Alapvető ismeretek informatika érettségihez

Programozás C#

A következők ismerete az informatika érettségi programozási feladatinak megoldásához szükségesek. Az említett adattípusok, műveletek és ciklusok ismerete elengedhetetlen a feladatok megoldásához. A System.Linq részben leírtak segíthetnek a feladatok megoldásában, ismeretük nem feltétlenül szükséges.

Ezen dokumentum segíti, de nem helyettesíti a felkészülést, esetenként eltérhet az iskolában tanultaktól. Az itt leírtakat mindenki a saját felelősségére használja!

Adattípusok

bool

Egy igaz (true) vagy hamis (false) érték.

byte

0-255 közötti egész szám.

int

-2.147.483.648 - 2.147.483.647 közötti (32 bites) egész szám.

Szöveg számmá alakítása

```
int a = int.Parse("15");                // 15
int b = System.Convert.ToInt32("15");   // 15
```

Ha hexadecimális számot szeretnék decimálissá alakítani, akkor a `Convert.ToInt32(szám, bázis)` függvényt használjuk:

```
int c = System.Convert.ToInt32("1a2b", 16); // 6699
```

Formázás

Egész számokat általában nem kell formázni. Ez alól kivételt képez, ha a számot hexadecimális számmá szeretnénk alakítani. Ehhez az x vagy X formátumot kell megadni. Attól függően, hogy kis- vagy nagybetűt használunk, a visszaadott értékben az a-f karakterek kis- vagy nagybetűvel lesznek írva.

```
6699.ToString("x"); // 1a2b
6699.ToString("X"); // 1A2B
```

`float`

Valós szám -3,40282347E+38 - 3,40282347E+38 között.

Formázás

Egész szám: `szam.ToString("0")`, egy tizedesjegyre kerekítve: `szam.ToString("0.0")`, két tizedesjegyre kerekítve: `szam.ToString("0.00")`, stb.

`char`

Egyetlen UTF-16 karakter. Egy karakter deklarálása aposztrófokkal történik: `'A'`

Karakter számmá alakítása

Egy karakter számmá alakítva megadja ezen karakter kódját.

```
int i = 'a'; // 97
```

Ez használható például indexként, vagy karakterek értékének számmá alakításánál.

```
int i = '7' - '0'; // 7
```

Függvények

- **`char.IsDigit`**: Megadja, hogy egy karakter 0-9 közötti szám-e
- **`char.IsLetter`**: Megadja, hogy egy karakter betű-e

`string`

Egy UTF-16 karakterekből álló, tetszőleges hosszúságú szöveg. Egy szöveget idézőjelekkel deklarálunk.

```
string szoveg = "Ez egy szöveg.";
```

Függvények és tulajdonságok:

- **Length**: Megadja a szöveg karaktereinek számát.
- **[index]**: Megadja az index pozíciójú karaktert a szövegben.
- **Split(karakter)**: A szöveget a megadott karakterek mentén „darabolja” és az így kapott szövegrészeket tömbben adja vissza.
- **IndexOf(keres)**: Megadja a keres szöveg vagy karakter helyét a szövegben. Ha a keresett karakter vagy szöveg nem található, akkor -1-et ad vissza, különben a keresett karakter vagy a keresett szöveg első karakterének pozícióját.
- **Substring(start[, hossz])**: Visszaadja a szöveg egy részét start pozíciójú karaktertől számítva hossz karakter hosszan. Ha nem adjuk meg a hossz paramétert, akkor a string végéig adja vissza a töredékszöveget.
- **StartsWith(keres), EndsWith(keres)**: Megadja, hogy a szöveg a keresett kifejezéssel kezdődik vagy végződik-e.

Példányosítás

- **Konstans szöveg:** `"Ez egy szöveg."`
- **Összefűzés:** `"Ma a hónap" + DateTime.Today.Day.ToString() + ". napja van."`
- **Interpoláció:** A szöveg elé egy $ jelet teszük. Ekkor a szövegben kapcsos zárójelek közé írt kifejezések kiértékelése után ezek szövegként ábrázolva a szövegbe kerül. Ez esetben a kifejezés után egy kettőspontot írva egy formátumot is megadhatunk.
```
$"A mai dátum: {DateTime.Today:yyyy. MM. dd.}"
// A mai dátum: 2020. 03. 29.
```
- **string.Format:** A `string.Format` függvény egy szöveget kap, melyben a változók helyett kapcsos zárójelek között a paraméterként megadott értékek indexe áll. Az index után szintén megadható a formázás.
```
string.Format("Az osztály létszáma: {0}, {1} fő hányzik, az osztály {2:0.00}%-a jelen van.", 27, 3, 24f / 27f * 100f)
// Az osztály létszáma: 27, 3 fő hányzik, az osztály 88,89%-a jelen van.
```

TimeSpan

Egy időpont vagy időtartam tárolására szolgáló struktúra.

```
TimeSpan idopont = new TimeSpan(15, 30, 0);
```

A fenti időpont értéke 15 óra 30 perc 0 másodperc.

TimeSpan értékek összeadhatók és kivonhatóak egymásból:

```
idopont = new TimeSpan(15, 0, 0) + TimeSpan.FromMinutes(30); // 15:30:00
```

Tulajdonságok

- **Hours**: Megadja az időpont óra komponensét. A példában 15.
- **Minutes**: Megadja az időpont perc komponensét. A példában 30.
- **Seconds**: Megadja az időpont másodperc komponensét. A példában 0.
- **TotalHours**: Megadja az eltelt időtartamot órában valós számként. A példában 15,5.
- **TotalMinutes**: Megadja az eltelt időtartamot percekben valós számként. A példában 930.
- **TotalSeconds**: Megadja az eltelt időtartamot másodpercekben valós számként. A példában 55800.
- **Days**: Időtartam esetén a napok számát.

Formázás

A formázást nem kötelező az itt leírtak alapján elvégezni. Ugyanúgy helyes, ha az egyes komponenseket egyenként egy szöveghez fűzzük. A következő formázási string-ek használhatóak az időpont formázására:

- **h**: Az óra komponens bevezető nulla nélkül.
- **hh**: Az óra komponens bevezető nullával.
- **m**: A perc komponens bevezető nulla nélkül.
- **mm**: A perc komponens bevezető nullával.
- **s**: A másodperc komponens bevezető nulla nélkül.

- **ss**: A másodperc komponens bevezető nullával.
- **Elválasztó karakterek**: Elválasztó karaktereket (pl. szóköz, kettőspont) két visszaperjellel \\ kell leírni.

```
idopont.ToString("hh\\:mm\\:s"); // 15:30:0
```

DateTime

Egy dátum és időpont tárolására szolgáló struktúra. Két dátum kivonható egymásból, az eredmény egy TimeSpan, ami megadja a két dátum között eltelt időt.

DateTime létrehozása

- **`new DateTime(nap, hónap, év[, óra, perc, másodperc])`**: Példányosít egy DateTime-ot a megadott év, hó, nap és opcionálisan óra, perc és másodperc értékekkel.
- **`DateTime.ParseExact(szöveg, formátum, formatProvider)`**: A szövegben található dátumot a megadott formátum szerint olvassa be. A formatProvider paraméter értéke null. A formátum részei a formázás pontban megadott szövegek a megfelelő módon összefűzve.

Tulajdonságok

- **Year**: A dátum év komponense.
- **Month**: A dátum hónap komponense.
- **Day**: A dátum nap komponense.
- **Date**: A dátum idő nélkül. (TimeOfDay = 00:00:00)
- **Hours**: Az idő óra komponense.
- **Minutes**: Az idő perc komponense.
- **Seconds**: Az idő másodperc komponense.
- **TimeOfDay**: Az idő komponens (TimeSpan típusú).

Formázás

- **yyyy**: Az évet adja meg.
- **MM**: A hónap bevezető nullával.
- **dd**: A nap bevezető nullával.
- **HH**: Az óra 24-órás formátumban bevezető nullával.
- **mm**: A perc bevezető nullával.
- **ss**: A másodperc bevezető nullával.
- **Elválasztó karakterek**: Az elválasztó karaktereke elé **nem** kell visszaperjel.

A 2019. októberi érettségi feladatban megadott dátumok beolvasása és kiírása:

Be: 20190326-0715

```
DateTime datum = DateTime.ParseExact("20190326-0715", "yyyyMMdd-HHmm", null);
datum.ToString("yyyy-MM-dd"); // 2019-03-26
```

Operátorok

Matematikai

Összeadás	Kivonás	Szorzás	Osztás	Maradék
+	**-**	*****	**/**	**%**
1 + 1 (2)	2 - 1 (1)	2 * 2 (4)	4 / 2 (2)	5 % 3 (2)

Két szám osztása esetén ha mindkét szám egész szám (`byte`, `int`, stb.), akkor a végeredmény is egész szám lesz. Ha az osztandó nem osztható az osztóval maradék nélkül, akkor az eredmény a hányados egész része (egészrészes osztás). **Ha a pontos hányadosra szükségünk van** (pl. százalékszámítás), akkor legalább vagy **az osztónak, vagy az osztandónak valós számnak kell lennie** (`float`, `double`)! Ehhez egy egész szám valós számmá alakítható:

```
int a = 52, b = 33;
var eredmeny1 = a / b; // 1
var eredmeny2 = (float)a / b; // 1,57575762
```

Számok egyel való növelése, csökkentése

Egy szám egyel növelhető a ++, valamint egyel csökkenthető a `--` operátorral.

```
a++; // 53
b--; // 32
```

A `System.Math` osztály különböző matematikai függvényeket tartalmaz. Ezek közül jó ismerni a következőket:

- **`Round(szam, tizedesjegyek, középértékkerekítés):`** A függvény egy számot és a tizedesjegyek számát várja paraméterként. A 3. paraméter értékét is meg kell adni, ez mindig `MidpointRounding.AwayFromZero` legyen, <u>különben hibás értéket kaphatunk</u>!
- **`Abs(szam):`** Visszaadja a megadott szám abszolút értékét.
- **`Sqrt(szam):`** Visszaadja a szám négyzetgyökét.
- **`Max(a,b), Min(a,b):`** A paraméterként kapott két szám közül a nagyobbat/kisebbet adja vissza.
- **`Floor(szam), Ceiling(szam):`** A megadott törtszámhoz legközelebbi/a számnál egyel nagyobb egész számot adja vissza. Ha a megadott szám egész szám, mindkét függvény a számot adja vissza.
- **`Pi:`** Konstans. A Pi értéke.

Logikai

Nem	És	Vagy
!	**&&**	**||**
!true (false)	true && false (false)	true || false (true)

Relációs

Egyenlő	Nem egyenlő	Kisebb (egyenlő)	Nagyobb (egyenlő)
==	**!=**	**< (<=)**	**> (>=)**
0 == 1 (false)	0 != 1 (true)	0 < 1 (true), 0 <= 1 (true)	0 > 1 (false), 0 >= 1 (false)

Gyűjtemények

Tömb

Előre meghatározott számú elemek gyűjteménye. A tömb elemeire azok indexével hivatkozunk. A gyűjtemények indexelése 0-val kezdődik. Azaz az első elem indexe 0, a másodiké 1, stb. Az utolsó elemre mindig a gyűjtemény mérete - 1. indexel hivatkozunk. A tömb példányosítása után minden elem értéke az adott adattípus alapértelmezett értéke. Számok esetében 0, referencia típusok esetén (pl. `string`) `null`, karaktereknél `'\0'`, `TimeSpan` esetén 00:00:00, `DateTime` esetén pedig 0001.01.01. 00:00:00.

Egydimenziós tömb:

```
byte[] tomb = new byte[10];
```

Egy 10 elemből álló tömb. A tömb harmadik eleme: `tomb[2]`. Az egydimenziós tömb méretét a Length tulajdonság adja meg.

Kétdimenziós tömb:

```
byte[,] tomb = new byte[5,2];
```

Egy kétdimenziós tömb, melynek 5 sora és 2 oszlopa van. A tömb második sorának első eleme: `tomb[1,0]`. A tömb egyes dimenzióinak méretét a GetLength(dimenzió) függvény adja meg, de deklaráció után, ha a dimenziók ismertek, akkor azok használhatóak. Az első dimenzió a 0., a második az 1. A fenti tömb elemein az alábbi két ciklussal tudunk végig menni:

```
for (int i = 0; i < 5; i++)
{
    for (int j = 0; j < 2; j++)
    {
        tomb[i, j] = (byte)(i + j);
    }
}

// byte[5, 2] { { 0, 1 }, { 1, 2 }, { 2, 3 }, { 3, 4 }, { 4, 5 } }
```

Lista (`List<T>`)

Egy előre nem meghatározott számú elemet tároló gyűjtemény. Az elemek típusát < > között adjuk meg.

```
List<Auto> autok = new List<Auto>();
```

Egy autókat tároló lista. A listához az `Add` függvénnyel adunk hozzá, a `Remove` függvénnyel pedig eltávolítunk elemeket. Az elemekre azok indexével hivatkozunk. A lista méretét a `Count` tulajdonság adja meg.

Szótár (`Dictionary<TKey, TValue>`)

Egy kulcs-érték párok tárolására használható gyűjtemény. Minden kulcs csak egyszer szerepelhet a gyűjteményben. Az egyes értékekre azok kulcsával hivatkozunk. Ha nem tudjuk, hogy egy kulcs szerepel-e a gyűjteményben, akkor azt a `ContainsKey(kulcs)` függvénnyel ellenőrizhetjük. Ha nem tudjuk, hogy a szótár biztosan tartalmazza- e a kulcsot, akkor ellenőrizni kell a kulcs létezését, mielőtt

a kulccsal megpróbálunk egy elemre hivatkozni, különben egy `KeyNotFoundException` kivételt kapunk. A kulcs és érték típusát < és > között adjuk meg.

Egy érték hozzáadása, módosítása:

```
Dictionary<string, int> szotar = new Dictionary<string, int>();
szotar.Add("hello", 1);
```

vagy

```
szotar["world"] = 2;
```

Ebben az esetben, ha a "world" kulcs nem szerepel a szótárban, akkor hozzáadásra kerül, különben az értékét módosítjuk. Az `Add` függvénnyel csak olyan értéket adhatunk hozzá, amelynek kulcsa nem tartozik a szótárhoz, különben `ArgumentException` kivételt kapunk.

Egy érték kiolvasása:

```
szotar["hello"];    // 1
```

A szótár minden elemén egy `foreach` ciklussal tudunk végig menni. A ciklus változója ez esetben `KeyValuePair<TKey, TValue>` típusú. A `Key` tulajdonság az adott elemhez tartozó kulcsot, míg a `Value` tulajdonság az értéket adja vissza.

```
foreach (var elem in szotar)
{
    if (elem.Key == "hello")
        Console.WriteLine(elem.Value); // 4
}
```

Ciklusok

Számlálós

```
for (int i = 0; i < tomb.Length; i++)
{
}
```

Egy tömb elemeinek vagy egy szöveg karaktereinek végig járására használhatjuk. Mindkét esetben a ciklus a tömb vagy szöveg méreténél egyel kisebb értékig számol (i **<** Length). A zárójelek között kettősponttal elválasztva a ciklus változóját, a folytatás feltételét valamint az iterátort adjuk meg.

Elöltesztelős, hátultesztelős

```
while (true)
{
}
```

Az elöltesztelős ciklus addig fut, amíg a kritérium igaz.

```
do
{
} while (true);
```

A hátultesztelős ciklus legalább egyszer lefut, utána csak addig, amíg a kritérium igaz.

Iteráló

```
foreach (var item in collection)
{
}
```

Ez a ciklus a gyűjtemény minden elemén végig menve végrehajtja a ciklus utasításait.

Ciklus elhagyása, továbblépés

Idő előtt kiléphetünk a `break;` utasítás használatával. Ha ki akarunk hagyni egy elemet, akkor a `continue;` utasítással egyből tovább léptethetjük a ciklust.

Véletlen szám generálása

Ha véletlen számot kell generálnunk, akkor erre a System.Random osztály egy példányát használhatjuk. **Mindig csak egy példányt hozzunk létre és azt használjuk!**[1]

```
Random rnd = new Random();
```

Egy véletlen egész szám generálásához használjuk a paraméter nélküli Next() függvényt:

```
rnd.Next();
```

Egy bizonyos intervallumba esö véletlen szám generálásához használjuk a Next(min, max) függvényt:

```
rnd.Next(0, tomb.Length);
```

Ezzel például egy véletlen indexet generálhatunk. A minimum értéke zárt, a maximum értéke nyílt: [min, max[. Tehát a legnagyobb véletlen szám amit generálunk az max-1.

[1] Ha több példányt hozunk létre egymás után (pl. egy ciklusban), akkor mindegyik példány uganazt a számot fogja generálni.

Fájlok kezelése

Fájlok kezeléséhez a System.IO névtérben található osztályok szolgálnak.

Elérési út

Ha a fájl a programmal megegyező könyvtárban, vagy egy almappában található, akkor nem kell a teljes elérési utat megadni, elegendő a fájl neve, vagy relatív elérési útját megadni. Ez igaz a fájlok létrehozására is: ha csak fájlnevet adunk meg a fájl létrehozásakor, akkor az a programunkkal azonos könyvtárban kerül létrehozásra.

Fájlok olvasása

Egy fájl teljes tartalmának szövegbe olvasásához a `File` osztály `ReadAllText` függvénye használható:

```
szöveg = System.IO.File.ReadAllText(fájlnév);
```

Egy fájl összes sorának beolvasása szöveg tömbbe a `ReadAllLines` függvénnyel:

```
sorok = System.IO.File.ReadAllLines(fájlnév);
```

Egy fájl soronkénti olvasásához a fájlt az `OpenText` függvénnyel nyitjuk meg:

```
using (var reader = System.IO.File.OpenText(fájlnév))
{
	var sor = reader.ReadLine();
	while (!reader.EndOfStream)
	{
		sor = reader.ReadLine();
	}
}
```

Az `OpenText` függvény megnyitja a fájlt olvasásra. A visszaadott `StreamReader` példány `ReadLine()` függvényével olvashatunk be egy sort a fájlból. Ha a fájl összes sorát be kell olvasnunk és nem tudjuk ezen sorok számát akkor az `EndOfStream` tulajdonsággal ellenőrizhetjük, hogy a fájl végére értünk-e. Ezt egy elöltesztelős ciklussal kombinálva egyesével beolvashatjuk a fájl sorait.

A fájl helyes kezeléséhez a `StreamReader` példányt vagy egy `using() {…}` utasítással körbe kell venni a példának megfelelően, vagy az olvasás végén az olvasót be kell zárni a `Close()` függvény meghívásával.

Fájlok írása

Egy teljes szöveg fájlba írása a `WriteAllText` függvénnyel történik:

```
System.IO.File.WriteAllText(fájlnév, szöveg);
```

Egy szövegeket tartalmazó tömb, vagy lista soronkénti fájlba írásához a `WriteAllLines` függvényt használhatjuk:

```
System.IO.File.WriteAllLines(fájlnév, sorok);
```

Egy fájl „manuális” írásához először létre kell hoznunk a fájlt a `CreateText` függvénnyel.

```
using (var writer = System.IO.File.CreateText(fájlnév))
{
```

```
        writer.Write(szöveg);
         writer.WriteLine(szöveg);
}
```

Ez a függvény egy `StreamWriter` példánnyal tér vissza, aminek a `Write` illetve `WriteLine` függvényeit használva tudunk a fájlba írni. A `Write` függvény az adott pozíciónál a fájlba írja a paraméterként kapott szöveget. A `WriteLine` függvény ugyanúgy működik, mint a `Write`, csak a szöveg után egy sortörést is a fájlba ír. Mindkét függvénynek több változata is van, melyek ismerete ajánlott.

A `StreamWriter`-t ugyanúgy, mint a `StreamReader`-t `using() {…}` utasítással kell körbevenni, vagy a használat után a `Close()` függvénnyel lezárni.

Saját osztály létrehozása

Egymással összefüggő adatok tárolására osztályt hozunk létre a `class` kulcsszó segítségével.

```
class Auto { }
```

Az osztályon belül mezőket vagy tulajdonságokat deklarálunk:

```
class Auto
{
    public string Gyarto;
    public float Fogyasztas { get; set; }
}
```

A mezők vagy tulajdonságok láthatósága **public kell, hogy legyen**, különben nem fogjuk tudni az osztályon kívül használni őket!

A `static` kulcsszó

A feladat megoldásához deklarált **függvényeket és változókat a `static` kulcsszóval kell ellátnunk**, hogy ezeket a Main() függvényből meg tudjuk hívni!

```
static Auto[] autok;
static void Feladat1() { }
```

System.Linq

Ha a programfájl elejéhez hozzáadjuk (ha nem került automatikusan hozzáadásra) a

```
using System.Linq;
```

utasítást, akkor a különböző gyűjteményekhez több kiterjesztett függvényt kapunk. Ezek használata nagyon praktikus és több bonyolultabb feladatot is egyszerűen elvégezhetünk.
A legtöbb függvény egy lambda kifejezést vár el paraméterként. A lambda kifejezés a
`változó => kifejezés`
formában adható meg. Ahol a változó egy változó neve, ami csak ebben a térben használható, a kifejezés pedig egy értéket ad vissza.

Az egyes függvények leírásánál a következő példát használjuk:

```
class Szemely
{
      public string Nev;
      public int Eletkor;
}
List<Szemely> szemelyek = new List<Szemely>();
int[] szamok = new int[10];
string[] nevek = new string[50];
```

- **`Where(lambda):`** Keresésre használható. Kiszűri a keresésnek megfelelő elemeket.
  `szemelyek.Where(sz => sz.Nev == "Éva")`
  Kiválogatja azon személyeket, akik neve Éva.
- **`Select(lambda):`** Az egyes elemeket átalakítja más típusúvá.
  `szemelyek.Select(sz => sz.Nev).ToArray()`
  Visszaadja az egyes személyek nevét, amit utána tömbbé alakítunk.
- **`OrderBy(lambda):`** Az egyes elemeket növekvő sorba rendezi.
  `szemelyek.OrderBy(sz => sz.Eletkor)`
  A személyeket életkoruk szerint növekvő sorba rendezi.
- **`OrderByDescending(lambda):`** Az egyes elemeket csökkenő sorba rendezi.
  `szemelyek.OrderByDescending(sz=>sz.Nev)`
  A személyeket nevük alapján az abc szerint csökkenő sorba rendezi
- **`ThenBy(lambda), ThenByDescending(lambda):`** Egy rendezett sort tovább rendezi a megadott kifejezésnek megfelelően.
  `szemelyek.OrderBy(sz=>sz.Nev).ThenBy(sz=>sz.Eletkor)`
  A személyeket nevük szerint, majd életkoruk szerint növekvő sorba rendezi. Ha pl. két Ádám szerepel a listában, egyikük 10, másik 22 éves, akkor előbb a 10 éves Ádám fog szerepelni. Ha nem rendezzük életkor szerint is a listát, akkor úgy vesszük, hogy az azonos nevű személyek sorrendje nem meghatározott.
- **`First():`** Visszaadja az első elemet. Hasznos például szűrés vagy sorba rendezés után.
  `szemelyek.First()`
  Az első személy a listában.
- **`Last():`** Visszaadja az utolsó elemet.
  `szemelyek.OrderBy(sz=>sz.Eletkor).Last()`
  A legidősebb személyt adja vissza.

- **`Min(), Min(lambda):`** Visszaadja a legkisebb elemet. Ha a típus maga rendezhető (pl. számok, karakterek, szövegek, DateTime, TimeSpan), akkor nem szükséges egy kifejezést megadni.
  ```
  szemelyek.Min(sz => sz.Nev);
  szamok.Min();
  nevek.Min();
  ```
  Visszaadja a névsorban első személy nevét, a tömbökben található legkisebb számot, illetve a sorba rendezett szöveg közül az elsőt. Megegyezik az OrderBy().First() függvényekkel.
- **`Max(), Max(lambda):`** Visszaadja a legnagyobb elemet. Ha a típus maga rendezhető (pl. számok, karakterek, szövegek, DateTime, TimeSpan), akkor nem szükséges egy kifejezést megadni.
  ```
  szemelyek.Max(sz => sz.Eletkor);
  szamok.Min();
  ```
  Megadja a legidősebb személy életkorát illetve a tömbben tárolt legnagyobb számot.
- **`Sum(), Sum(lambda):`** Visszaadja az egyes elemek, vagy a kifejezés által visszaadott értékek összegét.
  ```
  szemelyek.Sum(sz => sz.Eletkor);
  szamok.Sum();
  ```
  Megadja a személyek életkorának illetve a tömbben tárolt számok összegét.
- **`Count():`** Megadja egy gyüjtemény elemeinek a számát.
- **`Distinct():`** Megyadja a gyűjtemény különböző értékeit. Amelyik értekből több mint egy van, azokat is csak egyszer adja vissza. Ezt a függvény csak szöveg, számok és a fent leírt egyéb adattípusok esetén használható. Saját osztályok esetén a használata nem a kívánt eredményt fogja hozni (annak ismertetése nem része ennek a leírásnak).
  ```
  szemelyek.Select(sz => sz.Nev).Distinct();
  ```
  Először kiválasztjuk a személyek nevét, majd ezekből az egyes neveket kapjuk vissza (mindegyikből csak egyet, pl. ha két Ádám van, akkor Ádám csak egyszer szerepel).
- **`GroupBy(lambda):`** Az elemeket a megadott kifejezés értéke szerint csoportosítja. A visszaadott érték egy csoportosított gyűjtemény, melynek elemei az egyes csoportok. A csoportok Key tulajdonsága adja meg a csoporthoz tartozó értéket.
  ```
  var eletkorSzerint = szemelyek.GroupBy(sz =>
  sz.Eletkor).OrderBy(cs=>cs.Key);
  foreach (var csoport in eletkorSzerint)
  {
      var kor = csoport.Key;
      if (csoport.Count() > 2)
      {
          foreach (var szemely in csoport)
          {
          }
      }
  }
  ```
  A személyeket életkoruk szerint csoportosítottuk és sorba rendeztük. Ezután az egyes csoportokon végig menve megkapjuk a csoport életkorát. A csoport elemei azok a személyek, akik életkora a csoport életkorával megegyezik. De megvizsgáljuk, hogy a egyes csoportokban legalább két személy van-e és csak akkor megyünk az egyes személyeken végig.