Fogalmak

**Alapok**

**compiler:** fordítóprogram, ami lefordítja az adott nyelvet gépi byte nyelvre.

**program.cs**: C#-ban ezzel a nevezetű fájlal kezdődik az egész, amit később lefordítunk.

**szintaktikai hiba:** Abban az esetben, ha a programozási nyelv "nyelvtanában" hibáztunk valamit pl. sor végén” ; ” kell mindig a C#-ban, és a compiler nem fordítja le programkódot, hanem valamilyen hibát dob.

**szemantikai hiba:** 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.

**namespace:** névtéraz a logikai egység, melyben az azonosítónak egyedinek kell lennie a többi névtértől.

**modul:** egy önállóan is lefordítható forráskód C#-ban használata pl: using System;.

**Adatszerkezetek**

**változó:** A változó a memóriában lefoglalt bitek azonosítója és tartalma.

**deklarálás:** egy változónévvel ellátott típust hozunk létre érték nélkül pl: int szam1;

**definiálás**: más szóval inicializálás a változónévvel ellátott típusnak értéket is adunk. pl: int szam1=1

**típusok:** egy-egy algoritmusban a megadott típus értékének tárolására használt memória tartományának elnevezése pl int,double,string,char stb.

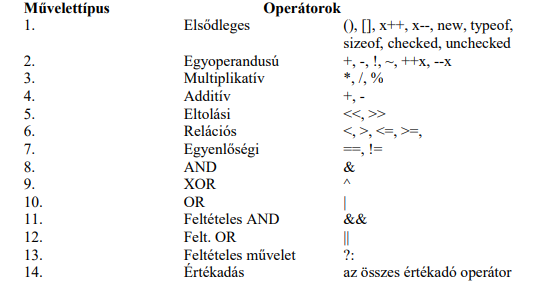
**prefix:** használati alakja pl: ++a itt először növeli az ’a’ változó értékét majd a többi részt.

**postfix:** használati alakja pl: a++ itt 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 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 tizesszámrendszerben. 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.

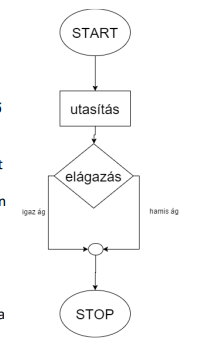
**Precedenciai rangsor**

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.

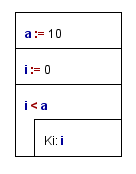


**Algoritmust leíró eszközök**

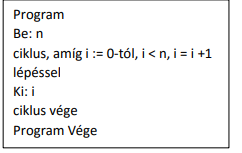
**A folyamatábra:** olyan algoritmust leíró eszköz, mely a programot gráfként írja le. Ez egy olyan irányított gráf, amely csomópontokból áll, egyetlen induló és befejező része van. A következő kép megmutatja ezeket a csomópontokat, utasításokat…, és talán így már könnyebben megérthető lesz később a 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 nehezebben áttekinthető, mint a folyamatábra. Itt egy téglalap írja le az egész programot, amelyet további téglalapokra, háromszögekre osztanak.



**Mondatszerű leírás:** programot mondatokkal írjuk le a Program szóval és a Program vége szavakkal ér véget például:

****

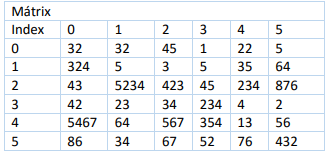
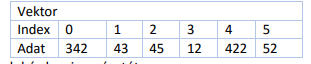
**Összetett adatszerkezetek: Tömbök**

**tömb:** 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:

****

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

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.

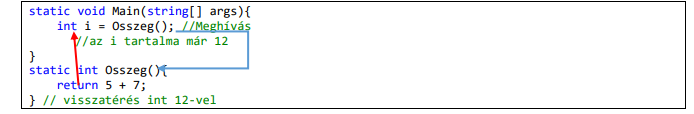
**Bővebben a mátrixról:** : a mátrix egy olyan kétdimenziós tömb melynek van szélessége és magassága is, míg a vektor csak 1 magas volt és bármekkora széles addig ez már legegyszerűbben egy táblázatként fogható fel, használata majdnem ugyan az, mint a vektornál csak itt már két indexet kell megadnunk, hogy elérjünk egy cellát. Deklarálása hasonló, mint a vektorok esetében, csak egy vesszőt kell beszúrnunk 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 dimenziókig tart. Szintén tudunk kapcsos zárójelekkel létrehozni egy már 0-tól eltérő adatokat tartalmazó tömböt, ilyenkor szintén nem kell megadnunk a tömb méretét. 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.

**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;

**Alprogramok**

**alprogram:** 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 szolgál, 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.

**eljárás:** : nincs visszatérési értéke, ezért void (üres) típusú:



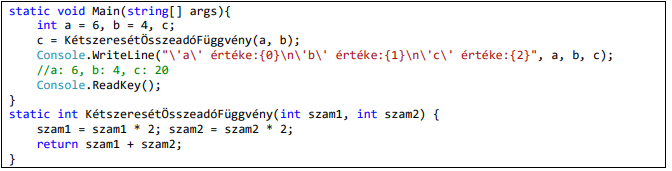
**függvény:** van visszatérési értéke, a visszaadott értéktől függ, a return utasítással adjuk meg a visszaadott értéket, ami minden esetben kötelező a megadása.

****

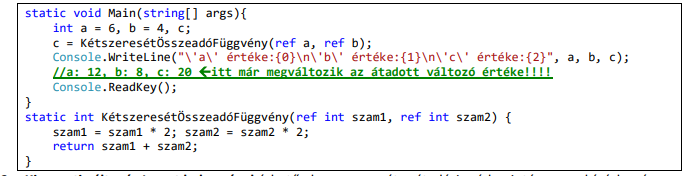
**rekurzió:** Az olyan műveletet, melynek végrehajtásakor a saját műveletsorait hajtja végre, tehát önmagát ismétli, rekurziónak nevezzük. Programozás esetén a függvények saját magukat hívják meg. Amikor egy függvény nem önmagát, hanem egy másik függvényt hív meg, majd ez a függvény pedig meghívja azt a függvényt, amiből meghívták (…), akkor azt kölcsönös rekurziónak nevezzük (pl. A() meghívja B()-t, majd B() meghívja A()-t…).

**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.

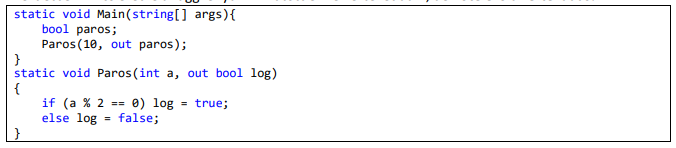
**Érték szerinti paraméterátadás:** Ha nem adunk meg semmilyen kulcsszót, akkor ilyen módon adódnak át a változók (kivéve tömbök, objektumok... ). Ilyenkor az alprogram meghívásakor megadott aktuális paraméterek értéke átmásolódik az alprogram egy-egy megfelelő formális paraméterbe, ezután a formális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

****

**Cím (Referencia) szerinti 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!

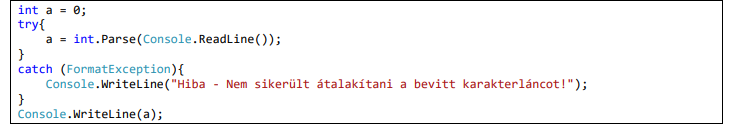
****

**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 érétke a függvényünk hívatott ennek értéket adni, de kötelező az értékadás!

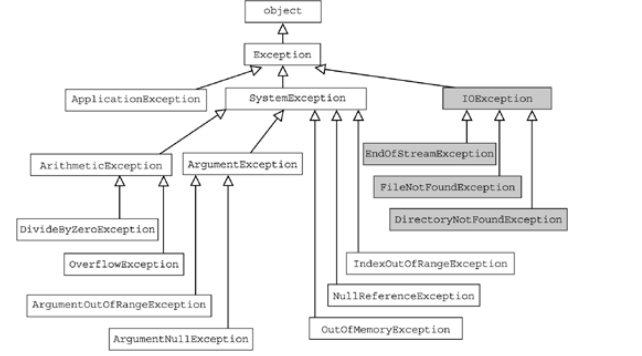
****

**Hibakezelés**

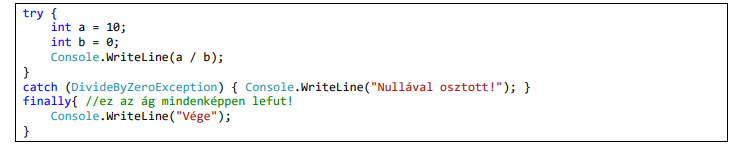
Abban az esetben eddig, ha egy nem kezelt kivétel (Exception) keletkezett be, akkor kifagyott a programunk és a hibát vagy a visual studio, vagy az operációs rendszer kezelte le, és egy hibaüzenetet dobott. A programunk, ekkor kifagyott. Például a string int-re való parsolásánál, ha nem egész számot 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 felejtsük a bekért változót már deklarálni a try előtt, különben nem tudjuk később használni!



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”:



**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:



**Generikusok**

**Lista:** egy dinamikus tömb, méretét automatikusan változtatja. A lista egy generikus típus, tehát bármilyen típusból (akár tömbből, osztályból, struktúrából, listából 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).

**Szótár (Dictionary):** 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 kulcsértékpárt adni, lehet akár int és string kulcs-értékpár is. Deklarációkor is megadhatunk már adatokat így:

****

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:



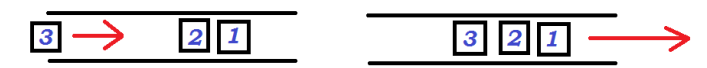
A kulcsoknak egyedinek kell lennie nem lehet két egyforma kulcs de az értékek ismétlődhetnek.

**Verem (Stack):** 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 kell létrehozni, de nem az Add metódussal tudunk értéket hozzáadni, hanem a .Push(érték) metódussal.

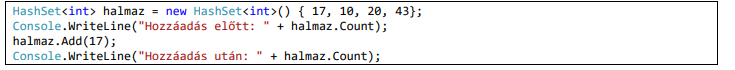
A képen sor, diagram, Téglalap, tervezés látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

**Sor (Queue):** 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.



**Halmazok (HashSet):** 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.



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.

Sorlista (ArrayList): 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.Példa:

A képen szöveg, Betűtípus, sor, képernyőkép látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

**Alapszíntű fájlkezelés**

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 névteret (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 kiterjesztésest 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 zárnunk az adatfolyamot az sw\_neve.Close() metódussal. 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 fájlfolyam közben, valamilyen IOException adódik, pl. tipikus hibaszokott lenni a rossz mappanév, ekkor például DirectoryNotFoundException-t kapunk.

A képen szöveg, Betűtípus, sor, képernyőkép látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

**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 fájlnak, 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.

A képen szöveg, képernyőkép, Betűtípus, sor látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

**Struktúrák:** A struktúrák olyan adategységek, amelyekben különböző típusú adatokat foglalhatunk egy adatszerkezetbe. Csakúgy, mint a felsorolásnál, a struktúrákat 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:

A képen szöveg, sor, Betűtípus, képernyőkép látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

A struktúra egy értéktípus, tehát értékadáskor csak a mezők értékei másolódnak át, így az egyes példányokon belül végzett módosítások nem hatnak a másik változóra. Egy struktúrából létrehozhatunk akár egy listát vagy tömböt is.

**A dinamikus típus (var):** A C#3.0 verziótól bevezettek egy általános típust, amikor egy változónak a var azonosítót adjuk, akkor a fordítóra bízzuk az adott változó típusának meghatározását, ezért muszáj deklarálás során értéket adnunk a változónknak.

**OOP (Object-Oriented Programing)**

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 lasabban 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 eljárás.

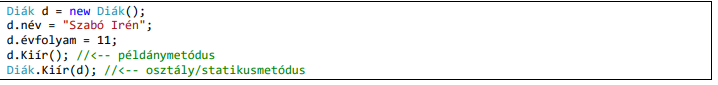
**konstruktor:** egy osztály azon metódusa, mely az objektum példányosításakor kerül meghívásra, paraméterekkel együtt. A konstruktort az osztály nevével kell megadni és public láthatósággal különben nem tudnánk elérni. A konstruktórból csinálhatunk többet is (túlterhelés), ekkor eltérő szignatúrájú konstruktorokat készítünk, melyek egymástól függetlenül használhatók.

**destruktor:** Az objektum adatmezőinek eltávolítását a ez végzi, mivel a Garbage Collector autómatikusan végzi a szemét eltakarítását nem determisztikusan (kiszámíthatóan) történik a destruktor meghívása, destruktort a ~OsztályNeve alakban tudjuk megadni, destruktort struktúránál nem lehet megadni mivel értéktípus.

**Példány és osztálymetódus/statikusmetódus, statikus mezők:** meghívjuk a Console osztály Write metódusát. Statikus metódus esetén nem férünk hozzá az osztályunk mezőihez, így például csak a paraméterben átadott példány elemeit tudjuk használni. Nézzünk két pédát: létrehozunk egy Diák osztályt egy példány és egy osztálymetódussal:

A képen szöveg, Betűtípus, képernyőkép, sor látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.



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 biggyezteni a static szót, de ha ezt csináljuk, akkor az osztályból nem lehet példányt létrehozni ~~(pl.: Diák d = new Diák();)~~



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

**jellemzők, tulajdonságok**: publikus adattagoknál, mezőknél nem tudunk ellenőrzést csinálni, hogy ha csak bizonyos értékeket adhassunk meg, akkor külön eljárást kell írnunk, viszont ez nem írható/olvasható közvetlenül, ezért jellemzőket, tulajdonságokat (property) kell írnunk. 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. Egy jellemző lehet csak írható (writeonline csak set metódus), csak olvasható (readonline – csak get metódus) vagy írható és olvasható is.

**get:** akkor kerül meghívásra, amikor értéket olvasunk ki a mezőből és a return utasítással adjuk meg

**set:** 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.

A képen szöveg, képernyőkép, szoftver, Betűtípus látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

**öröklödés:** 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. Létezzen egy Sokszögek nevezetű alaposztály (szakszerűen szülőosztály, ősosztály), 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 négyzetek. Minden gyermek örökli az ősök tulajdonságát és még hozzá is tesz a jellemzőihez, vagy felülírja azokat. 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.

A képen szöveg, képernyőkép, Betűtípus, sor látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

Elmondhatjuk, hogy a gyerekosztályok public és protected láthatóság esetén hozzáférnek az ősosztály mezőihez és metódusaihoz, de ezen kívül bővithetik 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 ősosztályban van paraméter nélküli konstruktor, az a leszármazott osztály konstruktorában hivódik meg, ha paraméterezve van az ősosztály konstruktora, 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.

**Is – a viszony:**

A képen szöveg, szoftver, Betűtípus, sor látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

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. (Megjegyzés: Nem tudom ki emlékszik a nem minden rovar bogár, de minden bogár rovar dologra, hát itt is ez van minden autó jármű, de nem minden jármű autó).





**polimorfizmus:** 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.

**A 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 felülbírálni, ha osztály előtt használjuk, azzal jelezzük, hogy az osztálynak nem lehetnek leszármazottjai.

**Absztrakt osztályok, eljárások:** Előfordul olyan eset, amikor olyan általános ősosztályt kell megcsinálni, aminek nem tudjuk teljesen kidolgozni egyes metódusait a „fejletlensége” miatt, de pl. már egy másik metódus használná. Az ilyen jellegű metódusok/osztályok megfelelő készítése az abstract szóval történik, ezzel a szóval megmondjuk a fordítónak, hogy az ősosztályban 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.

A képen szöveg, képernyő, szoftver, Betűtípus látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

**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 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.

**A képen szöveg, Betűtípus, képernyőkép, szám látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**

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 absztarktként van deklarálva, ebben az esetben majd az az osztály fogja definiálni ezeket a tagokat, amely az absztrakt osztályból öröklődik, pl.:

A képen szöveg, képernyőkép, sor, Betűtípus látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

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.

**Láthatósági, hozzáférési szintek**

**public:** bárhonnan hozzáférhetünk, az adatmezőt/eljárást lekérhetjük, módosíthatjuk.

**protected:** csak származtatott osztályból érhetjük el.

**private:** csakis az osztályon belül érhető el (helyi tagváltozó, saját), a leszármaztatott osztályok nem láthatják és nem is módosíthatják.

**internal:** a kód csak a saját fájlnév.cs-n belül érhető el, egy másik fájlnév.cs-ben nem.