a tömbbe az összes listában lévő (aktuális) adat átmásolódik.

```
List<int> lista = new List<int>() { 20, 31, 10};
int[] tömb = lista.ToArray();
```

Referencia másolás

Mint már említettem a lista egy referencia típus, tehát cím szerint adódik át, ennek eredményeként, ha egy már meglévő listából szeretnénk létrehozni egy újat, akkor nem a lista értékei, hanem a **címe másolódik** át az újonnan létrehozott listánkba, ezt nevezzünk **értékadáskor** bekövetkező **referencia másolásnak**, így a listákon **külön-külön végzett változtatások hatással lesznek a másik listára** is. Például ha én egy értéket akarok hozzáadni a régi listából létrehozott új listának, akkor a régi listába is fel fog kerülni az adott érték, vagy ha törölöm az egyik lista tartalmát, akkor a másiké is törlődni fog.

```
List<int> régilista = new List<int>() { 20, 31, 10};
List<int> újlista = régilista;
újlista.Add(40);
Console.WriteLine("régilista tartalma: ");
foreach (int item in régilista){
    Console.WriteLine(item + ", ");
}
Console.WriteLine("\nújlista tartalma: ");
foreach (int item in újlista){
    Console.WriteLine(item + ", ");
}
```

```
régilista tartalma: 20, 31, 10, 40,
újlista tartalma: 20, 31, 10, 40,
```

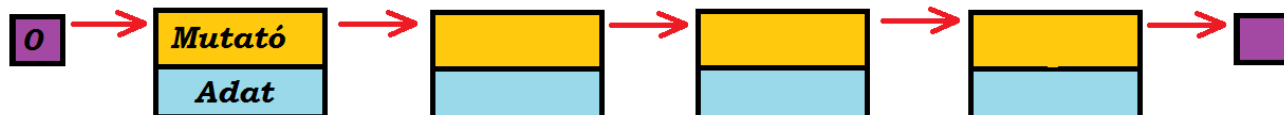
Összetett Listák

Tömbök esetében megtudtuk tenni azt, hogy egy vektoron belül létrehozzunk még tömböket, hát listák esetében se sincs ez másképp, így gyakorlatilag kiküszöbölhető az 1dimenziós határ. Nézzük meg hogy hogyan: A lista deklarációjánál a <> részek közé egy újabb List<T> veszünk fel, és ezután pedig a {} jelek közé új listákat hozunk létre vesszővel elválasztva, új belsőlistát a főlista.Add(new List<T>)- vel tudunk felvenni, a belső listákat az indexük segítségével tudunk elérni, és úgy felvenni bele elemeket. Létrehozhatunk struktúrából, tömbből, osztályból, halmazból is listákat, hasonlóan, mint itt, de azt nem tárgyalom ennél a résznél.

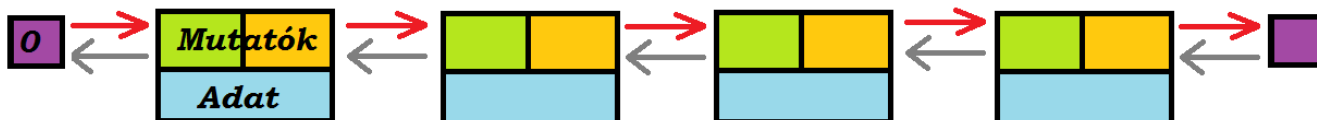
```
List<List<int>> lista = new List<List<int>>() { new List<int>(), new List<int>() };
lista.Add(new List<int>());
lista[1].Add(14);
for (int i = 0; i < lista.Count; i++){ //körbejárása for-ral
    for (int j = 0; j < lista[i].Count; j++){
        Console.WriteLine(lista[i][j]);
    }
}
foreach (List<int> belsőlista in lista){ //körbejárása foreach ciklussal
    foreach (int elem in belsőlista){
        Console.WriteLine(elem);
    }
}
```

Láncolt listák - *LinkedList* -Generikusok

A sima listák esetében az adatokat egy háttértömbben tároljuk el, így ez az adatszerkezet lassabb lesz akkor, ha sok mozgatót végzünk benne, pl. az Insert metódussal új elemeket szúrunk be a lista elejébe, ekkor az azt követő értékeket 1-el arrébb kell tolni, és ez nagyon időigényes. A lassúság kiküszöbölésére hozták létre a láncolt listákat (LinkedList), amit akkor érdemes használni, ha adatstruktúránkban sok mozgató van, fontos megjegyezni, hogy a láncolt lista nem biztos, hogy gyorsabb, mint a sima lista, amíg a listánkban csak a végére szúrunk be elemeket addig teljesen elég. A láncolt listák nagyobb helyet foglalnak, mint a sima elődjeik, a láncolt lista több, egymáshoz hivatkozás által összekapcsolt úgy. csomópontokból épül fel, ebben a csomópontban szerepel egy adat, valamint egy (egyszeresen láncolt lista) vagy kettő (kétszeresen láncolt lista)



hivatkozás, ami az előző és/vagy a következő csomópontra hivatkozik. Egy egyszeresen láncolt (fenn) lista egy null (lila négyzet) értékkel kezdődik, ami az első valódi csomópontra mutat, amiben szerepel a következő csomópontra hivatkozó mutató (sárga) és a csomópontához tartozó érték, adat(kék), a következő csomópont szintén ugyanígy épül fel, a legutolsó 'valódi' csomópont, pedig egy null típusra mutat, így tudhatjuk, hogy vége a láncolt listának.



Kétszeresen láncolt lista esetén 2db hivatkozás (zöld és sárga) van egy-egy csomópontban, az egyik az előző a másik a következő csomópontra mutat, C#-ban kétszeresen láncolt listák vannak.

Hasonlóan, mint a listák, szintén osztályból vannak létrehozva és ezért referencia típus, természetesen referencia másolás történik értékadásnál.

Mint a listák esetében is a LinkedList beírása után a <> jelek közé kerül a láncolt listánk típusa és a megszokott név, egyenlőségjel a new operátor valamint újra a LinkedList és el ne felejtjük a zárójeleket!

Példa: `LinkedList<string> láncoltlista = new LinkedList<string>();`

Ezután a `LinkedList.AddLast(érték)` metódussal tudunk a listánk végére beszúrni egy elemet, vagy például az `AddFirst(érték)` metódussal pedig a lista első helyére. Mit ír ki az alábbi program, milyen sorrendben?

```
LinkedList<string> láncoltlista = new LinkedList<string>();
láncoltlista.AddLast("vár");
láncoltlista.AddLast("kávé");
láncoltlista.AddFirst("autó");
foreach (string item in láncoltlista){
    Console.WriteLine(item);
}
```

A láncolt listák fontos metódusai

- **LinkedList.RemoveLast/First**

Töröli a láncolt lista utolsó vagy első elemét.

```
LinkedList<string> láncoltlista = new LinkedList<string>();
láncoltlista.AddLast("vár");láncoltlista.AddLast("kávé");láncoltlista.AddFirst("autó");
láncoltlista.RemoveFirst();
```

- **LinkedListNode<T> = LinkedList.Last/First**

Visszaadja egy adott listában szereplő első vagy utolsó csomópontot, amelyben megtalálhatjuk a következő csomópontra mutató hivatkozást. A `LinkedListNode`-ról bővebben később.

```
LinkedList<string> láncoltlista = new LinkedList<string>();
láncoltlista.AddLast("vár");láncoltlista.AddLast("kávé");láncoltlista.AddFirst("autó");
LinkedListNode<string> csomópont = láncoltlista.First;
```

- **LinkedList.AddAfter(LinkedListNode, érték)/AddBefore**

Egy adott listacsomópont mögé vagy elé szúr egy másik csomópontot, amelyben a megadott érték szerepel, és az új adat beépül a láncolt lista szerkezetébe.

```
//A láncoltlista azonosítójú LinkedList már létre van hozva, elemei: autó, vár, kávé
LinkedListNode<string> csomópont = láncoltlista.First;
láncoltlista.AddAfter(csomópont, "2.hely");
```

- **LinkedListNode<T> = LinkedList.Find(keresett érték)**

Egy LinkedList-ben visszaadja a megadott értékhez tartozó csomópontot.

```
//A láncoltlista azonosítójú LinkedList már létre van hozva, elemei: autó, vár, kávé
LinkedListNode<string> kerescsomópont = láncoltlista.Find("vár");
```

További metódusok, melyek ugyanúgy működnek, mint a listák (30 - 31. oldal) hasonló nevű metódusai: Clear, Remove, Count, Contains, viszont kapacitás nincs, mivel csomópontok szolgáltatják az adatszerkezetet, nem tömbök!

A Csomópontok - LinkedListNode -Generikusok

Egy listából létrehozhatunk egy-egy csomópontot, melyben az adott csomópontoz tartozó érték és a következő csomópont mutató hivatkozás van. A következő csomópontoz való ugráshoz a **.Next** metódust kell használnunk, hogy visszafelé közlekedjünk pedig a **.Previous** metódust kell használnunk. A **.Value** metódus az adott csomópontban eltárolt adatot adja vissza. Mivel tudjuk, hogy a láncolt listák utolsó, illetve első eleme null értékű, így egy while ciklussal is végig tudunk menni az adott láncolt listán, a léptetésről a Next gondoskodik.

```
LinkedList<string> láncoltlista = new LinkedList<string>();
láncoltlista.AddLast("vár"); láncoltlista.AddLast("kávé"); láncoltlista.AddFirst("autó");
LinkedListNode<string> csomópont = láncoltlista.First;
while (csomópont != null){
    Console.WriteLine(csomópont.Value);
    csomópont = csomópont.Next;
}
```

Szótár - Dictionary -Generikusok

Előfordulhat olyan eset, amikor egy karakterláncot meg kell feleltetnünk egy másik karakterláncnak, ilyenkor tipikusan a szótár adatszerkezetet kell használnunk. Ezek a szótárak felépítése felfogható két egymással kapcsolatban álló listaként, az egyik listában vannak a kulcsok a másik listában, pedig a kulcshoz tartozó megfelelő érték, ezért szótárak esetén két db típus kell megadnunk a <> jelek között, vesszővel elválasztva. Az első a kulcs típusa lesz a második a kulcshoz tartozó érték típusa, természetesen nem muszáj két db string kulcs-értékpárt adni, lehet akár int és string kulcs-értékpár is. Deklarációkor is megadhatunk már adatokat így:

```
Dictionary<string, string> szótár = new Dictionary<string, string>(){
    {"piros", "red"}, {"kék", "blue"} //...
};
```

Természetesen később is adhatunk hozzá kulcs-értékpárt, az **Add** metódussal: `szótár.Add("zöld", "green");` Mint már említettem a szótárban kulcs-értékpáros (KeyValuePair) tárolódik el, így egy foreach ciklusnál ezt kell megadnunk, valamint a kulcs és az érték típusát, ezután a **KeyValuePair** típusú változóban lesz eltárolva ezek az adatok. A kulcsot a **.Key** metódussal tudjuk elérni, míg az értéket a **.Value** metódussal. A szótár végigjárása:

```
foreach (KeyValuePair<string, string> item in szótár){
    Console.WriteLine("Kulcs: {0}, értéke: {1}", item.Key, item.Value);
}
```

A szótár fontos metódusai:

- **ContainsKey(keresett kulcs)/ ContainsValue(keresett érték)**

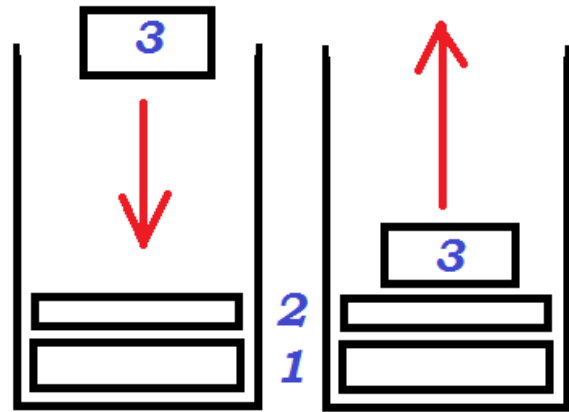
Tartalmazás vizsgálat, ha az adott kulcs(Key) vagy érték(Value) szerepel a szótárban true, hanem false ad vissza, a ContainsKey csak a kulcsok között, a ContainsValue csak az értékek között keres.

```
if (szótár.ContainsKey("zöld")) Console.WriteLine("Van zöld");
else Console.WriteLine("Nincs zöld");
```

További metódusok: Count, Remove (csak kulcs alapján), Clear. Ezek megtalálhatók a listáknál, 30-31. oldal.

Verem - Stack -Generikusok

Bonyolult lenne olyan listát kreálni, ami a **Last in first out** (LIFO) elvet követi, tehát amit utoljára beletettünk értéket azt tudjuk kivenni az elsőnek, az alatta lévő elemeket pedig csak akkor érjük el amikor már a fölötte lévő elemeket kipakoltuk. Az ilyen problémákra találták ki a vermet, bár nem foglyuk annyiszor használni, mint a listát, vagy majd a később megismert halmazt, de azért ismerkedjünk meg a használatával. Hasonlóan, mint a többi generikus, itt is hasonlóelven kel létrehozni, de nem az Add metódussal tudunk értéket hozzáadni, hanem a **.Push(érték)** metódussal, nézzünk egy példát, a veremhez 1-4-ig adunk hozzá elemet, és ezután egy foreach ciklussal végigjárjuk:



```
Stack<int> verem = new Stack<int>();
verem.Push(1); verem.Push(2); verem.Push(3); verem.Push(4);
foreach (int item in verem){
    Console.WriteLine(item);
}
```

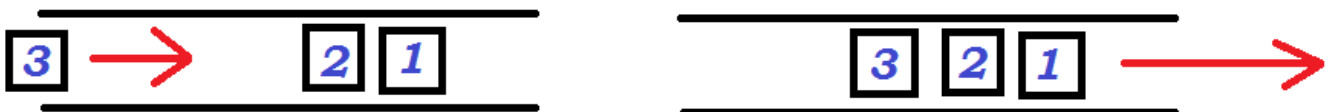
Ha kiíratunk az adatokat láthatjuk, hogy pont fordítva írta ki őket, mint ahogy hozzáadtuk, hiszen a verem csak mindig az utolsó elemet képes kivenni, és csak úgy tudja elérni a lejjebb lévő elemeket, ha a fölötte lévő elemeket már eltávolította, ezért írta ki fordítva a ciklus a benne tárolt értékeket. **A verem egyszerre csak az utolsó benne foglalt elemet tudja kiolvasni!** Nézzük meg, hogy hogyan tudunk kiolvasni belőle, ezt a **.Pop()** metódus végzi, mely visszaadja mindig az utolsó elemet, és ugyanakkor ki is törli belőle. Így felhasználható egy while ciklusban is, a verem méretét megadó Count segítségével:

```
while (verem.Count != 0){
    Console.WriteLine(verem.Pop());
}
```

Használható továbbá a listánál már megszokott alábbi metódusok: *Clear*, *Contains* és a *ToArray*, ezek megtalálhatók listáknál (30-31.o.).

Sor - Queue -Generikusok

Ha **First in first out** (FIFO) elvet valósítunk meg, tehát ami először jön, az először megy, a sor adatszerkezetet kell használnunk. Itt mindig a sorhoz elsőként hozzáadott érték kerül ki elsőnek. Szintén generikus típus a



Queue. Sorhoz a **.Enqueue(érték)** metódussal tudunk adatot felvenni, kivenni pedig a **Dequeue()** metódussal, mely visszaadja az éppen aktuális első értéket és ki is törli a sorból. Ugyanúgy, mint a veremnél a **belső elemeket nem tudjuk elérni**, csak akkor tudjuk elérni, ha a mellette levő elemeket eltávolítottuk. Így a Dequeue felhasználható arra is, hogy egy while ciklussal körbejárjuk és közben kis is töröljünk a tartalmát:

```
Queue<int> sor = new Queue<int>();
sor.Enqueue(1); sor.Enqueue(2); sor.Enqueue(3); sor.Enqueue(4);
while (sor.Count != 0) {
    Console.WriteLine(sor.Dequeue());
}
```

Természetesen egy foreach ciklussal is végigjárhatjuk, ami nem módosítja a tartalmát:

```
foreach (int item in sor) {
    Console.WriteLine(item);
}
```

Használható továbbá a listánál már megszokott alábbi metódusok: *Clear*, *Contains* és a *ToArray*, ezek megtalálhatók listáknál (30-31.o.).

Halmazok - HashSet -Generikusok

A halmazok olyan listák, amelyekben azonos értékű elemből csak egy szerepelhet, szintén generikusak. Csakúgy, mint listáknál halmazok esetében is a **Add(érték)** metódussal tudunk új adatot hozzáadni, viszont ha egy olyan értéket próbálunk újból **hozzáadni, amit már tartalmaz**, akkor **nem adódik hozzá és nem is keletkezik kivétel**.

```
HashSet<int> halmaz = new HashSet<int>() { 17, 10, 20, 43};  
Console.WriteLine("Hozzáadás előtt: " + halmaz.Count);  
halmaz.Add(17);  
Console.WriteLine("Hozzáadás után: " + halmaz.Count);
```

Így nem változik meg a mérete se a halmaznak, marad ugyanannyi, mint előtte. Ezt könnyen feltudjuk használni arra, hogy randomoláskor ne legyen még egy olyan szám, amit már generáltunk egyszer. Például számokat szeretnénk randomolni az ötös lottóhoz, lottó esetén nem lehetnek azonos számok. Mivel tudjuk, hogy ha olyan számot generálunk, ami már szerepelt, akkor nem növekszik a halmaz mérete. Példa:

```
Random r = new Random();  
HashSet<int> lottószámok = new HashSet<int>();  
while (lottószámok.Count < 5) {  
    lottószámok.Add(r.Next(0, 91));  
}  
Console.WriteLine("A sorsolt lottószámok: ");  
foreach (int item in lottószámok) {  
    Console.Write(item + ", ");  
}
```

Sima tömb, vagy lista esetén előfordulhattak volna azonos számok is, de halmaz esetén ez nem következhet be.

A halmaz fontos metódusai:

- **UnionWith(halmaz)**

A megadott halmaz és annak a halmaznak az unióját adja vissza, amelyikből meghívták, ezután a meghívott metódushoz tartozó halmazban lesz eltárolva az unió elemei, természetesen, ha valamelyik érték szerepel az egyik és a másik halmazban is, akkor csak egyszer adja hozzá. $unio = unio \cup halmaz1$

```
HashSet<int> unio = new HashSet<int>() { 10, 32, 4, 8};  
HashSet<int> halmaz1 = new HashSet<int>() { 20, 32, 12, 4};  
unio.UnionWith(halmaz1);
```

- **SymmetricExceptWith(halmaz)**

Visszaadja a metódust meghívó halmazba a halmazok metszetén kívüli elemeket, tehát a két halmaz metszetének komplementerét. $alaphalmaz = (alaphalmaz \cup halmaz1) \setminus (alaphalmaz \cap halmaz1)$, vagy $alaphalmaz$ és $halmaz1$ metszetének komplementere

```
HashSet<int> alaphalmaz = new HashSet<int>() { 10, 32, 4, 8};  
HashSet<int> halmaz1 = new HashSet<int>() { 20, 32, 12, 4};  
alaphalmaz.SymmetricExceptWith(halmaz1);
```

- **ExceptWith(halmaz)**

Kivonja a meghívott halmazból a megadott halmaz elemeit, a megmaradt értékeket a meghívott halmazba tárolja el. $alaphalmaz = alaphalmaz \setminus halmaz1$

```
HashSet<int> alaphalmaz = new HashSet<int>() { 10, 32, 4, 8};  
HashSet<int> halmaz1 = new HashSet<int>() { 20, 32, 12, 4};  
alaphalmaz.ExceptWith(halmaz1);
```

- **IntersectWith(halmaz)**

A megadott két halmaz metszetét adja vissza. $alaphalmaz = alaphalmaz \cap halmaz$

```
HashSet<int> alaphalmaz = new HashSet<int>() { 10, 32, 4, 8};  
HashSet<int> halmaz1 = new HashSet<int>() { 20, 32, 12, 4};  
alaphalmaz.IntersectWith(halmaz1);
```

Használható továbbá a listánál már megszokott alábbi metódusok: *Clear*, *Contains* és a *Remove*, ezek megtalálhatók listáknál (30-31.o.).

ArrayList- Sorlista - kollekció

A sorlistában eltérő típusú adatokat tudunk felvenni, a System.Collections névtérben találhatjuk, referencia típus, mivel minden féle típust feltudunk bele venni, ezért nem kell megadnunk semmilyen típusra utaló jelzőt. Mivel minden típus eltárolható benne, ezért ciklusoknál **var** típust kell használnunk, hogy minden változónál működjön. Az **Add** metódussal tudunk új adatot bevinni, továbbá használhatók az alábbi metódusok: *AddRange, Capacity, Clear, Count, Contains, IndexOf, Insert, InsertRange, Sort, (listák, 30-31)* valamint `pl.:sorlista[0].GetType()` visszaadja a típusát.

```
ArrayList sorlista = new ArrayList();
sorlista.Add("string"); sorlista.Add(7); sorlista.Add('c');
foreach (var item in sorlista){
    Console.WriteLine(item); }
```

Alapszintű fájlkezelés

System.IO, StreamWriter, StreamReader

Mivel a programunkban létrehozott változók csak a program futása alatt élnek, a program bezárása után, már nem lesz lehetőségünk folytatni ott, ahol abbahagytuk, ennek kiküszöbölése végett a fájlkezelést kell alkalmaznunk, a felhasználó által bevitt adatokat lementjük, majd beolvassuk a program futása közben. Hogy feltudjuk használni az ezekhez szükséges metódusokat **be kell emelnünk az IO névtérre** (`using System.IO`).

Írás - StreamWriter

Ahhoz, hogy írni tudjunk egy fájlba a **StreamWriter** osztályt kell használnunk és meg kell adnunk egy elérési utat és egy fájlnevet, amibe majd írni fogunk. Abban az esetben, ha nem adunk meg elérési utat az exe futtatható állományunk mellé kell tenni az adott fájlt, az esetben, hogy ha **nincsen meg a megadott fájl, akkor létrehozza**. Ha mi hozzuk létre a fájlt, UTF-8 karakterkódolást állítsunk be neki. A fájlneve mellett meg kell adnunk a kiterjesztést is, kezdetben elég a txt formátum. Miután példányosítottuk a StreamReader-t és meg is adtunk neki egy fájlt, a nevével már el is kezdhetjük a fájlba írás a `sw_neve.Write/ WriteLine(karakterlánc)` metódussal, ugyanúgy működnek, mint a Console osztály hasonló nevű metódusai a Write nem tesz sortörést, viszont a WriteLine igen. Miután befejeztük az adatfolyamot (stream-et), ki kell üríteni a puffert, mert ezek a metódusok egy pufferba írnak, és egy meghatározott időközönként ürítik csak ki ezeket az ideiglenes tárolókat, így nem biztos, hogy a metódusok lefutása után már minden adat benne lesz a fájlunkba. A puffer tartalmát a `sw_neve.Flush()` metódussal tudjuk kiüríteni a fájlba, ezután pedig le kell zárunk az adatfolyamot az `sw_neve.Close()` metódussal. Nézzünk egy példát:

```
StreamWriter sw_gyumolcsok = new StreamWriter("gyümölcsök.txt");
sw_gyumolcsok.WriteLine("alma"); //írás
sw_gyumolcsok.WriteLine("banán");
sw_gyumolcsok.WriteLine("ananász");
sw_gyumolcsok.WriteLine("körte");
sw_gyumolcsok.Flush(); //puffer kiürítése
sw_gyumolcsok.Close(); //adatfolyam lezárása
```

Ebben az esetben a .exe kiterjesztésű állományunk mellé hozta létre a fájlt a program (általában a bin\debug mappába), és 4db gyümölcsnevet írt bele külön-külön sorokba, ennek a fájlnak kódolása UTF-8-as így támogatja a magyar ékezetes betűket. Nézzünk meg már egy elérési úttal rendelkező fájl írását, mivel a \ jel speciális karakter ezért @módosítót kell használnunk a mappák megadásánál. Például itt a D merevlemez meghajtó, saját, azon belül is a fájlok mappába jön létre az alábbi szöveges állomány:

```
StreamWriter sw_elérésiúttal = new StreamWriter(@"D:\Saját\Fájlok\szöveg.txt");
sw_elérésiúttal.WriteLine("Elérési úttal rendelkező szöveges állomány");
sw_elérésiúttal.Flush(); sw_elérésiúttal.Close();
//el ne felejtsük a puffert kiüríteni és a fájlt lezárni!
```

Hogyha újra lefuttatjuk ezt a programot és más adatot írunk ki, akkor láthatjuk, hogy az előzően kiírt adat eltűnt, **ha egy létező állományba akarunk írni, akkor az állománytartalma kitörlődik** és bele íródik az új adat. Abban az esetben, hogyha valamilyen hiba lépne fel a fájlfolym közben, valamilyen **IOException** adódik, pl. tipikus hibaszokott lenni a rossz mappanév, ekkor például `DirectoryNotFoundException`-t kapunk. Javítsuk ki a programunkat hibakezeléssel az alábbi módon:


```
try {
    StreamWriter sw_elérésiúttal = new StreamWriter(@"D:\Saját\Fálok\szöveg.txt");
    sw_elérésiúttal.WriteLine("Elérésiúttal rendelkező szöveges állomány");
    sw_elérésiúttal.Flush(); sw_elérésiúttal.Close();
}
catch (DirectoryNotFoundException) {
    Console.WriteLine("Nem található egy mappa, valószínűleg rossz elérési utat adott meg!");
}
catch (IOException){
    Console.WriteLine("Hiba lépett fel a fájl írása közben!");
}
}
```

Olvasás – StreamReader

Fájlból való kiolvasáshoz a **StreamReader** osztályt kell használnunk, szintén meg kell adni egy elérési utat, valamint magát a fájlnevet, fontos hogy a fájl **UTF-8** kódolással legyen elmentve, különben nem tudunk ékezetes betűket olvasni. Abban az esetben, ha a fájl nem létezik **FileNotFoundException** kivétel dobódik. A **ReadLine()** metódussal egy darab sor tartalmát tudjuk beolvasni, a ReadLine-hoz tartozik egy mutató, abban az esetben, ha még egyszer kiadjuk ezt az utasítást már a 2. sort olvassa be és így tovább. Az **EndOfStream** azt adja vissza, hogy mikor van vége a fájlnek, ha a fájl végére értünk true, különben false értéket ad, így feltudjuk használni arra, hogy beolvassa egy while ciklussal egy fájl összes sorát: Példánkban az asztalon lévő fájlból olvasunk, és a sorok tartalmát egy List-ben tároljuk el.

```
StreamReader sr = new StreamReader(@"C:\Users\zolcsyx\Desktop\nevek.txt");
List<string> nevek = new List<string>();
while (!sr.EndOfStream) {
    nevek.Add(sr.ReadLine());
}
foreach (string item in nevek) {
    Console.WriteLine(item);
}
sr.Close();
```

Példa kivételkezeléssel:

```
try {
    StreamReader sr = new StreamReader(@"C:\Users\zolcsyx\Desktop\nevek.txt");
    List<string> nevek = new List<string>();
    while (!sr.EndOfStream) {
        nevek.Add(sr.ReadLine());
    }
    foreach (string item in nevek) {
        Console.WriteLine(item);
    }
    sr.Close();
}
catch (FileNotFoundException) { Console.WriteLine("Nem található a fájl!"); }
catch (IOException) { Console.WriteLine("Hiba keletkezett a fájl olvasása közben!"); }
```

Struktúrák

struct, mezők, konstruktor, metódusok

A struktúrák olyan adategységek, amelyekben különböző típusú adatokat foglalhatunk egy adatszerkezetbe. Csakúgy, mint a felsorolásnál, a struktúrát is az alprogramokon kívül kell létrehozni, C# esetén a **struct** kulcsszóval, ezután következik a neve, majd a kapcsos zárójelek között a magadott **mezők public láthatósággal** (OOP). Példánkban egy Diák struktúrát hozunk létre, név, életkor és lakhely mezőkkel:

```
struct Diák {
    public string név;
    public int életkor;
    public string lakhely;
}
//Main, egyéb eljárások
```

Miután ezt megtettük létre kell hoznunk egy példányt a Diák struktúrából az egyik alprogramban, és a nevére, valamint a mezőkre hivatkozva megadhatjuk neki az adattagjait, és le is kérhetjük:

```
static void Main(string[] args) {
    Diák d = new Diák();
    d.név = "Péter";
    d.lakhely = "Nyíregyháza";
    d.életkor = 15;
    Console.WriteLine("Neve: " + d.név);
}
```

Egy struktúrában megadhatunk **eljárásokat és függvényeket is, ezeket nevezzük metódusoknak**, például egészítsük ki, a Diák struktúrát egy Kiír() nevű eljárással, ami kiírja az egyén adatait és jegyeit, hozzunk létre még egy listát benne, amibe majd a diák jegyeit adhatjuk meg, valamint egy átlag függvényt, ami visszaadja a diák jegyeinek átlagát. A **struktúra deklarációjánál automatikusan lefutó metódust konstruktor**nak hívjuk, konstruktor eljárást a struktúra nevével azonos nevű metódussal tudunk csinálni, a konstruktorban példánkban így megadhatjuk a deklarációkor már az egyes mezőket paraméterezéssel, a konstruktorban fel kell használni az összes mezőt! A **this** szó, a **struktúrán belüli mezők azonosítására** szolgál, a régebbi keretrendszerekben kötelező volt a használata minden esetben, jelenleg elég akkor alkalmaznunk, amikor 2db azonos változónév van egy struktúrán belül, példánkban egy mezőnévként és egy paraméternévként szerepel a „név” szó. A this.név a struktúra mezőjére, míg a sima név a kapott paraméterre hivatkozik.

```
struct Diák {
    public string név; //<- erre mutat a this
    public int életkor;
    public string lakhely;
    public List<int> jegyei;
    public Diák(string név, int k, string h) {
        this.név = név; this.életkor = k; this.lakhely = h; this.jegyei = null;
        //^ mivel kötelező az összes adattagot beállítani ezért a jegyei null
    }
    public void Kiír() {
        Console.WriteLine("Név: " + this.név);
        Console.WriteLine("Életkor: " + this.életkor);
        Console.WriteLine("Lakhely: " + this.lakhely);
        Console.WriteLine("Jegyei: ");
        foreach (int item in this.jegyei) {
            Console.Write(item + ", ");
        }
        Console.WriteLine("\nA diák átlaga: " + Átlag());
    }
    public double Átlag() {
        int összeg = 0;
        foreach (int item in this.jegyei) {
            összeg += item;
        }
        return összeg / (double) this.jegyei.Count;
    }
}

static void Main(string[] args) {
    Diák d = new Diák("András", 17, "Sopron");
    d.jegyei = new List<int>() { 4, 5, 3, 5, 5, 4 };
    Console.WriteLine("A diák adatai:");
    d.Kiír();
    Console.ReadKey();
}
```

A **struktúra egy értéktípus, tehát értékadáskor csak a mezők értékei másolódnak át** (ellenben például a lista, aminél referencia másolás történik, lsd. 31.o.), így az egyes példányokon belül végzett módosítások nem hatnak a másik változóra. Egy struktúrából létrehozhatunk akár egy listát vagy tömböt is. Példa a listára és tömbre:

```
List<Diák> diákok = new List<Diák>();
Diák d = new Diák("András", 17, "Sopron");
diákok.Add(d); //vagy rögtön: diákok.Add(new Diák("András", 17, "Sopron"));
//vagy tömb esetén:
Diák[] diákokT = new Diák[10];
diákokT[0] = d;
```

Egyéb operátorok, bit műveletek

csonkolt operátorok, háromoperandusú operátor, bit operátorok

A **csonkolt logikai operátorok** (|, &) ugyanarra valók, mint a rendes társaik, csak hogy a csonkolt operátorok a **teljes feltételt kielemezik**, ellenben a sima logikai operátorokkal. Például a teljes vagy (||) operátor esetén már akkor igaz kiértékelést kapunk, ha az első vizsgált elem is már true, hiszen fölösleges megnézni a második operandust is, ezzel szemben a csonkolt operátor a teljes feltételt kiértékeli, hogy ez akkor miért is kell (?), jó kérdés, valószínűleg ez volt az eredeti vagy operátor, és később jelent meg a párja, csak aztán megmaradt.

```
if (false && true) Console.WriteLine();
else //rögtön az else ágra ugrik
if (false & true) Console.WriteLine();
else //hiába lesz már false, a második operandust is vizsgálja
```

A **háromoperandusú operátor**: C#-ban egyetlen háromoperandusú (úgy. ternáris) operátor van, ami egy if-else ág helyettesítésére szolgál, mivel háromoperandusú, ezért három értéket kell megadni a **? : operátornak**, szintaktikája: *feltétel ? ha igaz : ha hamis*. Például döntsük el egy számról, hogy pozitív vagy negatív:

```
int szám = int.Parse(Console.ReadLine());
//if-else -vel:
if (szám < 0)
    Console.WriteLine("Negatív");
else
    Console.WriteLine("Pozitív");
//A ternáris operátorral:
Console.WriteLine(szám < 0 ? "Negatív" : "Pozitív");
```

Nem csak ilyen esetekben használhatjuk a **? : operátort**, nézzünk egy abszolút érték függvényt:

```
static int AbszolútÉrték(int szám) {
    return szám > 0 ? szám : szám * -1;
    //feltétel      ^ha nem, akkor a -1-szeresét
    // ^ha igaz, önmagát adjuk vissza
}
```

Műveletek bitekkel, további operátorok:

Ha jobban belemélyedünk majd később az informatikába, és hálózatokkal is majd foglalkozunk, lehet, hogy nem lesz baj, ha ezeket megismerjük, emellett a bináris fájloknál sem baj, ha egy kicsit ismerkedünk vele.

Átváltás decimális (10-es) és bináris (2-es) számrendszerek között:

A legkönnyebb mód (szerintem), ha felírjuk a 2 többszöröseit, ameddig el nem érünk a számunknál nagyobb hatványig. Ezután megkeressük melyik az a szám, ami még belefér az adott számunkba, oda írunk egy egyest, majd levonjuk ezt a számot az eredeti számunkból, ha a következő hatvány nem fér el a maradékba, akkor 0 írunk, különben 1-et és újra levonjuk az adott hatványt... Nézzünk egy példát: a decimális számunk a 13:

Hatványok:	8	4	2	1
Bináris:	1	1	0	1
Maradék	13-8=5	5-4=1	1-2=-1	1-1=0 :)

Így a 13 binárisban: 1101, jóval egyszerűbben tudunk binárisból decimálisba átváltani, ehhez is írjuk fel az itt látható táblázatunkat és írjuk be a megfelelő helyre a biteket, ezután, ahol 1-et láttunk azoknak a

hatványait adjuk össze, ellenőrizzük le, hogy az 1101 valóban 13: $8 + 4 + 0 * 2 + 1 = 13$, tehát jó! C#-ban nagyon egyszerűen a `Convert.ToString(szám, 2)` metódussal tudunk decimálisból binárisba váltani

Bitenkénti és-&, vagy-|, kizáró vagy-^, valamint tagadás-~

A bitoperátorokat is a bitek közti műveletekre használhatjuk, két szám bitenkénti és kapcsolata után az eredmény az adott számok bináris alakban felírt megfelelő bitek és kapcsolata után kialakult szám lesz, példa:

```
byte a = 13, b = 7;
Console.WriteLine("a: " + Convert.ToString(a, 2)); //1101
Console.WriteLine("b: " + Convert.ToString(b, 2)); //0111
int c = a & b;
Console.WriteLine("c: " + Convert.ToString(c, 2)); //0101
```

Ugye ha $1 \& 1 = 1$, de ha csak az egyik is már 0, akkor az eredmény is 0, ugyanígy használható a többi bit operátor, a tagadás csak egyoperandusú, és vigyázzunk vele, mert a nem megjelenített 0-kat is megváltoztatja, így pl. a `byte(8bit) ~10110 = 11101001`, az alábbi táblázatokban az egyes eredmények láthatóak, igazságtábla:

Tagadás – Not Negáció		Képooperandusú	És – And Konjukció			Kétoperandusú	Vagy – Or Diszjukció			Kétoperandusú	Kizáró vagy – Xor Antivalencia			
a	~a		a	b	a & b		a	b	a b		a	b	a ^ b	
1	0		1	1	1		1	1	1		1	1	0	
0	1		0	1	0		0	1	1		0	1	1	
Egyoperandusú			1	0	0		1	0	1		1	1	0	1
			0	0	0		0	0	0		0	0	0	0

Bit eltolásra a >> és a << operátor használható, a >> jobbra tolja el, 1-szeres eltolás esetén a bitsorozatunk elé egy darab 0 szűrődik be, és az utolsó bit kitörlődik (vehetjük úgy is, hogy a semmibe tolódik el). Ha a balra tolás << operátort használjuk, akkor a bitsorozatunk mögé fog egy darab 0-ás beszűrődni és az előtte lévő számok pedig balra elcsúsznak, tolhatunk akár 2, 3 ... léptékkel is, szintaktikájuk: *eltolandó szám >> mennyit*

Például: `int c = a << 2; //2-vel eltolja balra az a változót`

```
Console.WriteLine("a: " + Convert.ToString(a, 2)); //1101 = 13 dec
int c = a >> 2;
Console.WriteLine("c: " + Convert.ToString(c, 2)); //0011 = 3 dec
```

A dinamikus típus – var

A C#3.0 verziótól bevezettek egy általános típust, amikor egy változónak a **var** azonosítót adjuk, akkor a **fordítóra bízunk az adott változó típusának meghatározását**, ezért muszály deklaráció során értéket adnunk a változónknak, különben hibát kapunk, természetesen, hogyha egy var típusú változónak szám értéket adunk meg, akkor a fordító egészszámként értelmezi, így csak az egész számokat adhatunk meg módosításkor. Ha visualstudióban rávisszük a kurzort az adott változóra kiírja, hogy milyen típusként fogja értelmezni a fordító.

```
var szám = 2; //int
var str = "string"; //string
var valami; //hibás, hiszen nincs kezdőértéke
str = 3; //hibás
szám = 17; //jó
```

A var-t használhatjuk akár struktúrák, listák, osztályok példányosítására is, én azt javaslom, hogy ne használjuk a var azonosítót, mert így nem látszik rögtön az adott változó szerepe, továbbá fordításkor időt pazarlunk vele.

```
var Struct = new Struktúra();
var lista = new List<string>();
```

A goto utasítás

A goto meghívását követően a vezérlés arra a pontra ugrik, ahol a meghíváskor megadott címke/azonosító van megadva, ez lehet a meghívás után (ebben az esetben átugrunk egyéb utasításokat), vagy előtt (ismétlés, mint egy ciklus gyakorlatilag). Formája: **goto ugrás; utasítás1; utasítás2; ... ugrás: utasítás** Ebben az esetben a goto miatt átugrotta a vezérlés az utasítás1-et és 2-őt. Csinálhatunk akár így ciklusokat is az alábbi módon:

```
int i = 0;
vissza: Console.WriteLine("Az i értéke: " + (++i));
if (i < 10) goto vissza;
```

Egy goto-hoz csak egy ugrás tartozhat, így nem lehet megadni többször ugyanazt a címkét. **Fontos: a goto nem része a struktúrált programozásnak, mert átláthatatlanná teszi a kódot, NE HASZNÁLJUK!**

C# Programozás

II. rész – OOP

OOP – Object-Oriented Programing

alapelv, felépítése, példa

Az objektum-orientált programozás gyakorlatilag egy programozási módszertan, a valós világ modellezésén alapul. Egy egy osztályt (class) többször is felhasználhatunk, így kevesebb időt vesz igénybe maga a fejlesztés, de az objektum-orientált program lassabban fut (elhanyagolható különbség). A OOP-ben az összetartozó adatokat és az azokkal műveleteket végző eljárásokat vagy függvényeket egy egységbe, másnéven osztályba szervezzük. Egy osztályt tekinthetünk egy mintának, tervrajznak is melyből példányokat tudunk létrehozni, ez lesz egy-egy objektum, amit az adott osztály alapján példányosítunk. Egy objektum változóit mezőnek, adattagnak vagy tulajdonságnak (attributum) hívjuk, az objektumhoz tartozó eljárásokat, függvényeket pedig metódusnak. Kezdetben nagyon hasonlít a struktúrákra, viszont a struktúra érték, míg egy osztály referenciatípus. Nézzünk egy példát, csináljunk egy Ember osztályt, melynek attribútumai: név, életkor, lakhely és munkahely. Csináljunk hozzá egy Kiír nevezetű metódust, mely kiírja egy adott emberpéldány adatait! Egy egy osztályt a class Program kívül kell létrehozni: **class** *Osztályneve* { *láthatóság típus mezőnév*;... *láthatóság eljárástípus metódusnév*(){}... }. Úgy mint a struktúráknál public láthatóságot adjunk meg, hogy miért majd később!

```
class Ember {
    public string Név;
    public int Életkor;
    public string Lakhely;
    public string Munkahely;
    public void Kiír() {
        Console.WriteLine("Név: " + Név);
        Console.WriteLine("Életkor: " + Életkor);
        Console.WriteLine("Lakhely: " + Lakhely);
        Console.WriteLine("Munkahely: " + Munkahely);
    }
}
```

Hozzunk létre egy ember példányt a Mainen belül, ez lesz egy Ember objektum, mivel ez referenciatípus, így ha egy meglévő emberből csinálunk egy másik emberpéldányt, akkor a hivatkozás fog létrejönni, ezért az adott emberpéldányokon elvégzett módosítások mind a két példányra kihatnak!

```
//class Program, Main...
Ember e = new Ember();
e.Név = "Tóth József"; e.Életkor = 32; e.Lakhely = "Eger"; e.Munkahely = "XYZ Kft.";
Ember e2 = e;
e2.Név = "Horvát Béla";
```

Láthatósági, hozzáférési szintek

public, private, protected, konstruktor, destruktork

Az objektumok adatmezőit gyakran elrejtjük más programozók elől, hogy azok ne férjenek hozzá, ne tudják módosítani, C#-ban 3 láthatóság van:

- **public:** bárholnan hozzáférhetünk, az adatmezőt/eljárást lekérhetjük, módosíthatjuk
- **protected:** csak származtatott osztályból érhetjük el, lsd: OOP Öröklődés, 45. oldal
- **private:** csakis az osztályon belül érhető el (helyi tagváltozó, saját), a leszármaztatott osztályok nem láthatják és nem is módosíthatják.

A C#-ban még megadhatunk láthatósági módosítókat az **internalt** és a **protected internal**, erről később;

Csináljunk egy kutya osztályt (nagyon erőltetett példa, de megteszi), a kutyának lesz egy neve, és egy éhségszintje, ami privát lesz, mivel az adott kutypéldányt csak etetéssel és játékkal lehet az éhségszintjét befolyásolni, így kiküszöböljük azt a problémát, hogy később mikor egy új kutyát hozunk létre a fejlesztő helytelenül használja fel az adott mezőt...

```

class Kutya {
    public string Név;
    private int ÉhségJelző = 50;
    public void Etet(int étel) {
        ÉhségJelző -= étel;
    }
    public void Játék() {
        if (ÉhségJelző <= 80) {
            ÉhségJelző += 50;
            Console.WriteLine("Játék...");
        }
        else Console.WriteLine("A kutya éhes, nem tudsz játszani vele!");
    }
}

class Program {
    static void Main(string[] args) {
        Kutya k = new Kutya();
        k.Játék();
        k.Játék();
        Console.ReadKey();
    }
}

```

Konstruktor, destruktork:

Mint már a struktúráknál említettem a konstruktornak hívjuk egy osztály azon metódusát, mely az objektum példányosításakor kerül meghívásra, paraméterekkel együtt. A konstruktort az osztály nevével kell megadni és public láthatósággal különben nem tudnánk elérni. A konstruktórból csinálhatunk többet is (túlterhelés), ekkor eltérő szignatúrájú konstruktorokat készítünk, melyek egymástól függetlenül használhatók. Az objektum adatmezőinek eltávolítását a **destruktor** végzi, mivel a Garbage Collector automatikusan végzi a szemét eltakarítását nem determinisztikusan (kiszámíthatóan) történik a destruktor meghívása, destruktor a *~OsztályNeve* alakban tudjuk megadni, destruktor struktúrájánál nem lehet megadni mivel értéktípus. Kutya konstruktora:

```

//class Kutya{...
//Konstruktor
public Kutya(string n, int eh) {
    this.Név = n; this.ÉhségJelző = eh;
}
public Kutya() { }
//Destruktor:
~Kutya() {
    Console.WriteLine("Destruktor...");
}
}

```

Túlterhelés esetén az osztályokban nem kell teljesen új konstruktort megadni, elég egy olyan konstruktort írni ami az összes adattagot beállítja, majd egy eltérő paraméterlistával rendelkező konstruktort írni, majd utána : -al this(paraméterek) alakban megadni az adott konstruktort, a nem megadott mezőket állítjuk értéktípusú(int, double) esetén 0-ra referencia típus esetén null értékre. Így az új konstruktorunk gyakorlatilag csak meghívja a teljes értékűt.

```

class Ember {
    public string név;
    public int életkor;
    //összes értéket beállító konstruktor
    public Ember(string név, int életkor) {
        this.név = név; this.életkor = életkor;
    }
    //csak a nevet beállító
    public Ember(string név) : this(név, 0){ }
}

```

Példány és osztálymetódus/statikusmetódus, statikus mezők

Az eddig elkészített metódusok mind példánymetódus volt, tehát csak egy osztály példányosítása után volt lehetőségünk használni, ezeknek semmilyen megkülönböztetésük nem volt, a statikus vagy osztálymetódusok a példányosítás nélkül vehetjük igénybe az *Osztálynév.Statikusmetódusnév(paraméterek)* –el, ilyen pl., amikor

meghívjuk a Console osztály Write metódusát. Statikus metódus esetén nem férünk hozzá az osztályunk mezőihöz, így például csak a paraméterben átadott példány elemeit tudjuk használni. Nézzünk két példát: létrehozunk egy Diák osztályt egy példány és egy osztálymetódussal:

```
class Diák
{
    public string név;
    public int évfolyam;
    public void Kiír() { // példánymetódus
        Console.WriteLine("Név: " + this.név);
        Console.WriteLine("Éf.: " + this.évfolyam);
    }
    static public void Kiír(Diák d) { // osztály/statikusmetódus
        // Console.WriteLine("Név: " + this.név); <-- Hibás lenne
        Console.WriteLine("Név: " + d.név);
        Console.WriteLine("Éf.: " + d.évfolyam);
    }
}
```

Akkor nézzük a Maint, csinálunk egy példányt, lássuk az eredményt:

```
Diák d = new Diák();
d.név = "Szabó Irén";
d.évfolyam = 11;
d.Kiír(); //<-- példánymetódus
Diák.Kiír(d); //<-- osztály/statikusmetódus
```

Hasonlóképpen használhatunk statikus mezőket is, mely csak közvetlenül az osztályon keresztül férhetünk hozzá, pl.:

```
class Állandók {
    public static double PI = 3.14;
}
```

Abban az esetben, ha csak statikus osztályok és mezők vannak az osztályunkban, lehetőségünk van statikus osztályok létrehozására is (ezzel vigyázzunk!!), ekkor az osztályunk elé kell biggyeztetni a static szót, de ha ezt csináljuk, akkor az osztályból **nem lehet példányt létrehozni** (pl.: `Diák d = new Diák();`)

```
static class Állandók {
    public static double PI = 3.14;
}
```

Ez esetben csak így használható fel az osztály pi mezője: `double pi = Állandók.PI;`

OOP Jellemzők, Tulajdonságok

property, set, get

Publikus adattagoknál, mezőknél nem tudunk ellenőrzést csinálni, hogy ha csak bizonyos értékeket adhassunk meg, akkor külön eljárást kell írunk, viszont ez nem írható/olvasható közvetlenül, ezért jellemzőket, tulajdonságokat (property) kell írunk. Ez egy **olyan speciális osztályelem, amely bár mező és változó módjára viselkedik, de olvasása és írása esetén a memóriaterületek közvetlen írása helyett, a megadott olvasó (get) és/vagy író (set) metódusok kerülnek meghívásra** és azok futnak csak le. A **get** akkor kerül meghívásra, amikor értéket olvasunk ki a mezőből és a *return utasítással* adjuk meg, a **set** akkor, amikor egy mezőnek értéket adunk vagy módosítjuk, a set metódusnál a megadott értékre a *value* azonosító hivatkozik. Általában ekkor a privát mezőknek, amit majd a tulajdonságok állítanak be, kisbetűvel kezdődő, míg a tulajdonság esetén nagybetűvel kezdődő, de ugyanazt adjuk. **Egy jellemző lehet csak írható (writeonline csak set metódus), csak olvasható (readonline – csak get metódus) vagy írható és olvasható is.** Visual Studio esetén a Ctrl + R, E billentyűkombinációt lenyomva az adott privát mezőn létrehozza a hozzá tartozó jellemzőt. Csináljunk mi is egy ilyen jellemzőt: Készítsünk egy Ember osztályt név, életkor és lakóhelyének irányítószám mezőkkel, csináljunk írható és olvasható jellemzőket az összes adattaghoz és kezeljük le azt, hogy ne tudjunk majd üres mezőt adni a név atribútumnak az életkor ne lehessen negatív, az irányítószám pedig 4jegyből álljon, különben dobjunk kivételt!

```

class Ember {
//Ha a tulajdonságokkal adjuk meg a konstruktort akkor is ellenőrzést végez
    public Ember(string n, int é, int isz) {
        Név = n; Életkor = é; Irányítószám = isz;
    }
    private string név;
    public string Név {
        get { return név; }
        set {
            if (value.Length != 0) név = value;
            else Exception("A név mező nem lehet üres!");
        }
    }
    private int életkor;
    public int Életkor {
        get { return életkor; }
        set {
            if (value >= 0) életkor = value;
            else Exception("Az életkor mező nem lehet negatív");
        }
    }
    private int irányítószám;
    public int Irányítószám {
        get { return irányítószám; }
        set {
            if (value.ToString().Length == 4) irányítószám = value;
            else Exception("Az irányítószám mező helytelen!");
        }
    }
    private void Exception(string s) {
        throw new FormatException(s);
    }
}

```

Ezután erre kivételt kapunk: `Ember e = new Ember("Józsi", -10, 9456);`

Csinálhatunk külön .cs-ékbe különböző osztályokat, ezt a Solution Explorer ablakban jobb egérrel a projektünkre kattintva az Add/Class menüben, megadjuk ott a nevét és Add. De honnan tudja a fordító mihez tartozik az adott osztály, ebben segít a névtér(namespace) hiszen mind a kettőnek ugyan az a névtére.

Készítsünk egy csak író és csak olvasható metódust egy Idő osztályhoz, a csak írható metódusban meg lehet adni egy bizonyos másodpercet, ekkor csak a set kell, az olvasható metódus pedig visszatér az eltárolt időt percben:

```

class Idő {
    private int másodperc;
    public int Másodperc { //csak írható
        set {
            if (value > 0) másodperc = value;
            else Exception("Nem lehet negatív!");
        }
    }
    public double Perc { //csak olvasható
        get { return másodperc / 60.0; }
    }
    private void Exception(string s) {
        throw new FormatException(s);
    }
}

```

Nézzük meg a main-t:

```

Idő idő = new Idő();
idő.Másodperc = 160;
Console.WriteLine(idő.Másodperc); //<-- mivel csak set metódust állítottunk be ez hibás
Console.WriteLine(idő.Perc);

```

OOP Öröklődés

öröklődés, protected, polimorfizmus

Mivel az OOP alapelve a való világ modellezése, ezért az egy csoportba tartozó dolgokat, a látszólag azonos tulajdonságokkal rendelkező elemeket közös osztályok alatt tudjuk összesíteni, nézzünk egy egyszerű példát.

Létezzon egy Sokszögek nevezetű alaposztály (szakszerűen

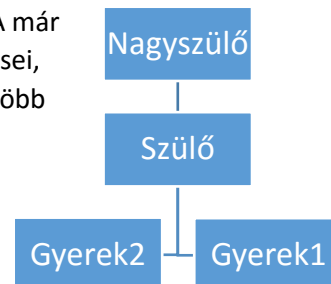
szülőosztály, őosztály), melynek gyermekei a 3, 4, 5... szögű síkidomok, a négyszögeken belül találhatjuk a trapézokat és a deltoidokat, a trapéz gyermeke a paralelogramma, annak a gyermeke a téglalapok és végül a

```
class Őosztály {  
    //metódusok, eljárások  
}  
class Gyermeosztály : Őosztály {  
    //metódusok, eljárások  
}
```

```
class Nagyszülő { }  
class Szülő : Nagyszülő { }  
class Gyerek1 : Szülő { }  
class Gyerek2 : Szülő { }
```

négyszetek. **Minden gyermek öröklí az ősök tulajdonságát és még hozzá is tesz a jellemzőihez, vagy felülírja azokat.** Nézzünk például egy Állatok őosztályt, ennek a gyermeke például a Gerincesek, a Gerincesek gyermekei pl. az Emlősök, Hüllők, Madarak és így tovább... **Minden osztály őse az Object, akkor is, ha ezt nem jelöljük.** Az öröklést az alábbi módon tudjuk megadni: A már

leszármaztatott osztálynak is lehetnek további örökösei, így egy hierarchia alakul ki közöttük, egy osztálynak több leszármazottja is lehet, **ugyanakkor csak egy őse!!** Egy adattag publikus megadása esetén az



őosztályban a leszármazott osztályok közvetlenül, míg a különálló osztályok példányosítás által tudják elérni az adott adattagot. Protected láthatóságg esetén csak a gyermekosztályok érik el az adott mezőt vagy metódust. Private által megadott mezők csak az adott osztályban belül érhetők el:

```
//PUBLIC láthatóság esetén  
class A {  
    public int IntA;  
    void Kiír() {  
        Console.WriteLine(IntA);  
    }  
}  
class B : A {  
    void Kiír() {  
        Console.WriteLine(IntA);  
    }  
}  
class C {  
    void Kiír() {  
        A a = new A();  
        Console.WriteLine(a.IntA);  
    }  
}
```

```
//PROTECTED láthatóság esetén  
class A {  
    protected int IntA;  
    void Kiír() {  
        Console.WriteLine(IntA);  
    }  
}  
class B : A {  
    void Kiír() {  
        Console.WriteLine(IntA);  
    }  
}  
class C {  
    void Kiír() {  
        A a = new A();  
        Console.WriteLine(a.IntA);  
    }  
}
```

```
//PRIVATE láthatóság esetén  
class A {  
    private int IntA;  
    void Kiír() {  
        Console.WriteLine(IntA);  
    }  
}  
class B : A {  
    void Kiír() {  
        Console.WriteLine(IntA);  
    }  
}  
class C {  
    void Kiír() {  
        A a = new A();  
        Console.WriteLine(a.IntA);  
    }  
}
```

Elmondhatjuk, hogy a gyerekosztályok public és protected láthatóság esetén hozzáférnek az őosztály mezőjéhez és metódusaihoz, de ezen kívül bővíthetik a tagváltozóit, új metódusokat hozhatnak létre és a szülőosztály metódusait akár felül is bírálhatják, más metódusokat készíthetnek egyazon név alatt. **A konstruktorok nem öröklődnek**, abban az esetben, ha az őosztályban van paraméter nélküli konstruktor, az a leszármazott osztály konstruktorában **hívódik** meg, ha paraméterezve van az őosztály konstruktor, akkor viszont a gyermek konstruktorában kell meghívni a **base** kulcsszóval. A base kulcsszó hasonlóan, mint a this az egy „családfához” tartozó mezőket érhetjük el vele, így ha olyan paraméter van, ami megegyezik egy osztály egy mezőjével akkor a base kulcsszóval azonosítható. Nézzünk egy példát: adott a következő őosztály konstruktorral (jobb)

```
class A {  
    private int a;  
    public A(int a) {  
        this.a = a;  
    }  
}
```

Mivel az leszármaztatott osztály örökli az „a” mezőt, ezért hibás az alábbi gyermekosztály konstruktora, ez jó:

```
class B : A {  
    protected int b;  
    public B(int b) {  
        this.b = b;  
    }  
}
```



```
class B : A {  
    protected int b;  
    public B(int b, int a) : base(a){  
        this.b = b;  
    }  
}
```



Ez a módszer akkor érdemes használni, hogyha az ősosztály mezője privát, így csak az ősosztály konstruktorával tudnánk beállítani a mezőt, különben

elég lenne a `base.mezőnév = xy;` beállítani. **Attól még, hogy mi privátnak állítjuk be az ősök mezőit, attól még öröklődnek, csak nem látjuk.**

Így meghívjuk az ősosztály konstruktorát a megadott paraméterrel, hogy állítsa be az „a” attribútum értékét.

Is – a viszony:

Csináljuk meg a következő ős és gyermekosztályt:

```
class Jármű {  
    public Jármű(int seb) {  
        sebesség = seb;  
    }  
    public int sebesség;  
    public int Megy(int h) {  
        return h * sebesség;  
    }  
}  
class Autó : Jármű {  
    public Autó(int seb, int ajtóksz, int cst) : base(seb) {  
        ajtóksz = ajtóksz; Csomagtér = cst;  
    }  
    public int ajtóksz;  
    public int Csomagtér;  
}
```

Jármű osztály leszármazottjai is egy jármű, az autó is egy jármű (az – egy... - is - a). Így a jármű osztályra megírt metódusoknak az Autó objektumokat, példányokat is átadhatjuk. (Megj.: Nem tudom ki emlékszik a minden légy bogár, de nem minden bogár légy dologra, hát itt is ez van minden autó jármű, de nem minden jármű autó). Például ez működik, nincs szintaktikai hiba:

```
static void JárműKiír(Jármű j, int h) {  
    Console.WriteLine(j.sebesség);  
    Console.WriteLine(j.Megy(h));  
}  
static void Main(string[] args) {  
    Jármű j = new Jármű(30);  
    JárműKiír(j, 10);  
    Autó a = new Autó(120, 5, 20);  
    //nincs szintaktikai hiba, hiába Jármű példányt vár az eljárás:  
    JárműKiír(a, 10);  
}
```

Mivel az autó is egy jármű a következő kód érvényes:

```
Autó a = new Autó(120, 5, 20);  
Jármű j = a;
```

De mivel egy jármű nem biztos, hogy autó a következő kód csak explicit konverzióval érvényes:

```
Jármű j = new Jármű(30);  
Autó a = j;
```



```
Jármű j = new Jármű(30);  
Autó a = (Autó) j;
```



Konverzió esetén futás idejű hibát kaphatunk, mert a jármű nem tartalmaz az autóban leírt metódusokat, mezőket. Így ha az adott jármű nem autó a következő hibát kapjuk konverzió esetén: `InvalidCastException` - Az objektum nem konvertálható Jármű típusról Autó típusra. Ennek elkerülése érdekében használjuk az **is** és az **as** operátort, az **is** true értéket ad vissza, ha a baloldalon megadott példány megfelel a jobb oldalon lévő típussal, az **as** operátor megvalósítja a konverziót akkor, ha az objektum megegyezik a jobb oldalon megadott típussal különben null értékre állítja be a példányt, példa:

```

static void JárműKiír_1(Jármű j, int h) {
    Console.WriteLine(j.sebesség);
    Console.WriteLine(j.Megy(h));
    if(j is Autó) {
        Autó a = (Autó)j;
        Console.WriteLine(a.ajtókSzáma);
        Console.WriteLine(a.Csomagtér);
    }
}
static void JárműKiír_2(Jármű j, int h)
{
    Console.WriteLine(j.sebesség);
    Console.WriteLine(j.Megy(h));
    Autó a = j as Autó;
    if (a != null) {
        Console.WriteLine(a.ajtókSzáma);
        Console.WriteLine(a.Csomagtér);
    }
}

```

Így egy metódusban több jármű adatot is beállíthatunk/kiírhatunk, anélkül hogy esetleg kivételt dobna. Mivel a kódunk más jármű leszármazottakra is működik ne csak a jármű mezőit irassuk ki, hanem a megfelelő elágazásokban nézzük meg, hogy az átadott objektum esetleg autó-e, akkor az autó osztály mezőit is írjuk ki.

Az ős metódusainak elfedése, újrainplementálása

Egy osztályhierarchián belül egy adott metódusnak más-más alakjai is létezhetnek, úgy hogy csak a program futása során derül ki, hogy melyik metódus fog lefutni, melyik fog meghívásra kerülni (polimorfizmus). C#-ban az alábbi módosítók adják meg, hogy a metódus felülbírálható-e, elfeledhető-e...

- **virtual**: A virtual módosítóval ellátott metódus a leszármaztatott osztályokban felülbírálható az override-el
- **override**: Az override módosítót a leszármazott osztályban annak a metódusunk elé kell tenni, aminek az ősosztályában az eljárásnál a virtual szerepel, így átdefiniálhatjuk, felülbírálhatjuk az ősmetódust
- **new**: A new módosító az ősosztályban létrehozott metódusokat elfedi, árnyékolja
- **sealed**: A sealed módosítóval ellátott metódust később nem tudjuk felülbíráltni, ha osztály előtt használjuk, azzal jelezzük, hogy az osztálynak nem lehetnek leszármazottjai

A virtual és az override használata:

Általános alakja az ősosztályban:

```

láthatóság virtual típus Eljárás(paraméterek) {
    //utasítások
}

```

Általános alakja a leszármaztatott osztályban:

```

láthatóság override típus Eljárás(paraméterek) {
    //utasítások
}

```

Az override módosítóval ellátott eljárások egyben virtual módosítóval elvannak látva, így a további leszármazott osztályok is felültudják definiálni. Nézzünk egy példát:

```

class Emlősök {
    public virtual void Evés() {
        Console.WriteLine("Etetés");
    }
}
class Kutya : Emlősök {
    public override void Evés() {
        Console.WriteLine("Etetés, farok csóválás");
    }
}
class Labrador : Kutya {
    public override void Evés() {
        Console.WriteLine("Etetés, ugatás");
    }
}

```


Vajon mi fut le az alábbi kód esetén a Mainben?

```
Emlősök e = new Emlősök(); e.Evés(); //Etetés
Kutya k = new Kutya(); k.Evés(); //Etetés, farok csóválás
Labrador l = new Labrador(); l.Evés(); //Etetés, ugatás
```

A new használata:

Az alábbi kódban nem jeleztünk semmitse a fordítónak a tagfüggvényekről, ezért figyelmeztetést mutat, mivel a Kutya tagfüggvénye eltakarja (shadow). Ha azt akarjuk, hogy ne jelezzon, javítsuk ki:

```
class Emlősök {
    public virtual void Evés() {
        Console.WriteLine("Etetés");
    }
}
class Kutya : Emlősök {
    public void Evés() {
        Console.WriteLine(
            "Etetés, farok csóválás");
    }
}
```




```
class Emlősök {
    public virtual void Evés() {
        Console.WriteLine("Etetés");
    }
}
class Kutya : Emlősök {
    public new void Evés() {
        Console.WriteLine(
            "Etetés, farok csóválás");
    }
}
```




Ugyanakkor a kutya további leszármazottjai a new módosító miatt nem tudják felülírni az Evés metódust, így ha azt szeretnénk, hogy továbbra is módosítható legyen a new szó után használjuk újra a virtual szócskát:

```
class Kutya : Emlősök {
    public new void Evés() {
        Console.WriteLine(
            "Etetés, farok csóválás");
    }
}
class Labrador : Kutya {
    public override void Evés() {
        Console.WriteLine("Etetés, ugatás");
    }
}
```




```
class Kutya : Emlősök {
    public new virtual void Evés() {
        Console.WriteLine(
            "Etetés, farok csóválás");
    }
}
class Labrador : Kutya {
    public override void Evés() {
        Console.WriteLine("Etetés, ugatás");
    }
}
```



A sealed használata, lezárt osztályok és metódusok


Ha a sealed kulcsszót egy osztály előtt használjuk, akkor azt mondjuk meg, hogy az osztálynak már nem lehetnek további leszármazottai, így például a Labrador osztályt több nem lehet származtatni:

```
sealed class Labrador : Kutya {
    public override void Evés() {
        Console.WriteLine("Etetés, ugatás");
    }
}
class SzomszédKutyája : Labrador {
    //hiba: Cannot derive from sealed type 'Labrador'
}
```



A sealed módosítót metódusoknál csak az override-al együtt használhatjuk, azt mondjuk meg, hogy az adott metódust már nem lehet a leszármaztatott osztályokban felbíráltni, így a Labrador osztály helytelen:

```
class Kutya : Emlősök {
    public sealed override void Evés() {
        Console.WriteLine("Etetés, farok csóválás");
    }
}
class Labrador : Kutya {
    public override void Evés() {
        Console.WriteLine("Etetés, ugatás");
    }
}
```



OOP – Polimorfizmus, késői és korai kötés

Csináljuk meg az alábbi programot:

```
class Fegyverek {
    public int Sebzés() { return 20; }
}
class Gépfegyver : Fegyverek {
    public int Sebzés() { return 5 * 20; }
}
public class Program {
    static void Elsüt(Fegyverek f) {
        Console.WriteLine("Sebzés mértéke: " + f.Sebzés());
    }
    public static void Main(string[] args) {
        Fegyverek f = new Fegyverek(); Elsüt(f);
        Gépfegyver gf = new Gépfegyver(); Elsüt(gf);
        Console.ReadKey();
    }
}
```

Ebben az esetben mivel nem írtuk felül (nem volt virtual és override), ezért mind a kettő fegyver elsütésénél 20-at írt ki a képernyőre a **korai kötés (early binding)** miatt, egészítsük ki tehát a virtualal az ősmetódust és overridal a Gépfegyver osztály metódusát:

```
class Fegyverek {
    public virtual int Sebzés() { return 20; }
}
class Gépfegyver : Fegyverek {
    public override int Sebzés() { return 5 * 20; }
}
```

Ha megint lefutatjuk a mainben lévő kódot mostmár eltérő sebzést kapunk. Miért is? **A nyelvnek azt a képességét, miszerint futási időben is képes dönteni késői (late binding), míg a deklarációkor, fordításkor lefutó típusmeghatározás a korai kötés**, például az első programrészben mikor még nem volt virtuális metódusunk fordításközben derítette ki a fordító, hogy melyik metódus fog lefutni, mivel Fegyver példányt vár az Elsüt eljárás, ezért került meghívásra mind a két esetben a 20-as, tehát a fegyverhez tartozó érték. A polimorfizmus azt a tulajdonságot jelenti, hogy futási időben dönti el, melyik metódus fusson le. Így például ha egy olyan Fegyver kollekciót hozunk létre, amelyben különböző fegyverek vannak, akkor nem kell aggódnunk amiatt, hogy egy foreach vagy for-ciklus esetén ugyanazok a metódusok futnak le, pl.:

```
Fegyverek[] fT = new Fegyverek[] { new Fegyverek(), new Gépfegyver() };
foreach (Fegyverek item in fT) {
    Console.WriteLine("Sebzés mértéke: " + item.Sebzés());
}
```

Mivel minden osztály az Object osztályból származik, örökli a ToString() nevezetű metódusát, ezt mi felül tudjuk írni:

```
class Fegyverek {
    public override string ToString() {
        return "Egyéb infók: . . . ";
    }
}
```

Ha nem hívjuk meg külön a ToString metódust és csak közvetlenül a változót írjuk ki, akkor is a ToString függvény értékét adja vissza:

```
Fegyverek f = new Fegyverek();
Console.WriteLine(f);
```

OOP - Beágyazott osztályok

Egy osztály tartalmazhat metódusokat, konstruktorokat, desktruktorokat és még akár egy belső osztály(oka)t is. Ezeket a belső osztályokat beágyazott (nested) osztályoknak nevezzük. A beágyazott osztályok alapértelmezetten private módosítóval vannak ellátva, így külső osztályok csak láthatóság módosítás esetén használhatják:

```
class Külső {  
    //Ha nem public lenne nem érnénk el:  
    public class Belső {  
        static public void Helló() { Console.WriteLine("Helló"); }  
    }  
}
```

A main-ben a pont operátor segítségével érhetjük el: `Külső.Belső.Helló();`

A beágyazott osztály hozzáfér az őt tartalmazó külső osztály minden tagjához (így a private láthatóságú tagokat és más beágyazott osztályokat is beleértve), de csak akkor, ha a beágyazott osztály egy referenciát tárol a külső osztályról:

```
class Külső {  
    private int szam = 15;  
    private Belső b;  
    public Külső() {  
        b = new Belső(this);  
    }  
    public class Belső {  
        private Külső k;  
        public Belső(Külső k) {  
            this.k = k;  
        }  
        public void KiírKülsőSzám(){  
            //ez hibás: Console.WriteLine(szam); Megoldás:  
            Console.WriteLine(k.szam);  
        }  
    }  
}
```

Ebben az esetben csináltunk a belső osztálynak egy olyan konstruktort, ami egy „Külső” osztálypéldányt vár, hogy onnan tudjuk kiolvasni a szám tartalmát, a külső osztályban szintén csináltunk egy konstruktort, ami a meghívása esetén átadja a saját példányát, önmagát (a this segítségével) a belső osztály konstruktorának.

OOP - Parciális osztályok

Lehetőségünk van c#-ban létrehozni **parciális, töredék** osztályokat. Ha egy osztály előtt a partial szócskát használjuk, akkor a fordítónak mondjuk meg, hogy az az osztály nem teljes, még további töredékei, darabjai vannak. Ezeket a töredékeket szintén a partial és a parciális osztályunk nevével látjuk el, fontos hogy azonos töredék osztályok csak azonos láthatósági módosítót használhatnak. Ugyanakkor az egyéb módosítókat elég egyszer jelezni az egyik töredék osztálynál, de annak ellentmondó más jelölés nem lehet. A parciális osztályok nem befolyásolják az osztály működését, csak segítik a fejlesztést, például átláthatóbbá teszik az osztályt, és több programozó is képes dolgozni egy-egy osztályon anélkül, hogy azokat később összekelljen másolni. A parciális osztályokat külön .cs-ben is elhelyezhetjük:

```
partial class PartialClass {  
    public void Kiír1() {  
        Console.WriteLine("Töredék 1 - Hy");  
    }  
}  
partial class PartialClass {  
    public void Kiír2() {  
        Console.WriteLine("Töredék 2 - Hy");  
    }  
}
```

Igy mind a két metódust használni tudjuk a mainben egy példány készítése esetén:

```
PartialClass pc = new PartialClass();
pc.Kiír1();
pc.Kiír2();
```

Beágyazott parciális osztályt a parciális osztály is tartalmazhat (a nem parciális is). Parciális osztályon belül lévő beágyazott parciális osztályok töredékei az őt tartalmazó osztály töredékei között oszlik szét.

Lehetőségünk van (C#3.0<) parciális metódusok használatát is, ekkor csak a metódus deklarációja és definíciója oszlik szét, parciális metódusnak nem lehet láthatósági módosítója így `alpé`. `privát` lesz, és `void`-al kell visszatérnie, ebben az esetben is a `partial` kulcsszót ki kell tenni minden előfordulásnál:

```
partial class PartialClass {
    partial void Kiír();
}
partial class PartialClass {
    partial void Kiír() {
        Console.WriteLine("Szia");
    }
}
```

OOP – Absztrakt osztályok, eljárások

Előfordul olyan eset, amikor olyan általános őosztályt kell megcsinálni, aminek nem tudjuk teljesen kidolgozni egyes metódusait a „fejletlensége” miatt, de pl. már egy másik metódus használná:

```
class Síkidom {
    protected int x;
    protected int y;
    public void Rajzol(){ }
    public void Animáció(int x_A, int y_A) {
        Console.Clear();
        x += x_A;
        y += y_A;
        this.Rajzol();
    }
}
```

A Síkidom osztályban az Animáció eljárás hiábahasználná már a Rajzol metódust, hiszen az nincs kidolgozva. Ebben az esetben hiába dolgoznánk ki a síkidom osztály gyermekosztályát a korai kötés miatt, akkor is a síkidomhoz tartozó metódus futna le. Akkor mit csináljunk? Próbáljuk meg `virtual` és `override` használatával:

```
class Síkidom {
    protected int x;
    protected int y;
    public virtual void Rajzol(){ }
    public void Animáció(int x_A, int y_A) {
        Console.Clear();
        x += x_A;
        y += y_A;
        this.Rajzol();
    }
}
class Vonal : Síkidom {
    protected int k;
    public override void Rajzol() {
        Console.SetCursorPosition(x, y);
        for (int i = 0; i < k; i++) {
            Console.Write("-");
        }
    }
}
```

Viszont ezzel a megoldással is több problémánk van: lehetséges, hogy a fejlesztő elfelejti felüldefiniálni egy virtuális metódust és erre semmilyen hibaüzenetet nem kap, ha használnák az adott eljárást nem történne semmi. Sőt fölöslegesen fordul le egy teljes értékű üres metódus és így helyet foglal (nem túl sokat). A másik probléma pl. ha nem void típusú lenne, akkor mégis valamilyen értéket visszakelljen adjon a függvény, hogy szintaktikailag helyes legyen, de mit adjunk vissza? Az ilyen jellegű metódusok/osztályok megfelelő készítése az abstract szóval történik, ezzel a szóval megmondjuk a fordítónak, hogy az őosztályban nem tudtuk megírni ezeket az eljárásokat, de deklaráljuk őket, hogy a gyermekosztályban használhatók legyenek. Ezeket a metódusoknál nem kell {} tenni csak egy pontosvesszőt a deklaráció végére. Fontos, hogy csakis absztrakt osztályban lehetnek absztrakt metódusok, és absztrakt osztály példányosítását megtiltja a fordító.

```
abstract class Síkidom {
    protected int x;
    protected int y;
    public abstract void Rajzol();
    public void Animáció(int x_A, int y_A) {
        Console.Clear();
        x += x_A;
        y += y_A;
        this.Rajzol();
    }
}
class Vonal : Síkidom {
    protected int k;
    public override void Rajzol() {
        Console.SetCursorPosition(x, y);
        for (int i = 0; i < k; i++) {
            Console.Write("-");
        }
    }
}
```

Érdekesség: Ha például az őosztályban olyan absztrakt metódust hozunk létre, amit a gyermekosztályban sem tudunk megírni, akkor azt a gyermekosztályt is abstract jelzővel kell ellátni, természetesen ez esetben sem példányosítható az absztrakt osztály.

Létrehozhatunk absztrakt tulajdonságokat jellemzőket is az alábbi módon, ha settert és gettert is megadtunk az absztrakt jellemzőnél, akkor mind a kettőt a gyermekosztályban be kell állítani:

```
abstract class Koordinátarendszer {
    public abstract int X { set; get; }
    public abstract int Y { set; get; }
}
class SíkbeliKoordinátarendszer : Koordinátarendszer {
    public int x;
    public int y;
    public override int X {
        get { return x; }
        set { x = value; }
    }
    public override int Y
    {
        get { return y; }
        set { y = value; }
    }
}
```

Az absztrakt osztályok gyakorlatilag egy közös felületet biztosítanak a leszármazottjaiknak. Ugyan absztrakt osztályokat nem lehet példányosítani, de ennek ellenére az alábbi kód érvényes:

```
Síkidom[] sT = new Síkidom[2];
sT[0] = new Vonal();
```

Ha jobban belegondolunk nem is történt példányosítás, ez esetben viszont igen, ezért szintaktikailag helytelen:

```
sT[1] = new Síkidom();
```

OOP – Interfészek

Az absztrakt osztályok és interfészek nagyban hasonlítanak egymásra, ha egy interfész egy osztály ősének állítunk, akkor **meghatározza az osztály felületét, előír egy mintát**. Az interfésznek nagy előnye, hogy míg egy osztály csak egy db osztályból öröklődhet, addig egy osztály több interfészt is használhat (*class Osztály : Interfész1, Int2, ...*), sőt interfészt akár struktúrák (38. o.) esetében is használhatunk. Az interfészek neveit konvenció szerint nagy I betűvel kezdjük. Ugyanúgy, mint az absztrakt osztályoknál a metódusokhoz, jellemzőkhöz nem tartoznak definíciók, csak deklarációk.

```
interface IKöszönés {
    void Reggel();
    void Este();
    void Napközben();
}
interface IÜdvözlés {
    void Szia(string név);
}
class Köszönés_Üdvözlés : IKöszönés, IÜdvözlés {
    public void Reggel() { Console.WriteLine("Jó reggelt!"); }
    public void Este() { Console.WriteLine("Jó estét!"); }
    public void Napközben() { Console.WriteLine("Jó napot!"); }
    public void Szia(string név) { Console.WriteLine("Szia {0}!", név); }
}
```

Az interfészek tagjainak nem lehet láthatósági módosítót megadni **alapértelmezetten publikusak lesznek**. Magának az interfésznek is alapértelmezetten publikus a láthatósága, de jelölhetjük internal-ként is. Ahogy a példában is látható egy interfészt használó osztálynak definiálni kell az interfészben deklarált összes tagját, különben szintaktikai hibát kapunk. Abban az esetben elhagyható ez, ha az osztály és tagjai absztraktként van deklarálva, ebben az esetben majd az az osztály fogja definiálni ezeket a tagokat, amely az absztrakt osztályból öröklődik, pl.:

```
interface IPélda {
    void Példa();
}
abstract class Absztraktpélda : IPélda {
    public abstract void Példa();
}
class PéldaClass : Absztraktpélda {
    public override void Példa() { Console.WriteLine("Példa"); }
}
```

Egyszerre származtathatunk egy osztályt egy osztályból és több interfészből is, ekkor az ősosztálynak kell az első helyen szerepelnie, majd utána vesszővel elválasztva következnek az interfészek, pl.:

```
class MyClass : PéldaClass, IPélda { }
```

Ez esetben nincs szintaktikai hiba, miért is? Mert már az IPélda interfész Példa metódusát a PéldaClass osztály már definiálta, és mivel megtettük ezt az osztályt az ősének, ezért a MyClass-ban is él a Példa metódus.

Ha egy interfész ősének egy interfészt teszünk meg, akkor később az interfészből létrehozott osztályoknál az ősinterfésznek is meg kell valósítani a tagjait:

```
interface IPélda {
    void Példa();
}
interface IPélda2 : IPélda {
    void Példa2();
}
class MyClass : IPélda2 {
    public void Példa() { Console.WriteLine("Példa1"); }
    public void Példa2() { Console.WriteLine("Példa2"); }
}
```

Az is és az as operátorrt használhatjuk az interfészeket megvalósító osztályokra is:

```
MyClass mc = new MyClass();
Console.WriteLine(mc is IPélda ? "Igaz" : "Hamis");
```

Mivel egy osztály több interfészből is származhat, előfordulhat névütközés, ezt a következő módon oldhatjuk meg:

```
interface IPelda {
    void Pelda();
}
interface IPelda2 {
    void Pelda();
}
class MyClass : IPelda, IPelda2 {
    void IPelda.Pelda() { Console.WriteLine("Pelda1"); }
    void IPelda2.Pelda() { Console.WriteLine("Pelda2"); }
}
```

Ebben az esetben nem használhatunk láthatósági módosítót. Nem tudjuk használni az osztály példányosítása után a Pelda metódust, ezért konvertálnunk kell a megvalósított interfészre:

```
MyClass mc = new MyClass();
((IPelda)mc).Pelda();
```

OOP – Operátor kiterjesztés

Mint az alapvető típusoknál megszokhattuk a különböző operátorokkal tudunk különböző műveleteket végezni, ezeket az operátorokat használni tudjuk az osztályokra is, ez esetben olyan **statikus függvényeket** kell megadnunk, ahol a függvény paraméterei az operandusok, visszatérési eredménye pedig a végeredmény, egy új osztálypéldány. Ezeket a függvényeket az alábbi módon kell megadni:

```
static public Osztály operator operátor(Osztály o1, Osztály o2){
    return new Osztály(...);
}
```

Természetesen a fenti esetben 2 operandusú operátorról beszélünk, ha kevesebb kell, akkor kevesebb paramétert kell megadni, vagy többet. Nézzünk egy példát: Csináljunk egy saját integer típust és adjuk meg a + operátort:

```
class EgészSzám{
    int szam;
    public int Szam { get { return szam; } set { szam = value; } }
    public EgészSzám(int szam) {
        this.szam = szam;
    }
    static public EgészSzám operator +(EgészSzám sz1, EgészSzám sz2){
        return new EgészSzám(sz1.szam + sz2.szam);
    }
}
```

Ezután használhatjuk is:

```
EgészSzám sz1 = new EgészSzám(13);
EgészSzám sz2 = new EgészSzám(20);
EgészSzám össz = sz1 + sz2;
```

Ezután nyugodtan használhatjuk megfelelő alkalmakkor a += operátort. Szokás szerint a paraméterek nevének lhs-t (left-hand-side) illetve rhs-t (right-hand-side) szoktak megadni. További felülírható operátorok: +, -, /, *, %, >, <, ++, -- és logikai operátorok...

Abban az esetben, ha logikai operátorokat adunk meg, akkor meg kell csinálni az operátor párját is, tehát a == is valamint a != definiálni kell, vagy < esetén a párját a >-t is... Nézzük meg mi lesz az EgészSzám osztályunk == és != statikus metódusa:

```
static public bool operator ==(EgészSzám sz1, EgészSzám sz2)
{ //ternáris operátor(?:) használatával, lsd.:40.o.
    return sz1.szam == sz2.szam ? true : false;
}
static public bool operator !=(EgészSzám sz1, EgészSzám sz2)
{ //hívjuk meg a == metódust és vegyük a tagadását, fontos hogy ne != írjunk hiszen azt ebben definiáljuk:
    return !(sz1 == sz2);
}
```


A ++/-- operátorok esetén csak a prefixes formát tudjuk megvalósítani, bár működik a postfixes forma, ugyanazt az eredményt kapjuk, mintha prefixest adtunk volna meg. Nézzük a példát:

```
static public EgészSzám operator ++(EgészSzám sz1){
    sz1.szam += 1;
    return sz1;
}
```

Az eredmény minden esetben azonos:

```
EgészSzám sz1 = new EgészSzám(2);
EgészSzám sz2 = ++sz1;
Console.WriteLine("Prefixes: {0}, {1}", sz1.Szam, sz2.Szam); //3, 3
EgészSzám sz3 = new EgészSzám(2);
EgészSzám sz4 = sz3++;
Console.WriteLine("Postfixes: {0}, {1}", sz3.Szam, sz4.Szam); //3, 3
```

Implicit és explicit konverzió esetén is megtudjuk mondani az osztálynak, hogy mit csináljon, ehhez az implicit és az explicit szavakra van szükségünk:

```
static public implicit operator EgészSzám(int sz) {
    return new EgészSzám(sz);
}
static public explicit operator EgészSzám(string sz) {
    return new EgészSzám(int.Parse(sz));
}
```

Használata:

```
EgészSzám sz1 = 10;
EgészSzám sz2 = (EgészSzám)"4";
```

OOP - Delegate, Event

A **delegate**-ek olyan típusok, amelyek egy-egy metódusra vagy **metódusokra mutatnak**, egy delegate megadásával olyan mutatót hozunk létre, amely a megadott metódusra hivatkozik. A metódusra hivatkozó delegate paraméterlistájának meg kell egyeznie a megadott metódus paraméterlistájával, a delegate ezután példányosítás után használható, konstruktorának a hivatkozott metódust kell megadnunk, majd a delegate objektum nevével hívjuk meg. Alakja: `delegate metódustípus Metódusnév(paraméterek);`

```
delegate void VoidDelegate(); //a delegate létrehozása
static void Metódus() {
    Console.WriteLine("A metódus lefutott!");
}
static void Main(string[] args) {
    VoidDelegate vd = new VoidDelegate(Metódus); //létrejön a hivatkozás
    vd(); //meghívja a megadott metódust, amire hivatkozik
}
```

A delegate-hez a + és a += operátorral újabb metódusokat adhatunk hozzá és a – valamint a -= operátorral kitudjuk törölni az adott metódusra vonatkozó hivatkozást, a meghívás esetén az összes hivatkozott metódus lefut:

```
delegate void Beléptetés(string str);
static void Üdvözlés(string név) {
    Console.WriteLine("Üdvözzöllek {0}!", név);
}
static void Üzenet(string név) {
    Console.WriteLine("Kedves {0}, köszönjük a belépésedet a rendszerbe...", név);
}
static void Main(string[] args){
    Beléptetés b = new Beléptetés(Üdvözlés) + new Beléptetés(Üzenet);
    Console.Write("Add meg a nevedet: ");
    string név = Console.ReadLine();
    b(név);
}
```

Egy delegate példánya hasonlóan működik, mint egy sima változó, átadhatjuk egyéb eljárásoknak, függvényeknek is, amelyek majd meghívják a benne lévő metódusokat.

```

class Class {
    public delegate void Meghívás(int i);
    static public void Ciklus(Meghívás mh) {
        for (int i = 0; i < 1000; i++) {
            mh(i);
        }
    }
}

class Program {
    static void Kiír(int i) {
        Console.WriteLine("i = {0}", i);
    }
    static void Main(string[] args) {
        Class.Meghívás mhKiír = new Class.Meghívás(Kiír);
        Class.Ciklus(mhKiír);
        Console.ReadKey();
    }
}

```

Az **event**-eket másnéven **események**et használó osztályok az **osztály állapotának megváltoztatásakor értesíthetnek más osztályokat**. Úgy, hogy az eseményt használó osztály egy eseménykezelővel meghívja azokat a metódusokat, amelyek feliratkoztak az eseményre. Az eseményhez meg kell adnunk egy delegate-t is, amivel az eseményhez megfelelő szignatúrát adjuk meg. Létezik beépített delegate (EventHandler), erről később. Általános alakja egy esemény definiálásnak: *Láthatóság(általában public) event EseményDelegateNév EseményNév*

```

class Szám {
    public delegate void EseménykezelőDelegate(string str);
    public event EseménykezelőDelegate ÁllapotváltozásEsemény;
    int szam = 0;
    public int Szam {
        get { return szam; }
        set {
            szam = value; //akkor indul be az esemény, ha megváltozik a szam mező:
            ÁllapotVáltozás();
        }
    }
    private void ÁllapotVáltozás() {
        if (ÁllapotváltozásEsemény != null) ÁllapotváltozásEsemény("Megváltozott a szam mező!");
    }
}

class Program {
    static void EseményKezelés(string str) {
        Console.WriteLine(str);
    }
    static void Main(string[] args) {
        Szám sz = new Szám();
        sz.ÁllapotváltozásEsemény += EseményKezelés; //feliratkozunk az eseményre
        sz.Szam = 21; //kiváltjuk az eseményt;
        Console.ReadKey();
    }
}

```

Az eseménykezelőknek általában 2 paramétert szoktak megadni (nem muszáj, csak szokás): egy objektumot, amin az esemény végbement, valamint a második paraméter az EventArgs osztály vagy egy őse. Hogy tudjuk, hogy mégis mi történt, csináljunk az EventArgs osztálynak egy olyan őst, ami képes egy üzenetet megjeleníteni. Ezután az osztályunkban módosítsuk az esemény delegate-nek a szignatúráját, valamint az eseménykezelő függvényt.

```

class Esemény : EventArgs {
    public string üzenet;
    //aki nem tudja, mit csinál a konstruktor lapozzon vissza!
    public Esemény(string str) : base()
    {
        üzenet = str;
    }
}

```

Nézzük, hogyan változott az osztályunk:

```
class Szám {
    public delegate void EseménykezelőDelegate(object o, Esemény e);
    public event EseménykezelőDelegate ÁllapotváltozásEsemény;
    int szam = 0;
    public int Szam {
        get { return szam; }
        set {
            szam = value;
            ÁllapotVáltozás();
        }
    }
    private void ÁllapotVáltozás() {
        if (ÁllapotváltozásEsemény != null)
            ÁllapotváltozásEsemény(this, new Esemény("A szám megváltozott!"));
        //a this-szel átadja az osztály saját magát
    }
}

class Program {
    static void EseményKezelés(object eseménytkiváltóosztály, Esemény e) {
        Console.WriteLine(e.üzenet);
    }
    static void Main(string[] args) {
        Szám sz = new Szám();
        sz.ÁllapotváltozásEsemény += EseményKezelés; //feliratkozunk az eseményre
        sz.Szam = 21; //kiváltjuk az eseményt;
        Console.ReadKey();
    }
}
```

DateTime, TimeSpan osztályok:

Előfordulhat, hogy programjainkban időegységgel kell dolgoznunk, erre egy nagyon jó megoldás a DateTime osztály, több konstruktora is van, ha nem adunk meg semmit, akkor alapértelmezetten 0-áll be, viszont ami nekünk fontos az alábbi két konstruktor:

```
int év = 2010, hónap = 11, nap = 25;
DateTime dt = new DateTime(év, hónap, nap);
int óra = 14, perc = 24, másodperc = 46;
DateTime dt2 = new DateTime(év, hónap, nap, óra, perc, másodperc);
```

Most nézzük meg, hogyan irathatjuk ki a megadott időpontot:

```
Console.WriteLine(dt2); //mindent kiír: 2010. 11. 25. 14:24:46
Console.WriteLine(dt2.ToShortDateString()); //rövidebb alakban írja ki csak a dátumot: 2010. 11. 25.
Console.WriteLine(dt2.ToShortTimeString()); //csak az időt jeleníti meg (óra:perc): 14:24
Console.WriteLine(dt2.ToLongDateString()); //hosszabb alakban: 2010. november 25., csütörtök
Console.WriteLine(dt2.ToLongTimeString()); //hosszabb alakban az időpontot: 14:24:46
```

A rendszerünk nyelvétől függően írja ki az időpontot, pl. angol esetén nap, hónap, év sorrendben. A **DateTime.Now** visszaadja a pontos jelenlegi időpontot, az **AddDays()**, **AddYears()**... függvények hozzáadják a megadott időegységet az objektumunkhoz. A **DateTime.Year/Mont/Day**... jellemzők visszadják az adott időpont egy-egy megadott részét.

```
DateTime most = DateTime.Now;
DateTime kétnapmúlva = most.AddDays(2);
DateTime egyévvelelőtt = most.AddYears(-1);
Console.WriteLine("{0}. van", most.Day);
```

Fontos metódus még a **DateTime.Parse(string)** függvény, ami megpróbálja átalakítani a megadott karakterláncot DateTime típusra, többféle formátum is megadható neki:

```
DateTime szülinap = DateTime.Parse("1999.07.23.");
DateTime szülinap2 = DateTime.Parse("1999/07/23");
```

Két időpont között eltelt időt úgy tudunk meghatározni, ha a két időpontot levonjuk egymásból, az eredmény egy **TimeSpan** objektum lesz, amiben a különbség eltárolódik, ezután a **TotalDays()**, **TotalHours()**... függvényekkel érhetjük el, hogy összesen hány nap, óra... a különbség:

```
DateTime szülinap = DateTime.Parse("1999.07.23.");  
TimeSpan hányévesvagyok = DateTime.Now - szülinap;  
Console.WriteLine(hányévesvagyok.TotalDays / 365);
```