



Hernyák Zoltán

# Magasszintű programozási nyelvek II.

Objektumorientált programozás  
a gyakorlatban





HERNYÁK ZOLTÁN

Magasszintű programozási nyelvek II.

draft változat

2013.04.14



Hernyák Zoltán

# Magasszintű programozási nyelvek II.

Objektumorientált programozás  
a gyakorlatban

Eger ♦ 2012



Szerző

Dr. Hernyák Zoltán  
főiskolai docens  
Eszterházy Károly Főiskola

Lektorálta

# Tartalomjegyzék

---

TARTALOMJEGYZÉK .....	3
1. BEVEZETÉS .....	7
2. AZ OOP TÖRTÉNETE .....	11
3. AZ OOP ALAPELVEI .....	14
4. AZ IMPERATÍV NYELVEK OSZTÁLYOZÁSA .....	17
5. EGYSZERŰ PÉLDA EGY OOP PROBLÉMÁRA .....	18
6. AZ ADATREJTÉS .....	22
6.1. A mező értékének védelme .....	24
6.2. A megoldás metódussal .....	25
6.3. Hibajelzés .....	28
6.4. Protected a private helyett .....	29
6.5. Miért a 'private' az alapértelmezett védelmi szint? .....	31
6.6. Property .....	32
6.7. Amikor azt gondolnánk, hogy nem kell alkalmazni védelmet .....	35
6.8. Egyszer írható mezők .....	35
6.9. Csak olvasható mezők .....	37
6.10. Hatékonyabb megoldás .....	39
7. METÓDUSOK .....	40
7.1. Példány szintű metódusok .....	42
7.2. Aktuális példány kezelése .....	44
8. A MAIN() FÜGGVÉNY .....	48
9. KONSTRUKTOROK .....	51
9.1. Konstruktorok példányosításkor .....	52
9.2. Konstruktor készítése .....	53
9.3. Több konstruktor készítése .....	54
9.4. Konstruktor hiánya .....	55
9.5. A paraméterek ellenőrzése .....	56
9.6. Egyszer írható mezők .....	58
9.7. Property és a kettős védelmi szint .....	58
9.8. Valódi egyszer írható mezők .....	59
9.9. Konstansok .....	60
10. AZ ADATTAGOK .....	62
10.1. Példányszintű mezők .....	62

---

10.2. Osztályszintű mezők.....	64
10.3. Konstansok .....	65
11. AZ ÖRÖKLŐDÉS .....	67
11.1. A mezők öröklődése .....	67
11.2. A mezők öröklődésének problémái.....	69
11.3. A base kulcsszó .....	72
11.4. A metódusok öröklődés.....	74
11.5. A metódusok öröklődésének problémái .....	75
11.6. A metódusok és a 'base'.....	78
11.7. A metódusok öröklődésének igazi problémája .....	80
12. TÍPUSKOMPATIBILITÁS .....	81
12.1. A típuskompatibilitás következményei .....	82
12.2. Az Object osztály .....	86
12.3. A statikus és dinamikus típus .....	87
12.4. Az 'is' operátor .....	88
12.5. A korai kötés és problémái .....	89
13. A VIRTUÁLIS METÓDUSOK.....	94
13.1. Az override és a property .....	95
13.2. Az override egyéb szabályai .....	96
13.3. Manuális késői kötés – 'as' operátor .....	97
13.4. Amikor csak a típuskényszerítés segít .....	101
13.5. A típuskényszerítés nem csodafegyver.....	101
13.6. A kenguruk története .....	104
14. PROBLÉMÁK A KONSTRUKTOROKKAL .....	105
14.1. Konstruktor hívási lánc.....	106
14.2. Konstruktor azonosítási lánc.....	107
14.3. Saját konstruktor hívása – 'this'.....	110
14.4. Saját konstruktor hívása és az azonosítási lánc .....	112
14.5. Ős konstruktor hívása explicit módon: 'base' .....	113
14.6. Osztályszintű konstruktorok .....	115
14.7. Private konstruktorok .....	118
14.8. A 'sealed' kulcsszó .....	119
14.9. Az Object Factory .....	120
15. INDEXELŐ .....	123
16. NÉVTEREK.....	128
17. AZ OBJECT OSZTÁLY MINT ŐS.....	134
17.1. GetType().....	134
17.2. ToString() .....	135



17.3. Equals()	137
17.4. GetHashCode()	138
17.5. Boxing - Unboxing	138
17.6. Object lista	141
17.7. Object paraméter	144
18. ABSTRACT OSZTÁLYOK	147
19. VMT ÉS DMT	155
19.1. A VMT segéd táblázat	156
19.2. A DMT segéd táblázat	161
20. PARTIAL CLASS	165
21. DESTRUKTOROK	167
21.1. Ha nem írunk destruktort	170
21.2. Mikor ne írjunk destruktort?	170
21.3. Mikor írjunk destruktort?	174
22. GENERIC	175
23. INTERFACE	180
23.1. Generic interface-k	187
23.2. Interface-k öröklődése	188
23.3. IEnumerable és a foreach	188
24. NULLABLE TYPE	191
25. KIVÉTELKEZELÉS	194
25.1. A kivétel feldobása	200
25.2. A hiba oka	202
25.3. A hiba kezelése	204
25.4. A hiba okának felderítése	208
25.5. A kivétel újrafeldobása	209
25.6. A kivételek szétválogatása	210
25.7. Saját kivételek	212
25.8. Finally	213
25.9. A kivétel keletkezése hibát okoz	216
25.10.	Try...
catch... finally...	217
25.11.	
Egymásba ágyazás	218
26. OPERÁTOROK	220
26.1. Egyoperandusú operátorok fejlesztése	221
26.2. Kétooperandusú operátorok fejlesztése	224
26.3. Típuskényszerítő operátorok fejlesztése	227

26.4. Záró problémák .....	229
26.5. Extensible methods.....	231
27. SZERELVÉNYEK .....	233
27.1. A Windows DLL .....	236
27.2. A DLL pokol.....	237
27.3. A .NET C# DLL .....	239
27.4. A DLL készítésének és felhasználásának lépései.....	239
27.5. A DLL és a GAC.....	246
27.6. A DLL és az OOP.....	247
27.7. A DLL kiegészítő védelmi szintjei.....	248
28. CALLBACK.....	249
28.1. Alkalmazáslogika és felhasználói felület .....	252
28.2. Dönts egyszer – használd sokszor .....	254
28.3. Nem kitöltött függvénypointerek.....	256
28.4. Példányszintű függvények.....	256
28.5. Függvénylista kezelése .....	257
28.6. Eseménylista .....	258
28.7. Származtatás másként .....	259
28.8. TIE osztályok.....	261
29. REFLECTION .....	264
29.1. Assembly betöltése dinamikusan .....	265
29.2. Saját assemblyre hivatkozás .....	265
29.3. Egy osztály megkeresése egy szerelvénnyel belsejében .....	266
29.4. Egy osztály metódusának megkeresése.....	267
29.5. Osztálysintű metódus meghívása I. ....	268
29.6. Osztálysintű metódus meghívása II.....	268
29.7. Példányszintű konstruktor megkeresése és példányosítás.....	268
29.8. Példányszintű metódus megkeresése és meghívása .....	269
30. ZÁRSZÓ .....	272
IRODALOMJEGYZÉK .....	274

# 1. Bevezetés

---

A programozás történeti folyamatait nem könnyű feltérképezni. Gyakran emlegetjük MOHAMED IBN MUSZAT (? i. sz. 800-850) mint egyik fontos személyiséget, aki matematikusként élt és alkotott. A programozás szempontjából a latin fordításban *Algorithm* néven megjelent *A hindu számokról* c. könyve miatt érdekes. Könyvében módszert adott arra, hogyan kell tízes alapú, helyértékes számokkal műveleteket végezni. Módszeres módon fogalmazta meg a műveletek elvégzésének lépéseit, így a világ egyik első algoritmusai tőle erednek. Maga az algoritmus tudományág is a könyv címéről kapta nevét.

A továbbiakban sokan és sokféleképpen fogalmaztak meg algoritmusokat, megoldási lépéssorozatokot. Ugyanakkor az elektronikus számítógépek megjelenéséig főképpen emberek értelmezték és hajtották végre azokat. ALAN TURING (1912–1954), a modern számítógép-tudomány atyja dolgozott ki egy absztrakt „gépet”, definiálta azokat a minimálisan szükséges feltételeket, utasításokat, melyek segítségével egy probléma megoldása leírható. E módon definiált egyfajta „univerzális” algoritmus-leíró nyelvet, amely kevés elemi lépést tartalmaz, és a lépések hatása állapotátmenetekkel jól definiált. Ez a leíró nyelv azonban nem kifejezetten alkalmas arra, hogy közvetlenül ezen adjunk meg algoritmusokat, de a más módon (leíró nyelv, folyamatábra stb.) megadottakat át lehet erre a nyelvre transzformálni. Így az algoritmusok már jól elemezhetőek, matematikai eszközökkel vizsgálhatóak.

A Turing-gép lett az alapja a Neumann-elvű számítógépeknek is. Ezen számítógépek memóriát tartalmaznak, amelyben az adatokat tároljuk. A memória képviseli (őrzi) a program aktuális állapotát. A program feladata nem más, minthogy egy kiinduló adatmennyiségből (állapotból), a memóriabeli adatokat módosítva, kiegészítve kiszámítsa a keresett értéket. A műveletvégző egység a processzor, amely az algoritmus elemi lépéseit hajtja végre. A processzor elemi lépéseit egy speciális programozási nyelven: a *gépi kód* utasításainak formájában kell megadni.

Valamely algoritmus leírására ennek megfelelően sokfajta lehetőségünk van. A programozás maga sem más, mint algoritmusok írása, csak a leírás módja különleges. Míg maga az algoritmus tárgy a vizsgált algoritmusokat próbálja eszközfüggetlen, platformfüggetlen módon megadni, addig a programozás tárgy ugyanezt egy kiválasztott programozási nyelven teszi. Az algoritmus tárgy igyekszik egyetlen konkrét, kisebb méretű problémára koncentrálni, a feladatot pontosan definiálni és a megoldási lépéssorozatot megadni. A programok írása során pedig általában a megoldandó feladat több algoritmus ötvözésével, összekapcsolásával készíthető el.

A számítógép véges erőforrásainak minél hatékonyabb kiaknázása végett az algoritmusok összekapcsolásakor azokat módosítjuk, átalakítjuk. Ez a programozás lényege. Azonban az átgondolatlan, nem megfelelő átalakítások okozhatják az így előállt kód hibás működését is, ezért a programok helyes futását valamilyen módon ellenőrizni kell, például teszteléssel.

---

A gépi kódú programozási nyelv sajnos nem kifejezetten alkalmas közvetlenül programozásra, algoritmusok leírására. Alacsony absztrakciós szintje mellett túlságosan könnyű hibát ejteni benne, majd azokat felderíteni is nagyon nehéz. További hátránya, hogy a gépi kódú nyelv processzorfüggő, vagyis az egyes processzorok saját gépi kódú nyelvei erősen eltérhetnek egymástól.

A magasabb absztrakciós szintű nyelvek, mint az assembly vagy a procedurális nyelvek (C, Pascal stb.) azonban a processzor számára értelmezhetetlenek (egyszerűen nem léteznek). Ezeken a nyelveken megírt programokat, forráskódokat egy program, a compiler fordítja át gépi kóddá, hogy a processzor azt végre tudja hajtani. A program indításakor már az átalakított, átfordított kód indul el, ami további bizonytalanságot szül, hiszen ha a compiler rossz, akkor az általunk megírt kód hiába hibátlan, a ténylegesen futó kód már hibásan fog működni. Ez gyakoribb eset, mint gondolnánk, főként ha a compilertől kódoptimalizálást kérünk (pl. futási sebességre, memória-kihasználásra).

A számítógépen futó programok egymásra is gyakorolhatnak káros hatást, zavarhatják egymás működését, amely az operációs rendszer hibájára vezethető vissza, mivel ezt nem lenne szabad megengednie. A virtualizálás, a futó szoftverek a hardvertől és egymástól minél kiterjedtebb elhatárolása ugyanakkor magára a programozásra is rányomta a bélyegét. Ma egyre elterjedtebb, hogy a compiler nem közvetlenül gépi kódra fordítja át a forráskódot, hanem egy magasabb absztrakciós szintű „gépi kódú” nyelvre, mely egy virtuális processzor gépi kódjaként fogható fel. A processzor utasításkészletére generált programot egy virtuális gép, egy virtuális futtató rendszer értelmezi és hajtja végre. Ezen megoldásnak komoly előnyei vannak, elsősorban biztonsági és működési szempontokból, míg hátrányaként elsősorban a futási sebességet és általában az erőforrásigényt szokták megjelölni.

A programozás ugyanakkor a külvilág igényeinek, nyomásának megfelelően kénytelen komoly teljesítményeket letenni az asztalra. Míg korábban néhány tíz képernyőnyi kód már elfogadható mennyiségű problémát tudott kezelni, a manapság készülő programok sok programozó több hónapos munkájának gyümölcsei. Példaképpen említjük meg, hogy a Windows NT 3.1 verziója (1993-ban) még csak 4-5 millió forráskódsorból állt, addig a rá egy évre megjelent Windows NT 3.5 már 7-8 millió, az 1996-os Windows NT 4.0 pedig 11-12 millió sorból állt. A Windows 2000 több mint 29 millió, a 2001-ben megjelent XP pedig 45 millió sorból<sup>1</sup>.

Miből épül fel egy program? Értelemszerűen kellenek adatok, amelyekkel dolgozik. Adatainkat változókból tároljuk, melyeket a memória tárol. A memória véges, így igyekszünk a helyfoglalást minimalizálni. Megfelelő típust választunk a tárolásra, melynek helyfoglalása a várható értékek befogadására képes, de feleslegesen nem igényel helyet. Az időtartamot is igyekszünk optimalizálni, csak a legszükségesebb adatokat tároljuk statikus élettartamú változókból, míg a legtöbb adat esetén dinamikus élettartamot választunk.

---

<sup>1</sup> Az adatok a [http://en.wikipedia.org/wiki/Source\\_lines\\_of\\_code#Example](http://en.wikipedia.org/wiki/Source_lines_of_code#Example) honlapról származnak.

Amennyiben több adatunk, több változónk egyetlen adatcsoport elemit tartalmazza, úgy élettartamuk kezdetének és végének is egyazon pillanatra kell essniük, erre használhatunk rekordot, listát, tömböket, és egyéb összetett adatszerkezeteket.

A program adatokon kívül utasításokat tartalmaz. A logikailag összetartozó utasításainkat, utasításcsoportjainkat jellemzően függvényekbe szervezzük. Programunk futása a függvényeink megfelelő sorrendben történő meghívásából áll. A függvények a globális statikus adatokkal, és a paramétereikben megkapott értékekkel dolgoznak.

Ezt a modellt nevezhetjük „hagyományos” programozási modellnek, melynek sok előnye és sok hátránya van. A hátrányok elsősorban nagyobb projektek esetén mutatkoznak meg. A programozók által készített függvényhalmaz nehezen tesztelhető, és ha az egyes függvények külön-külön megfelelnek a tesztnek, összekapcsolásuk továbbra sem feltétlenül jelent hibamentes működést. Az egyes adatok paramétereken keresztüli folyamatos átadása-átvétele gyakran feleslegesen terheli a processzort és a memóriát. Amennyiben valamely adat módosult, úgy nehéz eldönteni melyik függvény végezte el a módosítást, ami globális adatainkra is igaz. Emiatt az adatok értékére vonatkozó invariánsokat<sup>2</sup> nem könnyű betartatni. Az adataink értékeire vonatkozó megbízható védelmet csak a típus invariáns adja, vagyis ha egy adatunk típusa pl. *'sbyte'*, akkor biztosak lehetünk abban, hogy értéke  $-128-127$  közötti egész szám, de semmi másban nem lehetünk biztosak. Ha nekünk ennél szűkebb feltétel szükséges, de olyan típus nem áll rendelkezésre mely ezen szűkebb invariánst biztosítani tudná, akkor így zsákutcába jutottunk. Ugyanakkor a programozási nyelvben definiált alaptípusok körét bővíteni nem lehet, és ezzel együtt a programozási nyelvben létező operátorok működését sem lehet kiegészíteni. Valamint nem lehetséges olyan invariánsok definiálása, amelyben már két vagy több adat együttes értékkombinációjára van megfogalmazva a feltétel (pl. „ha *'A'* értéke páros, akkor *'B'* értéke nem lehet nagyobb, mint 10”).

Mielőtt áttekintenénk, milyen megoldásokat, lehetőségeket nyújt az objektumorientált programozás (továbbiakban OOP) ezekre a problémákra, szögezzünk le néhány dolgot:

- minden olyan program, amely megírható objektumorientált szemléletben - megírható hagyományos programozási szemléletben is,
- az OOP programozás során nem fogunk új programvezérlési szerkezeteket (ciklus, elágazás) megismerni, a függvények törzsében továbbra is a *for*, *if*, *foreach*, *switch* és társaik fognak szerepelni,
- továbbra is függvényeket fogunk írni, azoknak paramétereit adunk át, vesszük át,
- az OOP programok nem futnak gyorsabban (sőt, gyakran gyengébb a teljesítményük ilyen téren, mint a hagyományos stílusban tervezett és írt programoknak).

---

<sup>2</sup> Olyan állítás, mely a program teljes futási ideje alatt igaz értékű marad, pl. „az életkor adat értéke 18 és 60 közötti”

Előnyök, melyeket kapunk cserébe:

- programunk jobban áttekinthető egységekből fog állni (összetartozó adatok és függvények csoportjai),
- ezen csomagok tesztelése együttesen történhet, így a programunk hibátlan működése jobban biztosítható,
- sok esetben kevesebb függvény megírása is elegendő,
- az adataink értékére vonatkozó bonyolultabb garanciák, invariánsok is fenntarthatóak,
- új, teljes értékű típusokat hozhatunk létre, amelyekre operátorok működése is definiálható.

## 2. Az OOP története

---

A programozás története a programozási nyelvek generációjával jellemezhető. A gépi kódot nevezzük az első generációnak. A további generációk mindegyike fordítóprogramot feltételez: a magasabb generációs nyelveken megírt programokat át kell fordítani gépi kódra.

A második generációt assembly nyelveknek tekintjük, amely nyelv nagyon közel áll a gépi kódhoz. Bár sok és fontos fogalmat vezetett be a programozásba, de lényegében a gépi kódú programozás egy olvashatóbb formája. Az utasításai egy az egyben megfeleltethetőek a valamely gépi kódú utasításnak.

Az igazán nagy lépést a harmadik generációs, ún. procedurális, magas szintű nyelvek megjelenése hozta. A generáció programozási stílusát *moduláris* megközelítésnek is nevezik. Itt jelentős új programozási fogalmak jelentek meg, de egyik legnagyobb újdonság az volt, hogy egy utasítása már nem egy, hanem több gépi kódú utasításra volt csak lefordítható. Ez önmagában jelentősen növelte a programozók hatékonyságát, a kódolási sebességet.

A moduláris programozás központi eleme a *függvény*. A függvény valamely részfeladat megoldására készített, névvel ellátott kódrészlet. A függvény a feladat megoldása során más, korábban már megírt függvényeket is felhasználhat.

A függvények törzse, az utasítássorozat megfogalmazásában elrugaszkodott az alapokat jelentő gépi kód alacsony szintű programvezérlési szerkezeteitől is (pl. feltételes ugró utasítás), helyette bevezették a ma is használt szekvencia, szelekció, iterációs vezérlési szerkezeteket. Ha egy algoritmus (vagy program) leírása csak ezen három programvezérlési szerkezettel történik meg, akkor *sturktúrálnak* nevezzük. Két kutató, CORRADO BÖHM és GIUSEPPE JACOPINI fogalmazta meg azt a sejtését<sup>3</sup>, hogy minden kiszámítható függvény felírható pusztán e három vezérlési szerkezettel. Eszerint az akkoriiban a még igen elterjedt, magas szintű nyelvekben is fellelhető „goto” utasítás kizárhatóságára lehetett következtetni.

A Pascal nyelv egyik atyja, EDGSEER DIJKSTRA a „Goto utasítás káros hatásai” cikkével<sup>4</sup> újabb lökést adott ennek az iránynak. Ma még mindig rendelkeznek a programozási nyelvek ilyen „ugró” utasításokkal (pl. break, continue), mivel használatuk csökkentheti a kód bonyolultságát és növelheti a futási sebességet, hatékonyságot; de alkalmazásuk mindig megfontolandó és amennyiben lehetséges – kerülendő.

A magas szintű nyelvek elvei megfelelőnek tűntek, és tűnnek a mai napig. Mai napig is dolgoznak olyan programozók, akik csakis ezt a programozási paradigmát ismerik, és

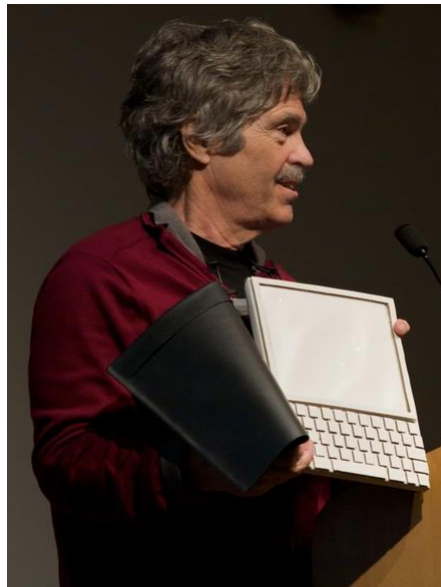
---

<sup>3</sup> Bohm, Corrado; and Giuseppe Jacopini (May 1966). "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules". *Communications of the ACM* 9 (5): 366–371. doi:10.1145/355592.365646

<sup>4</sup> Dijkstra, Edsger (1968). "Go To Statement Considered Harmful". *Communications of the ACM* 11 (3): 147-148. doi:10.1145/362929.362947. <http://www.acm.org/classics/oct95/>

ebben fejlesztik kiválóan működő alkalmazásaikat. Egyedüli, vagy néhány fős, szorosan egymás mellett dolgozó fejlesztők esetén ez nem jelent hátrányt. A bevezetőben részletezett, a szoftverfejlesztésre nehezedő nyomás azonban új utakra terelte a programozási nyelvek fejlődését.

Az objektumorientált programozás elveit ALAN CURTIS KAY<sup>5</sup> fektette le diplomamunkájában 1969-ben. Miután a Xerox Palo Alto-i kutatóközpontjában kezdett el dolgozni, folytatta és befejezte az alapelvek kidolgozását 1972-ben.



Megtervezett egy programozási nyelvet, melyet *Smalltalk*-nak nevezett el. Ez az első és máig is létező objektumorientált programozási nyelv, amelynek napjainkban is készülnek újabb és újabb változatai, de az alapelvek mindvégig ugyanazok maradtak.

Egy másik területen is úttörő munkát végzett: szerinte a személyi számítógépnek grafikus felhasználói felülettel kell rendelkeznie, melynek beviteli egysége az egér. A felhasználói felület a felhasználó ikonokon, menürendszeren, ablakokon kell, hogy alapuljon.

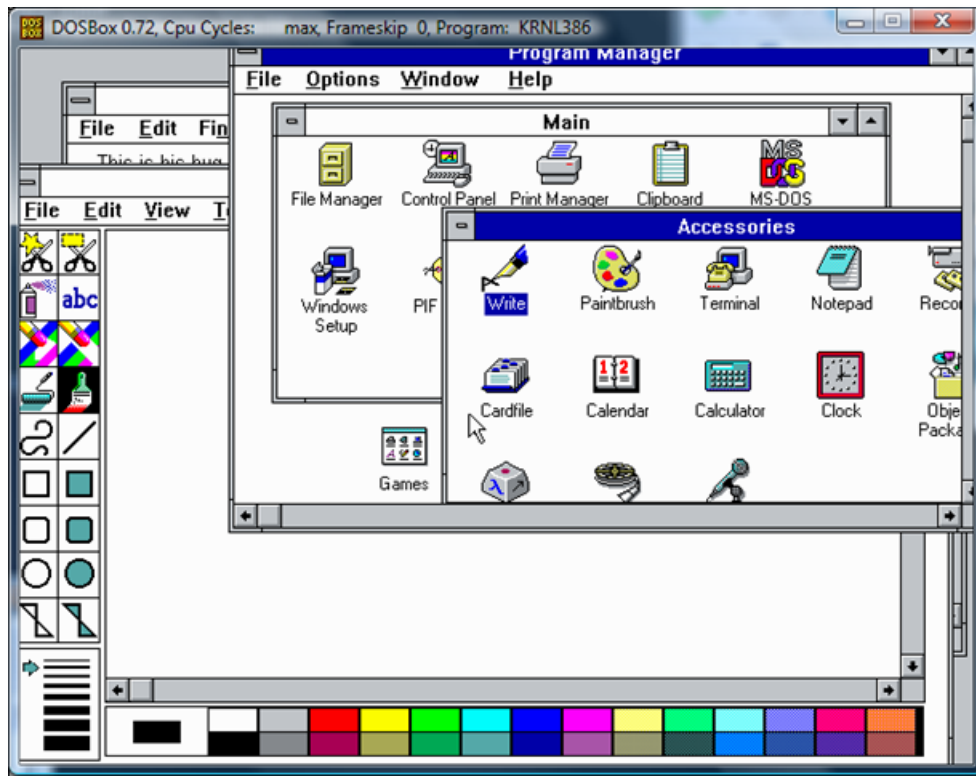
ALAN KAY 1973-ban egy hordozható számítógépet álmodott meg, amit Dynabooknak neveztek el. Egy könyv méretű, hordozható számítógép, ami vezetékek nélküli hálózati csatlakoztatást, jó minőségű színes képernyőt és igen nagy számítási teljesítményt foglalt volna magába. A terv ugyan terv maradt, de KAY meggyőzte a Xerox kutatási vezetőit, hogy dolgozzanak az elképzelésén. Végül összeállították, az akkoriban rendelkezésre álló csúcstechnológiából, az Alto névre keresztelt gépet, ami valójában egy miniszámítógép volt 256 KiB memóriával, egérrel, cserélhető merevlemez háttértárral. Grafikus felületű operációs rendszere szöveget és képeket is képes volt megjeleníteni képernyőjén, sőt hálózati képességekkel is felruházták: az első modemes munkaállomásnak tekinthetjük. KAY a hardver megálmodása után szoftvereket is tervezett, amelyek a mai grafikus felületen futó alkalmazások ősének tekinthetők.

---

<sup>5</sup> [http://en.wikipedia.org/wiki/Alan\\_Kay](http://en.wikipedia.org/wiki/Alan_Kay)



A Windows 3.1-ben már megtalálhatjuk ALAN KAY elképzeléseit.



Jelenleg tekinthetjük az OOP paradigmát a moduláris programozás egyfajta, sikeresnek bizonyult továbbfejlesztésének. Az OOP-t egyes kutatók negyedik generációs nyelvnek tekintik, mások a harmadik és negyedik generáció közé helyezik (s így 3.5 generációként aposztrofálják). Utóbbiak indoklása az, hogy az OOP-s megközelítésben a függvények törzse igazából ugyanazon építőegységekből épül fel, mint a moduláris programozásban, a kettő közötti különbség inkább csak a függvények csoportosítási, kódszervezési mód-szereiben rejlik.

### 3. Az OOP alapelvei

---

ALAN KAY eredeti elképzeléseinek megfelelően az OOP nyelvek három alapelvet kell, hogy támogassanak:

**egységbezárás,**

**öröklődés,**

**sokalakúság.**

Az **egységbezárás** (*encapsulation*) elve szerint azok az adatok, amelyek a programunkban valamely összetartozó értékcsoporthoz sorolhatók (pl. egy egyenlet együtthatói), valamint az adatokkal szorosan összetartozó függvények (amelyek az adatokkal dolgoznak) egységbe kell tartozniuk. Az egység jelenthesse azt, hogy a függvények nem hívhatóak meg, csak kitöltött adatokkal, illetve jelenthesse azt is, hogy az adatok csak a függvényeken keresztül változtathatók meg.

Ez az egységeket hívjuk **objektumosztálynak** (röviden osztálynak). Az osztály adattároló elemeit nem változóknak, hanem **mezőknek** (*field*), míg az osztályhoz tartozó függvényeket **metódusoknak** nevezzük. Az OOP nyelvekben is értelmezett a *változó* fogalma, de abban kifejezetten a függvények (metódusok) törzsében deklarált lokális változókat jelöli. A mező a függvények törzsein kívül deklarált, jellemzően dinamikus adattároló egységek neve. Mivel ezek korántsem ugyanazon jellemzőkkel rendelkeznek, így hibának számít, ha nem a megfelelő elnevezést használjuk.

A függvény a továbbiakban a hagyományos (procedurális) nyelvek szerinti megfogalmazásban olyan programozási elem, amely nem része objektumosztálynak. Hívásához egyszerűen le kell írni a függvény nevét. A metódus ellenben olyan függvény, mely valamely objektumosztály része, hívása sokkal bonyolultabb szintaktikai és szemantikai szabályok mentén történik. Emiatt szintén hibás, ha a két elnevezést nem megfelelően használjuk.

*Megjegyzés:* a tisztán OOP nyelvekben a függvény fogalma nem létezik (nem készíthető függvény, csak osztályba ágyazva), csak metódusok írhatóak. Szigorúan véve itt is hibás a metódusokat függvényeknek nevezni, de mivel itt nem értelmezhető félre az elnevezés, így gyakran előfordul mégis a függvény elnevezés használata a metódusokra is. Más vélekedések szerint az osztályszintű metódusokat szabad függvényeknek nevezni, míg a példányszintűek esetén mindenképpen a metódus megnevezést kell használni.

Fontos megjegyezni, hogy az objektumosztály egy absztrakt fogalom (később látni fogjuk, hogy lényegében egy típus). Vagyis önmagában egy objektumosztály megléte nem jelent feltétlenül működőképes adattárolást és funkcionalitást.

---

Az objektumosztály egy modell, egy terv. Hasonlóan, mintha lenne papíron egy autónk, amely tartalmaz adatokat (lóerő, ülések száma, sebességek száma, fékerő, gyorsulás stb.), és funkciókat (motor indul, leáll, gázt ad, fékez, kanyarodik). Ettől még nincs autónk. A terv alapján azonban nemcsak egyetlen autót tudunk készíteni, hanem sokat. A folyamat, amikor egy objektumosztályból ténylegesen létező **példányt** (*instance*) készítünk, az a **példányosítás**. Példányosításkor az osztályban definiált adattároló egységek helyet foglalván ténylegesen bekerülnek a memóriába. Ha több példányt készítünk, akkor ez többször is megtörténik. A példányokat gyakran objektumoknak is nevezzük.

*Megjegyzés:* egyes szövegezésekben keverődik az objektumosztály (osztály) és az objektum (példány) fogalma. Gyakran (hibásan) az objektumosztály elnevezést rövidítik objektumnak (pl. „tervezzünk egy objektumot”).

Az **öröklődés** (*inheritance*) elve szerint ha egy objektumosztály már elkészült (tartalmaz mezőket és metódusokat), és másik, hasonló adattartalmú és funkcionalitású osztályt kívánunk készíteni, úgy a már elkészült osztályt felhasználhatjuk kiindulási alapnak. Ekkor az új objektumosztály esetén deklarálhatjuk a kiinduló osztályt (nevével hivatkozva), és az új objektumosztály automatikusan átveszi az összes mezőt, metódust anélkül, hogy a forráskódban azokat fizikailag le kellene másolni.

A kiinduló osztályt a továbbiakban **szülőosztálynak** vagy **ősosztálynak** (*base class*, *parent class*, *super class*), az új, most készítendő osztályt **gyerekosztálynak** (*derived class*, *child class*) nevezzük. A gyerekosztály tehát minden mezőt, metódust tartalmaz (örököl) a szülő osztályból. Nem egyszerű copy-paste-ről van szó! Mivel a kapcsolat a forráskódban deklarált, így ha a szülőosztályt módosítjuk, és újra fordítjuk a forráskódokat, a gyerekosztály is átveszi automatikusan a módosításokat. Ez gyakori, ha a szülő osztály forráskódjában hibajavításokat végeznek, vagy újabb kiegészítéseket adnak hozzá. A gyerekosztályok az elvnek megfelelően a fordítási folyamat során azonnal és automatikusan átveszik a módosításokat.

A **sokalakúság** (*polymorphism*) a legnehezebben megérthető alapelv, de nagyon fontos. Alapvetően arról szól, hogy az egyes függvények, változók, osztályok többféle jelentéssel is felruházhatóak legyenek ugyanazon forráskódon belül. Az OOP szempontjából úgy értelmezhető, hogy lehessen definiálni egy leírást (*interface*), amelyen keresztül definiálható egy objektum működése (funkciói) anélkül, hogy megadnánk a tényleges tevékenységet, amit a funkció neve rejt.

El tudjuk képzelni azt a szituációt, mikor van egy központi vezérlő egység (tábornok), aki a rábízott elemeket tudja a hadszíntéren mozgatni egyszerű funkciókkal (parancsokkal), mint a „menj előre”, „fordulj balra”, „állj meg”. Másképpen fogalmazva a tábornok bármit elvezényel, elirányít, aki ezt a három funkciót tartalmazza, legyen az gyalogos katona, tank vagy harci vakond. Nyilvánvaló, hogy a funkciók végrehajtása az egyes elemekben teljesen másként van megvalósítva, de ez a tábornokot nem kell, hogy érdekelje. Számára az egyes elemek (példányok) intelligens egyedek, akik ismerik saját magukat, és tudják, mit, hogyan kell végrehajtaniuk. Kívülről elfogadják az utasításokat, de a külvilágnak azon kívül, hogy milyen utasításokat ismernek fel az egyes egyedek, mást nem is kell tudniuk.

A sokalakúság lehetővé teszi igazán magas szintű kódrészek kifejlesztését, melyek sokféle adattípussal is képesek hatékonyan együttműködni. Egy rendezőalgorithmus képes lehet tetszőleges típusú adatok sorozatát rendezni azon elv szerint, hogy a két adatelemet *megkéri*, hogy hasonlítsa össze magukat, adják meg, hogy melyikük a *nagyobb* (jelentsen ez a fogalom bármit). Amennyiben a sorrendjük nem megfelelő, *megkéri* a kollekción (tömb, lista), hogy cserélje fel a két adatelemet.

Ezen OOP alapelv megvalósítása okozza a legtöbb és legbonyolultabb fejlesztéseket. Késoi kötés, típuskompatibilitás, absztrakt metódusok és osztályok, dinamikus típus és egyéb fogalmak szükségeltetnek a teljes megértéshez. (Ezek tárgyalása a jegyzet jelentős részét teszi ki.)

Az alapelvek megoldása nincs szabályozva, ezért az OOP nyelvek között szintaktikai különbségek vannak, sőt több OOP nyelv a fenti elveken túlmutató, hasznos fejlesztéseket is tartalmaz. A C# az egyik legbővebb képességekkel rendelkező OOP nyelv, mely a szintaktika és a szemantika szempontjából is nagyon letisztult megoldásokat tartalmaz. Alapos megismerése után más OOP nyelveken programozva sok teljesen megegyező, vagy nagyon hasonló megoldással találkozhatunk, így a C# OOP képességeit tanulmányozva nagyon jó alapozást kaphatunk ebben a témakörben.

## 4. Az imperatív nyelvek osztályozása

---

Az objektumorientált programozás bizonyos alapelvek meglétét feltételezi a választott programozási nyelven. Elvei összeegyeztethetőek a hagyományos imperatív, eljárásorientált programozási nyelvek elveivel, ezért nagyon gyakori az, hogy egy már meglévő hagyományos programozási nyelv következő verziójába bevették az OOP alapelveket is. Az így létrejött programozási nyelv egyszerre hordozza az procedurális és az OOP szemléletet.

Ennek megfelelően az imperatív nyelveknek három szintjét különböztetjük meg:

**Procedurális programozási nyelv:** Nem alkalmazza az OOP, csak az eljárásorientált programozási nyelvek elveit. Ilyen nyelv például a Pascal és a C. Ezeken a nyelveken a függvény fogalmán kívül a „globális” változó is értelmezve van, mely egy adott modulon belül minden függvény számára hozzáférhető és módosítható. A lehetőség sajnos arra ösztönözheti a programozókat, hogy az adatok jelentős részét így tárolja, elkerülvén így a paraméterátadás és a függvény visszatérési értékének használatát. A nyelvek első verziói jellemzően még az OOP elvek kidolgozása előtt születtek meg.

**Tisztán OOP nyelv:** a nyelv tervezésekor már figyelembe vették az OOP alapelveit, sőt, a hagyományos szemlélet néhány fogalmát teljesen ki is dobták. Ennek megfelelően nincs függvény – mivel az egységbezárás elvét maximálisan alkalmazva, vagyis hogy minden egyes függvényt osztályba kell zárni – metódussá alakul. Nincsenek globális változók, hiszen minden ilyet is osztályba kell zárni, azok mezővé alakultak. Ezzel együtt persze megjelennek kisebb nehézségek is, látni fogjuk, hogy a szélsőségek kellemetlenségekké alakulnak. Hátrányaival szemben komoly előnyei vannak ezen nyelveknek és mára már bizonyítottak az ipar kihívásaival szemben is. Sikerük bizonyítja az előnyök erősségét. Ilyen nyelvek például a Java és a C#.

**OOP támogató nyelv:** egy meglévő hagyományos programozási nyelvet jellemzően sok programozó ismer, amelyben nagy mennyiségű forráskód készült már el korábban. A kompatibilitás, és a tudás megőrzése miatt érdemesnek tűnt az alapvetően nem OOP elvekre felkészített nyelvek szintaktikáját módosítani, és a beleépíteni az OOP ismeretek alkalmazhatóságát is. Az ilyen „felemás” nyelveken mindkét programozási szemlélet alkalmazható. Vagyis egy időben készíthetünk osztályon kívüli függvényeket, használhatunk globális változókat, valamint készíthetünk objektumosztályokat, mezőket, metódusokat. Egy felkészült, tapasztalt programozó kezében egy ilyen nyelv nagyon jó eszköz lehet. Egy kezdő programozó számára azonban sokszor ellentmondásosnak tűnik a szintaktika, nehezen tud választani melyik paradigmát alkalmazza az adott pillanatban. Ráadásul az OOP elvek utólagos beillesztése gyakran elbonyolította, nehézkessé tette a korábban egyszerű és letisztult szintaktikát. Ilyen nyelv például a Delphi vagy a C++.

Az OOP elvek használata mellett az eljárásorientált nyelvek minden lehetősége lefedhető, kis kompromisszumok mellett. Ugyanakkor egy szintaktikailag jobban letisztult, erősebb lehetőségekkel rendelkező megvalósítást kapunk, mely használatával biztonságosabban, kevesebb hibalehetőség mellett programozhatunk.

---

## 5. Egyszerű példa egy OOP problémára

---

Tegyük fel, hogy programunk téglalapokkal dolgozik. Téglalapunk egyik éle minden esetben vízszintes. A téglalapot vízszintes élei közül az alsó él bal oldali csúcsának  $x, y$  koordinátája, ezen vízszintes él (a oldal) hossza és a függőleges élének (b oldal) hossza jellemzi. A programnak a téglalap adatainak tárolásán túl, tudnia kell kiszámolni a téglalap területét, kerületét, és meg kell határoznia egy tetszőleges  $x, y$  koordinátájú pontról, hogy az adott téglalap belsejébe esik-e vagy sem.

Hagyományos programozási stílusban a téglalap adatait rekordba szerveznénk:

```
struct teglalap
{
    public double x;
    public double y;
    public double a_oldal;
    public double b_oldal;
}
```

Esetleg a pontot leíró rekordot is elkészítenénk:

```
struct pont
{
    public double x;
    public double y;
}
```

Végül elkészítenénk a szükséges függvényeket:

```
public static double kerulet(teglalap r)
{
    return (r.a_oldal + r.b_oldal) * 2;
}

public static double terület(teglalap r)
{
    return r.a_oldal * r.b_oldal;
}

public static bool benne_van_e(teglalap r, pont p)
{
    return (r.y <= p.y && p.y <= r.y + r.b_oldal &&
            r.x <= p.x && p.x <= r.x + r.a_oldal);
}
```

Egy lehetséges felhasználása a kódnak, egy 'Main()' függvény:

```
public static void Main()
{
    teglalap t = new teglalap();
    t.x = 10;
    t.y = 12;
    t.a_oldal = 22;
    t.b_oldal = 4;
    //
    double k = kerulet(t);
    double t = terület(t);
    //
    pont f = new pont();
    f.x = 12;
    f.y = 15;
    bool belso = benne_van_e(t, f);
}
```

Vegyük észre, hogy az adatok tárolását leíró adatszerkezet ('struct teglalap'), és adatszerkezettel dolgozó függvények kapcsolata nagyon laza. Felfedezhetjük a kapcsolatot, hiszen a függvények egyik paramétere egy téglalap típusú adat. De képzeljük el, hogy ha ez a néhány blokk a forráskódunkban szétszórtnan helyezkedik el, akkor az adatszerkezet módosítása után sok ponton jelentkeznek a javítási igények. További észrevétel az is, hogy az adatszerkezet „belsejét”, filozófiáját a feldolgozó függvényeknek alaposan ismernie kell. Zavaró, hogy a 'kerulet' függvényről nem tudjuk, minek számolja ki a kerületét, csak ha a paraméterezését is átnézzük (onnan tudjuk, hogy téglalap kerületet számol, mert paraméterként téglalapot adunk át).

Nézzük ugyanezt a példát OOP stílusban. Az osztályok kulcsszava 'class', ennek segítségével építjük fel a kódot:

```
class teglalap
{
    protected double x;
    protected double y;
    protected double a_oldal;
    protected double b_oldal;
    //
    public double kerulet()
    {
        return (a_oldal+b_oldal)*2;
    }

    public double terület()
    {
        return a_oldal*b_oldal;
    }

    public bool benne_van_e( pont p )
    {
        return (y<=p.y && p.y<=y+b_oldal &&
                x<=p.x && p.x<=x+a_oldal);
    }
    //
    public teglalap(double pX, double pY, double pA, double pB )
    {
        x = pX;
        y = pY;
        a_oldal = pA;
        b_oldal = pB;
    }
}
```

Az adatokat leíró mezőket a 'class' belsejébe helyezzük, összezárjuk egyetlen blokkba a függvényekkel (egységbezárás). A függvények részeivé válnak az adatstruktúrának, emiatt nem kell paraméterként kapni a téglalapot, mivel minden függvény eleve a saját téglalap mezőivel tud dolgozni. Ha két téglalapunk lenne, akkor az első a saját (nem paraméter), a második téglalap természetesen már paraméter lenne. A függvények (hogyan elérhessék a saját téglalap mezőit) nem tartalmazzák a 'static' módosítóval, hanem módosító nélküliek. Az utolsó, 'teglalap' nevű függvény speciális feladatú, a paramétereiben megkapott értékeket átmásolja a mezőkbe, meghatározva azzal a kezdő állapotot. Ez lesz a későbbiekben ismertetett konstruktor. Az előző 'Main()' függvénnyel egyforma feladatú, ám OOP stílusú 'Main()' függvény a következőképpen néz ki:

```
public static void Main()
{
    teglalap t = new teglalap(10,12,22,4);
    //
    double k = t.kerulet();
    double t = t.terulet();
    //
    pont f = new pont();
    f.x = 12;
    f.y = 15;
    bool belso = t.benne_van_e( f );
}
```

Az első sorban a téglalap példány ('t') elkészítése látszik. A 'new' operátor a szükséges memóriát foglalja le a mezőknek, mögötte a 'teglalap' nevű függvény (konstruktor) hívása látszik. A négy érték a mezők kezdőértékeit adják meg. A téglalap példányunknak nemcsak mezői, de függvényei is vannak, melyek hívásához először a példányt ('t'), a pont operátort, majd a meghívni kívánt függvény nevét kell megadni ('t.kerulet()'). Ez azt jelenti „számold ki a 't' példány mezői alapján a téglalap kerületét”. A 'kerulet()' függvény beseljében szereplő 'a\_oldal' és 'b\_oldal' ez alapján a 't' példány mezőit fogja jelenteni.

Az ebben a stílusban megírt kód – azok számára akik otthonosan mozognak az OOP területén – sokkal olvashatóbb. Jellemző „a készítsünk példányt és lássuk mit tud” gondolkodás. Ennek jegyében ha tudni akarjuk mit lehet egy téglalappal készíteni, csak leírjuk a 't.' párost a Visual Studioban (továbbiakban VS), és máris sorolja milyen mezői, milyen függvényei vannak. A fejlesztőeszköz tudja, hogy a pont, mint kiválasztó operátor azt jelöli, hogy a 'class teglalap' belsejében megadott mezőre vagy függvényre akarunk hivatkozni. A hagyományos, struktúrált programozásban készíthető lett volna hasonló segítség, a fejlesztőeszköz ott is ki tudta volna keresni azokat a függvényeket, amelyek paraméterezése szerint 'teglalap' típusú adatokkal dolgozik, de nyilván sokkal nagyobb idő- és energiaráfordítás árán. Később más eszközöket is megismerünk, amely az összetartozó kódok, kódrészletek csoportosítására is szolgálhat<sup>6</sup>.

---

<sup>6</sup> Névterek (namespace).

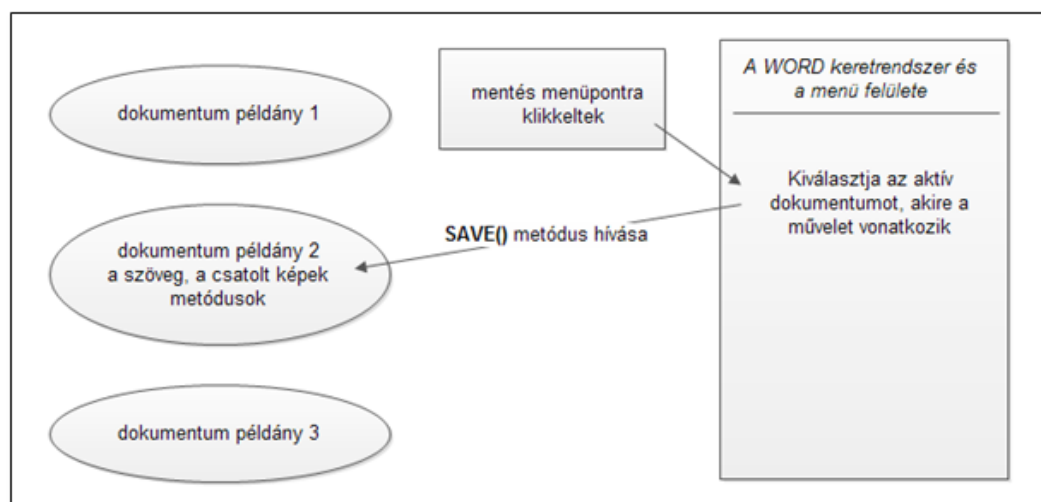


Egy programozási nyelvnek sok jellemzője van. Fontos, hogy a nyelv jól használható alaptípusokkal rendelkezzen (szám, betű, szöveg, logikai stb.), ezekre bőséges operátorműködés legyen, hogy kényelmesen lehessen kifejezéseket készíteni. A programvezérlési szerkezetek ismerős működéssel bírjanak, gyorsan meg lehessen szokni használatukat. Az egymásba ágyazást jól áttekinthető blokk-szerkezetek jellemezzék. Ezek azok az alapkövetelmények, melyek ha adottak, a programozók elkezdhetnek függvényeket, saját modulokat fejleszteni. További követelmény (egy nyelv sikerességének alapfeltétele), hogy a nyelvhez eleve létezzen bőséges függvénygyűjtemény. Így a programozók a magasabb szintű feladatokra tudnak koncentrálni, melyhez jól dokumentált, kézreálló alap függvényeket tudnak felhasználni. A túl bőséges függvénygyűjtemény azonban már hátrány is lehet, hisz a programozók képtelenek több ezer függvény nevét megjegyezni. Ha a függvények használatát több perces keresgetés előzi meg, akkor csökken a teljesítmény. A Win32 programozási környezetben több mint 2000 függvény alkotja azt az alapot, amelyből a fejlesztés kiindulhat. Ha ilyen sok függvényünk van, akkor a függvények nevei már nem segítenek eleget. Képzeljük el mikor egy programozó a Windows platformra írt programjában az egér kurzort az alap mutató nyíl kinézetéről homokórára akarja módosítani, amíg a programja a számolási feladatot végzi. Hogy hívhatják az egér kurzort átállító függvényt? SetMousePicture? MouseSetCursor? ChangeMouseIcon? (A helyes válasz egyébként LoadCursor + SetCursor páros.)

A Microsoft.NET 1.0 Base Class Library több mint 7000 osztályt tartalmaz, osztályonként számos függvénnyel. Ahhoz, hogy egy ilyen, már mennyiségileg is problémás library-t hatékonyan tudjon egy programozó kihasználni, OOP szemléletre van szükség.

## 6. Az adatrejtés

A nagyobb méretű kódbázison alapuló (esetleg több programozó munkáját felölelő) projektek összeállítása úgy történik, hogy önálló feladatkörrel rendelkező objektumosztályokat fejlesztünk ki. Az osztályok adatokat (mezőket) tartalmaznak, valamint számos függvényt (metódust), melyeken keresztül a példányok a tényleges tevékenységeket képesek elvégezni. Például a Word dokumentumszerkesztő esetén objektum tárolja a dokumentum szövegét, a fájl nevét és számos egyéb információt (utolsó módosítás dátuma stb.). A funkciója lehet a 'mentes()', melynek során csak a memóriában tárolt friss módosítások kerülnek tárolásra a lemezen. Az objektum maga tárolja a mentéshez szükséges fájlnevet is, így a funkció meghívása akár paraméter nélkül is elképzelhető.



Az objektum metódusai folyamatosan dolgoznak a mezőkben tárolt adatokkal. Első lépésben tanuljuk meg, hogyan lehet objektumosztályokat készíteni, mezőkkel, a külvilágból példányosítani, majd adatokat elhelyezni egy mezőbe!

Feladatunk elsőként legyen egy általános iskolás diák (Kiss Lajos, 12 éves, 7. C osztályos) adatainak tárolása!

```
class diák
{
    int életkor;
    string neve;
    int hanyadikos;
    char osztaly;
}
```

A fentiekben leírt kód inkább egy „rekord”, mint OOP, mivel egyelőre csak mezők vannak benne. Szükségünk lesz arra a kódrészre is, amelyik a példányosításért, az adatok feltöltéséért és úgy általában a példányunk működtetéséért felelős.

Ezt írjuk a 'Main()' függvénybe. Helyezzük a 'Main()' -t egy különálló osztályba. (Ezt a továbbiakban már nem fogjuk részletezni, de a jegyzet későbbi részeiben is a Main külön osztályba kerül.)

```
public static void Main()
{
    diak d = new diak();
    d.eletkor = 12;
}
```

int diak.eletkor

Error:  
'ConsoleApplication2.diak.eletkor' is inaccessible due to its protection level

Észrevehetjük, hogy a kód máris hibás. A VS kijelzi, hogy a 'd' példánynak nincs elérhető 'eletkor' mezője („*diak.eletkor is inaccessible due to its protection level*”<sup>7</sup>). Az új fogalom, amivel meg kell ismerkednünk a **védelmi szint** (*protection level*).

Három védelmi szint<sup>8</sup> áll rendelkezésre az OOP világában:

**private,**  
**protected,**  
**public.**

Elsősorban meg kell érteni, hogy a védelmi szintek **hatáskör módosítók!** A hatáskör, mint emlékszünk az azonosító azon tulajdonsága, amely megadja, hogy a forráskód mely részében szabad azt az azonosítót felhasználni, mely pontokon ismeri fel a fordító az azonosítót.

Az alapértelmezett védelmi szint a *private*, amely angol szó a szótár szerint bizalmas, magántermészetű, titkos, zártkörű stb. jelentésekkel bír. Az OOP világában is nagyjából helytálló a fordítás. Ha nem jelöljük külön a védelmi szintet, akkor minden esetben a *private* védelmi szint lép életbe. Ez a védelmi szint a mezők (és a metódusok) elérhetőségét az őt tartalmazó osztály belsejére korlátozza:

```
class diak
{
    private int életkor;
    private string neve;
    private int hanyadikos;
    private char osztaly;
}

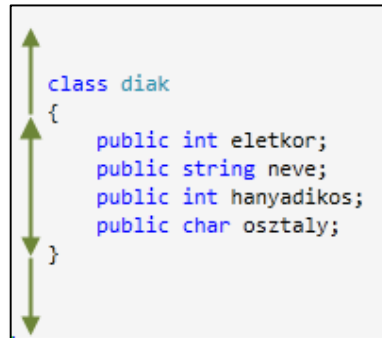
class FoProgram
{
    static void Main()
    {
        diak d = new diak();
        d.eletkor = 12;
    }
}
```

<sup>7</sup> A 'diak' osztály 'eletkor' mezője nem elérhető a védelmi szintje miatt

<sup>8</sup> Valójában 5, a maradék kettőt később, a DLL kapcsán ismertetjük.

A 'FoProgram' class, annak belseje, a 'Main()' függvény, e területen kívül esik, így ott a private mező nem érhető el.

Amire szükségünk van, az a *public* védelmi szint, mely nemcsak az adott osztály, de bármely más osztálybeli kód, így a 'Main()' függvény számára is a hozzáférést enged:



```
class diak
{
    public int eletkor;
    public string neve;
    public int hanyadikos;
    public char osztaly;
}
```

Ennek fényében megírható a főprogram:

```
public static void Main()
{
    diak d = new diak();
    d.eletkor = 12;
    d.neve = "Kiss Lajos";
    d.hanyadikos = 7;
    d.osztaly = 'C';
}
```

### 6.1. A mező értékének védelme

Az objektumok mezőibe a külvilág helyez el értékeket. A külvilág e mezőket értékadó utasítás segítségével lényegében bármely pillanatban megváltoztathatja. A metódusok, amelyek a mezőkben lévő adatokkal dolgoznak, minden egyes alkalommal le kellene, hogy ellenőrizzék, hogy az mezőkben lévő adatok megfelelőek-e. Tételezzük fel, hogy az adott iskolában csak A, B és C osztályok vannak. Ennek megfelelően az 'osztály' mező értéke csakis e 3 betű lehet. További problémák elkerülése végett nem megengedhető az osztálynevek esetében a kisbetűk használata sem. Megoldás lehetne a problémára enum megadása, de akkor ugyanezen osztályra ugyenezen enum már nem lenne használható egy másik iskolában, ahol esetleg D és E osztályok is vannak. De maradjunk az eredeti problémánál: szeretnénk elérni, hogy az 'osztály' mezőbe csak A, B, C értékek valamelyike kerülhessen be.

Célunk nyilvánvalóan nem valósítható meg, ha a mező publikus, és hozzáférhető a külvilág számára; mivel a külvilág a mező értékét futás közben tetszőlegesen sokszor módosíthatja, lényegében bármely pillanatban átírhatja. Amennyiben lennének metódusaink, melyek a mezőben lévő adatokkal végeznek műveletet, minden egyes alkalommal le kellene ellenőrizniük azok megfelelőségét. E plusz műveletek jelentősen lassítanák a kód futását. Nyilvánvaló például, hogy amennyiben mégsem módosította a külvilág a mező értékét, úgy felesleges az újraellenőrzés.

A problémára a hagyományos szemléletben, procedurális nyelvek esetén nincs jó megoldás. A programozók esetleg leírhatják a dokumentációban, hogy „ügyeljünk erre”. Elvileg biztosíthatunk a mezőbe íráshoz függvényt, ami ellenőrzi, hogy jó-e az osztály betűjele, de használata megkerülhető, mivel a függvény hívása nélkül is beállítható az érték.

```
static bool osztalyBeallit(diak p, char osztaly)
{
    if (osztaly != 'A' && osztaly != 'B' && osztaly != 'C')
        return false;
    else
    {
        p.osztaly = osztaly;
        return true;
    }
}
```

```
diak d = new diak();
osztalyBeallit(d, 'C');
diak k = new diak();
osztalyBeallit(k, 'X');
```

## 6.2. A megoldás metódussal

Az OOP világában azonban van megoldás a problémára, amely pontosan a védelmi szinteken alapszik. A mező védelmi szintjét nem publikusra vesszük, hiszen akkor a külvilág direktben tud a mezőbe értéket írni. A fordítóprogram csak a mező alaptípusát ellenőrzi értékadásakor, vagyis tetszőleges karaktert fogad el. Tehát a védelem megkerülhető. Válasszuk a mező elérhetőségét tehát `private`-ra. Így a külvilág nem fog tudni hibás értéket beírni a mezőbe, de sajnos helyes értéket sem, ugyanis a mezőhöz mindenféle hozzáférése tiltott lesz. Tehát készítsük el a fenti `'osztalyBeallit'` függvény OOP-s megfelelőjét, metódusként (és ne használjuk a `'static'` módosítót a függvény írásakor):

```
class diak
{
    public int életkor;
    public string neve;
    public int hanyadikos;
    // nem férhet hozzá kívülről, mert private
    private char osztaly;
    // ez meghívható kívülről, mert public
    public bool osztalyBeallit(char osztaly)
    {
        if (osztaly != 'A' && osztaly != 'B' && osztaly != 'C')
            return false;
        else
        {
            this.osztaly = osztaly;
            return true;
        }
    }
}
```

A metódus elérhetősége `'public'`. Védelmi szintjei megegyeznek a mezőknél leírtakkal.

A metódusok esetén a védelmi szint a *meghívhatóságot* jelöli, vagyis a `public` metódus nemcsak az osztály belsejében megadott más metódusokból hívható meg, hanem a teljes

programkód tetszőleges pontjáról. A metódus *bool* visszatérési értékű, megadja, hogy az osztály beállítását sikeresen végrehajtotta-e vagy sem. A metódusnak nincs szüksége a 'diak' paraméterre, mert az része egy konkrét diák példánynak. Az aktuális diák példányt, amelynek a mezőivel a metódus dolgozik a metódus törzsében a *this* kulcsszó azonosítja. Ennek megfelelően a *this.osztalynev = osztalynev;* sor értelme: a mezőbe helyezzük el a paraméterbeli értéket.

Emiatt a mezőbe már nem lehet direktben értéket írni (*private* mező nem elérhető), csak a publikus metóduson keresztül:

```
diak d = new diak();
// ez már nem megy a védelmi szint miatt
// d.osztaly = 'X';
bool siker = d.osztalyBeallit('X');
```

A fordítóprogram segítségével ekkor odáig terjed ki, hogy a nem publikus mezőbe való közvetlen beleírást megtiltja. A mező értékének változtatását a külvilág így csak a metódus hívásán keresztül tudja kezdeményezni. A metódus azonban minden esetben ellenőrzi a külvilág felől érkező értéket, és csak a kritériumoknak megfelelő értéket fogadja el és helyezi el a mezőbe. Emiatt a mezőben lévő érték minden esetben megbízható<sup>9</sup>, a metódusainkban nem szükséges azt újra és újra ellenőrizni.

Miközben biztosítottuk a külvilág számára a beállítás (írás) lehetőségét, ne feledkezzünk meg az olvasás lehetőségéről sem! Jelenleg, a *private* elérhetőség miatt nemcsak a direkt írás, de a direkt olvasás is tiltott.

A megoldást ismételten a metódusok írása jelenti, mivel a metódus az osztály része és így a *private* mezőkhöz is hozzáfér. A metódus ugyanakkor publikus, így a külvilág meg tudja hívni:

```
class diak
{
    public int életkor;
    public string neve;
    public int hanyadikos;
    // nem férhet hozzá kívülről, mert private
    private char osztaly;
    // ez meghívható kívülről, mert public
    public bool osztalyBeallit(char osztaly)
    {
        if (osztaly != 'A ' && osztaly != 'B' && osztaly != 'C')
            return false;
        else
        {
            this.osztaly = osztaly;
            return true;
        }
    }
    // a mező értékének kiolvasása
    public char osztalyLekerdez()
    {
        return this.osztaly;
    }
}
```

<sup>9</sup> Valójában nem teljesen igaz, hiszen a mező kezdőértéke még nem felel meg a kritériumoknak, de erre csak a konstruktor technika megismerése után tudunk megoldást keresni.

A főprogram már használja ezt az új függvényt:

```
diak d = new diak();
//
bool siker = d.osztalyBeallit('X');
char jelenlegi = d.osztalyLekerdez();
```

Jegyezzük meg, hogy a programozók gyakran az angol megnevezésekkel illetik a mezőket és metódusokat. Ennek sok oka van. Egyik, hogy az angol elnevezés gyakran rövidebb és kifejezőbb, mint annak magyar megfelelője, valamint nem kell az ékezetes betűkkel sem bajlódni. Így az író metódus neve (névadási hagyomány szerint) setXXXX, az olvasójé pedig getXXXX, ahol az XXXX helyébe a mező neve kerül. Lássuk a diák (*student*) osztály életkor (*age*) értékére vonatkozó megoldást. Az életkor csak 6 és 18 év közötti számérték lehet:

```
class student
{
    private int age;
    // írás
    public bool setAge(int value)
    {
        if (value < 6 || value > 18)
            return false;
        else
        {
            this.age = value;
            return true;
        }
    }
    // olvasás
    public int getAge()
    {
        return this.age;
    }
}
```

A főprogram részlete:

```
student d = new student();
d.setAge(12);
//
int actAge = d.getAge();
```

A setXXXX és getXXX névadás bár csak hagyomány, de használata sokat segít, mivel a mező nevének ismeretében a metódusok neve kitalálható. Java nyelvben ennél nagyobb jelentősége is van. Ott a property fogalma (lásd később) ismeretlen, de az azonos utótagú get... set... metóduspárt egyes fejlesztőrendszerek felismerik és property szintre emelik a kezelését.

### 6.3. Hibajelzés

Az OOP világában nem szokás a beállító (set) metódust *bool* visszatérési értékként definiálni. Az általa keletkező problémákról később, a kivételkezelés fejezetben lesz szó, ami majd lehetővé teszi a tényleges és alapos megértést. Már most jegyezzük meg, hogy OOP környezetben, ha egy metódust a külvilág *megkér* valamely tevékenység elvégzésére, melyet a metódus önhibáján kívül (a külvilág valamely hibájából) nem tud végrehajtani, akkor azt kivétel feldobásával jelzi. Egyelőre annyi is elég nekünk, hogy *false* értékkel való visszatérés helyett a *throw* kulcsszó segítségével jelezzük a hibát. A kivételkezelés alaposabb megértéséig két módot javasolunk az alkalmazásra. A

```
throw new ArgumentException("... a hiba megfogalmazása ...");
```

illetve a

```
throw new Exception("... a hiba megfogalmazása ...");
```

formákat.

Előbbit használjuk, ha a paraméterbeli érték (argumentum) hibájából nem lehet a tevékenységet végrehajtani, utóbbit minden más esetben. A „...hiba megfogalmazása...” részben szokás leírni a hiba pontos okát.

A 'throw' utasítást fogjuk fel egyelőre úgy, mintha a return egyfajta alternatívája lenne, vagyis ha a metódus 'throw' parancsot hajt végre, utána már további utasításokat nem fog, a vezérlés visszakerül a hívás helyére (mint return esetén). A különbség az, hogy a return esetén a visszatérés után a program fut tovább, míg 'throw' esetén a program tudomásul veszi, hogy hiba történt, és nem hajt végre további utasításokat, és a hívó kód is visszatér a hívás helyére. Ez addig folytatódik, míg végül a Main()-beli kiinduló hívás helyére tér vissza a végrehajtás (kihagyva minden köztes utasítást), végül a Main() is befejeződik<sup>10</sup>. Tehát a 'throw' végrehajtása további utasítások végrehajtásának átlépése miatt végső soron a program leállítását okozza. A megadott hibaüzenet szövege általában megjelenik a felhasználónak, aki ez alapján tudja értesíteni a programozót a hibáról.

```
class student
{
    private int age;
    // írás
    public void setAge(int value)
    {
        if (value < 6 || value > 18)
            throw new ArgumentException("Hibás, csak 6..18 fogadható el");
        else
            this.age = value;
    }
    // olvasás
    public int getAge()
    {
        return this.age;
    }
}
```

<sup>10</sup> A folyamat megszakítható a catch utasítás segítségével (lásd később).



Ennek megfelelően a `setAge()` metódus nem `bool`, hanem `void` lett. Ha a hibavizsgálat szerint probléma áll fenn, akkor a `throw` segítségével jelezzük, hogy nem sikerült a művelet végrehajtása. Az értékadó utasítást nem szükséges `else` ágba rakni, hiszen ha `throw`-t hajtunk végre, akkor az értékadás már nem történik meg. Ide a vezérlés csak akkor juthat el, ha nem volt `throw`, vagyis az érték (*value*) megfelelő.

## 6.4. Protected a private helyett

A két védelmi szint, a *public* és a *private* mellett a harmadikról, a *protected* védelmi szintről eddig nem esett szó. A *protected* védelem a kettő közé esik, de megértéséhez szükséges a gyerekosztályok fogalma. Az ősosztály valamely továbbfejlesztését gyerekosztálynak nevezzük. A gyerekosztály örökli az ősosztály mezőit és metódusait. Alap-sabban erről később lesz szó, addig nézzünk egy egyszerű példát:

```
class student
{
    private int grade;
    //
    public void setGrade(int newGrade)
    {
        if (newGrade < 1 || newGrade > 8)
            throw new ArgumentException("only 1..8 can be accepted");
        else
            this.grade = newGrade;
    }
    // ... cont ...
}
```

```
class fejlettebbDiak : diak
{
    public void evetLep()
    {
        if (this.hanyadikos == 8)
            ballagas();
        else
            this.hanyadikos++;
    }
    public void ballagas()
    {
        // nóta 1.. 2.. 3... ballag már a vén diák...
    }
}
```

A `fejlettebbDiak` osztály gyerekosztálya a *diak*-nak. Következésképpen eleve tartalmazza mind a 4 mezőt (pl. *hanyadikos* és *osztaly*), valamint a *hanyadikosBeallit* metódust. E mellett szeretne egy új metódust bevezetni, amelyet az ősosztály nem tartalmaz: az *evetLep()* függvényt. A metódusban használnánk az örökölt *hanyadikos* mezőt, de a VS hibát jelez: *protection level*-re hivatkozik. A *private* védelmi szint mellett ugyanis a mező öröklődik, de a hatásköre nem terjed ki a gyerekosztály belsejére. Ellentmondásos szitu-

áció, de később értelmet fog nyerni. Egyelőre annyit jegyezzünk meg, hogy ha a gyerekosztályban is el szeretnénk érni az örökölt mezőt, akkor a *protected* védelmi szintre lesz szükségünk. A *protected* védelmi szint a mezőhöz való hozzáférést az osztály kódján kívül a gyerekosztályok belsejére is kiterjeszti, de idegen kódokra, idegen osztályok belsejébe (mint pl. a `Main()` függvény) már nem.

A mezőkhöz való direkt hozzáférés bizalmi kérdés. Aki hozzáférhet a mezőhöz, az a típusa által megengedett bármely értéket elhelyezhet benne, ami korántsem biztos, hogy megfelel az objektumnak is. Az OOP-s kód ennek megfelelően 3 területre osztható fel megbízhatóság szempontjából.

Az első (legbelső) bizalmi körben csak maga az objektumosztály és a saját metódusok foglalnak helyet. Ez a *private* szint. A példányok mezőjéhez csak a publikus metódusokon keresztül férhet bárki hozzá, ők pedig gondosan ellenőrzik a külső körökből érkező adatokat, mielőtt azt elfogadnák vagy elhelyeznék valamely mezőben.



A második kör a *protected* szint. Itt a gyerekosztályok, és azok gyerekosztályai (unokák stb.) tartoznak. Egy *protected* mezőhöz a gyerekosztálybeli metódusok is hozzáférhetnek direktben. Ez a bizalmi szint a leggyakoribb, mert a direktben írás/olvasás művelete itt a leggyorsabb. A művelet ugyanis a publikus metódusokon keresztül sokkal lassabb. A gyerekosztályok önálló osztályok, önállóan vállalnak saját maguk működéséért felelősséget, így ha elrontják a mezőbeli értékek kezelését – hát magukra vessenek. Később látni fogjuk, hogy a gyerekosztályoknak joguk van az örökölt metódusokat felüldefiniálni (újraírni), így ha a gyerekosztály rosszul élne a mezőhöz való hozzáférés jogával, és hibás értéket helyezne bele, és ettől a mi valamely metódusunk hibásan működne, még mindig tudunk arra hivatkozni, hogy miért nem írta újra, miért nem alakította át a szóban forgó metódust ennek megfelelően.

A fentiek miatt a *private* védelmi szint használata valójában nagyon ritka, hisz a gyerekosztályok elől csak nagyon indokolt esetben rejtünk el mezőt. A leggyakoribb védelmi szint a *protected*, illetve a védelem nélküli mezők esetén a *public*.

A *public* védelmi szint a védelem hiányát jelöli. A publikus mezőkhöz mindenki hozzáférhet, értékét bármikor átírhatja. A publikus mezőkben lévő értékek ezért megbízhatat-

lanok, használatuk esetén érdemes minden egyes alkalommal ellenőrizni a bennük lévő értéket.

### 6.5. Miért a 'private' az alapértelmezett védelmi szint?

Amikor nem írunk védelmi szintet, az egyenrangú a *private* védelmi szint kiírásával. Ez az alapértelmezett védelmi szint. Miért éppen ez? Miért nem mondjuk a *protected* vagy a *public*?

Minden választás esetén vannak érvek és ellenérvek. A *private* mellett szóló első érvünk az, hogy leggyakrabban azért nincs feltüntetve a védelmi szint, mert a programozó egyszerűen megfélekedezik róla, így automatikusan a legerősebb védelemi szint lép életbe. Az osztály fejlesztője ebből mit sem érzékel, hisz az osztályon belüli metódusok tudják a mezőt használni. Ugyanakkor a külvilágot fejlesztő programozók azonnal érzékelik a „feledékenység” hatását, mivel ők semmilyen módon nem képesek a mezőhöz hozzáférni. Első dolguk, hogy jelezzék a feledékeny programozó felé a problémát, ezzel lehetőséget adva neki, hogy átgondolja a védelmi szint módosítását, a védelmi szint esetleges enyhítését.

Amennyiben a feledékenység hatására a *public* lépne automatikusan érvénybe, ő lenne az alapértelmezett védelmi szint, a feledékeny programozó akkor sem érzékelné ennek hatását, hisz az osztálybeli metódusokban ekkor is tudná a mezőt használni. A külvilág is érzékelné, hogy ő is képes a mezőt elérni, írni és olvasni is: de nem valószínű, hogy jeleznék a programozónak a „problémát”, csendben élveznék a feledékenység előnyeit.

A Delphi nyelvben van még egy jelenség. A Delphi szerint a mezők és metódusok védelmi szintjei csak az adott „forráskódon”, modulon kívül fejtik ki hatásukat. Ha ugyanabba a forráskódba rakjuk az osztályt és a külvilágot, akkor a külvilágbeli kód (pl. az ottani `Main()` függvény) problémamentesen eléri akár a *private* mezőket is. Ugyanakkor ha ezt a jól működő, letesztelt külvilágbeli kódot áttesszük (kiemeljük) egy másik forráskódba, akkor a fordító elkezd jelezni a védelmi szint megsértéséből fakadó hibákat. Oka, hogy a Delphi fordító feltételezi, hogy egy forráskódot egy programozó ír. Vagyis ugyanazon forráskódon belüli kód mindig megbízható, ott a védelmet még nem kell alkalmazni. Amint két külön forráskód van, azt akár két külön programozó is készíthette, a védelmet máris alkalmazni kell. Nem szerencsések azok a nyelvek, amelyeknek a szintaktikai szabályrendszerébe ilyen kivételek kerülnek be, de a hatékonyság szempontjából a megoldás előnyös. A „nem megbízható kód” a védett mezőkhöz a `property-n`, annak `'get'` és `'set'` részein keresztül férhet csak hozzá, míg a „megbízható kód” a mezőket közvetlenül írhatja és olvashatja. Ilyen szempontból a „megbízható kód” fogalma kiterjesztésre kerül az osztályt tartalmazó forráskód minden részére.

## 6.6. Property

A **property** (tulajdonság, jellemző) a védelemhez kapcsolható fogalom. Pusztán az OOP alapelvekből nem következik a léte, ezért vannak olyan OOP nyelvek, melyekben nem létezik a property fogalma, más nyelvek esetén pedig a megoldási módja különbözik. Most a C# nyelvi megvalósítást fogjuk megismerni.

A property egy kényelmi funkció, szintaktikai cukorka (*syntax sugar*), amely arra szolgál, hogy a sokszor használatos programozási elemeket olvashatóbbá, könnyebben használhatóbbá tegye. Arról van szó, hogy a *protected* rejtett mező olvasásához és írásához metódusokat készítünk. Ennek során gömbölyű zárójeleket is kell használni, valamint a mezőbe írás művelete (ami szokásosan egy értékadó utasítás kellene, hogy legyen) is függvényhívás, ahol az új érték a paraméterben kerül átadásra. Ez a tényleges tevékenység (mezőbeli új érték elhelyezése) szokásos külalakjától nagyon távol álló szintaktika.

A property szóra ha rákeresünk a szótárban, nem sok segítséget kapunk: a tulajdonság, ingatlan, birtok, vagyon stb. Az informatika világában a „jellemző” szóval tudjuk talán fordítani, de elterjedt a „tulajdonság” is. A property-t ha legegyszerűbben akarjuk megfogalmazni, akkor a „virtuális mező”-nek foghatjuk fel. Azért virtuális, mert nem létező mező, de szintaktikailag úgy néz ki, úgy viselkedik, mintha mező lenne. Típusa van, kiolvashatjuk az értékét, és el helyezhetünk benne a szokásos értékadó utasítással új értéket.

Ugyanakkor a property nem fizikailag létező mező. A property nem foglal el a memóriában helyet a példányok esetén – ilyen szempontból úgy viselkedik mint a metódusok. Valójában a háttérben mint látni fogjuk tényleges metódusokról van szó, csak a használata, meghívása szokatlan.

```
class diak
{
    // a védett mező
    protected int _hanyadikos;
    // és a publikus property
    public int hanyadikos
    {
        get
        {
            return this._hanyadikos;
        }
        set
        {
            if (value < 1 || value > 8)
                throw new ArgumentException("Csak 1..8 osztályba járhat");
            else
                this._hanyadikos = value;
        }
    }
    // ... folyt ...
}
```

A property szintaktikája kezdetben a mezőökre hasonlít. A védelmi szintje jellemzően publikus, hiszen pontosan a külvilág számára készül (de elképzelhető *protected* és *private* property is, mely esetben jellemzően csak a *set* rész kerül kidolgozásra).

A property-nek továbbiakban van típusa (*int*) és neve (*hanyadikos*). Ha ezen a ponton pontosvessző kerülne be, akkor ténylegesen mező lenne:

```
public int hanyadikos;
```

A property esetén azonban a sor végére nem kerül pontosvessző, hiszen a property definíciója nem fejeződik itt be. Másrészt a sor végére nem kerül gömbölyű zárójelpár sem:

```
public int hanyadikos()
{
    get
    {
```

Ez esetben ugyanis nem property készülne, hanem metódus (akinek üres paraméterlistája van, és törzse is, de a törzsét nem lehet kétfelé bontani). A property tehát nem mező, nem metódus, hanem ilyen sajátosságos a szintaktikája.

A property törzse két részre bontható, *get* és *set* részekre. A *get* rész felelős a virtuális mező kiolvasásáért, a *set* pedig az érték módosításáért. Másképpen: ha a külvilág ki szeretné olvasni a virtuális mezőnk értékét, akkor lefut a *get* szakasz, míg ha a külvilág értéket kíván beállítani, akkor lefut a *set* rész. A *set* rész belsejében a *value* kulcsszó segítségével hivatkozhatunk a külvilág által beállítani kívánt értékre.

```
diak d = new diak();
// a virtuális mező írási művelete -> set -> (value = 7)
d.hanyadikos = 7;
// a virtuális mező olvasási művelete -> get -> (visszaad 7-t)
int jovore = d.hanyadikos + 1;
```

Jelen esetben a property úgy viselkedik, mintha egy tényleges *int* típusú mező lenne, ugyanazzal a szintaktikával használhatjuk. Amennyiben értéket kívánunk adni a property-nek, úgy a példányosítás után egyszerű értékadó utasítással írhatunk bele értéket. Mivel az értékadó utasítás bal oldalán szerepel, így az értékre vonatkozó írási művelet kerül végrehajtásra, vagyis a *set* programrésze. A *set* belsejében a *value* ekkor a szóban forgó 7 értéket fogja képviselni. A *set* megvizsgálja, hogy kívül esik-e az [1, 8] intervallumon. Mivel nem esik kívül, így az értéket elmenti a védett, kívülről nem hozzáférhető fizikailag is létező mezőbe.

Később, mikor olvasnánk a property értékét a következő kifejezésben, akkor a property értékére vonatkozó olvasási műveletet kell végrehajtani, vagyis a *get* programrész fut le. Ez visszaadja a korábban a *set* által a fizikai mezőbe eltárolt értéket, a 7-et.

Jellemzően a property-k neve, és a fizikai mező neve hasonlít. Egyforma természetesen nem lehet, mivel az azonosítók a hatáskörön belül egyediek. A hasonlóságot többféleképpen is fenntarthatjuk. Hagyományos megoldást mutattunk be a fenti kódban: a külvilág elől rejtett fizikai mező neve aláhúzással kezdődik, míg a publikus property neve „csinosabb”. Más megoldás is választható, pl. a mező neve nagybetűs, a property neve pedig

kisbetűs kezdetű. Hasonló megoldás, ha a mező neve „f” betűvel kezdődik (mező = *field* angolul).

Nagyon kell ügyelni, hogy a property írásakor a property törzsében el ne rontsuk az azonosító nevét. A set a **mezőbe** helyezi az értéket, a get a **mezőből** olvassa ki az értéket. A mező neve aláhúzással kezdődik. Vizsgáljuk meg az alábbi (hibás) kódot:

```
// a védett mező
protected int _hanyadikos;
// és a publikus property
public int hanyadikos
{
    get
    {
        return this.hanyadikos;
    }
    set
    {
        if (value < 1 || value > 8)
            throw new ArgumentException("Csak 1..8 osztályba járhat");
        else
            this.hanyadikos = value;
    }
    // ... folytatás ...
}
```

A kód szerint ha a külvilág a *hanyadikos* property értékét ki akarja olvasni (get művelet), akkor a return miatt ki kell olvasni a *hanyadikos* értékét. A 'hanyadikos' azonban egy property, amelynek úgy kell kiolvasni az értékét, hogy le kell futtatni a 'get' részét, melynek belsejében a 'hanyadikos' property értékét kell kiolvasni, melyhez le kell futtatni a 'get' részét, melynek... .. ez a végtelen rekurzió.

Ennek oka, hogy a property get részében kiolvassuk magát a property-t (lemaradt az aláhúzásjel)!

Hasonló hibás működést tapasztalhatunk a *set* részben is. Ha az érték megfelelő, akkor a *value*t beírjuk a *hanyadikos*ba, amit úgy kell tenni, hogy le kell futtatni annak *set* részét, mely újra lefuttatja a *set* kódrészt... ez is rekurzió.

A fordítóprogram sajnos a fenti két hibát nem tudja kiszűrni, mert a kód szintaktikailag teljesen helyes.

## 6.7. Amikor azt gondolnánk, hogy nem kell alkalmazni védelmet

Nem szükséges védelmet alkalmazni egy mezőre, ha a mező értéke bármikor megváltozhat, de a típusa pontosan leírja a felvehető értékeket. Legegyszerűbb példa a logikai típusú mező. Ebbe, típusának megfelelően csak a *true* vagy *false* értékek kerülhetnek be. Nem lenne értelme pl. egy diák osztályban a *'beiratkozott-e'* információ tárolására szolgáló mezőre property-t készíteni az alábbi módon:

```
// a védett mező
protected bool _beiratkozott_e;
// a publikus property
public bool beiratkozott_e
{
    get
    {
        return this._beiratkozott_e;
    }
    set
    {
        if (value != true && value != false)
            throw new ArgumentException("csak true/false lehet");
        this._beiratkozott_e = value;
    }
}
```

Mivel a fordító amúgy is ellenőrzi, hogy csak true/false értékeket adhatunk meg a szóban forgó mezőnek, a fenti megoldás semmilyen plusz védelmet nem biztosít. Ráadásul a kapott kód jelentősen lassítja majd a teljes program működését, mivel a mezőbe írás a függvényhívás extra idején kívül még a felesleges feltételvizsgálatot is tartalmazza. A mező értékének kiolvasásához a *get* kódrész fut le, mely szintén lassabb, mint a közvetlen mezőkiolvasás.

```
// nem kell védeni
public bool beiratkozott_e;
```

Másik könnyű példa, mikor a mező lehetséges értékei egy *enum*-beli értékek.

## 6.8. Egyszer írható mezők

Gyakori eset a fejlesztések során, hogy valamely mező értékének beírását (megadását) engedélyezzük a külvilág felé, de módosítását már nem. Ezt nevezzük egyszer írható mezőnek.

Az egyszer írható mezők sokféleképpen megoldhatóak – az egyes programozási nyelvek különleges támogatást is adnak erre problémára. A következőkben ismertetett megoldás nem tipikus a C# nyelv esetén, mivel tartalmaz speciális konstrukciókat erre a feladatra – azonban a property alaposabb megértése miatt célszerű ezt a körülményesebb megoldást is tanulmányozni. (A tanulságok érdekesek lesznek, és a módszer átültethető más, hasonló problémák megoldására.)

Első lépésben az egyszer írható mezőnk védelmi szintjéről kell döntenünk. Könnyen belátható, hogy ne legyen *public*, hiszen akkor a külvilág korlátlan mértékben olvashatja és írhatja a mező tartalmát. A *protected* és *private* védelmi szintek közül viszont már bármelyik megfelel.

Másrészről tekintsük példaként a diák életkorának értékét. Legyen ez az a mező, amelynek értékét csak egyszer engedjük beállítani, a későbbiekben majd az 'oregedtel()' metódust hívjuk meg évente. Ez szerencsés eset, mert az életkor értékét egész számként (pl. int) tároljuk, de tudjuk, hogy negatív érték értelmetlen ebben a mezőben: így létezik olyan kezdőérték, mely egyértelműen jelzi, hogy a mező értékét beállítottuk-e már vagy sem!

```
class diak
{
    // a -1 lesz a mező kezdőértéke
    protected int _eletkor = -1;
    // a publikus property
    public int eletkor
    {
        get
        {
            return _eletkor;
        }
        set
        {
            if (_eletkor == -1) // még nincs beállítva
            {
                if (12 <= value && value <= 90) _eletkor = value;
                else throw new ArgumentException("csak 12..90 lehet");
            }
            else throw new Exception("Már be van állítva az érték");
        }
    }
    // ... folytatás ...
}
```

Az adott mező esetén létezik olyan speciális érték, amelyet a szabályok szerint nem vehet fel. A *set* belsejében ellenőrizzük, hogy a mező a kezdőértékén áll-e. Amennyiben igen, és a beírandó érték megfelel a szabályoknak (12..80 közötti) úgy elfogadjuk, és elhelyezzük a mezőben. A következő írási kísérlet már hibát (*ArgumentException*) okoz.

Van egy hiba a fenti megoldásban. Az olvasás (*get*) rész akkor is működik, amikor még nincs beállítva a mező értéke, ez esetben a kezdeti -1 értéket kapjuk meg a property olvasásakor. Könnyű orvosolni, de nem szabad róla elfeledkezni.

```
// a publikus property
public int eletkor
{
    get
    {
        if (_eletkor == -1)
            throw new Exception("még nincs beállítva az értéke");
        else
            return _eletkor;
    }
    // ... folytatás ...
}
```



A helyzet kis mértékben más akkor, ha a szituáció szerint a mező nem rendelkezik ilyen speciális értékkel. Például egy grafikus objektum X, Y koordinátája szinte bármilyen értéket felvehet működése során. Hiába kezdene szintén pl. a  $-1$  kezdőértéken, később ha az objektumot mozgatjuk és közben módosítjuk az X, Y mezők értékeit, a mozgatás során felvett  $-1$  érték hatására a mező újra írhatóvá válik.

Kivédésére alkalmazhatunk egy extra logikai mezőt, melynek *true/false* értéke jelzi, hogy a beállítás megtörtént-e már korábban vagy sem.

```
class grafikusObjektum
{
    protected bool _X_beallitva = false;
    protected int _X;
    public int X
    {
        get
        {
            if (_X_beallitva == false)
                throw new Exception("még nincs beállítva");
            return _X;
        }
        set
        {
            if (_X_beallitva == false) _X = value;
            else throw new Exception("korábban már be volt állítva");
        }
    }
}
```

Érezhető, hogy a megoldási módszer univerzálisabb, és egyszerűbb is. Célszerűtlen azonban az alkalmazása, hiszen igényli a plusz egy logikai mező jelenlétét és kezelését, mely a memória-kiosztás szempontjából igen kedvezőtlen. A csak olvasható mezők legmegfelelőbb módszerére a 9.8 (Valódi egyszer írható mezők) fejezetben térünk vissza.

## 6.9. Csak olvasható mezők

A **csak olvasható mező** szintén mindennapos a gyakorlatban. A csak olvasható mező az objektum valamely jellemzőjét írja le, mely direktben nem módosítható, hanem egyéb tevékenységeink hatására kerül beállításra. Ilyen pl. a vektorok mérete (*.Length* mező), a listáké (*.Count* mező), vagy a Console osztály *CapsLock* mezője, mely jelzi, hogy a CAPS LOCK gomb be van-e nyomva.

Csak olvasható mezők esetén a *public* védelmi szint nem használható, hiszen a külvilág kódja nemcsak olvasásra, de írásra is lehetőséget kap. A *protected* és *private* szintén nem ad önmagában megoldást, mert ekkor nemcsak az írástól, de az olvasás lehetőségétől is megfosztjuk a külvilágot.

A megoldást a property-k adják ismét. Tudnunk kell, hogy a property törzse nem kötelezően tartalmazza mindkét részt, a 'get' és 'set' részeket. Létezik olyan property, mely csak az egyik vagy a másik részt tartalmazza. Nyilván értelmetlen az a property, amely egyiket sem.

A példa, amelyen keresztül bemutatjuk a csak olvasható mezőt legyen az autó objektum benzinmennyisége. Az autónk benzintartályában lévő aktuális benzinmennyiség kiolvasható, például az autó vezérlő elektronikája által. Az írásának engedélyezése azonban tiltott, mert kétértelmű lenne: a 'skoda.benzin = 30'; a kód alatt érthetnénk a „benzin mennyisége legyen 30 liter”, vagy, hogy „töltsünk a benzintartályba még 30 liter”. Első esetben nem derül ki mennyivel kellett pótolni a benzinmennyiséget, hogy elérjük a 30 litert. Második esetben nem tudni, hogy volt-e hely a benzintartályban még 30 liternek. Tovább is folytathatnánk a filozófiai eszmefuttatásunkat, de fogadjuk el tényként, hogy az autó osztály kialakításánál ez a döntés született: a benzinmennyiség mező csak olvasható legyen! Lássuk a megoldást:

```
class auto
{
    protected double _benzinMennyiseg;
    public double benzin
    {
        get
        {
            return _benzinMennyiseg;
        }
    }
    // ... folytatás ...
}
```

A külvilág kiolvashatja a 'benzin' mező értékét, de módosítani nem tudja:

```
static void Main(string[] args)
{
    auto p = new auto();
    p.benzin = 12.4;
    Console.WriteLine(p.benzin);
}
```

A mező írására nincs lehetőség. A VS jelzi is, hogy a property az értékadás bal oldalán a 'set' hiányában nem szerepelhet – miközben az olvasási művelet egy sorral lejjebb már hibátlan. A „Property or indexer 'auto.benzin' cannot be assigned to -- it is read only”<sup>11</sup> hibaüzenet is jelzi.

A csak olvasható mező probléma szorosan kapcsolódik az egyszer írható mezők problémájához. Az egyszer írható mező 'set' része ugyanis egyetlen egyszer kap csak szerepet, második és további hívásakor, futásakor már csak hibaüzenetet ad. A 9.6 (Egyszer írható mezők) fejezetbeli megoldás során a 'set' részt nem adjuk meg, azt más módon orvosoljuk. A 'get' részben pedig kihagyjuk a beállítottság ellenőrzését, mert egyéb okból garantálható, hogy a mező értékének beállítása korábban lefut, mint ahogy a 'get' hívására sor kerülhetne.

<sup>11</sup> „Property or indexer XY cannot be assigned to – it is read only” fordítása: „A tulajdonság vagy indexelő XY értéke nem beállítható – csak olvasható”.

## 6.10. Hatékonyabb megoldás

A Delphi programozási nyelvben a C#-beli lehetőségnél ügyesebb módszert találtak a property-k kezelésével kapcsolatban. A C# nyelvi property megoldás legfőbb problémája a 'get' rész, amely függvényként viselkedik, a háttérbeli viselkedése szerint lassú és körülményes – ugyanakkor jellemzően egyetlen 'return'-t tartalmaz, amely megadja a nem public mező értékét.

Ha belegondolunk, a property 'get' része végül is direkt módon enged hozzáférést a mező tartalmához – de használata lassítja a program futását. A Delphi nyelven megoldható, hogy az írás függvényen keresztül menjen (a 'set' rész), de az olvasás közvetlenül a mezőt olvassa.

```
type yourClass = class
  private
    FCount: Integer;           { belső tárolásra }
    procedure SetCount (Value: Integer); { írási metódus }
  public
    property Count: Integer read FCount write SetCount;
end;
```

Delphi nyelvi példa a *private* védelmi szintre definiálja az 'FCount' mezőt, valamint a 'SetCount' eljárást. Publikusan definiál egy 'Count' property-t, amelyet ha valaki olvas (*read*) akkor közvetlenül az 'FCount'-hoz irányítódik a lépés, míg a property írása (*write*) a 'SetCount' metód hívását fogja kiváltani. A 'read' kulcsszó mögött vagy egy megfelelő típusú mező neve (mint esetünkben), vagy egy megfelelő típusú értékkel visszatérő paraméter nélküli függvény neve adható meg. Utóbbi esetben a property olvasása a függvény hívását fogja okozni.

Delphi nyelven a fentieknek megfelelő property-k olvasása a fordítóprogram jó működésének megfelelően a generált kód szintjén már egyenértékű a rejtett mező olvasásával, így a generált kód futási sebessége maximális. Delphi nyelven a csak olvasható mező létrehozása is egyszerűbb, mint C# nyelven:

```
type yourClass = class
  private
    FCount: Integer;           { belső tárolásra }
  public
    property Count: Integer read FCount;
end;
```

A 'Count' property esetén csak a 'read' rész van megadva, így írására vonatkozó kísérleteket a fordítóprogram nem fogja engedélyezni. A property olvasása ugyanakkor végső soron a mező olvasásával egyezik meg. Olyan, mintha a mező írását *private* szintre, olvasását *public* szintre emeltük volna.

## 7. Metódusok

---

Azon osztályokat amelyek csak mezőkből állnak – **rekord**oknak nevezzük. A rekordok szükségszerű velejárói a programozásnak, nem túl bonyolult szerkezetek. Segítségükkel valamely szempont szerinti adatokat csoportosíthatunk, foglalhatunk egységbe. Egy ilyen rekord példány komoly előnye, hogy egyetlen lépésben adhatunk át egy csoportnyi adatot függvényeknek paraméterként:

```
class kör
{
    public double X_koord;
    public double Y_koord;
    public double sugar;
}

static double kerulet( kör k )
{
    return 2 * k.sugar * Math.PI;
}
```

A függvények, amelyek rekordot kapnak paraméterként, kiválóan képesek működni. Írásukkor azonban egy dologról jellemzően el szoktunk feledkezni: a 'class' kulcsszóval létrehozott típusok referencia típuscsaládba sorolandók. A referencia típusú paraméterek azonban kaphatnak a hívás helyéről 'null' értéket is. Az alábbi első példa jól működik, a második azonban hibát fog okozni:

```
kör k = new kör();
k.sugar = 12.5;
double k1 = kerulet(k);    // jól működik
double k2 = kerulet(null); // ez hibát fog okozni
```

Valójában a rekord paraméterű függvényeknek minden esetben illene ellenőrizniük, hogy paramétere nem 'null' értékű-e.

```
static double kerulet( kör k )
{
    if (k==null) throw new ArgumentNullException("A k értéke null");
    return 2 * k.sugar * Math.PI;
}
```

Az OOP szemlélet szerint a függvényeket is valamely osztályba kell helyezni. Természetesen megoldható, hogy a függvényünk ne a 'kör' osztályba kerüljön:

```
class kör
{
    public double X_koord;
    public double Y_koord;
    public double sugar;
}

class számítások
{
    public static double kerulet( kör k )
    {
        if (k==null) throw new ArgumentNullException("A k értéke null");
        return 2 * k.sugar * Math.PI;
    }
}
```

A 'static' módosító miatt a függvény hívásához meg kell adni azt is, hogy melyik osztályból, melyik függvényt kívánjuk meghívni:

```
kör k = new kör();
k.sugar = 12.5;
double k1 = számítások.kerulet(k); // jól működik
```

De az is megoldható, hogy bekerüljön a 'kör' osztályba:

```
class kör
{
    public double X_koord;
    public double Y_koord;
    public double sugar;
    //
    public static double kerulet( kör k )
    {
        if (k==null) throw new ArgumentNullException("A k értéke null");
        return 2 * k.sugar * Math.PI;
    }
}
```

Ez esetben hívása:

```
kör k = new kör();
k.sugar = 12.5;
double k1 = kör.kerulet(k); // jól működik
```

Ugyanakkor ha a 'static' módosító nélkül írjuk a függvényünket, az komoly változásokat okoz.

## 7.1. Példány szintű metódusok

A 'static' módosítójú metódusok lényegében a hagyományos programozás függvényeivel szinte teljesen egyenértékűek tervezés és felhasználás szempontjából is. A függvények fogalmára erőltették rá az OOP szemléletet. Ha egy, a hagyományos programozási szemlélethez szokott rutinos programozó kerül át OOP-s világba, az első, amit megjegyez és megtanul: „a függvényeidet rakd be egy 'class { ... }' környezetbe, és kész is vagy”. A 'static' metódusok neve **osztálysztintű metódusok**.

A 'static' módosító nélküli függvények azonban jelentős különbségeket mutatnak. Nevük a továbbiakban **példányszintű metódus**.

A példányszintű metódusok olyan függvényekként foghatóak fel, melyeknek kötelező paramétere egy, az adott osztályból készített rekord, amely rekord nem lehet 'null' értékű.

Ez az szabály nagyon fontos. Kitértünk arra, hogy a rekord paraméterű függvény igazából mindig kaphat 'null'-t is, és ezt elvileg minden esetben meg kellene vizsgálnia. Erre jellemzően az a válasz, hogy „Minek? Miért adna a kívülág át'null' értéket? És ha átad, akkor futási hiba keletkezik, na és? Ez az én hibám?”. Nem nyilvánvaló a válasz, hogy a függvény írója-e a hiba vagy sem. Első körben a válasz könnyű: „Nem az ő hibája”. Ugyanakkor elképzelhető, hogy a függvény több lépést is tesz, több feladatot is elkezd vagy végrehajt, mielőtt a null érték végzetes hibát okozna. Ekkor már nem annyira egyértelmű a „nem az ő hibája” válasz. A függvények megvalósításakor egyszerű elv, hogy első lépésben a függvény ellenőrizze le az összes kapott paramétert, hogy azok megfelelőek-e, mielőtt bármilyen tevékenységbe is belekezdene. Persze fárasztó és könnyen elfeledhető lépés. Ezért is nagyon hasznos, hogy ezt az ellenőrzést (a rekord nem 'null' érték) a fordítóprogram átveszi.

Figyeljük meg, milyen változásokon megy keresztül a static függvényünk, hogy példányszintű metódussá válhasson:

- elsősorban elveszíti a static jelzőjét,
- másodsorban elveszíti a rekord paramétert,
- mivel nincs 'k' paraméter, a mezőkre továbbiakban már nem tudnánk 'k.sugar' néven hivatkozni,
- a példányszintű metódus teljesen beleolvad a példányba, ezért a mezőkre mindenféle előtag nélkül, direktben hivatkozhatunk ('k.sugar' helyett csak 'sugar').

```
class kör
{
    public double X_koord;
    public double Y_koord;
    public double sugar;
    // !!! STATIC !!! és !!! PARAMÉTER !!! nélküli metódus
    public double kerulet()
    {
        return 2 * sugar * Math.PI;
    }
}
```

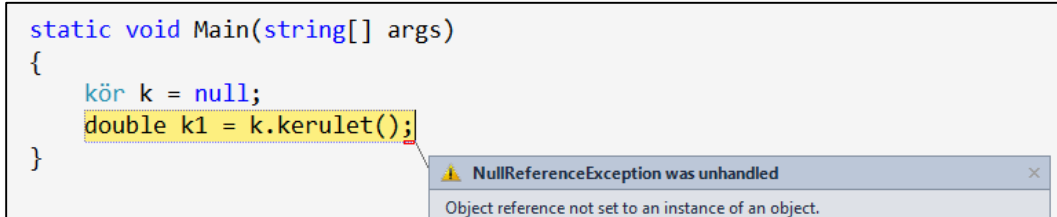
A függvényhívás szintaktikája átalakul:

```
kör k = new kör();
double k1 = kör.kerulet(k);
```

helyett:

```
kör k = new kör();
double k1 = k.kerulet();
```

Itt meg is kaphatjuk a választ. A 'null' ellenőrzés kikerül a függvényhívás helyére. Ha a 'k' példányváltozó értéke 'null', akkor a futás közbeni hiba a függvény hívás helyén automatikusan létrejön, és valószínűleg le is állítja a program futását. Ellenőrzésére tehát a függvényünknek már nem kell időt és energiát vesztegetnie.



```
static void Main(string[] args)
{
    kör k = null;
    double k1 = k.kerulet();
}
```

NullReferenceException was unhandled  
Object reference not set to an instance of an object.

Amennyiben alaposabban megfigyeljük a 'kerulet()' függvény törzsét, több érdekes gondolatunk is támadhat. Először is: milyen példány 'sugar' mezőjére hivatkozunk, mikor nincs is megjelölve ott példány. Mi van akkor, ha egyáltalán nincs is példánya a 'kör'-nek a programban, akkor mit csinál majd ez a függvény? Másodszor amikor több példány is van, akkor melyik példány sugar mezőjét használja ez a függvény?

## 7.2. Aktuális példány kezelése

A válasz mindkét kérdésre a hívás helyén keresendő, illetve a háttérműködésben.

Első kérdésre a válasz (ha nincs is példány): akkor a függvény nem meghívható! Kísérletezhetünk, de a példányszintű metódusok hívási szintaktikája:

**példánynév.metódusnév (paraméterek)**

Vagyis a hívás helyén szerepeltetni kell példányt. Példánynak lennie kell! Ez következménye a módosított szintaktikának – a rekord paraméterű függvény még hívható lett volna rekord példány nélkül, de az OOP-s változatbeli metódus nem hívható meg a példány nélkül. Többek között, emiatt a szoros kapcsolat miatt nevezük a metódust példányszintű metódusnak.

A második kérdés (több példány is van) válasza szintén a hívás helyén keresendő.

```
kör k1 = new kör();
kör k2 = new kör();

double d1 = k1.kerulet();
double d2 = k2.kerulet();
```

Amikor több példányunk is van ('k1' illetve 'k2'), akkor a hívás helyénél kell specifikálni melyik példánnyal kívánjuk a műveletet végezni. Eddig könnyű és világos. De honnan tudja a függvény törzse, melyik példányt használtuk a hívás helyén? Nem világos a két pont közötti kapcsolat.

Lényeges annak megértése, hogy nincs is OOP programozás!<sup>12</sup> A számítógép utasítás végrehajtó egysége, a mikroprocesszor, nem ismer ilyen absztrakt fogalmakat, hogy példány vagy metódus. Az egész csak a fordítóprogram trükkje. Mi példányszintű függvényt írunk, és nem adunk át neki rekord paramétert. A fordító olvassa a magas absztrakciós szintű, OOP stílusú forráskódunkat, és hagyományos programozásbeli fogalmakra vezeti vissza. Ezt írják az OOP programozók, mert a fordítóprogram ezt várja el tőlük:

```
double d1 = k1.kerulet();
```

A fordítóprogram pedig ezt a kódot érti alatta:

```
double d1 = kör.kerulet(k1);
```

Vagyis ugyanott tartunk, ahonnan indultunk. Illetve majdnem. Ha mi írtuk ezt, akkor megszóltak minket, hogy nem vagyunk OOP-s programozók, illetve a 'k1' paraméterrel nekünk kellett bíbelődni, deklarálni mint paraméter, és ellenőrizni, hogy nem 'null' értékű-e. Az OOP-s változatban a példány paramétert helyettünk deklarálja a fordító, és leellenőrzi, hogy null értékű-e! Ez azért valami!

<sup>12</sup> Mátrixbeli gondolat: az órakulumnál mondta Neonak a kanálhajlító kisfiú: „Ne próbáld elhajlítani, mert az lehetetlen. Helyette inkább próbáld felismerni az igazságot!” – „Miféle igazságot?” – „Hogy nincs kanál.”



De hol ez az extra paraméter? Ahogy mi írtuk meg a 'kerulet()' metódust, úgy annak nincs is paramétere!

A példányszintű metódusoknak minden esetben van egy extra paramétere, mely a hívás helyén megadott példányt kapja értékül. Az extra paraméter deklarálását és kezelését a fordítóprogram végzi, a paraméter neve: **this**.

Látjuk már, hogy a hívás helyén megadott példány eljut a függvényünkig, a függvény törzséig, de nem látjuk ott az alkalmazását. Még mindig nem érthető, hogy a függvény törzsében szereplő 'kerulet' mező melyik példány kerulet mezője?! Nos, ez megint csak a fordítóprogram egy trükkje. A fordítóprogram „megengedi”, hogy a metódus törzsében a mezőhivatkozás során egyszerűen csak leírjuk a mező nevét. De ő is tudja, hogy a mező a példány megnevezése nélkül (akihez a mező tartozik) értelmetlen. Van példány a függvény törzsénél, paraméterként kaptuk, 'this' a neve a példánynak. Ezért amikor mi azt írjuk, hogy:

```
return 2 * kerulet * Math.PI;
```

a fordító ezt olvassa:

```
return 2 * this.kerulet * Math.PI;
```

A teljes kód tehát (amit mi írunk):

```
class kör
{
    ...
    public double kerulet()
    {
        return 2 * sugar * Math.PI;
    }
}

class FoProgram
{
    public static void Main()
    {
        kör k = new kör();
        double d1 = k.kerulet();
    }
}
```

Amit a fordítóprogram olvas ki belőle:

```
class kör
{
    ...
    public static double kerulet( kör this )
    {
        return 2 * this.sugar * Math.PI;
    }
}

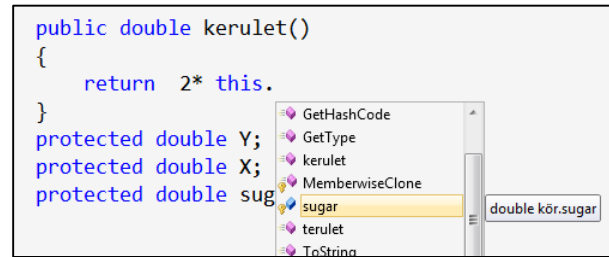
class FoProgram
{
    public static void Main()
    {
        kör k = new kör();
        double dl = kör.kerulet( k ); // ellenőrizve k != null !!!!!
    }
}
```

Eszerint azt gondolhatjuk: nem vétünk nagy hibát, ha eleve így írjuk meg a kódunkat. De igen, vétünk! Ugyanis a fordítóprogramnak még számtalan további trükkje van, amelyet be tud vetni kényelmünk érdekében, hogy kiszűrje a hibákat és megkíméljen minket további hibalehetőségektől. Ha így írjuk meg a kódunkat, az egyik legerősebb OOP technika – a késői kötés használata – is lehetetlenné válik, és nem engedjük meg a fordítóprogramnak, hogy jobban megértse és alkalmazhassa rá a megfelelő kódgenerálási trükkjeit.

A 'this' kulcsszó a példányszintű metódusok törzsében valódi, jelenlévő kulcsszó, azonosító. Az extra paraméter azonosítója. Ami egyrészt azt jelenti, hogy ilyen nevű azonosítót mi már a metódusban nem deklarálhatunk (mivel a 'this' kulcsszó, foglalt szó, semmiképpen sem).

```
public double kerulet()
{
    int this = 2;
    return Identifier expected; 'this' is a keyword I;
}
```

Mivel a 'this' a metódusban a paraméter-példány neve, így akár fel is használhatjuk azt a metódus törzsében. A 'sugar' mezőre valóban hivatkozhatunk oly módon, hogy 'this.sugar'. A 'this.' előtag hasznos sok esetben, melyekre később még teszünk utalást, de már most kezdhethetjük szokni a használatát. A fordítóprogram egyáltalán nem emel kifogást az ellen, ha kiírjuk. A 'this.sugar' a forráskódban azt hangsúlyozza, hogy: ennek a példánynak a sugar mezője. További előny, ha beírjuk, hogy 'this.', a pont leütése után a VS azonnal ad segítséget, listát, hogy a példány milyen mezőkkel és metódusokkal rendelkezik.



A 'this' kulcsszó a példányszintű metódusok törzsében az aktuális példányt azonosítja. A törzsben ez a kulcsszó fel is használható, a mezőkre való hivatkozás során. A 'this' a fordítóprogram által automatikusan kezelt extra paraméter neve. A 'this' emiatt foglalt szó (kulcsszó). A 'this' típusa megegyezik azzal az osztálytípussal, amelyben a metódus szerepel.

## 8. A Main() függvény

---

A programunk utasításokból áll. Az utasítások adott sorrendben kerülnek végrehajtásra. A szekvencia végrehajtási szabály értelmében abban a sorrendben, ahogy a forráskódban is szerepelnek. Kell egy kezdőpont, amely megadja melyik a legelső végrehajtandó utasítás. Onnantól kezdve egyértelmű, hogy melyik a következő.

A program kezdőpontját a C nyelvben szokásosan egy speciális nevű függvény, a 'main()' függvény jelöli. A C# nyelv a C nyelv szintaktikai alapjaira épült, így hasonló választással a kezdő függvény neve a 'Main()' függvény lett (nagy M-mel írva).

Az OOP nyelveken minden függvényt osztályba kell helyezni (egységbezárás elve). Így a 'Main()' függvény sem állhat osztályon kívül, tehát bele kell helyezni valamely osztályba. Melyikbe? Teljesen mindegy! Az az elv, hogy a programban lennie kell pontosan egy 'Main()' függvénynek, valahol! A valahol az pontosan azt jelenti, hogy mindegy melyik osztályban.

A 'Main()' függvény azonban nem szabad, hogy példányszintű legyen! Miért? A példányszintű függvények meghívásához példányra lenne szükség. A példányosítás utasítás. Hova kerüljön ez az utasítás, ha a legelső utasítás a 'Main()' függvény belsejébe kell, hogy kerüljön? (Tyúk és tojás probléma) Ha a 'Main()' függvényt véletlenül nem jelöljük meg a static jelzővel (osztálysintű), akkor a fordítóprogram nem fogja felismerni, hogy ez az a 'Main()' függvény, amely a program indulását képviseli.

Mi legyen a 'Main()' függvény védelmi szintje? Teljesen mindegy. Kivételesen akár private is lehet, nem befolyásolja a működést. Ennek értelmében a legegyszerűbb 'Main()' függvény alakilag így néz ki:

```
static void Main()  
{  
  
}
```

Nem árt tudni, hogy a 'Main()' függvény vagy 'void', vagy 'int' visszatérési típusú. Utóbbi esetet akkor használjuk, ha a programunk valami olyan tevékenységet végez, amelyet batch-ből (script) hívunk meg. Az ilyen jellegű indítások esetén illik jelezni a hívó scriptnek, hogy a program futása sikeres volt-e vagy sem. Szokásosan 0 érték jelöli, hogy nem volt hiba, minden más érték valamiféle hibára utal. Amikor 'void' a visszatérési típusunk, az ugyanaz az eset, mintha 'int' lenne, és 0-t adnánk vissza (ezt helyettünk megteszi a fordítóprogram):

```
static int Main()  
{  
    // ..  
    return 0;  
}
```

Ha programunkat command ablakból indítjuk, akkor induláskor adhatunk át neki parancssori paramétereket is.

```

C:\TEMP>
C:\TEMP>
C:\TEMP>filetitkosit c:\proba.txt c:\proba.txt.titkos

```

A program neve után írt paramétereket a 'Main()' függvény kapja meg, melyek string-ek, számuk attól függ, hány ilyen paramétert adtunk át. Ha kezelni kívánjuk ezeket a paramétereket, akkor a 'Main()' függvényhez paramétert is kell definiálnunk:

```

static void Main(string[] args)
{
}

```

Az 'args' vektorban két elem lesz:

```

args[0]      =>  "c:\proba.txt"
args[1]      =>  "c:\proba.txt.titkos"

```

Ügyeljünk rá, hogy az operációs rendszer számára a paraméterek elválasztó jele a szóköz. Vagyis az alábbi paraméterezés mellett

```

C:\TEMP>
C:\TEMP>
C:\TEMP>filetitkosit c:\Documents and Settings\leiras.txt c:\leiras.titkos

```

az args vektorban 4 elem lesz:

```

args[0]      ⇨   "c:\Documents"
args[1]      ⇨   "and"
args[2]      ⇨   "Settings\leiras.txt"
args[3]      ⇨   "c:\leiras.titkos"

```

Szerencsére a Windows újabb változatai már az operációs rendszer szintjén is értik az idézőjelet:

```

C:\TEMP>
C:\TEMP>
C:\TEMP>filetitkosit "c:\Documents and Settings\leiras.txt" c:\leiras.titkos

```

Ekkor az args vektor újra 2 elemű lesz:

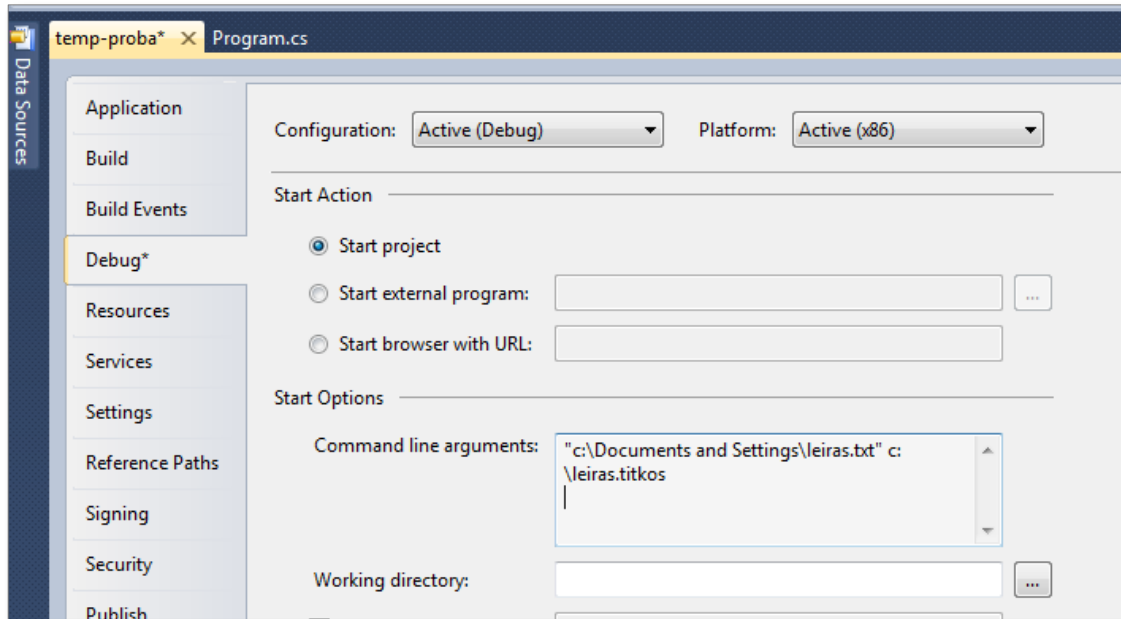
```

args[0]      ⇨   "c:\Documents and Settings\leiras.txt"
args[1]      ⇨   "c:\leiras.titkos"

```

Ha a programunkat nem parancssorból, hanem a VS-ből indítjuk el, akkor is tudunk átadni neki parancssori paramétereket (tesztelési céllal). Kattintsunk jobb egérgombbal a

Solution Explorer ablakban a Projectre, válasszuk ki a Properties (legalsó) menüpontot, majd keressük ki a Debug beállító fület, és a Command Line Arguments részhez írjuk be a Main()-nek átadandó paramétereiket:



## 9. Konstruktorok

---

A konstruktorok kulcsfontosságú elemei az OOP szemléletnek. Sok probléma van, melyet a konstruktorok segítségével egyszerűen meg lehet oldani. De a konstruktorok legfontosabb feladata a példányok kezdőérték beállításával kapcsolatos.

Vegyük példának a korábban már említett 'diak' osztályt, és az 'eletkor' mezőt. Az életkor mező szabálya, hogy csak 18..60 év értéket vehet fel, amit garantálnunk kell. Minden tudásunkat felhasználva a tényleges életkor mezőt levédjük (protected), és készítünk hozzá publikus property-t:

```
class diak
{
    protected int _eletkor;
    public int eletkor
    {
        get
        {
            return _eletkor;
        }
        set
        {
            if (value < 18 || value > 60)
                throw new ArgumentException("csak 18..60 lehet");
            else _eletkor = value;
        }
    }
    // ... folytatás ...
}
```

Úgy érezzük, pompás munkát végeztünk, megnyugodhatunk. Ha jól használjuk az objektumot, akkor jól is működik:

```
diak d = new diak();
d.eletkor = 22;
Console.WriteLine("eletkora = {0}", d.eletkor);
```

Azonban könnyen előfordulhatnak hibás használatból eredő problémák. Szabályunk szerint az életkor mező értéke 18..60 közötti szám, *minden esetben!* Ezt az objektum garantálja, és mivel a mező olvasható is, a külvilág is feltételezi hogy ez az érték valóban 18..60 közötti:

```
diak d = new diak();
// kihagyjuk: d.eletkor = 22;
Console.WriteLine("eletkora = {0}", d.eletkor);
```

Amennyiben létrehozunk egy diák példányt, az életkor mező értéke nulla<sup>13</sup> lesz, ami nem felel meg a szabálynak. A WriteLine kiírásában látni fogjuk, hogy az életkor értéke nulla. Ha a külvilág erre nem számít, hibát generálhat, ami nem megengedhető.

Egyik megoldás, hogy a mezők kezdőértékét megadjuk, például a mezők deklarációja során kezdőértékkadás formájában:

```
class diak
{
    protected int _eletkor = 18;
    public int életkor
    {
        get
        {
            return _eletkor;
        }
    }
}
```

Nem minden esetben használható ez a megoldás. Mit adnánk meg a diák nevének, NEPTUN kódjának vagy nemének kezdőértékként? Ráadásul a külvilágot is megzavarjuk vele, mivel nem követeljük meg e mezők értékének beállítását, így már sokkal könnyebb végképp elfeledkezni róluk. Ha mégis megtesszük, akkor sem elegáns a példányosítás, több utasítást is használnunk kell. Az első utasítás a példány létrehozása, a továbbiak a mezők kezdőértékeinek beállítása. Nem elegáns.

### 9.1. Konstruktork példányosításkor

Másik oldalról megközelítve: eddig különösebben nem figyeltünk arra, miért pont így néz ki a példányosítás szintaktikája:

```
diak d = new diak();
```

Az értékadás bal oldalán deklaráljuk a 'diák' típusú 'd' változót. Mivel ez egy referencia típuscsaládba tartozó változó, elsődleges memóriaigénye 4 byte. A tényleges adatok a másodlagos memóriaterületen kerülnek tárolásra. A másodlagos területet a 'new' kulcsszó segítségével foglaljuk le. A 'new' rendkívül ügyes, megfelelő mennyiségű memóriát foglal le a diák példány számára<sup>14</sup>. Sokkal érdekesebb kérdés, hogy a 'new' után álló 'diak()' az pontosan micsoda?!

---

<sup>13</sup> Az *int* típusú mezők kezdőértéke minden esetben nulla, amit a futtató rendszer garantál.

<sup>14</sup> Pontosán mennyit jelent, elsősorban a példányszintű mezőktől függ, de sok egyéb tényezőtől is.



Vegyük észre, hogy a 'diak()' alak egy függvényhívásra hasonlít. Ami zavaró, hogy a 'new' áll előtte. Hiányában egészen függvényhívásra emlékeztetne a szintaktika:

```
string s = bekeres(); // ez valóban függvényhívás
diak d = diak();      // ez is annak néz ki
```

Nos, a zavaró hasonlóság oka: ez itt valóban egy függvényhívás. Egy speciális függvény hívása. Sok szempontból speciális. Elsősorban a neve. A függvény neve 'diak', ami egyezik egy osztály nevével (class diak).

Másodsorban a hívási szintaktika: ezen függvényt ily módon lehet csak meghívni, a 'new' kulcsszó után. A két lépés párban áll, kiegészítik egymást.

Az ilyen jellemzőjű függvényt, metódust nevezzük speciálisan az OOP világában **konstruktor**nak. A konstruktor tehát egy metódus, egy függvény. Feladata: a frissen elkészülő példány alaphelyzetbe állítása. Az alaphelyzet nagyon fontos. Egy objektumpéldány a létrehozásának pillanatában garantálni köteles a helyes állapotot: minden mezőnek a szabályoknak megfelelő értékekkel kell rendelkeznie. Nem helyes az a hozzáállás, hogy egy diak példány létrehozásának pillanatában még nem, csak sorozatos értékadások után van helyes állapotban:

```
diak d = new diak();
// -- itt már létezik de még nem jó
d.neme = nemek.ferfi;
d.neptun_kod = "QQDK23";
d.eletkor = 22;
// --- itt már jó
```

## 9.2. Konstruktor készítése

Készíthetünk konstruktor függvényt, mely megkönnyíti a külvilág felé a példányosítás folyamatát, és egyúttal előírhatjuk a kötelezően megadandó adatokat is. A konstruktor írásakor az osztály belsejébe egy, az osztály nevével egyező nevű metódust kell készítenünk. A konstruktor nem kötelezően bár, de jellemzően public, és nincs jelölt visszatérési típusa, még a void-ot sem szabad kiírni:

```
enum nemek { ferfi, no }
class diak
{
    public diak(int pEletkor, nemek pNeme, string pNeptunkod)
    {
        _eletkor = pEletkor;
        neme = pNeme;
        neptun_kod = pNeptunkod;
    }
}
```

Elkészítettük a diak osztály egy konstruktorát. A konstruktorunk három paraméteres, ezek: életkor, nem és a neptun kód. E pillanattól kezdve használata a külvilág felé köte-

lező. Vagyis a diák példányosításakor már nem működik a megszokott, egyszerű módszer:

```
diak d = new diak();
```

diak.diak(int pEletkor, nemek pNeme, string pNeptunKod)

Error:  
'ektf.diak' does not contain a constructor that takes 0 arguments

A külvilág a 'new' kulcsszó után köteles meghívni az adott osztály konstruktorát. Mivel a konstruktor neve egyezik az osztály nevével<sup>15</sup>, így a konstruktor függvény nevét nem kell sokáig keresgélni. Amikor a 'new' lefoglalja a memóriát, a mezők felveszik a típusuknak megfelelő kezdőértékeket (a bool mezők false, a szám típusú mezők nulla, a referencia típusúak a null értéket stb.). A továbbiakban lefutnak a mezők mellé írt kezdőértékadások. Végül meghívásra kerül a 'new' után kötelezően megadandó konstruktor függvény is. Jelen példában a konstruktorfüggvény háromparaméteres, így a hívásakor is meg kell adni a három paramétert:

```
diak d = new diak(22,nemek.ferfi, "QQDK23");
```

### 9.3. Több konstruktor készítése

Nemcsak egyetlen konstruktort készíthetünk – sőt, általában több konstruktor áll rendelkezésre ugyanazon osztályhoz. A konstruktorok mindegyikére azonos szabályok vonatkoznak: neve egyezik az osztály nevével. Az *overloading* szabály megengedi, hogy egyforma nevű függvények létezzenek egyazon környezetben mindaddig, míg a paraméterezésük különböző. Jelen esetben ez lesz segítségünkre. Ugyanazon osztálybeli konstruktorok csak ebben különböznek egymástól, így ügyelnünk kell rá:

```
enum taplalkozas { husevo, novenyev, mindenevo }
enum neme { him, nosteny }
class allat
{
    public allat(taplalkozas t, neme n)
    {
        // ...
    }
    public allat(allat anya, neme n)
    {
        // ...
    }
}
```

<sup>15</sup> Nem minden nyelvben egyezik meg, Delphi-ben pl. szabadon választható a konstruktor neve, de általában Init-nek vagy Create-nek nevezik el.

A fenti esetben vagy úgy hozunk létre egy új állatot, hogy megadjuk, mit eszik és mi a neve, vagy megadjuk az anyaállatot (a kis állatka nyilván ugyanazt eszi, mint a mama) és az új állat nemét.

```
allat m = new allat( taplalkozas.novenyevo, neme.nosteny );
allat p = new allat( m, neme.him);
```

## 9.4. Konstruktore hiánya

A jegyzet korábbi fejezeteiben többször készítettünk osztályokat, de nem írtunk hozzá konstruktort, mégis tudtunk példányosítani:

```
class kor
{
    public double x;
    public double y;
    public double sugar;
}

class Program
{
    public static void Main()
    {
        kor k = new kor();
        k.x = 12.4;
    }
}
```

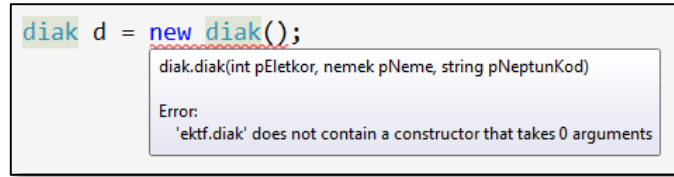
A kód egyenértékű azzal az esettel, ha a kör osztálynak lenne paraméter nélküli konstruktora. Mivel ezen konstruktornak nincs paramétere, a mezők pedig amúgy is felveszik típusuknak megfelelő kezdőértéket – a konstruktor törzse is üres:

```
class kor
{
    public double x;
    public double y;
    public double sugar;
    // ..
    public kor()
    {
    }
}
```

Nem megengedhető az, hogy egy class ne rendelkezzen konstruktorral. Amennyiben a programozó nem készít konstruktort, a fordítási folyamat során generálódik számára egy, a fentieknek megfelelő konstruktor. A konstruktor publikus lesz, nincs paramétere, és a törzse sem tartalmaz utasítást. Egyetlen fontos szerepe van: lehetővé teszi példány készítését az adott osztályból. Mivel a C# nyelvi szintaktikai szabályok előírják, hogy a példányosítás során a 'new' után kötelező meghívni a konstruktort, így konstruktornak léteznie kell. Ezt biztosítja a leírt mechanizmus.

Jegyezzük meg azonban, hogy a fordítóprogram ezen funkciója csak akkor lép működésbe, ha az osztályhoz a programozó egyáltalán nem készít konstruktort. Ha készít, akkor

a fordító már nem adja hozzá az üres konstruktort. Oka, hogy az alábbi képen is látható hiba felbukkan. Amennyiben a diák osztályhoz készítünk paraméteres konstruktort, a példányosítás a paraméter nélküli konstruktorral (annak hiányában) nem lehetséges:



### 9.5. A paraméterek ellenőrzése

Gyakori eset, hogy egy mezőre szabály vonatkozik. Emiatt property-t készítünk, get és set résszel. A set rész belsejében a value aktuális értékére vonatkozó ellenőrzések már elkészültek, ugyanakkor a mező a konstruktoron keresztül is értéket kaphat. Legyen a példa a diák életkora, a szabály pedig a korábbi 18..60 közötti érték:

```
class diak
{
    // védett mező
    protected int _eletkor;
    // property
    public int életkor
    {
        get
        {
            return _eletkor;
        }
        set
        {
            if (value < 18 || value > 60)
                throw new ArgumentException("csak 18..60 lehet");
            else _eletkor = value;
        }
    }
    // konstruktor
    public diak(int pEletkor)
    {
        _eletkor = pEletkor;
    }
}
```

A fenti kód részt hagy az objektum védelmi rendszerén. Az életkor mező értékét a külvilág értékadással nem tudja elrontani, a set minden esetben ellenőrzi a szabályt. De a konstruktor a paraméterül kapott értéket ellenőrizetlenül másolja a mezőbe. Ily módon mégiscsak létezhet olyan diák példány, amelynek életkor értéke nem megfelelő:

```
public static void Main()
{
    // 12 éves diák
    diak d = new diak(12);
}
```

Ezt elkerülendő, a konstruktor le kell, hogy ellenőrizze valamely módon a paraméter értékének megfelelőségét:

```
// konstruktor
public diak(int pEletkor)
{
    if (pEletkor < 18 || pEletkor > 60)
        throw new ArgumentException("csak 18..60 lehet");
    _eletkor = pEletkor;
}
```

A megoldás természetesen működik, de sok problémát vet fel. A teljes ellenőrző algoritmus (mely a példában ugyan csak egy if, de lehetne sokkal összetettebb is) két helyen is szerepel a kódban: a set belsejében, és a konstruktor belsejében is. Nemcsak a kód másolása a problémás, hanem a redundancia is. Ha a szabály módosul, úgy most már két helyen kell módosítani.

E helyett érdemes használni a már leprogramozott és tesztelt set törzset a konstruktorból is:

```
// konstruktor
public diak(int pEletkor)
{
    // nem a mezőbe, a propertybe !
    eletkor = pEletkor;
}
```

Az első (hibás) példában a konstruktor még közvetlenül a mezőbe írta be a paraméter értékét. Utóbbi esetben már a property set részét használjuk, így a paraméter átmegy az ellenőrzésen.

## 9.6. Egyszer írható mezők

Korábban volt szó az egyszer írható mezőkről. A konstruktorok segítségével az egyszer írható mezők problémája könnyen kezelhető. Az egyetlen írási lehetőséget, a mező kezdőértékét a konstruktor paraméterén keresztül lehet megadni, így sokat egyszerűsödik a megvalósítás:

```
// védett mező
protected int _eletkor;
// property
public int eletkor
{
    get
    {
        return _eletkor;
    }
}
// konstruktor
public diak(int pEletkor)
{
    if (pEletkor < 18 || pEletkor > 60)
        throw new ArgumentException("csak 18..60 lehet");
    // a mezőbe helyezzük, nincs set a propertyben
    _eletkor = pEletkor;
}
```

A fenti kódban az életkor property csak olvasható. A get részben nem szükséges annak ellenőrzése, hogy a fizikai mező, az '\_eletkor' értéke beállításra került-e korábban, hiszen a konstruktoron keresztül az garantáltan megtörtént. A konstruktor fogadja a mező kezdőértékét, és ellenőrzi annak megfelelőségét.

## 9.7. Property és a kettős védelmi szint

Az egyszer írható mezők problémájának egyfajta megoldása, amikor csak 'get' részt írunk, az érték beállítását pedig a konstruktoron keresztül végezzük el. Utóbbi csak egyszer futtatható le, így a mező értéke csak egyszer állítható be. Ez a megoldás nem túl elegáns. A mező kezelésére vonatkozó szabály a konstruktorban van, túl távol. Logikusabb helyen lenne a property set részében. Ráadásul, ha több konstruktorunk is van, amely ilyen paramétert kezel, akkor mindegyikbe át kellene másolni az ellenőrzést. Több indok is szól amellett, hogy az ellenőrzést kiemeljük a konstruktorból.

Nem készíthetünk azonban publikus set részt, mert akkor a kívülág is tudná használni. Ha készítünk set-et, akkor emiatt a property védelmi szintjét érdemes protected-re állítani, de akkor set és a get is protecteddé válna, a kívülág elveszíti az olvasási lehetőségét is. A megoldás az lenne, ha a property get részét publikusra, a set része pedig protected-ra módosítanánk. Ez nem probléma. Amennyiben a property mindkét része egyforma védelmi szinttel rendelkezik, úgy azt a property deklarációsor a kívülág részén kell beállítani. Az eltérő védelmi szintek esetén az egyik védelmi szintet kívül, a másik elem védelmi szintjét belül kell megadni:

```
public int életkor
{
    get
    {
        return _életkor;
    }
    protected set
    {
        if (value < 18 || value > 60)
            throw new ArgumentException("csak 18..60 lehet");
        else _életkor = value;
    }
}
```

Az eltérő szintek esetén kívül kell megadni a megengedőbb védelmi szintet (public), és belül kell azt tovább szigorítani. Ezért nem működik az alábbi megoldás:

```
protected int életkor
{
    public get
    {
        return _életkor;
    }
    set
    {
        if (value < 18 || value > 60)
            throw new ArgumentException("csak 18..60 lehet");
        else _életkor = value;
    }
}
```

## 9.8. Valódi egyszer írható mezők

Vegyük észre, hogy a property-n keresztüli egyszer írható mező valójában nem egyszer írható. A példány metódusai tetszőlegesen sokszor módosíthatják a példány életkor mezőjének értékét, csak a kívülág képtelen erre a mezőbe írás módjával.

Létezik azonban a C#-ban valódi, csak egyszer írható mező. 'readonly' kulcsszóval kell megjelölni a mezőt, és akár publikus védelmi szinttel is elláthatjuk:

```
// publikus, de readonly mező
public readonly int életkor;
// konstruktor
public diak(int pEletkor)
{
    if (pEletkor < 18 || pEletkor > 60)
        throw new ArgumentException("csak 18..60 lehet");
    else életkor = pEletkor;
}
```

A readonly mező értékét csak a konstruktorban lehet beállítani. Később már a mező nem vehet részt írási műveletben, beleértve az osztály további metódusait is:

```
public void életkorNoveles()
{
    életkor = életkor + 1;
}
```

A readonly field cannot be assigned to (except in a constructor or a variable initializer)

### 9.9. Konstansok

A konstansok és a valódi egyszer olvasható mezők rendkívül hasonlóak egymáshoz. Azonban vannak különbségek:

```
// konstans
public const int maxEletkor = 80;
// publikus, de readonly mező
public readonly int életkor;
// konstruktor
public diak(int pEletkor)
{
    if (pEletkor < 18 || pEletkor > 60)
        throw new ArgumentException("csak 18..60 lehet");
    else életkor = pEletkor;
}
```

Első különbség a deklaráció. A konstansokat a 'const', az egyszer írható mezőket a 'readonly' kulcsszavakkal lehet deklarálni. A konstansoknál a konstans értékét a deklaráció során meg kell adni, mivel a konstansok értékét csakis ott lehet definiálni. Az egyszer írható mezők értékét a konstruktorban kell definiálni.



Második különbség a külvilág szemszögéből az, hogy a konstans az osztály része; a konstans értékének kiolvasásához nem kell példány. Az egyszer írható mező azonban példányszintű, így kiolvasásához példány kell (mivel a mező értéke példányonként akár eltérő is lehet):

```
diak d = new diak(22);
int e = d.eletkor; // readonly mező kiolvasása => 22
int me = diak.maxEletkor; // konstans kiolvasása => 80
```

Megjegyezzük, hogy mint minden mező, a readonly mező kezdőértéke is megadható kezdőértékadással, akár a konstansok esetén. Ez azonban gyakorlatilag hiba, két okból. Az egyik, hogy ekkor a readonly mező lényegében már konstans (értékét a konstruktoron kívül más már nem tudná módosítani), viszont példányszintű mező lenne, tehát az értékének lekérdezéséhez példányt kell létrehozni – ami értelmetlen, hisz az érték nem példányfüggő. A másik ok, mivel példányszintű mező, minden példány számára saját mező készül a memóriában, amelyet a 'new' le is foglal. Ez memóriapazarlás, hisz bármely példányt használunk a mező értékének lekérdezéséhez, ugyanazt az értéket kapjuk:

```
class diak
{
    // konstans
    public const int maxEletkor = 80;
    // csak olvasható mező
    public readonly int max_osztondij = 90000; // HUF
```

Ha olyan konstanst szeretnénk létrehozni, amelynek lekérdezési lehetőségét példányhoz szeretnénk kötni, használjuk az alábbi módszert:

```
class diak
{
    protected const int _max_osztondij = 90000; // HUF
    public int max_osztondij
    {
        get
        {
            return _max_osztondij;
        }
    }
}
```

A konstans marad konstans, de védelmi szintje protected. Emiatt a külvilág nem tudja 'diak.\_max\_osztondij' módon elérni. A property viszont példányszintű, így a get meghívásához példányt kell készíteni. A get rész pedig megadja a konstans értékét mint visszatérési értéket.

## 10. Az adattagok

---

Az OOP-ben adattagoknak nevezzük az osztályok adattárolással foglalkozó részeit. Három eset, három típus létezik:

**példányszintű mező,**  
**osztálysintű mező,**  
**konstans.**

### 10.1. Példányszintű mezők

A példányszintű mezők olyan adatokat tárolnak, amelyek az objektumok példányai esetén eltérő értékeket tartalmazhatnak. Egy kör objektum esetén a kör példányok sugarai egymástól eltérőek lehetnek, akár a körök x és y koordinátái. A diák objektum esetén a diákok nevei, neptun kódjai, születési évei különbözőek. Minden példány esetén más. A példányszintű mező lényegében a rekord adatszerkezetbeli „mező” fogalommal egyenértékű, ezért a „példányszintű” jelzőt gyakran el is hagyjuk.

A példányszintű mezők deklarálásának szintaktikája:

```
[védelmi szint] <típus> <mezőnév> [= kezdőérték];
```

A védelmi szint lehet: public, protected, private. Ha nem adjuk meg egyiket sem, akkor az alapértelmezett védelmi szint a private. A kezdőérték megadása elhagyható, hiányában a mező kezdőértéke a típusának megfelelő nulla érték. Számok (int, double stb.) esetén nulla, logikai típusnál false, karakternél nulla kódú karakter, míg referencia típus-osztálybeli esetekben null.

Példányszintű mezőkből a memóriában kezdetben nulla darab van. Amikor példányosítunk, akkor a 'new' operátor foglalja le a helyet az új példány számára. A memóriaigényt elsősorban a példányszintű mezők száma és típusa határozza meg. A private mezőknek is ugyanúgy helyet kell foglalni, mint a protected és public mezőknek.

Az OOP szemlélet szerint a mezők védelmi szintje jellemzően protected, ritkábban private. A public mező védelem nélküli, értékét a külvilág tetszőleges időpontban a típusának megfelelő korlátok között módosíthatja. Emiatt a publikus mezőkkel dolgozó metódusok azt ellenőrizni kötelesek mielőtt ténykednének velük.

---

A példányszintű mező **hatásköre** a védelmi szintjétől függ. A private mezőkre csak az ugyanazon osztálybeli metódusok, konstruktorok, property-k törzsében lehet hivatkozni. A protected mezőkre az előbbieken felül a gyerekosztálybeli metódusok, konstruktorok és property-k is hivatkozhatnak. A public mezőkre a program szövegében bárhol hivatkozhatunk.

A példányszintű mezők **élettartama** a példány élettartalmával jellemezhető. A példányosításakor kerül be a memóriába (new + konstruktor lefutása), jellemzően a program indítást követően valamely későbbi időpontban. A mező mindaddig létezik, amíg a példány meg nem szűnik. C# nyelvben a példány megszüntetése automatikus folyamat, a Garbage Collector (GC) észleli, hogy a példány memóriacímét a program elvesztette (semmilyen változó és mező nem tárolja már a memóriacímet, sem közvetlenül, sem közvetve). A program számára a példány ekkor már nem létezik, de a memóriában még a mezők fizikailag tárolódnak, amíg a GC ténylegesen fel nem szabadítja a helyét, ami legkésőbb a program leállásakor megtörténik.

```
class diak
{
    protected int életkor;
    protected string neve = "-- ismeretlen --";
    protected string nemzetiseg = "HUN";
}
```

```
class kor
{
    public int X_koord;
    public int Y_koord;
    public double sugar;
}
```

A példányszintű mezőkre az ugyanazon osztálybeli példányszintű metódusok, konstruktorok, property-k törzsében közvetlenül hivatkozhatunk (a mező nevének leírásával), vagy a 'this' kulcsszó segítségével 'this.mezőnév' alakban. Ha a hatáskör legalább protected, akkor a mezőre a gyerekosztálybeli metódusok is hivatkozhatnak, ugyanezen szintaktikával. A public mezőkre a külvilágbeli kódok is hivatkozhatnak, számukra a szintaktika azonban más – nekik azonosítani kell a példányt is melynek a mezőjére hivatkoznak: 'példánynév.mezőnév' formában. Vagyis három szintaktika létezik:

```
mezőnév
this.mezőnév
példánynév.mezőnév
```

A példányszintű mezők értékét beállíthatjuk kezdőérték megadás formájában, de nem jellemző. Leggyakrabban a konstruktoron keresztül adjuk meg (a konstruktor a paramétereiből veszi a kezdőértéket). A mezők értékét a property-k segítségével szoktuk módosítani, de bármely metódus is módosíthatja őket.

A példányszintű mezők esetén a **readonly** jelző használható. Ilyenkor a mező értékét csak a kezdőérték megadása, vagy a konstruktor állíthatja be. A továbbiakban a property-k és a metódusok már nem módosíthatják.

## 10.2. Osztályszintű mezők

Az osztályszintű mezők olyan adatok, amelyeket a program szempontjából elég egyetlen egyszer eltárolni. Ilyen például, hogy a KRESZ szempontjából a lakott területen belüli megengedett legnagyobb sebesség 50 km/h, a nagykorúság kezdete 18 év, a minimálbér 78 000 Ft<sup>16</sup>, a könyvek áfa kulcsa 5%, vagy a benzin ára 420 Ft/l. Ezek olyan értékek, melyeket gyakran konstansként is felvehetnénk, de mégsem konstansként tesszük. A könyvek áfa-ja pl. bármikor változhat. Célszerűbb tehát az értéket valamiféle konfigurációs helyről (.ini file, .xml file, adatbázis) kiolvasni a program indulásakor, vagy bekérni billentyűzetről, illetve egyéb adatforrásból beszerezni.

Az osztályszintű mezőket gyakran magyarázzák úgy, hogy az értéke nem az egyes példányokat jellemzi, hanem az adott osztály minden példányára jellemző. Ez az indoklás is segíthet abban, hogy elképzeljük mi a szerepe az osztályszintű mezőknek.

A nem OOP környezetben, harmadik generációs nyelvekhez szokott programozók első-sorban úgy értelmezik az osztályszintű mezőket, hogy azok a globális változók. Olyan változóknak tekintik, amelybe ha értéket helyeznek, a függvények azt ott megtalálják, kiolvashatják, módosíthatják. Ez is igaz lehet, bár ebben a megfogalmazásban sok a pontatlanság. A globális jelző ugyanis a hatáskörét jellemezné, ami az OOP világában inkább a védelmi szintjét mutatja egy mezőnek. Az osztályszintű mezők a példányszintű mezőktől ugyanakkor jellemzőbb módon az élettartamukban térnek el.

Az osztályszintű mezők deklarálásának szintaktikája:

```
static [védelmi szint] <típus> <mezónév> [= kezdőérték];
```

Az osztályszintű mező szintaktikája tehát egyetlen ponton tér el a példányszintűtől: a 'static' jelzőben. A védelmi szint és a 'static' kulcsszó sorrendje felcserélhető, tehát a következő szintaktika is helyes:

```
[védelmi szint] static <típus> <mezónév> [= kezdőérték];
```

Az osztályszintű (statikus) mezők **hatáskörére** vonatkozó szabályok megegyeznek a példányszintű mezőknél leírtakkal. A private mezőkre csak az osztály, a protected mezőkre a gyerekosztályok, a public mezőkre pedig a teljes program szövegéből bárhonnan hivatkozhatunk.

Az osztályszintű mezők élettartama statikus, vagyis a mezők a program indulásakor kerülnek be a memóriába, és a program futásának végéig ott is maradnak. A létezésük nem kötődik semmi más szemponthoz. Az adott osztályszintű mezőből egy van a memóriában, akkor is ha az osztályból egyetlen példány sem készül, akkor is ha sok példány készül belőle. Az osztályszintű mezők élettartamát tehát a GC működése nem befolyásolja.

---

<sup>16</sup> 2012-ben.

```
class diak
{
    protected static int minEletkor = 17;
    public static string foiskola_elotag = "EKF";
    static public int neptun_kod_hossza = 6;
}
```

```
enum elsobbseg { balkez, jobbkez }
class kresz
{
    static public int maxSebessegLakottTeruleten = 50;
    public static double megengedettAlkoholSzint = 0;
    static public elsobbseg alapertelmezettSzabaly = elsobbseg.jobbkez;
}
```

Az osztálysztí mezők már akkor is léteznek és elérhetőek, amikor az osztályból még nem is készült példány. Ezért az osztálysztí mezőkre külső programkódból 'osztálynév.mezőnév' alakban hivatkozhatunk. Az osztályon belüli kódok (konstruktorok, metódusok, property-k) belsejében az osztálysztí mezőkre közvetlenül hivatkozhatunk szintén 'osztálynév.mezőnév' alakban. Amennyiben a védelmi szint legalább *protected*, úgy a gyerekosztályokban is hivatkozhatunk a mezőkre, 'mezőnév', vagy az eredeti osztály nevével 'osztálynév.mezőnév' alakjában.

Mivel a 'this' kulcsszó az aktuális példányt azonosítja, az osztálysztí mező viszont nem köthető példányhoz, így a 'this.mezőnév' alak nem létezik, használata szintaktikai hiba. Vagyis kétféle szintaktika létezik:

mezőnév

osztálynév.mezőnév

Az osztálysztí mezők értékét beállíthatjuk kezdőérték megadásával (jellemzően), vagy nagyon ritkán az osztálysztí konstruktoron keresztül (a konstruktor nem veheti át paraméterként, jellemzőbb, hogy számolt értéket állít be, lásd később). A mezők értékét a property-k segítségével is módosíthatjuk (nem jellemző), de leginkább az osztálysztí metódusok szokták módosítani.

Osztálysztí mezők esetén is használható a *readonly* jelző. Ekkor a mező értékét csak a kezdőérték megadása, vagy az osztálysztí konstruktor állíthatja be. A továbbiakban a property-k és a metódusok már nem módosíthatják.

### 10.3. Konstansok

A konstansok olyan adatokat tartalmaznak, melyeknek értéke a program írásakor ismert, és várhatóan nem is fog változni később. Az adatok akár a program szövegébe a felhasználás helyére is beírhatóak lennének, de érdekesebbnek érezzük névvel azonosítani a szóban forgó értéket, növelve a forráskód olvashatóságát. Ha az értékről tudjuk, hogy fix, de a program írásakor a programozó valamiért bizonytalan az értékben, akkor mindenképpen érdemes konstanst használnia. A program szövegében mindenütt a nevével használhatja, az érték beállítását pedig annak tisztázásakor egyszer kell csak megadni.

A konstansok OOP környezetben szintén valamely osztályba helyezendők. Az osztály neve későbbiekben a konstans nevéhez hozzáadódik, így érdemes a konstanst olyan osztályba elhelyezni, amelynek neve köthető a konstanshoz is. A PI konstanst pl. a Math osztályba helyezték, holott pl. a String osztályba is helyezhető lenne, de ott senkinek nem jutna eszébe keresni. Illetve a Math.PI névről mindenkinek van sejtése milyen értéket jelent, míg a String.PI nevet olvasván többen elbizonytalanodnának, hogy ez a név milyen értéket takarhat.

Az konstansok deklarációjának szintaktikája:

```
const [védelmi szint] <típus> <mezőnév> = kezdőérték;
```

Hasonlóan az osztályszintű mezőkhöz, a védelmi szint és a 'const' kulcsszó sorrendje felcserélhető. A védelmi szintre vonatkozó megjegyzések is ugyanazok. A privát konstansokra csak az osztályon belüli kódok hivatkozhatnak.

A konstansok **hatásköre** a védelmi szintjüktől függ. A konstansok értékét a külvilág nem képes módosítani, ezért a konstansok jellemzően publikusak, így az értékük a program szövegében bárhol hozzáférhető. A privát és protected konstansoknak is van létjogosultsága, ha az általuk hordozott információ a külvilág számára érdektelen vagy titkos.

A konstansok **élettartamáról** nem szokás beszélni, ugyanis a szó klasszikus értelmében nincs élettartamuk. Ha szóba kerül, mindenki természetesnek veszi, hogy a program akár legelső utasításában, kifejezésében már felhasználható a konstans; már létezik és felvette az értékét, és a program utolsó sorában is még létezik és ugyanazt az értéket képviseli. Tehát élettartama statikusnak tekinthető. Azért nem szokás mégsem erről így nyilatkozni, mert a konstansok tárolása nem a változókkal együtt (nem a változók tárolására szolgáló adatszegmensben, erre a célra allokalált területen) történik, hanem leggyakrabban a kódterületen kerülnek tárolásra. Egyes (optimalizált) fordítási folyamatokban még ez sem igaz, a konstans értéke nem kerül sehol tárolásra, hanem a hivatkozás helyén, a kifejezésben maga a konkrét értéke kerül be (kifejtett konstans), mintha literálként hivatkoztunk volna az értékére. (Jegyezzük meg, hogy a fordító dönthet akár úgy is, hogy a konstanst a változóterületen helyezi el, a többi változó között.) A programozók általában úgy gondolnak a konstansokra, hogy ennek tárolása és kezelése a fordítóprogram belügye; pontos működését, tárolási és kezelési mechanizmusát firtatni illetlen dolog.

A konstansokra az osztályon belüli kód közvetlenül a nevével hivatkozhat. Gyerekosztályon belüli kód esetén (ha hozzáfér) szintén közvetlenül annak nevével történik. Külvilágbeli kódra pedig az 'osztálynév.konstansnév' formában. E módon a konstansok és az osztályszintű mezők egyforma szintaktikával hivatkozhatók. A különbség az, hogy a konstans értéke nem módosítható, az osztályszintű mező értéke viszont igen. A readonly osztályszintű mező már-már konstans, hisz értéke az osztályszintű konstruktort leszámítva szintén nem módosítható. Ugyanakkor az osztályszintű mező fizikailag egy változó, tárolása a változók memóriaterületén történik meg. Így hát a readonly mező elvileg megfelelő trükkökkel, a fordítóprogram kijátszásával akár később is módosítható lenne, míg a konstansok tárolása egészen más módon működik, így a háttérbeli tárolása és kezelése egészen más szabályrendszeren alapszik.

# 11. Az öröklődés

---

## 11.1. A mezők öröklődése

Az öröklődés az OOP második alapelve. Az öröklődés kimondja, hogy új objektumosztály fejlesztésekor fel kell tudnunk használni egy már meglévő objektumosztályt. Az öröklődés során átvesszük az ős objektumosztály mezőit, metódusait. Lehetőségünk van a továbbiakban új mezőkkel, metódusokkal kiegészíteni azt, bővítvén a kiinduló osztály funkcionalitását, illetve biztosítani kell, hogy a gyerekosztály az örökölt, meglévő funkciók működését is módosíthassa.

Másképp fogalmazva: a gyerekosztály hozzáadhat és módosíthat, de el nem dobhat elemket az ősosztály tudásához képest. Vagyis a gyerekosztály mindent tud, amit az ősosztály, csak néhány dolgot másképp tud, valamint többet is tudhat mint az ősosztály tudott. (Ez később még fontos lesz!)

Az öröklés deklarációja egyszerű: amikor a gyerekosztályt deklaráljuk, meg kell adni, hogy kit tekintsen a fordító ősosztálynak.

```
// nincs deklarált ős
class negyzet
{
    public double a_oldal;
}

// őse a "negyzet"
class teglalap : negyzet
{
    public double b_oldal;
}
```

A 'negyzet' osztálynak egy mezője van. Ha példányosítunk (new kulcsszó) akkor a példány memóriaigénye elsősorban az az egy double mező lesz (8 byte). A 'teglalap' osztálynak két mezője van, az örökölt 'a\_oldal' és az általa hozzáadott 'b\_oldal', tehát egy téglalap példány memóriaigénye  $2 \times 8$  bájt<sup>17</sup>.

A téglalap osztálybeli metódusok úgy használhatják az 'a\_oldal' mezőt, mintha az a téglalap osztály sajátja lenne, egyenrangúként a ténylegesen saját 'b\_oldal' mezővel:

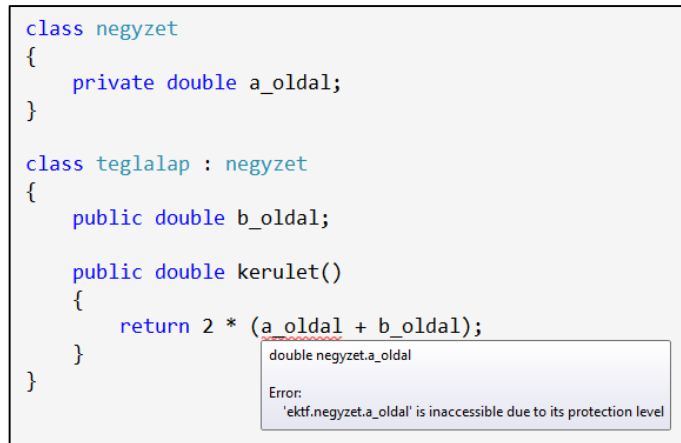
```
class teglalap : negyzet
{
    public double b_oldal;
    public double kerulet()
    {
        return 2*(a_oldal+b_oldal);
    }
}
```

---

<sup>17</sup> A tényleges memóriaigény nem pontosan ennyi, befolyásolhatja a szóhatárra igazítás, illetve további technikai mezők jelenléte az objektumban.

Gondolhatunk úgy is az öröklődésre, mintha a fordító első lépésben szeretné bemásolni (copy-paste alapon) a teljes négyzet osztálybeli forráskódot a téglalap osztályunkba, majd utána kezdené olvasni csak a téglalap osztályunkba helyezett kódot. Ránézésre jól mutatja az öröklődés működését.

De ez nem teljesen igaz, mert a négyzet osztályban 'private' módosítójú mezőkhöz a téglalap osztály is hozzáférne, hiszen ha fizikailag is átkerülne a kód a téglalapba, akkor ez beleértődne a viselkedésbe:



```
class negyzet
{
    private double a_oldal;
}

class teglalap : negyzet
{
    public double b_oldal;

    public double kerulet()
    {
        return 2 * (a_oldal + b_oldal);
    }
}
```

double negyzet.a\_oldal  
Error: 'ektf.negyzet.a\_oldal' is inaccessible due to its protection level

A hibaüzenet azt jelzi, hogy a 'private a\_oldal' mezőhöz annak védelmi szintje miatt nem férhetünk hozzá. De nem azt jelenti, hogy nincs is 'a\_oldal' mezője a téglalapnak, csak annyit, hogy van 'a\_oldal' mező, csak nem férhetünk hozzá közvetlenül!

A direkt hozzáférés mellett van indirekt lehetőség is. Ha pl. a négyzet osztály tartalmaz olyan függvényt vagy property-t, amelynek segítségével kiolvashatjuk az 'a\_oldal' mező értékét:

```
class negyzet
{
    private double _a_oldal;
    public double a_oldal
    {
        get { return _a_oldal; }
    }
}

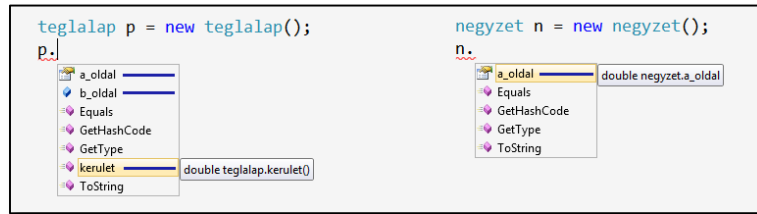
class teglalap : negyzet
{
    public double b_oldal;

    public double kerulet()
    {
        return 2*(a_oldal+b_oldal);
    }
}
```

Ez esetben örököltük a privát '\_a\_oldal' mezőt, és a publikus 'a\_oldal' property-t is. Utóbbit védelmi szintje miatt a gyerekosztály is használhatja. Ha a téglalap osztályból készítünk példányt, úgy természetes, hogy a példánynak van 'a\_oldal' csak olvasható property-je, 'kerulet()' metódusa, és 'b\_oldal' mezője.



A négyzet példány esetén csak a property van jelen (a listában szereplő egyéb metódusokra, Equals(), GetHashCode(), GetType() és ToString() később térünk ki):



## 11.2. A mezők öröklődésének problémái

Az öröklődés szabályainak megismerése után tisztázzunk néhány extrém esetet is! Ezeket az eseteket azért szükséges átbeszélni, mert bár a való életben egyáltalán nem, vagy csak nagyon ritkán fordulnak elő, de segítségükkel alaposabban megérthetjük az öröklődés működését.

Tegyük fel, hogy van egy 'első' nevű osztályunk, három mezővel:

```
class első
{
    private int a = 1;
    protected int b = 1;
    public int c = 1;
}
```

Egy ilyen 'első' példány összesen  $3 \times 4$  byte memóriát köt le, 3 mezője van. Készítsünk egy 'második' osztály, amelynek őse az 'első':

```
class második : első
{
    public double a = 2.0;
    public double b = 2.0;
    public double c = 2.0;
    public double d = 2.0;
}
```

Az alábbi üzeneteket láthatjuk a VS-ban:

```
3 'ektf.masodik.c' hides inherited member 'ektf.elso.c'. Use the new keyword if hiding was intended.
2 'ektf.masodik.b' hides inherited member 'ektf.elso.b'. Use the new keyword if hiding was intended.
```

A 'második' osztálynak eleve van három mezője az öröklés miatt, melyet újabb négy mezővel egészítünk ki. Összesen tehát 7 mezője van, a példány memóriaigénye  $3 \times 4 + 4 \times 8$  byte (3 int és 4 double mező memóriaigénye).

Azonban problémás, hogy a példány összesen 7 mezője csak 4 különböző néven osztozik. Az 'a', 'b' és 'c' mezőnevek duplán szerepelnek. A 'd' mezőnév egyedi, olyan nevet csak egy mező birtokol.

Először is jegyezzük meg, hogy ez a szituáció kerülendő. Ne nevezzünk el egyforma névvel mezőket. Erre egyébként a VS is figyelmeztet („*masodik.c hides inherited member elso.c*” = „*a 'masodik' osztály c mezője eltakarja az 'elso' osztály c mezőjét*”). Ez általában tervezési probléma. (Miért is neveznénk pont így el a mezőnket, mikor annyiféle név közül választhatunk.) A terv egyszerű módosításával a második osztálybeli mezők átnevezésével az ütközés feloldható.

Tegyük fel, hogy ezt mégsem akarjuk. A helyzet ekkor válik igazán problémássá!

Először is vegyük észre, hogy az 'a' nevű mezővel nincs baja a VS-nak, ott nem jelez hibát. Miért? Mert az 'a' mező privát, vagyis hatásköre nem terjed ki 'masodik' osztályra, így az abban deklarált másik 'a' mezővel nem fedik át egymás hatáskörét. A 'd' mező nincs jelen az első osztályban, ezért vele sincs gond.

Az első osztálybeli 'b' és 'c' mezők hatásköre azonban már kiterjed a 'masodik' osztályra is, ezért a második osztályban a mezők direkt módon is elérhetőek lennének. Az ütközést jelzi „warning” alakban a VS. A warning (figyelmeztetés) hiba is hiba, érdemes foglalkozni vele.

A figyelmeztetés a VS „nyugtalanágát” jelzi. Miközben ez nem számít szoros értelemben vett szintaktikai hibának, mégis tervezési hibára utal. A VS tehát azt kéri tőlünk: gondoljuk át a kódrészt. Ha véletlenül okoztuk a problémát, kifejezetten hasznos a warning jelzés. A „nyugtalanág” nem múlik el, amíg a VS-nek nem jelezzük, hogy döntésünk végleges és tudomásul vesszük következményeit. Döntésünket a kérdéses mező előtti 'new' kulcsszó kiírásával jelezzük.

```
class masodik : elso
{
    public double a = 2.0;
    new public double b = 2.0;
    new public double c = 2.0;
    public double d = 2.0;
}
```

Először is jegyezzük meg, hogy ez a 'new' itt nem ugyanaz, mint a példányosításkor használt 'new', legalábbis ami a jelentéstartalmát illeti. A példányosításkori 'new' memóriát foglal a példány számára, kalkulálva a mezők alapján a memóriaigényt, majd meghívja a konstruktort. Ez a 'new' itt egyszerűen azt jelenti „új”, szerepe csak jelzés értékű. Annyit tesz, hogy a fordítóprogram a továbbiakban nem zaklat minket a figyelmeztető üzenettel.

Most figyeljük meg, hogy a kód egyes területein, hogyan működnek az átfedő mezők!

```
class elso
{
    private int a=1;
    protected int b=1;
    public int c=1;

    public void elso_proba()
    {
        this.a = 11;
        this.b = 11;
        this.c = 11;
    }
}
```

Az 'elso' osztályban deklarált mezők ebben az osztályban értelemszerűen elérhetők. Az itteni kód belsejében még névütközési problémák nincsenek, mivel a gyerekosztályban deklarált mezők hatásköre az ős felé nem terjeszkedik.

A 'masodik' osztályban írt kód elvileg elérhetne a hét mezőből hatot (a privátot nem). Ugyanakkor az örökölt 'b' és 'c' mezők elnevezését eltakarják a saját ugyanilyen nevű mezők. Ennek megfelelően az itt megírt kód (metódustörzs, property vagy konstruktor törzse) elsősorban az újabb, double mezőkhöz fér hozzá:

```
class masodik : elso
{
    public double a = 2.0;
    new public double b = 2.0;
    new public double c = 2.0;
    public double d = 2.0;

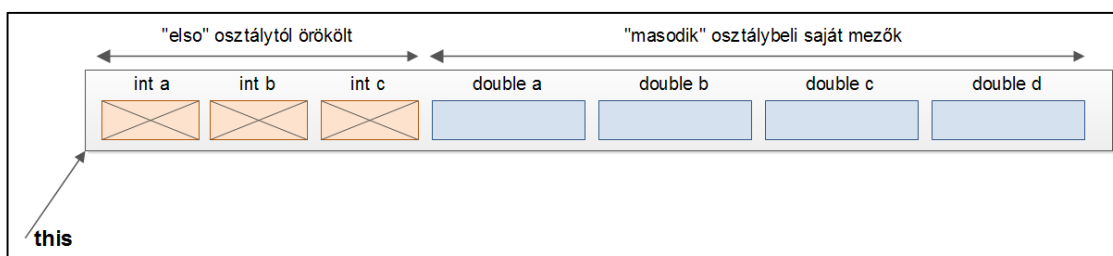
    public void masodik_proba()
    {
        this.a = 22.2;
        this.b = 22.2;
        this.c = 22.2;
        this.d = 22.2;
    }
}
```

A kérdés mindössze az: ha már vannak (mert örököltük), az int típusú 'b' és 'c' mezőket, hogyan érjük el?

Nem azon múlik, hogy a mezők neve elé odaírjuk-e a 'this.'-t, vagy sem. Mint korábban említettük, a 'this.' nélküli mezőhivatkozásokat a fordítóprogram automatikusan 'this.'-tal egészíti ki:

```
public void masodik_proba()
{
    b = 33.3;           // mindkettő ugyanarra a
    this.b = 22.2;      // mezőre hivatkozik
}
```

Ahogy a 'this' látja a 'masodik' osztálybeli példányokat:



A 'this'-en keresztül nem érhető el az 'int a', mert védelmi szintje private. Nem érhető el a 'int b' és 'int c' sem, mert ezen azonosítókat eltakarják az új mezők.

Nem működő próbálkozás, ha megpróbáljuk a fordítót értesíteni arról, hogy az 'elso' osztályban deklarált 'b' mezőt szeretnénk elérni, legalábbis az 'elso.b' módon ez nem megy. A szintaktika ugyanis a statikus osztályszintű mezők elérésének szintaktikája:

```
public void masodik_proba()
{
    elso.b = 33.3;
}
```

class ektf.elso

Error:  
An object reference is required for the non-static field, method, or property 'ektf.elso.b'

### 11.3. A base kulcsszó

Alapvetően két módszer van, hogy elérjük az őssztálybeli mezőket. Az egyikhez az 'as' típusmódosító operátort kell használnunk, melyről csak később lesz szó. A másik a 'base' kulcsszó alkalmazása. A 'this' kulcsszó az aktuális példányra mutat, végül is a 'base' is. De ha a 'masodik' osztály belsejében használjuk a 'this'-t, akkor a 'this' típusa 'masodik' lesz, vagyis a 'this.' esetén a második osztálybeli mezőkhöz férhetünk hozzá, az eltakart mezőkhöz nem. A 'base' esetén is ugyanez a helyzet, az aktuális példány mezőire utalhatunk, de a 'base' típusa az őssztály, tehát jelen esetben 'elso' típusú. A 'base' esetén nem láthatjuk az új mezőket, csak az őssztályban már meglévőket. Így a 'base.b' az őssztálytól örökölt int típusú mezőre fog mutatni:

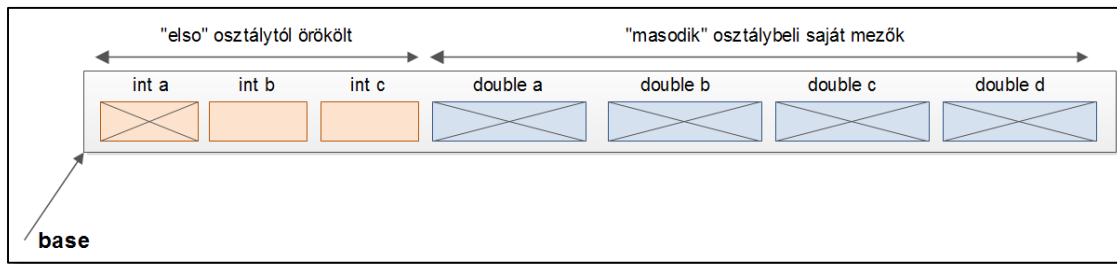
```
public void masodik_proba()
{
    base.b = 33;
}
```

b  
c  
elso\_proba

int elso.c

Jegyezzük meg, hogy a 'base' használata esetén sem férhetünk hozzá az 'a' mezőhöz. He-lyileg a 'base.a' hivatkozása a második osztály belsejében van, ahova az 'a' mező hatás-köre nem terjed ki. (Tehát nincs kiskapu!)

Ahogy a 'base' látja a 'masodik' típusú példányt:



A 'base' szerint nem érhető el az 'int a' mező, annak védelmi szintje miatt. Nem érhető el a 'double' mezők egyike sem, ezek egyszerűen nem is léteznek a 'base' számára. Ugyanakkor elérhetőek az ősoosztálybeli 'int b' és 'int c' mezők.

A 'base' használhatóságának komoly korlátja van: csak a közvetlen ősünkre lehet vele hivatkozni. Ha készítenénk egy 'harmadik' osztály, őseként választva a 'masodik'-at, akkor ezen harmadik osztálynak is pontosan ugyanaz a 7 mezője lenne, mint a másodikonál volt. A harmadik osztály belsejében írt kód esetén a 'this' típusa 'harmadik', a 'base' típusa az őse, vagyis 'masodik'. Ennek megfelelően az ottani 'base.b' már nem 'int' típusú mező lesz, hanem 'double', amely elfedi az 'int'-et:

```
class harmadik : masodik
{
    public void harmadik_proba()
    {
        base.
    }
}
```

Visual Studio autocomplete dropdown for `base.` showing:

- a
- b
- c

Selected item: `double masodik.b`

A 'base' segítségével tehát csak egy szinttel tudunk visszább lépni az őseink listájában. Ez persze nem jelenti azt, hogy a 'harmadik' osztályban már nem tudunk az örökölt, végül is létező 'int' típusú 'b' mezőhöz hozzáférni. Az 'as' operátor segítségével majd megoldható lesz.

## 11.4. A metódusok öröklődés

A metódusok öröklődésére vonatkozó szabályok nagyon hasonlóak a mezőkhöz. A private metódusok öröklődnek, bár a gyerekosztályból nem hívhatóak meg. A protected és public metódusok is öröklődnek, de azok használhatóak és meg is hívhatóak.

```
class elso
{
    private int a = 1;
    protected int b = 1;

    public void kiiras()
    {
        Console.WriteLine(" elso.kiiras ");
    }
}
```

Ezt figyelembevéve készítsük el a második osztályt is:

```
class masodik : elso
{
    public double b = 2.0;

    public void teszt()
    {
        kiiras();
    }
}
```

A 'teszt()' metódus belsejében meghívhatjuk az örökölt 'kiiras()' függvényt. Tehetjük, hisz a védelmi szintje public, tehát a gyerekosztály is meghívhatja. Ha példányokat készítünk, a függvények hívhatóak:

```
elso e = new elso();
e.kiiras();
masodik m = new masodik();
m.kiiras();
m.teszt();
```

Az 'e' példány esetén természetesen nem hívható a 'teszt' függvény, a típusa miatt ő ezt a függvényt nem láthatja. Ennek nem az az oka, hogy az 'e' példány létrehozásakor a teszt metódus nem jön létre, hisz a példányosításnak nincs köze a metódusokhoz. A példányosítás során csak a mezők számára foglalódik le hely, és hívódik meg a konstruktor. A probléma az, hogy az

```
e.teszt();
```

metódushívás a fordító belső működéséből fakadóan az alábbi módon értelmezné:

```
elso.teszt( e );
```

Itt két probléma is látszik. Az egyik, hogy az 'első' osztályban nincs 'teszt()' függvény. A másik, hogy bár a 'második' osztályban van 'teszt()' függvény, de paramétereként egy 'második' típusú példányt várna, és az 'e' nem az.

Teljesen hasonlóan működik a property öröklődése is (ezekre már korábban mutattunk példákat). Ha a property public vagy protected, akkor a gyerekosztálybeli metódusok vagy más property-k törzsében hivatkozhatunk rájuk.

Nagyon fontos újra és újra ismételni: a példány memóriaigényét csakis a mezők száma és típusa befolyásolja. A metódusok (legyenek azok örökölték vagy saját fejlesztések) példányosításkor nem igényelnek memóriát! Szintén nem számítanak az osztályszintű mezők és konstansok száma és típusa! A példányosítás során egy új garnitúra példányszintű mezőnek foglalódik hely a memóriában. Az, hogy a példányosítás előtt a metódus nem hívható meg – az csak a fordító szintaktikai trükkje. Ne támadjon az az érzésünk, hogy a példányosítás eredményeképp nemcsak mezőink, de metódusaink is lesznek. A metódusok előtte is rendelkezésre álltak, a program indulásakor már betöltődtek a memóriába (egyszer), csak éppen nem tudtuk őket meghívni! A metódusok akkor is csak egyszer szerepelnek a memóriában, ha nincs egyetlen példányunk se, és akkor is egyszer szerepelnek a memóriában, ha több tucat példányunk van.

### 11.5. A metódusok öröklődésének problémái

A metódusok öröklődése tehát a mezők öröklődésével párhuzamba állítható, hasonló elvek mentén működő dolog. A problémák akkor kezdődnek, ha egyazon nevű metódust szeretnénk a gyerekosztályban is készíteni, mint amelyet örököltünk az ősosztályunktól. Az alábbiakat kell végiggondolni:

- Az ősosztályban ez a *metódus private*? Ez esetben nincsenek problémáink, mivel a private metódust a gyerekosztály nem ismeri fel, nem hívhatja, semmilyen módon nem is tud a jelenlétéről. A gyerekosztályban ugyanilyen nevű metódus készíthető, de használni (más metódusokból meghívni) csak a sajátot fogjuk tudni.

- Az őssztálybeli metódus nem private, amit a gyerekosztályban készítünk metódust annak neve egyezik vele, de a gyerekosztálybelinek *más a paraméterezése?* Ha más a paraméterezés, akkor hiába egyforma a két metódus neve, nincs átfedés a hatáskörökben. Amikor meghívjuk (valamelyik) metódust, a nevén felül a paraméterezését is ellenőrzi a fordító, s így egyértelműen el tudja dönteni melyik változatot kívánjuk meghívni. Ez végül is ugyanaz az eset, mintha nem is lenne egyforma a két metódusnév. A szabályt **overloading** szabálynak neveztük korábban, és az OOP-ben is működik.
- Nem private, ugyanaz a név, ugyanaz a paraméterezés? Nos, az igazi probléma itt kezdődik!

Első kérdés, amit tisztáznunk kell: miért akarunk ilyet írni a gyerekosztályba? Örököltünk egy metódust, de ugyanolyan névvel és paraméterezéssel készítünk egy másikat a saját osztályunkban? A szituáció nagyon hasonló a mezők esetére, ahol ha örököltünk egy mezőt, akkor az tervezési hibára utal, ha ugyanolyan névvel készítünk egy másikat.

Metódusok esetén ez szinte sosem tervezési hiba, hanem szándékos; és pontosan a hibák elkerülése végett szoktunk ilyet végezni. Gondoljuk át, mi lenne ha a négyzet osztályunkba készítenénk egy kerület metódust, majd a téglalap gyerekosztály ezt örökölné:

```
class negyzet
{
    public double a_oldal;

    public double kerulet()
    {
        return 4 * a_oldal;
    }
}

class teglalap : negyzet
{
    public double b_oldal;
    // + orokolt a_oldal mezo
    // + orokolt kerulet() fv
}
```

A négyzet példányok jól működnek. A téglalap példányokkal mi a helyzet?

```
negyzet n = new negyzet();
n.a_oldal = 12;
double nk = n.kerulet(); // 48
// ---
teglalap t = new teglalap();
t.a_oldal = 10;
t.b_oldal = 20;
double tk = t.kerulet(); // 40 !?
```

A négyzet kerület metódusa az 'a\_oldal' értékét szorozza 4-gyel, ez így van rendjén. A téglalap öröklí a kerület metódust, de az még mindig a négyzet szerint számol kerületet, így a téglalap esetén nem a  $2 \times (10 + 20)$  lesz a kerület értéke, hanem a  $4 \times 10$  ( $4 \times a\_oldal$ ). Mi a megoldás?



Első gondolatunk: írjunk saját kerület metódust. A gondolat jó, a megvalósítással valami gond van, mert a VS aláhúzza és hibának jelöli:

```
class teglalap: negyzet
{
    public double b_oldal;
    // + orokolt a_oldal mezo
    // + orokolt kerulet() fv -- de az nekunk nem jo
    public double kerulet()
    {
        return 2 * (a_oldal + b_oldal);
    }
}
```

A hiba oka „teglalap.kerulet() *hides inherited member negyzet.kerulet()*” – vagyis szinte szóról szóra ugyanaz a hiba megfogalmazása is mint a mező esetén tapasztaltuk (a téglalap osztály kerület metódusa eltakarja az örökölt négyzet osztálybeli kerület metódust):

```
public double kerulet()
{
    return 2 * (a_oldal + b_oldal);
}
```

'ektf.teglalap.kerulet()' hides inherited member 'ektf.negyzet.kerulet()'. Use the new keyword if hiding was intended.

A megoldást is megadja a hibaüzenet: „*use new keyword if hiding was intended*” – „*használjuk a new kulcsszót ha ez szándékos*”. Ugyanaz a gondolatmenet, mint amit leírtunk a mezők esetén. A VS először is aggódik, hogy csak véletlenül írtunk olyan metódust, amelyet örököltünk. A 'new' kulcsszó segítségével megnyugtathatjuk, hogy szándékosan:

```
new public double kerulet()
{
    return 2 * (a_oldal + b_oldal);
}
```

A főprogram e pillanattól kezdve jól működik:

```
negyzet n = new negyzet();
n.a_oldal = 12;
// negyzet.kerulet( n )
double nk = n.kerulet(); // 48
// ---
teglalap t = new teglalap();
t.a_oldal = 10;
t.b_oldal = 20;
// teglalap.kerulet( t )
double tk = t.kerulet(); // 60 jó!
```

Amennyiben a gyerekosztály örököl egy `protected` vagy `public` metódust, jogában áll kifejleszteni egy ugyanolyan nevű és paraméterlistájú metódust. Ez a metódus az örökölt metódus a gyerekosztály szempontjából történő javítás. Az új metódus elfedi, eltakarja az örökölt metódust, emiatt az új metódusnál fel kell tüntetni a `'new'` kulcsszót.

## 11.6. A metódusok és a `'base'`

A mezőknél mutattuk be a `'base'` kulcsszót, melynek segítségével az őszosztálybeli mezőre hivatkozhatunk akkor is, ha a gyerekosztályban azt felüldefiniáltuk. Hasonlóan lehet az őszosztálybeli metódusokat meghívni, property-ket használni a gyerekosztályban akkor is, ha a gyerekosztály ugyanolyan névvel felüldefiniálta azokat.

Vegyük a következő példát! Készítsünk olyan objektumosztályt, amelyik listába gyűjt egész számokat, de csak a pozitívakat. Ezen felül megadja egy csak olvasható property segítségével a számok összegét.

```
class egesz_szamok
{
    protected int summa = 0;
    protected List<int> gyujto = new List<int>();
    public void hozzaad(int szam)
    {
        if (szam < 0) throw new ArgumentException("csak pozitiv");
        else
        {
            gyujto.Add( szam );
            summa += szam;
        }
    }
    public int osszege
    {
        get { return summa; }
    }
}
```

Az osztály továbbfejlesztéseként módosítsuk a viselkedést oly módon, hogy maximum 30 számot fogadhatunk el. Ehhez a `'hozzaad()'` metódust el kell takarnunk az új változattal, hogy ne is tudják a korábbi változatot használni (melynek segítségével tetszőleges sokat is hozzá tudnának adni):

```
class jav_egesz_szamok : egesz_szamok
{
    new public void hozzaad(int szam)
    {
        if (gyujto.Count > 30) throw new Exception("tul sok szam");
        else base.hozzaad(szam);
    }
}
```

A 'else' részen azért nem szabad hozzáadni még a gyűjtő listához, mert még azt is le kellene ellenőrizni, hogy az érték pozitív-e, valamint növelni kellene a számok összegét is. Ezeket az ősosztálybeli metódus mind megteszi, csak meg kell hívni a 'base' segítségével.

Hasonló probléma, ha egy halakat szimuláló program fejlesztése során szeretnénk egy hal objektumosztályt készíteni, melyből majd sok hal példányunk lesz az akváriumunkban. A halaknak van súlya (kilogrammiban, törtérték is lehet). Mivel mindenféle halunk lesz, így a hal súlyaként elfogadunk bármilyen, nullánál nagyobb, és mondjuk 120 kg-nál kisebb értéket. Készítsük el a hal súlyát tároló mezőt (protected) és a publikus property-t hozzá:

```
class hal
{
    protected double sulya;
    public double sulya
    {
        get { return _sulya; }
        set
        {
            if (value < 0 || value > 120)
                throw new ArgumentException("nem megfelelo suly");
            else _sulya = value;
        }
    }
}
```

A gyerekosztályban felveszünk egy „életben van-e” jellegű logikai mezőt. Ha a hal nincs életben már, akkor a súly property sem működik. Ha azonban életben van, akkor a működése nem változott az ősosztálybeli viselkedéshez képest:

```
class elohal : hal
{
    protected bool elo_e = true;
    new public double sulya
    {
        get
        {
            if (elo_e) return base.sulya;
            else throw new Exception("nem elo hal");
        }
        set
        {
            if (elo_e) base.sulya = value;
            else throw new Exception("nem elo hal");
        }
    }
}
```

### 11.7. A metódusok öröklődésének igazi problémája

A metódusok öröklése további problémákat vet fel. Hogy bemutassuk az egyiket, gondolkodjunk el azon, ha a négyzet osztályt kiegészítjük egy 'kiir\_kerulet()' függvénnyel az alábbiak szerint (s melyet a téglalap is örököl, aki ráadásul felüldefiniálja a kerulet metódust), akkor mit ír ki a végén bemutatott főprogram:

```
class negyzet
{
    public double a_oldal;

    public double kerulet()
    {
        return 4 * a_oldal;
    }

    public void kiir_kerulet()
    {
        Console.WriteLine("kerulet = {0}", kerulet() );
    }
}
```

```
class teglalap : negyzet
{
    public double b_oldal;

    new public double kerulet()
    {
        return 2 * (a_oldal + b_oldal);
    }
}
```

```
negyzet n = new negyzet();
n.a_oldal = 12;
n.kiir_kerulet();
// ---
teglalap t = new teglalap();
t.a_oldal = 10;
t.b_oldal = 20;
t.kiir_kerulet();
```

A kód szintaktikailag hibátlan, mégsem az történik amit szeretnénk. Hogy tovább fokozzuk a problémát, bemutatunk még egy olyan kódrészletet, aminek megértése még várat magára (ahogy az is, hogy ez a kódrészlet miért nem működik az elvárásoknak megfelelően):

```
static void kiiras(negyzet p)
{
    p.kiir_kerulet();
}

public static void Main()
{
    negyzet n = new negyzet();
    n.a_oldal = 12;
    kiiras(n);
    // ---
    teglalap t = new teglalap();
    t.a_oldal = 10;
    t.b_oldal = 20;
    kiiras(t);
}
```

## 12. Típuskompatibilitás

---

Képzeljük el, hogy van egy objektumosztályunk, belőle egy példány: 'p'. Ennek a 'p'-nek számos mezője és metódusa van. Készítünk egy gyerekosztályt, s abból egy példányt: 'k'. A továbbiakban vizsgáljuk meg, mi a viszony a két példány között:

- Ha 'p'-nek van valamiféle mezője, akkor olyan mezője a 'k'-nak is van! Miért? Mert örökölte!
- Ha 'p'-nek van egy property-je, akkor olyan property-je a 'k'-nak is van! Miért? Mert örökölte!
- Ha a 'p'-nek van egy metódusa, akkor olyan metódusa 'k'-nak is van! Miért? Mert örökölte!

Megállapíthatjuk tehát, hogy a gyerekosztálybeli példánynak, a 'k'-nak biztosan megvan minden olyan mező, metódus, property, ami a 'p'-nek is. Ez egyszerű következménye az öröklődés szabályának, és annak a ténynek, hogy a gyerekosztály az öröklődés során nem döntheti el mit örököl és mit nem. Nem választhat ki mezőket, metódusokat, hogy „ezeket nem kérem, a többi jöhet”. S bár sokszor nagyon hasznos lenne, de akkor elbuknánk mindazon lehetőségeket, amelyek az alábbiakban kerülnek ismertetésre.

Jelenleg azonban elmondhatjuk, hogy a gyerekosztály OOP szempontból minden esetben legalább annyi mindent tartalmaz, mint az ősosztály (mezők, metódusok stb.). Ahol egy ősosztálybeli példány ('p') alkalmazható (mert képes minden adatot eltárolni amit el kell tudni tárolni, és metódusai segítségével képes minden műveletet elvégezni, amire szükség van) – ott alkalmazható a gyerekosztálybeli példány ('k') is! Amit a 'p' el tudott tárolni, azokat a 'k' is el fogja tudni tárolni pontosan ugyanazon nevű és típusú mezői segítségével, amely tevékenységeket a 'p' el tudott végezni, azokat a 'k' is el fogja tudni végezni pontosan ugyanazon nevű és paraméterezésű metódusaival.

Az ilyen szituációkat a való életben úgy fogalmazzuk, hogy 'k' szakszerűen képes helyettesíteni 'p'-t. Az informatika világában nem helyettesítésről, hanem **kompatibilitásról** szoktunk beszélni, így azt mondhatjuk, hogy a 'k' kompatibilis 'p'-vel. Igazából a gyerekosztály bármely példánya képes helyettesíteni az ősosztály bármely példányát, tehát a kompatibilitás nemcsak az egyes példányok között igaz, hanem a teljes típusok között is.

A gyerekosztály az öröklődés miatt az ősosztálytól minden mezőt, metódust, property-t átvesz. Emiatt a gyerekosztály példányai képesek helyettesíteni az ősosztály példányait. Általánosságban: **a gyerekosztály típuskompatibilis az ősosztályával!**

Ha egy 'A' osztálynak egy 'B' osztály gyerekosztálya, akkor a 'B' kompatibilis az 'A' osztállyal. Ha a 'B'-nek is van egy gyerekosztálya 'C', akkor a 'C' kompatibilis a 'B'-vel. Kérdés: a 'C' kompatibilis-e az 'A'-val?

```
class A { /* ... */ }
class B : A { /* ... */ }
class C : B { /* ... */ }

public static void Main()
{
    A a;
    B b;
    C c = new C();
    b = c;
    a = b;
}
```

Mivel 'C' kompatibilis a 'B'-vel, így helyes a 'b = c' értékadás. Mivel a 'B' kompatibilis az 'A'-val, így helyes az 'a = b' értékadás. Mi kerül az 'a' változóba? A 'b' értéke. Mi van a 'b'-ben? A 'c' értéke. Vagyis végső soron az 'a' változóba a 'c' értéke kerül.

```
A a;
C c = new C();
a = c;
```

Rövidebben is leírható ez az értékadás: 'a = c'. Mivel a 'c' típusa kompatibilis a 'b'-vel, a 'b' pedig az 'a' típusával, így az értékadó utasítás típushelyes. További indoklás is járhat hozzá, gondoljuk végig: egy 'c' példánynak is megvan minden mező, metódus, property, ami megvan egy 'A' típusú példánynak is. Honnan? Örökölte a 'B' osztálytól, aki örökölte az 'A' osztálytól.

A típuskompatibilitás tranzitív, minden osztály kompatibilis nemcsak a közvetlen ősével, hanem annak ősével is, felfelé, egészen a kezdetig. Az osztályok kompatibilisek minden direkt és indirekt ősükkel!

## 12.1. A típuskompatibilitás következményei

A típuskompatibilitás kulcsfontosságú fogalom, ugyanis az értékadó utasítás ezt a szabályt használja fel. Ismétlésképp:

Az értékadó utasítás helyes, ha a bal oldalon álló változó típusa egyezik a jobb oldalon álló kifejezés típusával, vagy ezen kifejezés típusa kompatibilis vele.

Ennek megfelelően, ha a téglalap osztály a négyzet osztály gyerekosztálya, akkor helyes az alábbi értékadás:

```
teglalap t = new teglalap();
negyzet n = new negyzet();
n = t;
```

Sőt, egyszerűbben írva:

```
teglalap t = new teglalap();
negyzet n;
n = t;
```

Még egyszerűbben írva:

```
teglalap t = new teglalap();
negyzet n = t;
```

Legegyszerűbben írva:

```
negyzet n = new teglalap();
```

Vegyük sorra, miért is működnek a fenti értékadó utasítások. Először is vegyük észre, hogy bal oldalt minden esetben az 'n' áll, akinek a típusa 'negyzet'. Minden értékadó utasítás jobb oldalán végső soron egy 'teglalap' típusú dolog áll – s a téglalap a korábban leírtak szerint típuskompatibilis a 'negyzet'-tel. Vagyis minden értékadó utasítás helyes!

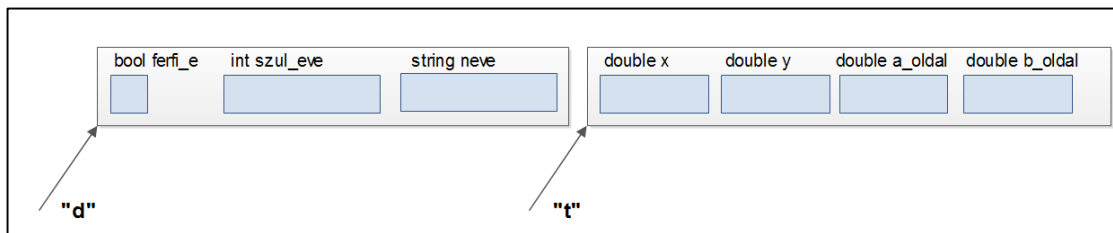
Gondoljunk bele, a 'class' típussal létrehozott típusok – vagyis az objektumosztályok – mind a referencia típuscsaládba tartoznak. A példány változók mindegyike 4 byte helyigényű, és csak egy memóriacímet tartalmaznak. Emiatt ha van pl. egy 'e' és 'f' példányunk, mindegy milyen objektumosztályokból, fizikailag akár az 'e = f;' akár az 'f = e;' értékadások működhetnének – hiszen az értékadás során technikailag az egyik 4 byte-os memóriacímet kell átírni egy másik, 4 byte-os memóriacímek fogadására kiválóan alkalmas változóba.

Mi történne, ha ezt a fordítóprogram ellenőrzés nélkül megengedné? Nézzük az alábbi példát! Legyen egy diák osztályunk és egy téglalap osztályunk az alábbiak szerint:

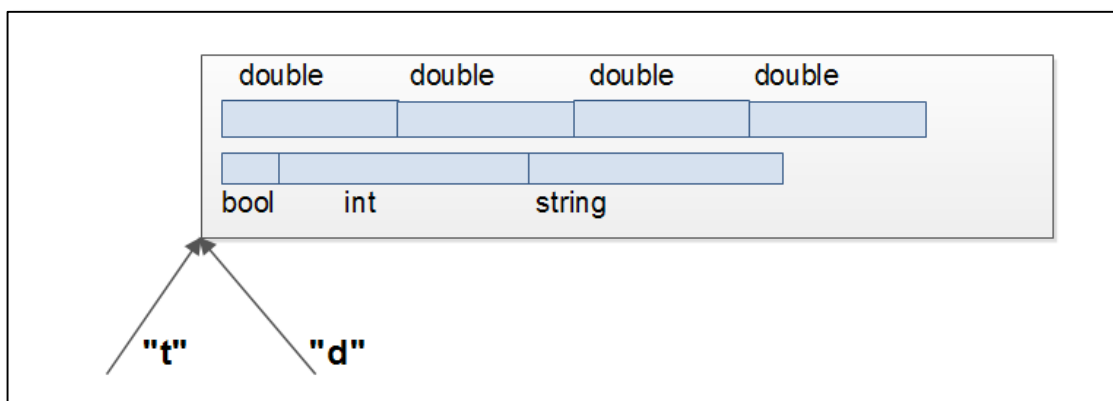
```
class diak
{
    protected bool ferfi_e;
    protected int szul_eve;
    protected string neve;
    // ... további mezők ...
}

class teglalap
{
    protected double x;
    protected double y;
    protected double a_oldal;
    protected double b_oldal;
    // ... további mezők ...
}
```

Hozzunk létre egy 'd' nevű példányt a diákból, és egy 't' nevű példányt a téglalapról. Ekkor a memória az alábbiak szerint néz ki:



Ha végrehajtanánk egy 'd = t;' értékadást, akkor ennek értelmében a 'd'-be átmásolnánk a 't'-beli memóriacímet (4 byte), s így a 'd' is a téglalapra mutatna:



Ezáltal a memóriaterületet kétféleképpen is értelmeznénk. A 'd.ferfi\_e = true;' értékadás pl. beállítaná a bool mezőt true-re, ami egyben a téglalap 'x' mezőjének első byte-ja is (ahol a double érték első byte-ja foglal helyet). Ezekután a 'double f = t.x;' értékadás (ami kiolvassa a double mezőnk értékét) meglepő eredményt adna. Értelemszerűen ez nem megengedett.

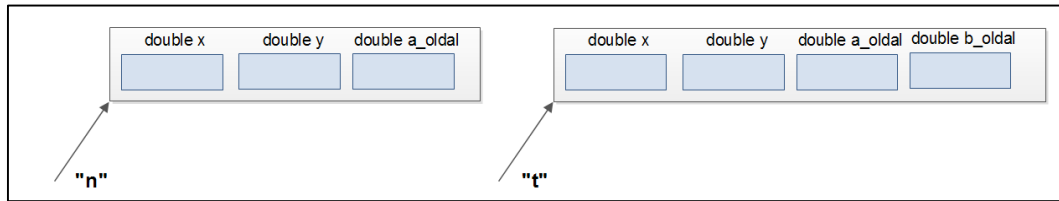
Viszont a típuskompatibilitás mellett nem lehet probléma:

```
class negyzet
{
    protected double x;
    protected double y;
    protected double a_oldal;
    // ... további mezők ...
}

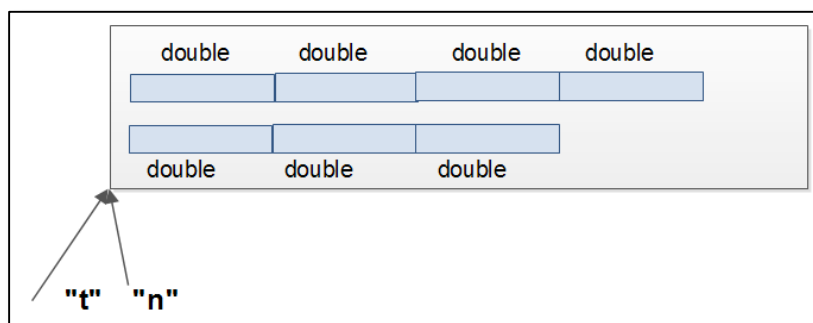
class teglalap:negyzet
{
    protected double b_oldal;
    // ... további mezők ...
}
```



Ekkor a téglalap öröklí a mezőket a négyzettől, vagyis a téglalap példányok memóriaterületén az első részek felépítése és szerkezete egyezik a négyzet példányokéval:



A típuskompatibilitás miatt a `'n = t;` értékadás elvégezhető. Ekkor az `'n'` változó is a téglalap területére fog mutatni, de `'n'`-en keresztül csak az első három mező érhető el (az `'n'` példánynak nincs `'b_oldal'` mezője, így hiába van a memóriaterületen annak is hely foglalva, a fordító nem engedi meg az `'n.b_oldal'` leírását):



Ha `'n.x'` mezőt módosítjuk, akkor a memóriában a `'t.x'` mező is módosul, hiszen a két objektum ezen mezői a memóriában ugyanazon a helyen foglalnak helyet. Az `'x'` mezők típusa egyforma (öröklődés), és pozíciójuk a memóriaterület belsejében is ugyanaz. Ez tehát biztonságos.

Lássuk a korábban bemutatott értékedásokat!

```
teglalap t = new teglalap();
negyzet n = new negyzet();
n = t;
```

Először létrehozunk egy téglalap példányt, és a memóriacímét eltároljuk a `'t'` változóban. Második lépésként a négyzetet hozzuk létre, memóriacímét az `'n'` változóban tároljuk el. A harmadik lépésben az `'n = t'` értékadás segítségével az `'n'`-be átmásoljuk a `'t'`-beli memóriacímét. Az `'n'`-ben tárolt korábbi memóriacím, a négyzet típusú példány memóriacíme azonban felülíródik, elveszett. A lefoglalt memóriaterületet a Garbage Collector később fel fogja szabadítani. Ennek ellenére értelmetlen lefoglalni egy memóriaterületet, mellyel műveletet nem végzünk, és értelmetlen felesleges munkával terhelni a GC-t.

```
teglalap t = new teglalap();  
negyzet n;  
n = t;
```

Itt már az 'n' nem kapja meg a felesleges értéket, csak deklarálásra kerül, és a következő sorban veszi át a 't'-beli memóriacímét. Itt a memóriában már csak egyetlen téglalapnyi példánynak foglaltunk le helyet, és két változónk, a 'n' és a 't' is ismeri ezen terület memóriacímét.

```
teglalap t = new teglalap();  
negyzet n = t;
```

A harmadik kísérlet során a deklarációt egyszerűen összevontuk a kezdőértékadással.

```
negyzet n = new teglalap();
```

A legutolsó esetben is ugyanezt kapjuk. Csak itt már kiiktattuk a 't' változót is, a frissen létrehozott téglalap példány memóriacímét egyenesen az 'n' változóba helyezzük el. Az értékadás ebben a formában látszólag értelmetlen, de működik!

Ugyanis hiába téglalapnak foglaltunk le helyet (4 double), és a téglalap konstruktorát hoztuk létre, az 'n' példányváltozón keresztül csak az első három mezőjéhez férhetünk hozzá, és csak azokat a metódusokat hívhatjuk meg, amelyek már a négyzet osztályban is léteztek.

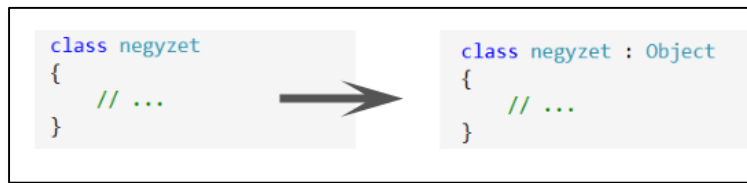
## 12.2. Az Object osztály

Amennyiben nem jelölünk meg egyértelműen ősosztályt, azt gondoljuk, hogy nincs is ősosztályunk. Ám ekkor a fordító egy alapértelmezett ősosztályt jelöl ki ősnek. Az ősosztály az **Object** nevű objektumosztály.

Mivel a gyerekosztályok kompatibilisek az ősosztályukkal, így az ősosztály nélküli osztályok típuskompatibilisek az Object osztállyal. Ugyanakkor ha kijelöljük az ősosztályt, akkor meg vele kompatibilis a gyerekosztályunk, s a tranzitivitás miatt annak őseivel is, vagyis kompatibilisek vagyunk az Object osztállyal.

```
A C# nyelvben minden objektumosztály típuskompatibilis az  
System névtér Object osztályával. Ezen osztály aliasneve „object”  
– gyakran így találkozhatunk a forráskódokban a névvel.
```

Vagyis ezt írjuk – és ezt érti a fordítóprogram:



Hogy minden osztály kompatibilis az `Object` osztállyal, annak fontos következményei vannak. Az `object` típusú változókba történő értékadás (bal oldal típusa `object`) esetén a jobb oldali kifejezés típusa lényegében bármilyen lehet – hisz minden típus kompatibilis az `object`-tel (vagyis a jobb oldala garantáltan kompatibilis a bal oldalával):

```

object ne = new negyzet();
object donald = new kacsa();
object szam = 2 * Math.Sin(30 * Math.PI / 180);
    
```

Ennek ebben a formában még nincs sok értelme, de ha tovább vizsgáljuk a kérdést, nagyon is sok felhasználási lehetőséget fogunk találni.

### 12.3. A statikus és dinamikus típus

Nézzük az értékadást:

```

negyzet n = new teglalap();
    
```

Látszik, hogy `n` típusa (a deklarációja szerint) `'negyzet'`, míg (mint korábban leírtuk) a példány, aminek a memóriacímét őrzi az egy teljes értékű `'teglalap'` (minden mezője megvan ami egy téglalapnak meg kell legyen, és a téglalap konstruktra fut le, állítja be a mezők kezdőértékét).

Mi most az `n` típusa akkor valójában? A típusa `'negyzet'`, vagy inkább `'teglalap'`?

Új fogalmakkal kell megismerkednünk. A deklaráció típusa nagyon fontos, és általában egyezik azzal a típussal, amely a létrejött példány sajátja is:

```

Random rnd = new Random();
    
```

Amennyiben a két típus eltér egymástól, saját nevet kell adni az egyik, illetve a másik típusnak. A deklaráció során megadott típust „statikus típus”-nak nevezzük, míg a valóban létrejött példány típusának a neve „dinamikus típus” (valódi típus). Vagyis az `n` statikus típusa `negyzet`, a dinamikus típusa `teglalap`. Az `'rnd'` statikus és dinamikus típusa is `Random`.

Nyilván nincs semmi gond mindaddig, míg a két típus egyforma. Ekkor minden az elvárásoknak megfelelően tud működni. Azonban sok izgalmas kérdést vet fel az az eset, amikor a két típus különböző.

```
static void kiiras(negyzet p)
{
    p.kiir_kerulet();
}

public static void Main()
{
    negyzet n = new negyzet();
    n.a_oldal = 12;
    kiiras(n);
    // ---
    teglalap t = new teglalap();
    t.a_oldal = 10;
    t.b_oldal = 20;
    kiiras(t);
}
```

A fenti példán is látható egy ilyen izgalmas eset. A kiírás függvény paraméterként egy 'negyzet' típusú példányt vár a 'p' paraméterében. A hívás helyén átadhatunk egy 'n' négyzet példányt. A függvényhíváskor a háttérbeli működés miatt a paraméterváltozó ('p') felveszi a hívás helyén szereplő kezdőértéket ('n' értékét), vagyis végrehajtódik egy 'p = n' értékadás. Végrehajtató, hisz a 'p' és az 'n' statikus típusa is egyforma. A továbbiakban ugyanezt a függvényt meghívjuk úgy is, hogy egy 't' téglalap példányt adunk át a függvénynek. Ekkor a 'p = t' értékadás hajtódik végre, s mint láttuk, a típuskompatibilitás miatt végre is hajtható. A 'p' paraméter statikus típusa 'negyzet' lesz (így deklaráltuk a paraméterlistában), de dinamikus típusa téglalap, hisz a második esetben egy téglalap példányra fog mutatni a memóriában.

## 12.4. Az 'is' operátor

Mint láthattuk, van olyan eset, hogy egy függvény paraméterét adott típusra deklaráljuk, de nem tudhatjuk, valójában mit kapunk a hívás során. Persze egy dolgot tudhatunk – amit kapunk az vagy ugyanaz az osztály egy példánya, vagy valamely gyerekosztálybeli példány. De ennél többet tényleg nem tudhatunk.

Felmerülhet a kérdés, van-e lehetősége a függvénynek, hogy a megvizsgálja ténylegesen mit is kapott? Mi a valódi (dinamikus) típusa a paraméternek?

Nos, erre az 'is' operátor segítségével van lehetőség. Az 'is' operátorral meg tudjuk vizsgálni a valódi típust:

```
static void kiiras(negyzet p)
{
    if (p is teglalap)
    {
        // amit kaptunk az teglalap
        Console.WriteLine("ez egy teglalap");
    }
    else
    {
        // amit kaptunk az nem teglalap
        Console.WriteLine("ez egy negyzet");
    }
}
```

A 'p is teglalap' egy kifejezés, logikai típusú eredményt ad. Egész pontosan azt vizsgálja meg, hogy a „p dinamikus típusa kompatibilis-e a 'teglalap' típussal”. Ki kompatibilis a 'teglalap' típussal? Maga a 'teglalap' típus, és a gyerekosztályai (tetszőleges mélységben).

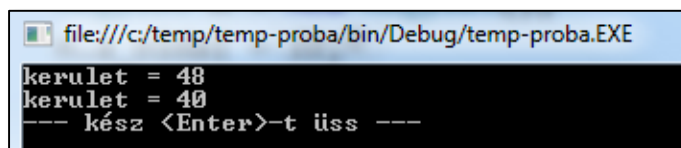
## 12.5. A korai kötés és problémái

Térjünk vissza a előbb bemutatott problémára. Ott a négyzet osztályban volt egy 'kiir\_kerulet' metódus, amely meghívta a 'kerulet' függvényt, és a kapott értéket kiírta a képernyőre. A 'teglalap' osztályban a kerulet függvényt felüldefiniáltuk, mivel a téglalap kerületét máshogyan kell számolni, de a kerulet kiírásához nem nyúltunk. A főprogram példányosított mind a négyzet, mind a téglalap osztályból, a példányokat átadta a 'kiiras' nevű függvénynek, aki meghívta a kerulet kiírását.

```
static void kiiras(negyzet p)
{
    p.kiir_kerulet();
}
```

Ha lefuttatjuk a kódot, az derül ki, hogy nem jól működik:

```
negyzet n = new negyzet();
n.a_oldal = 12;
kiiras(n);
// ---
teglalap t = new teglalap();
t.a_oldal = 10;
t.b_oldal = 20;
kiiras(t);
```



```
kerulet = 48
kerulet = 40
--- kész <Enter>-t üss ---
```

A négyzet kerülete jó érték,  $4 \times 12$ , de a téglalapé nem jó:  $2 \times (10 + 20)$ -nak kellene lenni, de helyette  $4 \times 10$ -nek tűnik, a kiírás szerint.

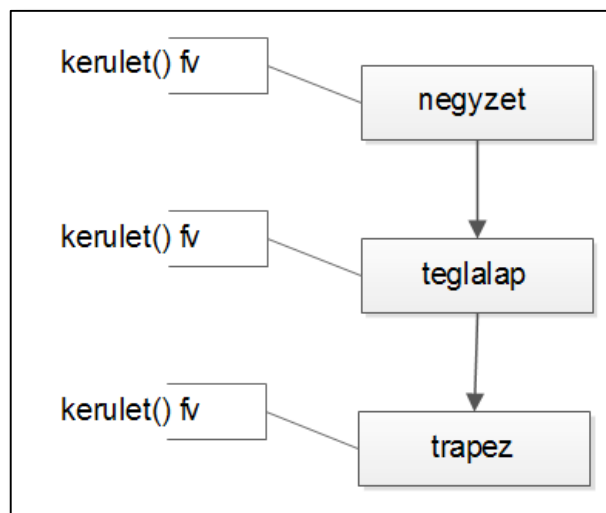
Miért? Másodjára egy téglalap példányt adunk a függvénynek, s a téglalapban a kerület metódust javítottuk, annak jól kellene működnie! A probléma oka a **korai kötés**.

A nem OOP programozási szemlélet esetén, az overloading szabály mellett előfordulhat, hogy a kódban egyforma nevű, de nem egyforma paraméterezésű függvények szerepelnek. A fordítóprogramnak a konkrét függvényhíváskor el kell dönteni, melyik függvényt akarjuk meghívni. Ez a döntési folyamat a **kötés** (*binding*).

A kötés (binding) során a fordítóprogram a függvényhívást konkrét függvénnyel kapcsolja össze.

Az OOP környezetben a binding sokkal összetettebb probléma. Ugyanis könnyen előfordul, hogy örököltünk egy metódust, majd ugyanazzal a névvel és paraméterezéssel készítettünk egy javítását. Tehát itt a metódushívásnál általában ugyanazon névvel és paraméterezéssel több metódus is elérhető, több is létezik.

Tételezzük fel, hogy van egy olyan objektumosztályunk, amelynek ősei is rendelkeznek 'kerület()' nevű, paraméter nélküli metódussal. Az objektumosztályok minden szintjén felüldefiniáljuk a metódust a **new** kulcsszó segítségével:



Ha a trapéz osztályból példányosítunk és hívjuk meg a kerület metódust, akkor egyértelműnek tűnik a válasz arra a kérdésre: melyik kerület metódus fog lefutni?

```
trapez tr = new trapez();
double ker_trapez = tr.kerulet();
```

A 'tr' példány számára valójában három kerület metódus is rendelkezésre áll, hiszen ő örökölte a négyzetét, a téglalapét is, mielőtt a trapéz osztály kifejlesztette volna a sajátot. Úgy érezzük, hogy mégis a trapéz osztálybeli fog lefutni, hisz a 'tr' példány számára az a „legmegfelelőbb”. Persze a „legmegfelelőbb” fogalma nem létezik az OOP-ben, ezért inkább másképp fogalmazzuk meg. A 'tr' példány típusának pontos ismeretében a fordító a fejlesztési fában visszafele haladva keresi azt a metódust, amelyet hívni kell. Ha a tra-

péz osztályban nem lenne ilyen nevű és paraméterezésű metódus, akkor egy szinttel feljebb lépve a téglalap osztályban ellenőrizné a metódus meglétét. Ha a keresés során egyik szinten sem találna 'kerulet()' metódust (aminek védelmi szintje is megfelelő), akkor jelezne hibát („metódus nem található”).

De a 'tr' példány melyik típusát használja fel kiindulási pontként? A dinamikus vagy a statikus típusát?

```
teglalap tx = new trapez();
double tx_kerulet = tx.kerulet();
```

Egyszerű tesztprogrammal ellenőrizhetjük a fordító viselkedését. Hamar kiderül, hogy a statikus típusból indul ki, mivel a fenti esetben a 'tx' példány kerületének számításához a téglalapbeli kerulet függvényt fogja felhasználni (ha F11-el lépésenként hajtjuk végre a programot, akkor látni is fogjuk, hogy a 'tx.kerulet()' hívásakor a téglalap osztálybeli változatba lép be a program).

Nem értjük miért is teszi ezt a program? Pedig nagyon egyszerű okból történik. Képzletben két (jelenleg) egymást követő utasítást távolítsuk el egymástól jó messzire. Lehet, hogy nem is ugyanazon függvényben van a két utasítás. Lehet, hogy formailag nem, de működésében néz csak ki így:

```
static void kiiras(teglalap p)
{
    double dk = p.kerulet();
}

public static void Main()
{
    trapez tr = new trapez();
    kiiras(tr);
}
```

Itt látszik, hogy a téglalap típusú 'p' változó valójában egy trapéz példány memóriacímét kapja meg, tehát mint az előző esetben: statikus típusa téglalap, dinamikus típus a trapéz. Meghívjuk a kerulet metódust a 'p.kerulet()' módon. Melyik kerulet függvény fusson le?

Tanulság, ne várjunk csodát a fordítóprogramtól. A fordítóprogram nagyon sokszor úgy viselkedik, mintha a program sorait nem is sorban egymás után, hanem az utasításokat egymástól függetlenül olvasgatná és értelmezné. Amit az előző sorban olvasott és megértett, azt a következő sor olvasásakor már „elfelejti”. Csak az adott sorra koncentrálni, abból próbálja kihozni a maximumot. A

```
double tx_kerulet = tx.kerulet();
```

sor megértésekor annyira hagyatkozik, amit tud. Tudja, hogy a 'tx' deklarált (statikus) típusa téglalap. Már elfelejtette, hogy a 'tx'-be mi valójában trapézt raktunk. A 'tx.kerulet()' hívásakor tehát csak a statikus típust ismeri, csakis erre hagyatkozhat.

A 'kerulet()' metódus keresését tehát a fában a téglalap osztály magasságában kezdi, és halad felfele. Mivel a téglalap osztályban talál ilyen kerület metódust (new-val felüldefiniálva), ezért ezt a sor az alábbi módon értelmezi:

```
double tx_kerulet = teglalap.kerulet( tx );
```

A 'kiiras' metódus használata mellett könnyebben belátható, hogy mást nem is tehet. Ott semmilyen ismerete nem is lehet arról, hogy a 'p' példánynak mi a dinamikus típusa, csak a 'p' statikus típusa ismert számára. Szintén ugyanez a helyzet – a 'p' statikus típusa szerint téglalap, így a 'p.kerulet()' hívás a téglalap osztálybeli metódus hívását jelenti.

Még mindig felmerül bennünk a kérdés, hogy miért nem veszi figyelembe a dinamikus típust? Nos, a fordítóprogram szeret döntést hozni. Ha ő fordítási időben (amikor bőven van idő) ráér ezen gondolkodni, megvizsgálni a statikus típust, és eldönteni melyik metódust kívánjuk mi meghívni – hát megteszi. Ez nekünk azért jó, mert amikor a program fut, és eléri a metódushívási pontot a futás, akkor a korábbi döntés miatt már lehet is ugrani az adott függvényre, és folytatni a program futását.

Ha nem így lenne, hanem a fordítóprogram itt nem hozna döntést, akkor semmilyen döntés nem tudna itt születni arról, hogy melyik kerület metódust is kívánjuk meghívni. Ehhez meg kellene várni míg a program konkrétan eléri az adott pontot, akkor ott (futás közben) ellenőrizni a 'p' dinamikus típusát (akkor már ismert lenne, a konkrét esetben) és meghozni a döntést – melyik kerület függvény kerüljön végrehajtásra. Mivel ez futás közbeni esemény, egyértelműen lassítja a program futását. Hányszor? Mindannyiszor, ahányszor a program futása eléri ezt a pontot. Nem számít hanyadik alkalommal történik meg. A 'p' dinamikus típusa minden alkalommal lehet más-más, tehát futás közben minden alkalommal újra és újra ki kell értékelni.

Tudni kell, hogy van arra lehetőségünk hogy mégis ezt az utóbbi viselkedést írjuk elő – de ekkor igényünket jelezni kell a fordítóprogram felé. A 'new'-val történő felüldefiniálás a jelzést nem hordozza magában.

A kötés az a folyamat, amikor a fordítóprogram a metódushívást összekapcsolja a megfelelő metódussal. A fordító ennek során a példány statikus típusát tudja csak figyelembe venni. A döntés fordítási időben születik meg, így futás közben már nem kell erre időt vesztegetni. Sőt biztosítja a program maximális futási sebességét. Ezt a viselkedést **korai kötésnek** (*early binding*) nevezzük.

A korai kötés az OOP világába lényegében a korábbi programozási módszerek örökségének köszönhető. A moduláris programozásban ugyanis ismeretlen fogalom a dinamikus típus, így nem volt igény a másik típusú kötésre. Az OOP világában azonban ez a kötési típus valójában sok hibára ad okot, hisz a fentiek alapos megértése és végiggondolása



hiányában nem ismernénk fel, és nem értenénk a kódunk esetleges hibás működési lehetőségét.

## 13. A virtuális metódusok

---

Az OOP-ben korán felismerhető és megérthető probléma az örökölt metódusok felüldefiniálási igénye. Erre az eddig megismert módszer – a 'new' kulcsszó segítségével – azonban nem biztosítja a kellő rugalmasságot. A típuskompatibilitás maximális kihasználhatósága mellett gyakori az az eset, amikor a statikus és a dinamikus típus eltér egymástól. Láthattuk, hogy a metódushívás során a statikus típus a döntő, így könnyen előfordulhat, hogy nem a megfelelő, nem a legújabb fejlesztésű metódus kerül hívásra.

Korábban jeleztük, van lehetőségünk arra, hogy kérjük a fordítót: ne a statikus típus alapján hozzon döntést. Ekkor felvállaljuk, hogy a rugalmasságért majd futási sebességgel kell fizetnünk, hiszen a fordító elhalasztja a döntést későbbi időpontra. A későbbi időpont futás közbeni időpont lesz, tehát a futás során, amikor már a dinamikus típus is ismert, akkor fog a megfelelő metódus kiválasztásra kerülni. Az elhalasztott kötési típust **késői kötésnek** (*late binding*) nevezzük.

Hogy a késői kötést használhassuk, ne alkalmazzuk a 'new' kulcsszót a metódusok felüldefiniálásakor. A 'new' egy kényszerűségből használt kulcsszó, csak arra alkalmas, hogy „megnyugtassuk” a fordítóprogramot.

A késői kötés kérést a fordítóprogram felé a metódusok felüldefiniálásakor kell megtenni. Két kulcsszót kell megismernünk: 'virtual' és 'override'.

A 'virtual' (virtuális) jelzőt a metódus elé az őssztály írja (esetünkben a négyzet). Az az őssztály, amely még nem örökölte a metódust, hanem ő írja a metódus első változatát. (A 'new' kulcsszavas világban ennek az osztálynak még nem kellene használnia a 'new' kulcsszót.)

A 'virtual' jelző a fordítóprogram számára hordoz üzenetet. Azt jelzi, hogy a metódust a gyerekosztályok *nagy valószínűséggel* felül fogják írni a saját változatukkal. Emiatt utasítja a fordítót, hogy ha később valahol ezt a metódust hívja a program, ott ne a korai kötést, hanem a késői kötés módszerét alkalmazza.

```
class negyzet
{
    public double a_oldal;

    virtual public double kerulet()
    {
        return 4 * a_oldal;
    }
}
```

Az 'override' kulcsszót a gyerekosztályok használják (ne a 'new'-t!), amikor felüldefiniálják az örökölt virtuális metódust.

```
class teglalap : negyzet
{
    public double b_oldal;
    override public double kerulet()
    {
        return 2 * (a_oldal + b_oldal);
    }
}
```

Ebből következik, hogy a 'virtual' kulcsszót csak egyszer, az ősosztályban kell használni, a gyerekosztályok már minden alkalommal az 'override'-ot használják:

```
class trapez : teglalap
{
    public double c_oldal;
    override public double kerulet()
    {
        return 2 * c_oldal + a_oldal + b_oldal;
    }
}
```

### 13.1. Az override és a property

Nemcsak metódusok esetén használható a virtual + override páros. Mivel a property-k is valójában metódusok, így property-k esetén is alkalmazhatók.

```
class kacsa
{
    protected double _suly;
    virtual public double suly
    {
        get { return _suly; }
        set
        {
            if (value <= 0) throw new ArgumentException("nem jo suly");
            else _suly = value;
        }
    }
}
```

A gyerekosztályban a property felüldefiniálása során nem szükséges mind a 'get', mind a 'set' részt felülírni. Elfogadható pl. ha csak a 'get' részt definiáljuk felül, a 'set' rész ekkor változatlanul marad:

```
class adv_kacsa : kacsa
{
    protected bool _elo_e = true;
    public override double suly
    {
        get
        {
            if (elo_e == false) throw new Exception("nem elo kacsa");
            else return _suly;
        }
    }
}
```

## 13.2. Az override egyéb szabályai

A virtual + override során az alábbiakkal vannak problémák:

- private,
- mezők,
- osztályszintű metódusok és property-k,
- módosult paraméterlista.

Nem kell hangsúlyozni, hogy 'private' védelmi szintű metódusra, property-re nem alkalmazható a virtual jelzés. Ezeket a gyerekosztály bár örökli, de védelmi szintje miatt nem ismerheti fel. Nem lehetséges az „elfedés”, felüldefiniálás. A „projekt.kiiras(): virtual or abstract members cannot be private” hibaüzenet fordítása: „a kiiras() metódus: virtual vagy abstract módosítójú elemek nem lehetnek private védelmi szintűek”.

```
class projekt
{
    // private metódusra (propertyre sem) alkalmazható
    virtual private void kiiras()
    {
        Console.WriteLine("hello world");
    }
}
```

'ektf.projekt.kiiras()': virtual or abstract members cannot be private

Szintén nem minősül felüldefiniálásnak, ha a gyerekosztályban a metódus felüldefiniálásakor más paraméterezést használunk. Az override során csak a metódus törzsét módosíthatjuk, sem a nevét, sem a paraméterezését, sem a visszatérési érték típusát nem változtathatjuk meg. Más paraméterezés esetén az overloading szabály ad támpontot a működésre. A „teglalap.kerulet(bool): no suitable method found to override” hibaüzenet fordítása „nem található bool paraméterezésű 'kerulet' metódus amelyet felülírhatnánk”.

```
class teglalap: negyzet
{
    public double b_oldal;
    override public double kerulet( bool ellenorzes )
    {
        double k = 2 * (a_oldal + b_oldal);
        if (ellenorzes && k<0) throw new Exception("A kerulet negativ!");
        else return k;
    }
}
```

'ektf.teglalap.kerulet(bool)': no suitable method found to override

Mezőkre viszont egyszerűen nem működik a virtual + override. Mezők esetén csak a 'new' kulcsszó segítségével végezhetünk felüldefiniálást, de mint korábban erről szó volt: kerülendő. A „the modifier 'virtual' is not valid for this item” hibaüzenet fordítása „a 'virtual' módosító kulcsszó nem alkalmazható erre az elemre”.

```
class projekt
{
    // mezőkre nem alkalmazható a virtual
    public virtual double adatmezo;
}
```

The modifier 'virtual' is not valid for this item

Osztályszinten nem működik a `virtual` és az `override`. Ennek legfőbb oka, hogy a késői kötéshez dinamikus típus szükséges, ami példányt feltételez. Osztályszintű metódusok és property-k használatakor példány sincs, így nincs dinamikus típus sem. A „*static member projekt.kiiras() cannot be marked as override, virtual or abstract*” hibaüzenet fordítása „*osztályszintű kiiras() metódus mellett nem használható az override, virtual, abstract módosítók*”.

```
class projekt
{
    // static metódusra (propertyre sem) alkalmazható
    public static virtual void kiiras()
    {
        Console.WriteLine("hello world");
    }
}
```

A static member 'ektf.projekt.kiiras()' cannot be marked as override, virtual, or abstract

Feltételezzük, hogy a gyerekosztály örököl valamely metódust, de szeretné felülírni. Az ősosztályban azonban a programozó nem írta a metódus mellé a `'virtual'` jelzőt, emiatt a gyerekosztályban nem alkalmazható az `'override'`, csak a `'new'`. Viszont mint már tudjuk, a `'new'` esetén nem működik a késői kötés, így a programunk hibásan működhet, régi metódusváltozatot használhat az új helyett. Sajnos, ekkor nem sokat tudunk tenni. Az ősosztályba nem írhatjuk be utólag a `'virtual'`-t annak programozója helyett.

Ezt elkerülendő a Java programozási nyelven a `virtual` és `override` kulcsszavak eleve nem is léteznek, használatuk automatikus. Minden metódus első változata azonnal megkapja a `virtual` jelzőt, és a gyerekosztályokban az ugyanazon nevű és paraméterezésű változatok automatikusan `override` módosítót kapnak.

Úgy érezzük, ennek csak előnyei vannak. Egy dologról ne feledkezzünk meg: a `virtual` és `override` legfőbb előnye, hogy utasítja a fordítóprogramot a hívási pontokon való késői kötések alkalmazására. Azonban a késői kötés lassúbb futást is eredményez. Nem gond, ha a lassúbb futás és a kód helyes működése között kell döntenet, egyértelmű a késői kötés alkalmazása. Az automatizmus mellett azonban nincs lehetőségünk a korai kötés választani: minden metódushívásunk késői kötésű – vállalva annak sebességét is.

### 13.3. Manuális késői kötés – `'as'` operátor

Maradjunk annál a problémakörnél, hogy az ősosztály nem írta be a `'virtual'` jelzőt, így a gyerekosztály programozója nem használhatja az `'override'`-ot, mégis helyesen kell működtetni a programot.

Induljunk ki a *négyzet*  $\Rightarrow$  *téglalap*  $\Rightarrow$  *trapéz* szituációból. A *négyzet* gyerekosztálya a *téglalap*, amelynek a gyerekosztálya a *trapéz*. Minden osztálynak saját `'kerulet()'` metódusa van, de a *négyzet* osztály programozója nem használta a `'virtual'`-t.

Szeretnénk készíteni egy 'kiiras()' metódust, amely paraméterként kap egy négyzetet, és kiírja annak területét:

```
static void kiiras(negyzet p)
{
    double dk = p.kerulet();
    Console.WriteLine("A terület = {0}", dk);
}
```

A típuskompatibilitás értelmében ha 'negyzet' típusú a paraméterünk, akkor valójában átadhatunk négyzeten felül annak minden gyerekosztályából képzett példányt is:

```
negyzet ne = new negyzet();
teglalap te = new teglalap();
trapez tr = new trapez();
kiiras( ne );
kiiras( te );
kiiras( tr );
```

A 'new' használata miatt azonban a 'kiiras' belsejében a 'p.kerulet()' minden esetben a négyzetbeli 'kerulet()' függvény lesz (mivel a 'p' statikus típusa négyzet, és korai kötést alkalmaztunk).

Az 'is' operátor segítségével megvizsgálhatjuk a dinamikus típust, így pótolhatjuk a késői kötés hiányát. Próbálkozzunk az alábbival:

```
static void kiiras(negyzet p)
{
    double dk = 0;
    if (p is trapez) dk = p.kerulet();           // trapéz esete
    else if (p is teglalap) dk = p.kerulet();    // téglalap esete
    else dk = p.kerulet();                       // négyzet esete
    Console.WriteLine("A terület = {0}", dk);
}
```

Bár a kód jól néz ki, de a 'dk = p.kerulet()' minden ágon gyanúsan egyforma. Ha kipróbáljuk a működést, észlelhetjük a hibás számolást a kiírásokban.

Vizsgáljunk meg egy ilyen elágazás ágat alaposabban:

```
if (p is trapez) dk = p.kerulet();           // trapéz esete
```

Meg kell értenünk, hogy a fordító az 'if'-ről annyit tud, hogy zárójelben egy feltételt kell megadni, majd azt egy utasítás fogja követni (vagy egy utasításblokk). A feltétel és az utasítás között nem feltételez speciális értelmi összefüggést. Jelen esetben az 'is' operátor eredménye egy logikai érték, így szerepelhet feltételben. Most nézzük az utasítást: meg kell hívni a 'p' példány 'kerulet()' függvényét. Milyen logikát is kell alkalmazni?

1. A 'kerulet()' metódus virtual? Nem. Akkor korai kötés.
2. A korai kötésnél a 'p' statikus típusa dönt.
3. A 'p' statikus típusa 'negyzet'.
4. Akkor a 'p.kerulet()' esetén a négyzet osztálybeli 'kerulet()' függvényt kell meghívni.

Mi megvizsgáltuk, hogy az 'if' feltételében a 'p' dinamikus típusa trapéz-e. Azt várnánk, hogy a fordító az 'if' ezen ágán ennek megfelelően, az ismeret birtokában, a trapéz kerület függvényét hívja meg. De nincs ilyen működés, a fordítóprogram ilyen összefüggéseket nem alkalmaz. A 'p.kerulet()' hívását minden előzménytől mentesen, azoktól függetlenül értelmezi, és hozza meg a döntését.

Ha azt szeretnénk, hogy ne a négyzet kerület függvénye hívódjon meg korai kötés közben, akkor az alábbiakat kell végiggondolni:

- késői kötés nem fogunk tudni alkalmaztatni, mert ahhoz meg kellett volna jelölni a metódust már fejlesztés közben a virtual jelzővel,
- a korai kötés minden esetben a példány statikus típusa alapján működik,
- tehát a statikus típust kell módosítani!

Típusmódosítás alatt típuskényszerítést értünk. A típuskényszerítés során a fordító statikus típusról alkotott feltételezését módosítjuk. Típuskényszerítést az első félévben tanultaknak megfelelően a zárójelezett típuskiírással is végrehajthatunk:

```
if (p is trapez) dk = (trapez)p.kerulet();
```

A próbálkozás nem nagyon sikeres. Lássuk az okait:

- az értékadás jobb oldalán valójában három operátor foglal helyet,
- ha több operátor is van, akkor fontos az operátorok prioritása,
- az egyik operátor a zárójelezett típuskényszerítés,
- a második operátor a pont operátor a metódus kiválasztásakor,
- a harmadik a függvényhívó operátor, a metódus neve után álló zárójelpár<sup>18</sup>,
- a három operátor közül a pont operátor a legerősebb, majd a zárójelpár, végül a típuskényszerítés.

Ebben a kifejezésben a *pont* operátor a legerősebb, így elsőként ő értékelődik ki. Másodikként a függvény hívó operátor, harmadikként a típuskényszerítés. Ennek megfelelően a kifejezés értelmezése: „*hívd meg a p.kerulet függvényt, és a visszaadott érték típusát fogd fel trapéz típusúként*”. Mivel a kerület függvény double-lel tér vissza, azt nem lehet trapéz típusra kényszeríteni. Így talán már érthető a hiba oka. A „*cannot convert type 'double' to 'trapez'*” fordítása tehát „*nem konvertálható a double típus trapézra*”.

```
if (p is trapez) dk = (trapez)p.kerulet();
```

```
double negyzet.kerulet()
```

Error:

Cannot convert type 'double' to 'ektf.trapez'

<sup>18</sup> Klasszikus értelemben véve a fv() alakban a () a függvény hívást jelző operátor a C, C++ nyelveken létezik. A C# is C alapú nyelv, így nem tekinthető nagy hibának a ()-t függvényhívó operátornak nevezni. A C# nyelvben az operátor precedencia táblázatban is fel van tüntetve, bár itt ezen operátor szerepe (és ismerete) jóval kisebb, lévén, hogy nem használata általában szintaktikai hibát eredményez.

Amikor az operátorok kiértékelése nem megfelelő sorrendben történik prioritásuk miatt, zárójelezést kell alkalmaznunk. A jelenlegi működés az alábbi zárójelezés szerint került értelmezésre:

```
if (p is trapez) dk = (trapez) ( p.kerulet() );
```

Nekünk másik működésre van szükségünk, így az alábbi zárójelezést kell használnunk:

```
if (p is trapez) dk = ((trapez)p).kerulet();
```

Vagyis a 'trapez' típus köré írt zárójelpár a típuskényszerítés jele, míg a második zárójelpár a 'p'-re alkalmazza a típuskényszerítést. Innentől kezdve a fordító tudomásul veszi, hogy a 'p' statikus típusa trapéz. A továbbiakban a kerület metódus hívására továbbra is korai kötést alkalmaz, de ez esetben a trapéz-beli változatot fogja hívni.

A dupla zárójelezés kényelmetlen, nem elegáns, és véletlen elhagyása véletlen félreértésekhez vezethet. Szerencsére van egy hasonló feladatú, de más szintaktikájú típuskényszerítés is. Az első, hagyományos forma a zárójelbe írt típusnév. A második alak az 'as' operátort használja:

```
((trapez)p)                      (p as trapez)
```

Formailag az 'as' operátor nagyon hasonlít az 'is' operátorra, előre kerül a példány neve ('p'), az 'as' operátor, és a típus neve, amire típuskényszerítünk:

```
if (p is trapez) dk = (p as trapez).kerulet();                      // trapéz esete
```

Ennek segítségével tehát a teljes kód:

```
static void kiiras(negyzet p)
{
    double dk = 0;

    if (p is trapez) dk = (p as trapez).kerulet();                      // trapéz esete
    else if (p is teglalap) dk = (p as teglalap).kerulet(); // téglalap
    else dk = p.kerulet();                                              // negyzet esete
    Console.WriteLine("A kerulet = {0}",dk);
}
```

A négyzet esetében nem kell típuskényszeríteni, mivel a 'p' statikus típusa eleve négyzet.



### 13.4. Amikor csak a típuskényszerítés segít

Mivel a mezőkre nem alkalmazható a `virtual + override`, itt minden esetben „korai kötés”-t alkalmazunk. Ahhoz, hogy elérjük a megfelelő osztálybeli mezőt, a példány típusát a megfelelő osztályra kell kényszeríteni valamilyen módszerrel (mindegy melyikkel). Az „A base kulcsszó” fejezetben volt szó arról, hogy ha a gyerekosztály felüldefiniál egy mezőt a `'new'` segítségével, akkor az őosztálybeli még elérhető a `'base'`-zel, de az öröklődési sorban feljebb már nem tudunk lépni.

A típuskényszerítés alkalmazható a példányszintű metódusok belsejében a `'this'` nevű (aktuális) példányra is. Így tetszőleges ősünkhöz visszatérhetünk:

```
class trapez : teglalap
{
    new public double a_oldal;
    //
    public void mezok_kiprobalasa()
    {
        double negyzet_a = (this as negyzet).a_oldal;
        double teglalap_a = (this as teglalap).a_oldal;
        double trapez_a = this.a_oldal;
        // ...
    }
}
```

Feltételezve, hogy az `'a_oldal'` mezőt minden szinten felüldefiniáltuk, a fenti módszerrel bármely ősünk szintjén definiált mezőt le tudjuk kérdezni. A trapéz osztály belsejében (harmadik sor) szereplő metódusok belsejében a `'this'` statikus típusa trapéz, így a típuskényszerítés felesleges.

### 13.5. A típuskényszerítés nem csodafegyver

A típuskényszerítés megkerülhetetlen, ha mezőkhöz kívánunk hozzáférni. A szituáció eleve kerülendő (mezők felüldefiniálása a `'new'` kulcsszóval). Másrészt a mezőkhöz `property`-t is definiálhatunk, amelyeken keresztül a mezők elérhetőek, és a `property`-k már rendelkezhetnek `virtual + override` jelöléssel.

A metódusok esetén eleve rendelkezésünkre áll a `virtual + override`. Csak akkor kerülünk bajba, ha az őosztály programozója nem használta a `virtual`-t, így a gyerekosztályok programozói nem használhatják az `override`-t. Ekkor a

```
static void kiiras(negyzet p)
{
    double dk = 0;

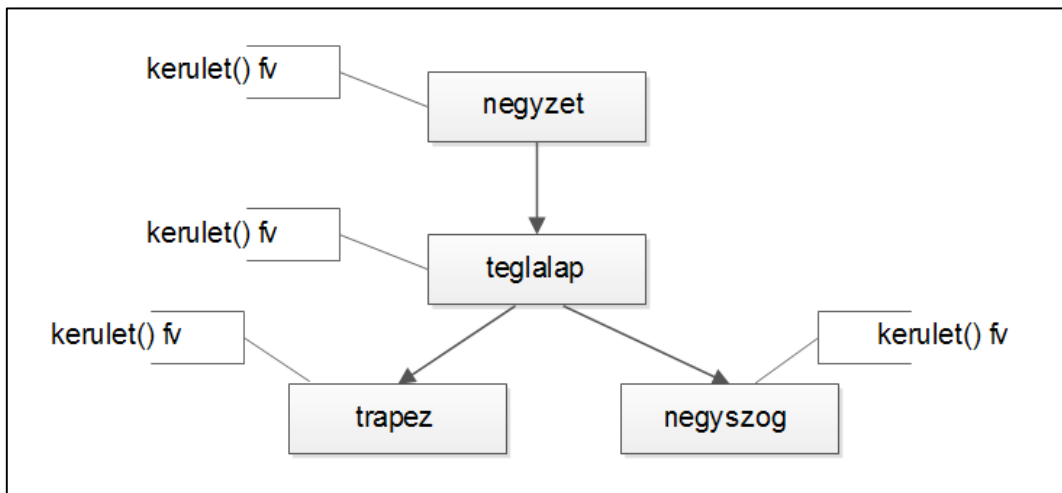
    if (p is trapez) dk = (p as trapez).kerulet();    // trapéz esete
    else if (p is teglalap) dk = (p as teglalap).kerulet(); // téglalap
    else dk = p.kerulet();                          // negyzet esete
    Console.WriteLine("A kerulet = {0}", dk);
}
```

kód nyújthat megoldást.

De vegyük észre az alábbiakat:

- ha a szóban forgó 'kerulet()' metódus virtual lett volna, akkor nem lett volna szükség az if-ekre (sebesség),
- a megoldás során annyi if-et kell írni, ahány osztályban felüldefiniáltuk a 'kerulet()' metódust,
- ha kész is vagyunk az összes if-fel, bármikor keletkezhet új osztály, amikor vissza kell ide jönnünk, hogy az újabb if-et beleírjuk a kódba,
- de csak ha az új osztály is override-olja a kerulet metódust.

Lássuk mi történne, ha pl. téglalap osztályból egyszer csak származtatnánk egy általános négyszög gyerekosztályt. Mivel a négyszögnek akár mind a négy oldala különböző hosszúságú, így természetes, hogy a kerulet metódust felüldefiniálja.



Mi történne, ha a kiírás függvénynek négyszög példányt adnánk át, de az if-ekhez nem nyúlnánk?

```

static void kiiras(negyzet p)
{
    double dk = 0;

    if (p is trapez) dk = (p as trapez).kerulet(); // trapéz esete
    else if (p is teglalap) dk = (p as teglalap).kerulet(); // teglalap
    else dk = p.kerulet(); // negyzet esete
    Console.WriteLine("A kerulet = {0}", dk);
}
  
```

Ne feledjük, az 'is' operátor nem pontos típusegyezést vizsgál, csak típuskompatibilitást. Lássuk, a négyszög kivel típuskompatibilis? Minden osztály az ősoztályaival típuskompatibilis, vagyis a téglalappal is és a négyzettel is.

Az első 'if' a trapézal típuskompatibilitást vizsgál, ami false eredményt ad, hisz azzal nem kompatibilis. A második a téglalappal, az true eredményt ad, azzal típuskompatibilis. Ekkor a négyszög példányt típuskényszerítjük téglalapra, és meghív-

juk a kerület metódust; a téglalap kerület metódusát. Ez nem jó a négyszögre, tehát bele kell nyúlnunk az új osztály megjelenésekor a kódba:

```
static void kiiras(negyzet p)
{
    double dk = 0;

    if (p is trapez) dk = (p as trapez).kerulet(); // trapéz esete
    else if (p is teglalap) dk = (p as teglalap).kerulet(); // teglalap
    else if (p is negyszog) dk = (p as negyszog).kerulet(); // négyszög
    else dk = p.kerulet(); // negyzet esete
    Console.WriteLine("A kerulet = {0}", dk);
}
```

De így rossz helyre raktuk be a vizsgálatot! A téglalap vizsgálata megelőzi a négyszögét, s mivel a téglalappal kompatibilis a példány dinamikus típusa, így ez az if teljesül, a parancsok letutnak. A felépítés miatt a következő vizsgálat már nem kerül kiértékelésre, a négyszögünk vizsgálatát hiába építettük be – még mindig hibásan működik.

Ha a fenti megoldást kell választanunk, jól gondoljuk végig milyen sorrendben írjuk fel a feltételeket. A fejlesztési fa levélelemeivel kell kezdeni, és haladni felfelé, a gyökér felé: (trapéz | négyszög)  $\Rightarrow$  téglalap  $\Rightarrow$  négyzet:

```
static void kiiras(negyzet p)
{
    double dk = 0;

    if (p is trapez) dk = (p as trapez).kerulet(); // trapéz esete
    else if (p is negyszog) dk = (p as negyszog).kerulet(); // négyszög
    else if (p is teglalap) dk = (p as teglalap).kerulet(); // teglalap
    else dk = p.kerulet(); // negyzet esete
    Console.WriteLine("A kerulet = {0}", dk);
}
```

Tehát sok a hibalehetőség az 'is'-'as' pár használata mellett is. Ráadásul egy nagyobb projekt esetén sem garantálhatjuk, hogy minden objektumosztályt ismerni fogunk, és be tudjuk illeszteni a listába. Nem tudunk univerzális megoldást készíteni. Ha azonban a metódust megjelöltük virtual-lal, és maga a fordító alkalmazza a késői kötést ezen a ponton, akkor a kód ilyen egyszerűen felírható:

```
static void kiiras(negyzet p)
{
    double dk = p.kerulet();
    Console.WriteLine("A kerulet = {0}", dk);
}
```

### 13.6. A kenguruk története<sup>19</sup>

*„Objektum-orientált kód újrafelhasználása miatt érte egy kis kellemetlenség az ausztrál hadsereget. Manapság ugye már egyre jobb helikopter-szimulátorokat csinálnak, amelyek szinte teljesen ugyanazt nyújtják, mint az igazi repülés. Domborzat, időjárás, növényzet: mind teljesen élethű. Az ausztrálok úgy gondolták, hogy az állatokat is be kellene rakni, mivel azok a menekülésükkel információt szolgáltathatnak az ellenségnek a környéken repülő helikopterről.*

*A kutatás-fejlesztés főnöke üzent a programozóknak, hogy tegyenek a programba néhány kenguru-falkát is. A programozók öreg rókaként persze nem kezdtek vadul kódot írni, hanem elővettek egy már meglévő részt: a gyalogságot. A menekülési algoritmus ugyanaz maradt, mindössze a bitmap képeket kellett lecserélni, a futás sebességét megnövelni és már készen is volt a kenguru-csapat.*

*Történt aztán egyszer, hogy amerikai katonák jöttek látogatóba az ausztrálokhoz. A helyi nagyfiúk persze egyből vakítani kezdtek az amerikaiaknak: mélyrepülésben húztak a nagy kenguru-nyáj felé, mire azok jól szétspricceltek. Az amerikai katonák elismerően bólogattak a mutatvány láttán... aztán döbbentek egy nagyot, amikor a kenguruk visszatértek az egyik domb mögül és Stinger-rakétákkal zárótűz alá vették a szerencsétlen helikoptert.*

*A programozók ugyanis elfelejtették kivenni ezt a részt a kódból (az összes attribútum öröklődött). A tanulság az ausztrál programozók számára az lett, hogy óvatosan kell bánni a kódok újrafelhasználásával, az amerikaiak számára pedig az, hogy az ausztrál vadvilág tényleg olyan veszélyes, mint ahogy azt beszélük. A nagyfőnökök egyébként örültek az esetnek, mert a pilóták megtanulták a leckét: azóta mindegyikük szigorúan elkerüli a kengurukat.”*

---

<sup>19</sup> Forrás: <http://sirkan.iit.bme.hu/~kapolnai/fun/reusecode.txt>. A történet nem hiteles, de érdekes.

## 14. Problémák a konstruktorokkal

---

Az öröklődés miatt a konstruktorokra még egyszer ki kell térnünk. Egyelőre annyit tudunk a konstruktorokról, hogy segítségükkel inicializálhatjuk a mezőinket a konstruktor paramétereinek segítségével. Az osztályunkban több konstruktor is lehet, mindaddig míg azok paraméterezése eltérő. Ezen felül fontos ismeret, hogy lehet paraméter nélküli konstruktorunk is, sőt, ha nem írunk egyetlen konstruktort sem, akkor a rendszer generál neki egy paraméter nélküli konstruktort, amelynek a törzse nem tartalmaz egyetlen utasítást sem.

Amennyiben vannak őssztályaink, és a gyerekosztály fejlesztésébe kezdünk, tisztában kell lennünk az őssztálybeli konstruktorokkal.

Az első kérdés, amire a választ keressük: vajon csak az az egy konstruktor fut le, amelyet a példányosításkor meghívunk? Hozzuk létre a négyzet  $\Rightarrow$  téglalap  $\Rightarrow$  trapéz fejlesztési fát, minden osztályban egyetlen paraméter nélküli konstruktort írjunk, melybe egy 'WriteLine' segítségével egysoros szöveget írunk ki:

```
class negyzet
{
    public negyzet()
    {
        Console.WriteLine("a negyzet konstruktora");
    }
}
```

```
class teglalap : negyzet
{
    public teglalap()
    {
        Console.WriteLine("a teglalap konstruktora");
    }
}
```

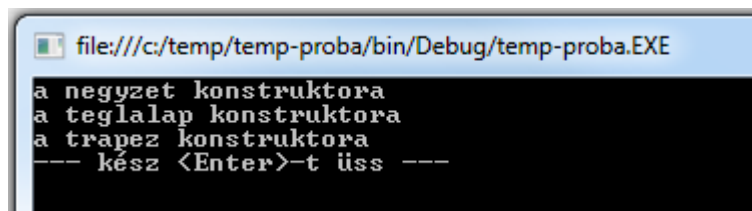
```
class trapez : teglalap
{
    public trapez()
    {
        Console.WriteLine("a trapez konstruktora");
    }
}
```

Készítsünk példányt a trapézból:

```
public static void Main()
{
    trapez tr = new trapez();
    Console.WriteLine("--- kész <Enter>-t üss ---");
    Console.ReadLine();
}
```

---

És futtassuk le a programot:

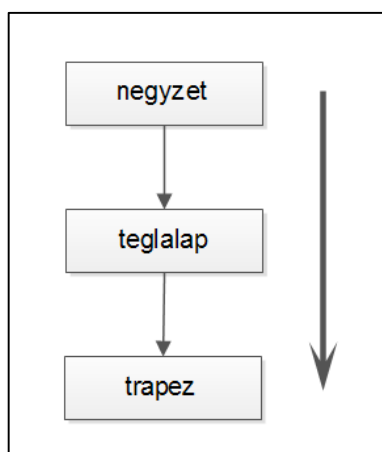


```
file:///c:/temp/temp-proba/bin/Debug/temp-proba.EXE
a negyzet konstruktora
a teglalap konstruktora
a trapez konstruktora
--- kész <Enter>-t üss ---
```

Észlelhetjük, hogy nemcsak egyetlen konstruktor futott le, hanem több konstruktor is. A jelenséget meg kell vizsgálnunk!

### 14.1. Konstruktor hívási lánc

Amit azonnal láthatunk, hogy ugyan a példányosításkor csak a trapéz konstruktort hívjuk meg, de mégis lefutnak az őszosztály konstruktorai. Sőt, ha alaposan megfigyeljük, először az első őszosztályé, a négyzeté, majd a fejlesztési fában lefelé haladva ugyanebben a sorrendben kerül végrehajtásra mindegyik. Legvégül annak a gyerekosztálynak a konstruktora fut le, amelyből a példányt valójában készítjük:



A fenti viselkedést **konstruktor hívási lánc**nak nevezzük, amely szerint a példány elkészítésében az őszosztályok konstruktorai is részt vesznek. Vajon ez véletlen vagy szándékos? Hogy kiderüljön, módosítsunk például a téglalap osztályon: az ottani konstruktornak legyen paramétere, melynek segítségével be lehet állítani a téglalap B oldali mezőjének értékét:

```
class teglalap : negyzet
{
    public double b_oldal;
    public teglalap(double b_oldal)
    {
        this.b_oldal = b_oldal;
        Console.WriteLine("a teglalap konstruktora");
    }
}
```

Ha megpróbáljuk lefordítani a programot, hibát jelez a trapéz osztály konstruktora környékén:

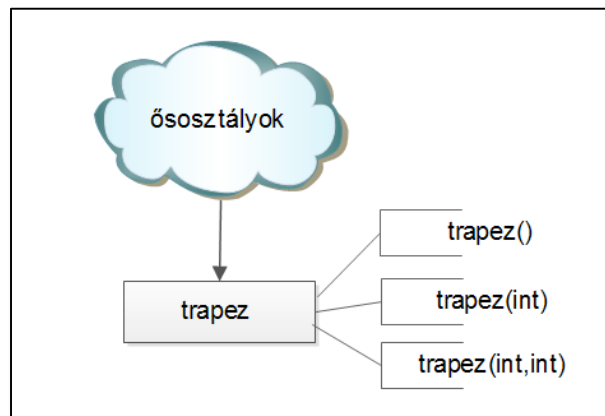
```
class trapez : teglalap
{
    public trapez()
    {
        Console.WriteLine("a trapez konstruktora");
    }
}
```

'ektf.teglalap' does not contain a constructor that takes 0 arguments

A probléma az lesz, hogy nincs biztosítva az ősosztály konstruktorának meghívhatósága, ezért a gyerekosztály nem elfogadható. Jelzi, hogy az ősosztályok konstruktorának lefutása nem opcionális, hanem kötelező! Ha valamiért ez nem biztosított, akkor az szintaktikai hiba, és a program kódja le sem fordítható. Ez erős szabályt és erős megkötést jelent.

## 14.2. Konstruktor azonosítási lánc

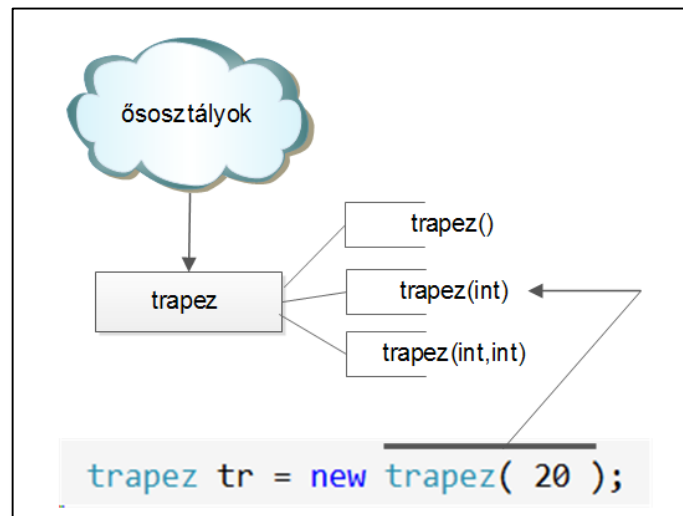
Lássuk a hibajelenség valódi okait. A példányosításkor kiválasztjuk a megfelelő osztályt, a 'new' segítségével memóriát foglalunk, majd meghívjuk a konstruktort. Melyik konstruktort hívjuk meg? Az adott osztálynak több konstruktora is lehet! Nézzük meg a trapéz osztály esetén. Feltételezzük, hogy három konstruktora is van:



A konkrét példányosítás során választunk konkrét konstruktort:

```
trapez tr = new trapez( 20 );
```

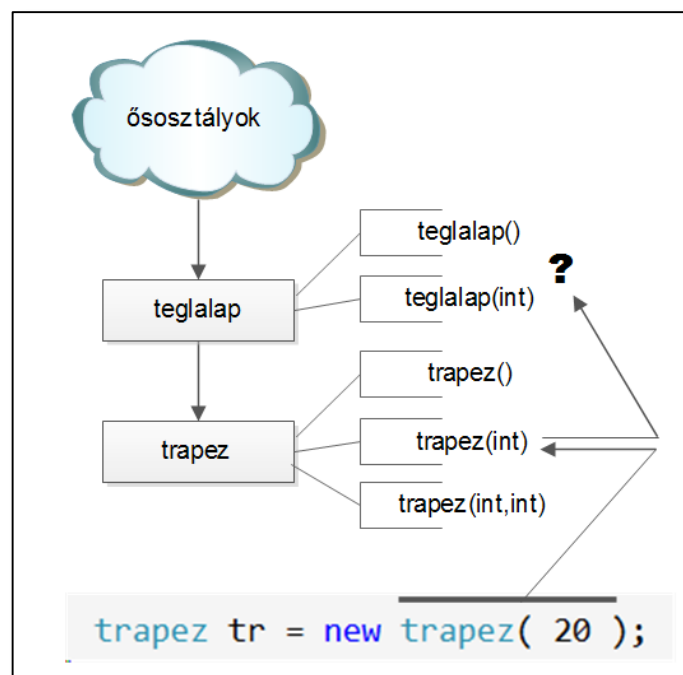
Ez esetben azt a konstruktort választottuk ki (az aktuális paraméterlista szerint) amelyik éppen egy 'int' típusú értéket vár paraméterként:



A kiválasztás tehát a példányosított osztály konstruktorának azonosításával kezdődik. Ha nem sikerül (nincs ilyen paraméterezésű konstruktor, vagy annak védelmi szintje szerint nem elérhető a példányosítás helyén) akkor a példányosításnál jelentkezik a szintaktikai hiba:

```
trapez tr = new trapez( 20, "alma" );
```

Miután a fordító azonosította a példányosítás során használandó konstruktort, szeretne kiválasztani egy konstruktort az őszosztályból is:





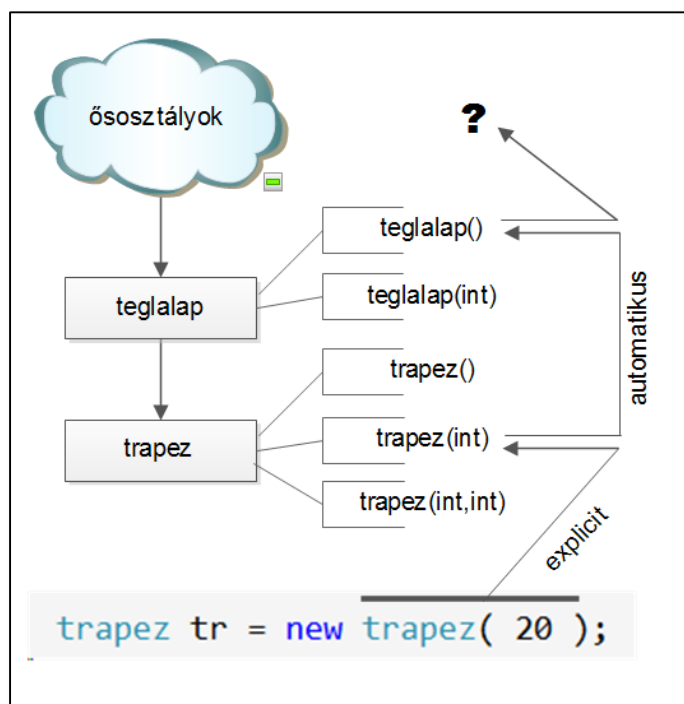
Az ősosztálybeli konstruktor kiválasztásának szabályai:

1. csak a public vagy protected konstruktorok jöhetnek szóba,
2. ha van 'this()' saját másik konstruktor hívása, akkor azt hívjuk meg,
3. ha van 'base()' ősosztálybeli konstruktor hívás, akkor azt hívjuk meg,
4. ha van paraméter nélküli, akkor az kerül meghívásra,
5. szintaktikai hiba (ha az előző 4 pont alapján nem kerül meghívásra ősosztálybeli konstruktor, akkor szintaktikai hibát kapunk a fordítótól).

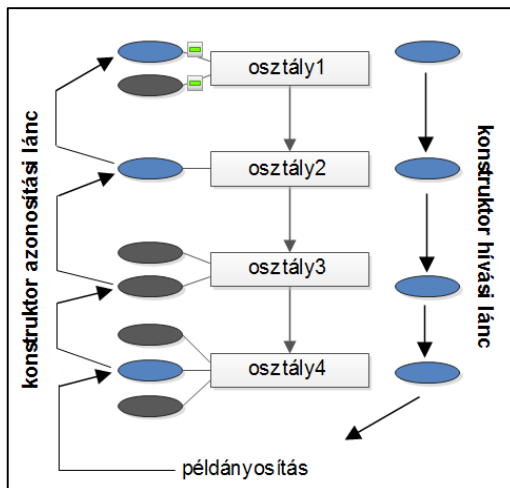
Nézzük meg egyelőre a 4-es szabályt. Ha az ősosztályban van paraméter nélküli elérhető (elérhető védelmi szintű) konstruktor, akkor az kerül kiválasztásra. Mikor van ilyen konstruktor? Ha

- írunk paraméter nélküli public vagy protected konstruktort,
- egyáltalán nem írunk konstruktort, s ekkor a rendszer generál egy public védelmi szintű, üres paraméterezésű és törzsű konstruktort.

Tehát ha egy ősosztályba írunk paraméter nélküli konstruktort, vagy egyáltalán nem írunk konstruktort, akkor a gyerekosztálybeli konstruktorokhoz a láncban automatikusan hozzáfűződik a konstruktor az ősosztályból.



Amennyiben minden őssztályban van paraméter nélküli konstruktor, a teljes folyamat észre sem vesszük. A kiválasztás az őssztályok felé felfelé haladva automatikusan megtörténhet. Amikor a teljes hívási lánc azonosításra kerül, akkor a példányosítás a legfelső konstruktor lefutásával kezdve az azonosítási lánc elemein lefelé haladva történik meg:



### 14.3. Saját konstruktor hívása – 'this'

Amikor egy osztálynak több konstruktora is van, előforduló probléma, hogy egyik konstruktor működése visszavezethető egy már elkészített másik (saját) konstruktor hívására. Főleg akkor fordul elő, ha az egyik konstruktorunk paraméterezése a másik paraméterezés egyfajta egyszerűsítése, vagyis néhány paraméter hiányzik:

```
class negyzet
{
    protected double a_oldal;
    public negyzet()
    {
        this.a_oldal = 0;
    }

    public negyzet(double a_oldal)
    {
        this.a_oldal = a_oldal;
    }
}
```

Az első konstruktor 'new negyzet()' valójában a második konstruktor hívásával 'new negyzet(0)' kiváltható lenne. Hogyan tudnánk az összefüggést a kódban is feltüntetni?

```
public negyzet()
{
    this.negyzet(0);
    negyzet(0);
}
```

Ez logikusnak tűnt, hiszen a konstruktorok végül is egyfajta metódusok, hívásuk azonban úgy tűnik nem hagyományos szintaktikával történik. Ennek több oka is van, pl. a konstruktoroknak nincs valódi visszatérési értékük (még void sem), a fenti szintaktikával pedig void-os visszatérési értékű metódusok hívása történik.

Ez most egy olyan történés, mely nem logikus, tehát nem megérteni, hanem megjegyezni szükséges: a saját konstruktor hívásának speciális szintaktikája van.

```
class negyzet
{
    protected double a_oldal;
    public negyzet() :this( 0 )
    {
    }

    public negyzet(double a_oldal)
    {
        this.a_oldal = a_oldal;
    }
}
```

A saját másik konstruktorunkat a 'this' kulcsszó segítségével hívhatjuk meg. A 'this' után fel kell tüntetni a függvényhívó operátort, a két zárójelpárt. A zárójelpárba az aktuális paramétereket kell írni, amely alapján eldönthető melyik másik saját konstruktort akarjuk hívni. A hívás helye is speciális: a konstruktor aktuális paraméterezése mögé, de még a törzse előtt kell feltüntetni a hívást:

```
class kacsa
{
    public kacsa(double sulya, string neve)
        :this(sulya,neve,true)
    {
        // ...
    }

    public kacsa(double sulya, string neve, bool eletben)
    {
        // ...
    }
}
```

Akkor is lehetséges, ha a másik (meghívandó) konstruktor private (amelyet egyébként nem tudnánk a külső – példányosítást végző – kódból közvetlenül meghívni)

```
enum nemek { ferfi, no }
class diak
{
    public diak(int eletkor, string neptun_kod)
        :this(eletkor,neptun_kod, nemek.no )
    {
    }

    private diak(int eletkor, string neptun_kod, nemek neme)
    {
    }
}
```

Lehetséges a továbbhívás, vagyis a 'this'-szel hívott konstruktor tovább hív egy (harmadik) saját konstruktort:

```
class teglalap : negyzet
{
    public teglalap() : this(0)
    {
    }

    public teglalap(double a_oldal)
        : this(a_oldal, a_oldal)
    {
    }

    public teglalap(double a_oldal, double b_oldal)
    {
    }
}
```

Természetesen tilos kört kialakítani a hívásokból, vagyis visszahívni egy olyan konstruktort, amelyből kiindulva eljuthatunk erre a pontra. Sajnos, a VS nem érzékeli szintaktikai hibának (mivel külön-külön a konstruktorok szintaktikailag helyesek):

```
class teglalap : negyzet
{
    public teglalap() : this(0)
    {
    }

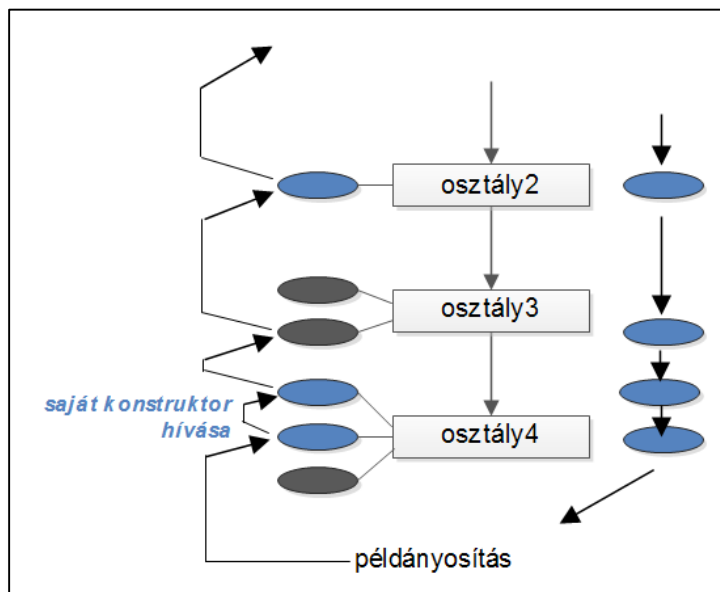
    public teglalap(double a_oldal)
        : this(a_oldal, a_oldal)
    {
    }

    public teglalap(double a_oldal, double b_oldal)
        : this(a_oldal)
    {
    }
}
```

Ha ilyet készítünk (hívási kör kialakítása), akkor a példányosítás elvileg végtelen ideig fog futni, mivel a példányosításkor a konstruktorok végtelen rekurzióban egymást hívják, és sosem készül el a példányunk. Gyakorlatilag nincs végtelen rekurzió, mivel a számítógép memóriája véges. Ezért az eset futási hibához vezet.

#### 14.4. Saját konstruktor hívása és az azonosítási lánc

Amikor az azonosítási láncot keresi a fordító, akkor figyelembe veszi a 'this' saját konstruktor hívást is. Ha ilyennel találkozik, akkor beleveszi a hívási láncba a meghívott másik saját konstruktort:



Mivel hívási kör kialakítása hibás, ezért lennie kell olyan saját konstruktorunknak, amelyből nem hívunk saját másik konstruktort. Ezen konstruktorig eljutva a keresés feljebb jut az őszosztály szintjére.

### 14.5. Ős konstruktor hívása explicit módon: 'base'

Ha az őszosztályban van elérhető paraméter nélküli konstruktor, akkor annak kiválasztása automatikus. Azonban ha nincs, akkor explicit módon (manuálisan) kell azt megoldani. Erre nem használható a 'this', hiszen az saját osztálybeli (másik) konstruktort hív meg. Helyette a 'base' kulcsszót alkalmazzuk. A továbbiakban a 'base' használata az őszosztálybeli konstruktor meghívására ugyanazon szintaktikával történik, mintha a 'this'-t használnánk:

```
class teglalap: negyzet
{
    public teglalap(double a_oldal) ←
    {
        // ...
    }
}

class trapez : teglalap
{
    public trapez(double a_oldal, double b_oldal)
    {
        :base(a_oldal)
        // ...
    }
}
```

Mikor szükséges használni a base-t? Ha az őszosztályban van elérhető paraméter nélküli konstruktor, akkor nem szükséges, annak kiválasztása automatikus. Ha azonban, csak paraméteres konstruktorok érhetőek el, akkor **a base használata kötelező!** A paraméterezést a fordító nem fogja tudni feltölteni helyettünk!

Valójában azon konstruktor mögött, ahova nem írunk semmit, a `'base()'` szerepel, vagyis meghívja az ősosztály paraméter nélküli konstruktorát:

```

class teglalap: negyzet
{
    public teglalap(double a_oldal)
    {
        // ...
    }
}

class teglalap: negyzet
{
    public teglalap(double a_oldal) : base()
    {
        // ...
    }
}

```

Ennek megfelelően amikor nem írunk konstruktort (pl. a négyzet osztályhoz), akkor a fordítás során generált konstruktor valójában így néz ki:

```

class teglalap : negyzet
{
    public teglalap():base()
    {
    }
}

```

Ha egy osztály a gyerekosztályai felé csak paraméteres konstruktort ad, akkor a gyerekosztályokban konstruktor írása kötelező, mivel az ősosztálybeli konstruktorokat ekkor a **base** segítségével manuálisan kell kiválasztani és felparaméterezni.

Következők tehát a lehetőségeink:

- Nem írunk konstruktort az osztályunkba. Ez esetben generálódik hozzá egy publikus, paraméter nélküli konstruktor, mely automatikusan meghívja az ősosztálybeli paraméter nélküli konstruktort, ami működik, ha van az ősosztályban ilyen elérhető konstruktor. Ha nem jelöltük meg, hogy explicit módon ki az ősosztályunk, akkor az `Object` lesz az, és neki van ilyen konstruktora. Ezért az „első” szintű objektumosztályok esetén konstruktorok írására gyakorlatilag nincs szükség.
- Írunk saját konstruktort az osztályhoz. Ezen konstruktor mellett:
  - a. Nem szerepeltetünk semmit, ami megfelel a `'base()'` szerepeltetésének.
  - b. A `'base( ... )'` segítségével őskonstruktort hívunk explicit módon. Az ősosztályban lennie kell adott paraméterezésű konstruktornak `protected` vagy `public` elérési szinttel.
  - c. A `'this( ... )'` segítségével saját másik konstruktort hívunk. Lehet akár `private` védelmi szinttű is. Ügyeljünk arra, hogy ne alakítsunk ki rekurzív körbehívást a saját konstruktorok között, mert azt fordításkor nem jelzi ki a fordítóprogram, de futás közben már hibát fogunk kapni.

## 14.6. Osztálysztintű konstruktorok

A témához szorosan nem kapcsolódik (konstruktorok és az öröklődés), de a konstruktorok témájának lezárásaképp meg kell említeni, hogy léteznek osztálysztintű konstruktorok is.

Az eddigi konstruktorokat példányszintű konstruktoroknak nevezzük, mivel futásuk a példányosításhoz kötődik. Szerepük a példány kezdő állapotba állítása, a mezők kezdőértékeinek beállítása. Az osztálysztintű konstruktor feladata is teljesen hasonló, az osztálysztintű mezők kezdőértékét állítja be. Nem véletlen a többszám elhagyása – osztálysztintű konstruktorból nem lehet több, legfeljebb egyetlen egy.

Az osztálysztintű konstruktor neve szintén egyezik az osztály nevével, de három megkövetés van, ami a példányszintű esetén nincs:

- static módosítóval rendelkezik,
- nem lehet paramétere,
- nem lehet kiírni neki védelmi szintet (private).

A 'static' módosító szerepe egyértelmű: mutatja, hogy osztálysztintű, és nem példányszintű a konstruktor. A paraméter kötelező hiánya arra vezethető vissza, hogy (mint látni fogjuk) automatikusan kerül meghívásra, és a rendszer nem fog paramétereket kitalálni és átadni eközben. Mivel nem lehet paramétere, így nem is készíthető ilyen konstruktorból több (mindegyiknek ugyanaz lenne a neve, és a paramétere, ami nem megengedhető az overloading mellett sem).

Két kérdés van amit tisztázni kell ez ügyben:

- Mikor kell osztálysztintű konstruktort írni?
- Mikor fog az lefutni?

Az osztálysztintű konstruktorok elsődleges feladata az osztálysztintű mezők kezdőértékének beállítása. Egyszerűbb esetekben ez a mezők mellé írt kezdőértékadással kezelhető, vagy teljesen megfelel a mező alapértelmezett (típusának megfelelő) kezdőértéke. Az esetek legnagyobb részében így is van:

```
class Beallitasok
{
    public static int numOfClient = 20;
    public static string serverVersion = "0.9a";
    public static bool debugMode; // = false alapértelmezett
```

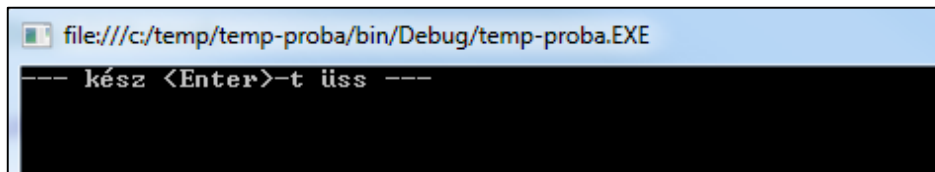
A gond akkor kezdődik, ha a mező kezdőértékét nem lehet ilyen egyszerűen, egysoros kifejezéssel beállítani. Mielőtt keresünk ilyen esetet, tegyünk meg egy kísérletet: készítsünk egy mezőt és egy osztálysztintű konstruktort a 'Beallitasok' osztályba, ami kiír egy egysoros üzenetet:

```
class Beallitasok
{
    public static int counter = 0;
    static Beallitasok()
    {
        Console.WriteLine("Beallitasok osztalyszintu konstruktor");
    }
}
```

Készítsünk el ezen felül egy szinte semmi hasznosat nem végző főprogramot:

```
public static void Main()
{
    Console.WriteLine("--- kész <Enter>-t üss ---");
    Console.ReadLine();
}
```

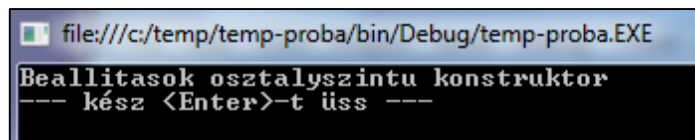
Ha lefuttatjuk a programot, azt látjuk, hogy az osztályszintű konstruktor valószínűleg nem futott le, mivel az ő kiírását nem látjuk a képernyőn:



Annyit módosítsunk a főprogramon, hogy a 'counter' mező értékét növeljük meg 1-gyel:

```
public static void Main()
{
    Beallitasok.counter++;
    Console.WriteLine("--- kész <Enter>-t üss ---");
    Console.ReadLine();
}
```

Az újabb futás esetén látható amit kerestünk:



Lefutott tehát az osztályszintű konstruktor, holott a kódból explicit módon nem is hívtuk meg. (Nem is tudnánk, mivel nincs védelmi szintje feltüntetve, ezért az alapértelmezett védelmi szint, a private lép érvénybe. Így a főprogramból nem is tudnánk rá hivatkozni, nem tudjuk meghívni.)

Az osztályszintű konstruktor hívása automatikusan történik, de pontos időpontját nem tudjuk megadni. A rendszer azt garantálja, hogy mielőtt az első hivatkozás történne az osztályra a kódban (osztályszintű mező, metódus, példányosítás stb.), előtte le fog futni.

Az első kísérletkor azért nem futott le, mert a programban nem is hivatkoztunk az adott osztályra. Második kísérletkor mivel hivatkoztunk a counter mezőre (növeltük), így a konstruktor automatikusan lefutott még a 'counter++' előtt.



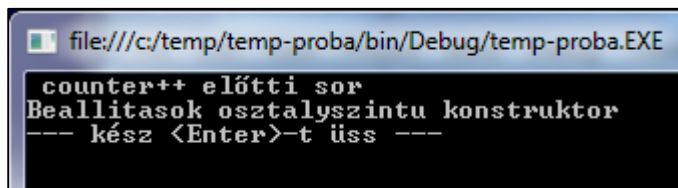
A rendszer igyekszik optimalizálni az osztályszintű konstruktorok futtatását: a hívást eltolni a program lehetőleg legkésőbbi időpontjára (ha lehet teljesen ki is hagyni ha nincs rá szükség).

Nagyon rossz lenne az, ha az osztályszintű konstruktorok mindegyike már a program indulásának első pillanataiban lefutna. Persze megoldható. Ekkor azonban könnyen előállna az az eset, hogy a program indulásakor sok idő telne el mielőtt a legelső, általunk a Main()-be írt utasítás végre elindulna.

Tegyük még egy próbát:

```
public static void Main()
{
    Console.WriteLine(" counter++ előtti sor");
    Beallitasok.counter++;
    Console.WriteLine("--- kész <Enter>-t üss ---");
    Console.ReadLine();
}
```

Láthatjuk, hogy az osztályszintű konstruktor lefutása nem a program indulásakor azonnal, hanem később történik meg:



Így hát a „mikor” kérdést alaposan körbejártuk:

- nem lehet tudni pontosan mikor fut le,
- egy garantált csak: mielőtt az osztályra bármi módon hivatkoznánk, előtte le fog futni,
- lehet, hogy le sem fut (ha nem hivatkozunk az osztályra).

Már csak az a kérdés: miért kellene ilyet írunk? Nos, van amikor a mezők kezdőértéke egyszerű kezdőértékadással nem kiszámolható. Például mert a mezők értékét fájlból vagy adatbázisból szeretnénk feltölteni, esetleg több más mező értékétől csak algoritmussal leírható módon függ.

### 14.7. Private konstruktorok

Amennyiben nem szeretnénk, hogy gyerekosztályt készíthessenek az általunk fejlesztett osztályból, két lehetőségünk is van:

- private konstruktor készítése,
- sealed kulcsszó használata.

Ha az osztályunkba nem írunk konstruktort, úgy abba egy public védelmi szintű konstruktor kerül be, melynek nincs paramétere sem. Emellett gyerekosztály készíthető, hisz még a 'base' explicit használatára sincs szükség, a gyerekosztály konstruktora implicit módon ki tudja választani az őse osztályából ezt a konstruktort. Így ha nem írunk konstruktort, akkor semmi sem akadályozza meg a többi programozót, hogy gyerekosztályt készítsen a mi osztályunkból.

Nyilván ugyanez a helyzet akkor, ha mi magunk készítünk konstruktort, de azok közül legalább egy public vagy protected van. Ez esetben a gyerekosztály szintén elkészíthető, hisz van meghívható ősosztálybeli konstruktor.

Ha azonban készítünk konstruktort (konstruktorokat), de azok mindegyik private, akkor a gyerekosztály készítője bajban van – hisz nem képes azokat még a 'base' segítségével sem meghívni. Ekkor az osztályunkból gyerekosztály nem készíthető!

```
class teglalap: negyzet
{
    private teglalap() : base()
    {
    }
}

class trapez : teglalap
{
    public trapez()
    {
    }
}
```

'ektf.teglalap.teglalap()' is inaccessible due to its protection level

Jegyezzük meg: a private konstruktorok mellett gyerekosztály nem készíthető. (De, hogy példány készíthető-e az osztályunkból, az már egy másik kérdés.)

## 14.8. A 'sealed' kulcsszó

A másik módszer, amivel megakadályozhatjuk gyerekosztály készítését az osztályból, hogy a 'sealed' (lepecsételt) jelzővel látjuk el magát az osztályt.

A „cannot derive from sealed type teglalap” fordítása: „nem származtatható a teglalap nevű lepecsételt típusból”:

```
sealed class teglalap: negyzet
{
    protected int a;
    public teglalap(int a)
    {
        this.a = a;
    }
}

class trapez : teglalap
{
    public trapez()
    {
    }
}
```

'ektf.trapez': cannot derive from sealed type 'ektf.teglalap'

A 'sealed' nemcsak osztályokra, hanem metódusokra, property-kre is alkalmazható, amennyiben azok késői kötésűek. Ekkor a gyerekosztályok már nem override-olhatják a metódust (property-t). A „cannot override inherited member kerulet() because is it sealed” fordítása: „nem lehet override segítségével felülírni az örökölt kerulet() metódust, mert lepecsételt”:

```
class teglalap: negyzet
{
    public override sealed double kerulet()
    {
        return base.kerulet();
    }
}

class trapez : teglalap
{
    public override double kerulet()
    {
        return base.kerulet();
    }
}
```

'ektf.trapez.kerulet()': cannot override inherited member 'ektf.teglalap.kerulet()' because it is sealed

Természetesen nem szabad a 'sealed' jelzőt a metódus első bevezetésekor (együtt a virtual kulcsszóval) használni. Értelmetlen jelölni 'virtual'-al (gyerekosztályok valószínűleg felül fogják definiálni; működjön rá a késői kötés, hogy mindig a legmegfelelőbb változat hívódjon meg) és letiltani azonnal a felülírhatóságot a 'sealed'-del:

```
class negyzet
{
    public virtual sealed double kerulet()
    {
        return 0;
    }
}
```

'ektf.negyzet.kerulet()': cannot be sealed because it is not an override

## 14.9. Az Object Factory

Térjünk vissza arra az esetre, amikor az osztályunkban csak `private` konstruktor van. Ekkor nemcsak gyerekosztályt nem készíthetünk az adott osztályból, de példányosítani sem lesz könnyű. Ugyanis a példányosítás helyszíne is általában kívül esik az adott osztályon (pl. a `Main` függvényben), ahol a `private` védelmi szintű osztályelemek nem érhetőek el, nem hívhatóak meg.

Ez ugyan nehezíti, de nem teszi teljesen lehetetlenné a példányosítást. A példányosítás folyamatát, a `'new'` memórafoglalást és a konstruktor hívását be kell vinni az osztály belsejébe, valamely metódusba, ami még nem probléma:

```
class negyzet
{
    private negyzet()
    {
    }

    public void keszits_peldanyt()
    {
        // itt belül hívható a private konstruktor
        negyzet n = new negyzet();
    }
}
```

Azonban ha metódusunk szintén példány szintű (static nélküli), akkor nem jutottunk előre, hiszen a példányszintű metódus meghívásához példányra lenne szükségünk. A megoldás, hogy a metódus legyen osztályszintű, és készítsen példányokat (az elkészített példány referenciáját, memóriacímét adja meg visszatérési értéként):

```
class negyzet
{
    private negyzet()
    {
    }

    public static negyzet Create()
    {
        // itt belül hívható a private konstruktor
        negyzet n = new negyzet();
        // a kész példány visszaadható
        return n;
    }
}
```

Ekkor a példányosítás más szintaktikájú, hisz nem tudjuk a szokásos `'new + konstruktor'` módon készíteni a példányt, de az osztályszintű metódus meghívása pótolja ezt:

```
static void Main(string[] args)
{
    negyzet t = negyzet.Create();
}
```

A szóban forgó osztályszerű metódus (akinek feladata nem más, mint a példány elkészítése a külvilág számára, és annak visszaadása) neve a szakzsargonban **Object Factory** (objektumgyár).

Az Object Factory metódus természetesen paramétereket is vehet át, és adhat át a konstruktoroknak:

```
class negyzet
{
    protected double a_oldal;
    private negyzet(double a_oldal)
    {
        this.a_oldal = a_oldal;
    }

    public static negyzet Create(double d)
    {
        // itt belül hívható
        negyzet n = new negyzet( d );
        // a kész példány visszaadható
        return n;
    }
}
```

A példányosítás során át kell adni a kért paramétereket:

```
static void Main(string[] args)
{
    negyzet t = negyzet.Create(12.5);
}
```

Sok okból készíthetünk Object Factory jellegű metódusokat, néha akkor is, amikor van elérhető konstruktor is. Ebben a példánkban megmutatjuk, hogyan lehet Object Factory segítségével megvalósítani azt az esetet, amikor a valamely osztályból csak egyetlen példány készülhet, és mindenkinek ugyanezt a példányt kell használni<sup>20</sup>. Legyen az osztály egy naplófájl (log) író osztály. A program eseményeit fájlba írja ki. Hogy mindenki ugyanabba a fájlba írja az eseményeit (és az állományt csak egyszer lehet megnyitni), az alábbiak szerint valósítható meg a feladat:

<sup>20</sup> Ezt *singleton* programozási mintának nevezik.

```

class fileLog
{
    public static fileLog peldany = null;
    static string[] dows = new string[] { "7-vasarnap",
        "1-hetfo", "2-kedd", "3-szerda", "4-csutortok",
        "5-pentek", "6-szombat" };
    // .....
    public static fileLog Create()
    {
        // ha már kész a példány, akkor azt returnolni
        if (peldany != null) return peldany;
        //
        var now = DateTime.Now;
        var basePath = Path.GetDirectoryName(
            Assembly.GetExecutingAssembly().Location );
        var date = now.ToString("yyyy-MM-dd");
        var dow =
            (int)CultureInfo.InvariantCulture.Calendar.GetDayOfWeek(now);
        var dowstr = dows[dow];
        var fn = String.Format("{0}-{1}.log", date, dowstr);
        fn = Path.Combine(basePath, fn);
        peldany = new fileLog( fn );
        peldany.w.WriteLine(":: --- {0} {1} LOG --- :: ", date, dowstr);
        peldany.w.WriteLine(":: log file opened {0} {1}",
            now.ToString("yyyy-MM-dd HH:mm:ss.ff"),
            dowstr.Substring(2));
        peldany.w.Flush();
        return peldany;
    }
    // .....
    protected StreamWriter w = null;
    public fileLog(string filenev)
    {
        this.w = new StreamWriter(filenev, true, Encoding.UTF8);
    }
    // .....
    public void writeln(string esemeny, params object[] parms)
    {
        this.w.WriteLine(esemeny, parms);
    }
}

```

## 15. Indexelő

---

Egy osztály fejlesztése során gyakran választhatunk a metódus vagy a property írása között. Utóbbit a szebb szintaktika miatt választjuk, hiszen a `get` metódus két üres zárójelpárt tartalmaz (paraméter nélküli metódus), és a `set` metódus egy paraméteres változata helyett is sokkal olvashatóbb az értékadásos forma.

Speciális eset, amikor valamely mezőnk nem egy, hanem több érték tárolására is képes (a mező lehet egy vektor vagy egy lista). A mezőket általában nem érdemes megosztani a külvilággal (`public`), mert referencia típusú mezők. Ha külvilág hozzáfér, nagyon el tudja rontani az értékét (akár null értékkel is felülírhatja).

Hogyan tudjuk egy ilyen mező kezelését a külvilág felé átadni? Legyen a példánk egy horgász ember, aki a zsákjában csak adott mennyiségű (súly kérdés) halat tárolhat. A halak külön objektumosztály, a mezőjük tartalmazza a súlyt. A horgász zsákját egy lista reprezentálja, amelybe a halakat elhelyezhetjük. Garantálni kell, hogy ne rakhasson bele több halat a külvilág, mint amennyi a zsák teherbírása.

```
class hal
{
    protected double _sulya;
    public double sulya
    {
        get { return _sulya; }
        set
        {
            if (value < 0 || value > 50.0)
                throw new ArgumentException("nem megfelelő hal súly");
            else _sulya = value;
        }
    }
    public hal(double sulya)
    {
        this.sulya = sulya;
    }
}
```

A hal osztály után lássuk a horgász osztályt:

```
class horgasz
{
    protected List<hal> zsak = new List<hal>();
    protected double zsak_sulya = 0;
    //
    public void zsak_berak(hal h)
    {
        if (h == null)
            throw new NullReferenceException("a hal nem lehet null");
        if (h.sulya + zsak_sulya > 20.0)
            throw new ArgumentException("a hal mar nem fer el a zsakban");
        zsak.Add(h);
        zsak_sulya = zsak_sulya + h.sulya;
    }
    //
    public hal zsak_eleme(int index)
    {
        if (index < 0 || index >= zsak.Count)
            throw new IndexOutOfRangeException();
        return zsak[index];
    }
}
```

A 'zsak\_berak()' függvény segítségével lehet halat rakni a zsákba. Esetünkben a zsák teherbírása 20 kilogramm. A 'zsak\_sulya' mezőben tartjuk nyilván, hogy mennyi a zsákunk aktuális súlya. Mikor halat adunk a zsákhoz, növeljük a mező értékét is. A zsákban lévő halat a 'zsak\_eleme()' függvény segítségével kérdezhethetjük le, megadva hányadik elemre vagyunk kíváncsiak.

Használata pl. az alábbi módon történhet:

```
horgasz lajos = new horgasz();
hal h1 = new hal(2.5);
lajos.zsak_berak(h1);
hal h2 = new hal(4.2);
lajos.zsak_berak(h2);
hal h = lajos.zsak_eleme(0);
```

Ez a kód nem elegáns, nem szép. Az alábbiakban megtanuljuk, hogyan kell indexelőt írni, és annak segítségével mennyivel szebben megírhatjuk a fenti kódot.

Az indexelő egy olyan property, melynek paramétere is lehet., hogy ne keveredjen a paraméterezése a metódusok szokásos paraméterezésével, így az indexelő paraméterezését szögletes zárójelbe rakjuk a szokásos gömbölyű zárójelpárok helyett. Emellett az indexelő neve sajnos kötött, kötelezően 'this'. Egy példa:

```
class horgasz
{
    protected List<hal> zsak = new List<hal>();
    protected double zsak_sulya = 0;
    //
    public hal this[int index]
    {
        get
        {
            if (index < 0 || index >= zsak.Count)
                throw new IndexOutOfRangeException();
            return zsak[index];
        }
    }
}
```



Esetünkben a ember zsákjának kiolvasását egyszerűen az indexének megadásával lehet kezdeményezni. A 'this' nevet nem is kell használni, automatikus:

```
hal h = lajos[0];
// valójában
// h = lajos.this[0]
```

Amennyiben van lehetőség egy már korábban hozzáadott hal cseréjére (indexével azonosítva melyiket akarjuk cserélni) úgy indexelő nélkül az alábbi függvényt kellene kialakítani:

```
public void zsak_csere(int index, hal h)
{
    if (h == null)
        throw new NullReferenceException("a hal nem lehet null");
    if (index < 0 || index >= zsak.Count)
        throw new IndexOutOfRangeException();
    hal csere = zsak[index];
    if (h.sulya-csere.sulya + zsak_sulya > 20.0)
        throw new ArgumentException("nem fer tobb hal a zsakba");
    zsak[index] = h;
    zsak_sulya = zsak_sulya - csere.sulya + h.sulya;
}
```

Használata:

```
hal h3 = new hal(3.5);
lajos.zsak_csere(0, h3);
```

Ezzel szemben indexelő esetén egy set szekciót kell kialakítani:

```
public hal this[int index]
{
    set
    {
        if (value == null)
            throw new NullReferenceException("a hal nem lehet null");
        if (index < 0 || index >= zsak.Count)
            throw new IndexOutOfRangeException();
        hal csere = zsak[index];
        if (value.sulya - csere.sulya + zsak_sulya > 20.0)
            throw new ArgumentException("nem fer tobb hal a zsakba ");
        zsak[index] = value;
        zsak_sulya = zsak_sulya - csere.sulya + value.sulya;
    }
}
```

Használata:

```
hal h3 = new hal(3.5);
lajos[0] = h3;
```

Vagy egyszerűbben (de ez nem az indexelő érdeme miatt):

```
lajos[0] = new hal(3.5);
```

A jelen esetben írt indexelő vázlata tehát:

```
public hal this[int index]
{
    set { /* ... tartalom ... */ }
    get { /* ... tartalom ... */ }
}
```

Ez az indexelő 'int' típusú indexeket használ, és ezen keresztül 'hal' típusú elemeket érhetünk el könnyedén. Tegyük fel, hogy szeretnénk még egy ilyen listát tárolni a horgászban, pl. időpontokat (dátum), amely napokon voltunk pecázni. Mivel sűrűn járunk, ez is egy sok elemű lista lenne DateTime típusú elemekből:

```
public DateTime this[int index]
{
    set { /* ... tartalom ... */ }
    get { /* ... tartalom ... */ }
}
```

Problémás lenne a használat során, hiszen mindkét esetben a listaelemekre az egész szám típusú sorszámukkal kellene hivatkozni (olvasáskor és íráskor is):

```
hal h      = lajos[0];
DateTime m = lajos[0];
//
lajos[0] = new hal(3.5);
lajos[0] = DateTime.Now;
```

A szabály az alábbi: egy osztálynak **lehet több indexelője** is, de az indexelő paraméterében különböznie kell. Az tehát kevés, ha az indexelő típusa (hal és DateTime) különbözik, a paraméterezés a döntő.

Ezenkívül elképzelhető olyan indexelő is, amely nem egy paraméteres, hanem kettő vagy több:

```
public DateTime this[int a, int b]
{
    set { /* ... tartalom ... */ }
    get { /* ... tartalom ... */ }
}
```

Ennek használata (valamilyen 'p' példány esetén pl.):

```
p[0, 1] = DateTime.Now;
DateTime n = p[2, 3];
```

Ilyen indexelővel rendelkeznek a vektorok:

```
double[] t = new double[20];
t[2] = 32.4;
double x = t[1];
```

Ilyen indexelővel rendelkeznek a listák:

```
List<bool> l = new List<bool>();
// ... lista feltöltése
l[2] = true;
bool x = l[1];
```

Valamint van egy speciális „lista”, amit **Dictionary**-nak nevezünk. Míg egy hagyományos listában (amely mondjuk diákokat tartalmaz) az elemek indexelése minden esetben egész számok, a Dictionary esetén megválasztható az index típusa, akár string is lehet. Jelen esetben értelmezzük úgy, hogy a diák NEPTUN kódja lesz az index (lényeg, hogy egyedi azonosító legyen). Egy Dictionary példány definiálásakor meg kell adni először az index típusát (string), majd, hogy milyen típusú elemeket kívánunk tárolni a listában (diak):

```
Dictionary<string, diak> l = new Dictionary<string, diak>();
//
l["BZQQC3"] = new diak("lajoska");
l["N67ERO"] = new diak("petike");
//
l.Add("SEB8IR", new diak("gizike"));
//
diak d = l["BZQQC3"];
```

Új elemet kétféleképpen helyezhetünk a kezdetben üres Dictionary-hoz:

- vagy az Add() metódusa segítségével, melynek paramétere az elem azonosítója (kulcsa) és maga az elhelyezendő elem,
- vagy egyszerűen az indexelő segítségével az adott azonosítóhoz hozzárendeljük, tároljuk az elhelyezendő elemet.

Az Add() hibát fog dobni (kivételt) ha olyan azonosítójú elem már létezik a gyűjteményben, hiszen az azonosítónak egyedinek kell lennie. Az indexelő segítségével biztonságosabb a hozzáadás olyan értelemben, hogy ha az adott azonosítóval már létezik elem, akkor azt egyszerűen lecseréli (felülírja).

## 16. Névterek

---

Az OOP környezetben fontos szempont a „csoportosítás”. Az egységbezárás elv alkalmazása arra sarkall minket, hogy összeszedjük: milyen mezők és műveletek szükségesek a programozási feladat megvalósítására. Ugyanazon feladatkörben több osztály is készülhet, hogy később kiválaszthassuk közülök a számunkra legalkalmasabbat. Ilyen célok miatt készülnek a gyerekosztályok is, amelyek jellemzően valamilyen szempont mentén specializálódnak.

Ha jól dolgozunk, hamarosan maroknyi osztály, enum lesz a birtokunkban. Hogyan tudjuk őket csoportosítani?

A névterek (*namespace*) segítségével. A névterek egy magasabb szintű csoportosítást tesznek lehetővé, elsősorban egy témakörbe eső osztályokat csoportosíthatjuk vele:

```
namespace elolenyek
{
    class allat { /* ... */ }
    class haziallat : allat { /* ... */ }
    class kutya : haziallat { /* ... */ }
    class cica : haziallat { /* ... */ }
    class sziami: cica { /* ... */ }
    /* stb */
}
```

Minden névtérnek van neve, melyet a 'namespace' után kell megadni. Ez a név azonosító, tehát az azonosító névképzési szabályai alkalmazandók (betű, számjegy, aláhúzás, nem kezdődhet számjeggyel stb.). A névtér neve az osztály nevét kiegészíti, a fenti példában a 'cica' osztály teljes neve 'elolenyek.cica'. Ebben a szintaxisban is használható, akár példányosításkor is:

```
static void Main(string[] args)
{
    elolenyek.cica m = new elolenyek.cica();
}
```

Ezt a formát, amikor az osztály neve mellett megadjuk az őt tartalmazó névtér nevét is **minősített névnek** hívjuk. A szabály szerint teljesen egyedinek kell lennie, tehát egy névtéren belül két egyforma nevű osztály nem létezhet.

Hogy ne kelljen minden egyes osztály nevére való hivatkozáskor megadni a névtér nevét is, a 'using' kulcsszót használhatjuk. Általában a forráskód elején szoktuk kiadni. Használata mindössze annyit jelent, hogy a 'using' után megadott névterekben szereplő osztályok esetén nem kell kiírni a névtér nevét is:

```
using elolenyek;
class Program
{
    static void Main(string[] args)
    {
        cica m = new cica();
    }
}
```

---

Egyéb szerepe a `using` kulcsszónak nincs. Tehát ha kitörölnénk a forráskódunk elején szereplő `using`-okat, akkor annyi változás történne, hogy minden egyes osztályunk előtt meg kellene adni a névtér nevét is. Ez vonatkozna pl. a `Console` osztályra is, aki a `System` névtérben került megadásra:

```
// using System; <- törölve
static void Main(string[] args)
{
    System.Console.WriteLine("Adj meg egy számot 1..10 között");
    int a = int.Parse( System.Console.ReadLine() );
}
```

Emiatt figyelmeztetjük a Pascal/Delphi környezetből érkező programozókat, hogy a C# `using` kulcsszava csak alakjában hasonlít a Pascal-beli `'uses'` kulcsszóra. Pascal esetén ha a `uses`-t nem írtuk ki, az adott modulban, unitban definiált függvényekre egyáltalán nem lehetett a kódban hivatkozni. A `'uses'` valójában a linkernek szóló üzenet volt, ha nem adtuk meg a unit nevét, a linker nem szerkesztette hozzá a futtatható programhoz, és emiatt a hivatkozott függvény nem is szerepelt volna a végső programban. A C# `using`-ja azonban nem ilyen szerepkörű, nélküle is elérhetőek az adott osztályok, csak kényelmetlenebb szintaktikával. A Pascal `uses`-át valójában a *projekt management* ⇒ *add reference* menüpontja pótolja (rőla később lesz szó), a `dll`-ek kapcsán.

Fontos tudni, hogy a `using` után csak a névtér nevét adhatjuk meg, osztálynév már nem szerepelhet ott. Jó lenne a `Console` osztály metódusait így hívni:

```
using System.Console; // <- ez hiba egyébként !!

static void Main(string[] args)
{
    // a Console osztály Write metódusát hívnánk, de így nem lehet !!
    Write("Adj meg egy számot 1..10 között");
}
```

Következő hasznos tudnivaló, hogy a névterek is hierarchikus rendszerbe illeszthetőek, másképpen fogalmazva: egymásba ágyazhatóak. Ekkor a belső beágyozott névtér neve összeadódik a külső rétegbeli névvel:

```
namespace elolenyek
{
    namespace haziallatok
    {
        class allat { /* ... */ }
        class kutya : allat { /* ... */ }
        class cica : allat { /* ... */ }
    }
    namespace haziasitott
    {
        class allat { /* ... */ }
        class galamb : allat { /* ... */ }
    }
}
```

Ez esteben a 'kutya' osztály teljes neve:

```
elolenyek.haziallatok.kutya
```

Amíg a galamb minősített neve:

```
elolenyek.haziasitott.galamb
```

A névterek tetszőleges mélységben egymásba ágyazhatóak, így nincs akadálya akár az alábbi osztálynévnek sem:

```
System.Runtime.Serialization.Formatters.Soap.SoapFormatter
```

Ez úgyis előállhat, hogy a névtér nevének eleve összetett nevet választunk:

```
namespace elolenyek.haziallatok
{
    class allat { /* ... */ }
    class kutya : allat { /* ... */ }
    class cica : allat { /* ... */ }
}
namespace elolenyek.haziasitott
{
    class allat { /* ... */ }
    class galamb : allat { /* ... */ }
}
```

A névterek segítenek tehát az osztályok csoportosításában. Ha tudjuk milyen feladatra keresünk osztályt, akkor könnyen tudjuk azonosítani, hogy melyik névtérben kell keresnünk. Egy adott problémakörbe eső osztályok ugyanis jellemzően egy névtérbe (alnévtérbe) kerülnek. A Microsoft a .NET Framework tervezésekor következetesen a System névteret használta, ebbe készített alábontásokat, alnévtereket:

- **System.Collections:** összetett adatszerkezetek (tömbök, listák, verem, sor, ...),
- **System.Collections.Generic:** általános típusú összetett adatszerkezetek( List<int>, ...),
- **System.Drawing:** rajzoláshoz szükséges osztályok (ecset, toll, képformátumok, festővásznon, ...),
- **System.IO:** alkönyvtárak, állományok kezelése,
- **System.Reflection:** futás közbeni típusinformációk kezelése, attribútumok, dll-ek (assembly) kezelése,
- **System.Runtime.Remoting:** távoli metódushívások kezelése,
- **System.Threading:** többszálú programok írása, kritikus szakaszok kezelése, lock-olás,
- **System.Web:** web alapú kommunikációk kezelése.

A VS2010-ben, ha egy konzol típusú alkalmazást hozunk létre, akkor az alábbi névterek-re vonatkozó using-ok kerülnek be automatikusan:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Runtime.InteropServices;  
using System.Reflection;  
using System.IO;
```

Nagy részükre nincs is szükség, így a legtöbbet akár ki is törölhetjük. Érdekes megtartani a System-et (a Console miatt elsősorban), és az alatta lévő, a Generic végződéssel (a listák miatt). Esetleg az IO névtérrel is, ha a fájlrendszert, vagy magukat a fájlokat kezeljük (text fájlt olvasunk vagy írunk). A többiek eltávolítása a legtöbb esetben semmilyen gondot nem okoz.

```
using System;  
using System.Collections.Generic;  
using System.IO;
```

De vajon érdemes-e foglalkozni ilyen kérdésekkel egyáltalán, hogy a felesleges using-okat kitöröljük? Hogyan működik valójában a using?

Amikor a kódban hivatkozunk valamely osztályra, de nem adjuk meg a névtér nevét, akkor a fordítónak kell azt kitalálnia, hogyan gondolkodhat? Ellenőrzi az adott osztály nevét, hogy melyik névtérben szerepel ilyen nevű osztály (alapul véve a using listát). Ha egyikben sem szerepel, akkor baj van, ismeretlen az osztály, hibát kell jelezni. Ha pontosan egyben találja meg, akkor minden rendben van. Ha több (using-gal megnyitott) névtérben is szerepel ilyen nevű osztály, akkor is baj van, hiszen nem eldönthető melyikre gondoltunk.

Ilyen eset könnyen előállhat. Van egy osztálynév, 'Timer' a neve. Ilyen nevű osztály több névtérben is szerepel:

- System.Timers.Timer
- System.Threading.Timer
- System.Windows.Forms.Timer

Amennyiben pl. mindhárom névtérrel using-gal felnyitnánk<sup>21</sup> a programban, úgy a Timer osztályra hivatkozáskor probléma lenne:

<sup>21</sup> Konzol típusú programok esetén a Windows.Forms nem hozzáadható, csak ha magát a névtérbeli osztályokat tartalmazó dll-t előtte hozzáadjuk az „add reference” segítségével.

```

using System.Timers;
using System.Threading;
using System.Windows.Forms;

namespace Program
{
    class FoProgram
    {
        static void Main(string[] args)
        {
            Timer t = new Timer();
        }
    }
}

```

'Timer' is an ambiguous reference between 'System.Timers.Timer' and 'System.Threading.Timer'

Mint a képen is láthatjuk, a VS hibát jelez. A hiba oka, hogy nem egyértelmű számára melyik Timer osztályra is gondoltunk. Ez a hiba könnyen előállhat, ha sok using-ot használunk a program elején. Mi a teendő? Meg kell adni annak a Timernek a minősített nevét, amelyikre gondoltunk:

```

using System.Timers;
using System.Threading;
using System.Windows.Forms;

namespace Program
{
    class FoProgram
    {
        static void Main(string[] args)
        {
            System.Timers.Timer t = new System.Timers.Timer();
        }
    }
}

```

Ezzel a megoldással akkor is kezelni tudjuk a helyzetet, ha több, megnyitott névtér is tartalmazná az adott osztályt.

Ugyanakkor hátránynak tekinthető, hogy sok using a fordítási folyamatot lassítja. A fordítónak sok névtérbe kell belenézni, hogy azonosíthassa: a kódban szereplő osztályok pontosan melyik névtérbeli osztályok. Nyilván minél több névtérbe kell belenéznie, annál lassúbb a folyamat. (De valójában egy megfelelő gépen ez az a különbség, hogy a kód 1,2 másodperc, vagy 1,3 másodperc alatt fordul le. Tapasztalatok szerint valójában nem érdemes ezzel a kérdéssel foglalkozni.)

Egy névtérbe bármikor tehetünk még újabb elemeket be. Egy névteret lehetetlen bezárni, bármikot lehet folytatni, akár ugyanabban a forráskódban, akár más forráskódokban. A System névtérbe is tehetnénk saját osztályokat, bár a Microsoft azt kérte, hogy ezt ne tegyük, mivel fenntartja a lehetőséget, hogy később még ő is telepíthet osztályokat, és akkor ütközhet a mi osztályunkkal.



```
namespace elolenyek.haziallatok
{
    class allat { /* ... */ }
}

/* .... */
// bezártuk a névteret //
/* .... */

// de újra megnyitjuk és újabb osztályokat teszünk bele
namespace elolenyek.haziallatok
{
    class kutya : allat { /* ... */ }
    class cica : allat { /* ... */ }
}
```

Érdekesség a névterekkel kapcsolatban, hogy a `using` egyik használati módja mellett lehetőségünk nyílik névterekre alias neveket kialakítani:

```
using gen = System.Collections.Generic;
```

Ekkor a 'gen' azonosítóhoz rendeltük a jóval hosszabb névtérnevet. A program további részében ha a névtérbeli osztályra szeretnénk annak teljes, minősített névvel hivatkozni, akkor azt megtehetjük a 'gen' alias segítségével is:

```
// System.Collection.Generic.List<double> szamok = new ...
gen.List<double> szamok = new gen.List<double>();
```

Erre szükségünk is lesz, mert a `using` ezen alakja az említett névteret nem nyitja fel, tehát a benne lévő osztályok eléréséhez a minősített névre lesz szükségünk, mégha ebben a rövidített alakban is.

Amennyiben egy osztályt mégsem helyezünk el névtérbe, hanem névteren (*namespace*) kívül írjuk bele a kódba, akkor azt mondjuk, hogy ez az osztály a **globális névtérbe** került.

```
using System;
using System.Collections.Generic;

class allat
{
    /* ... */
}
```

Erre van lehetőség, de kerülendő. Ugyanis nincs különösebb indok arra, hogy miért ne rakjuk be valamiféle névtérbe. A névtér neve lehet akár a saját nevünk, a monogramunk, cég (iskola) neve, bármi. Ha nem használunk névteret, később nem lesz lehetőségünk névütközés esetén a minősített névvel feloldani azt.

## 17. Az Object osztály mint őse

---

Amennyiben objektumosztály fejlesztésébe kezdünk, és nem jelöljük meg ki legyen az őse – akkor automatikusan az Object osztály lesz az őse. Az Object osztály a System névtérben van, teljes neve tehát System.Object, alias neve „object”. Vagyis az alábbiak mindegyike egyforma hatású:

```
using System;
class Sajat:Object
{
```

```
class Sajat:System.Object
{
```

```
class Sajat:object
{
```

```
class Sajat
{
```

Az Object osztály néhány alapvető metódust tartalmaz, melyet ennek megfelelően minden más osztály is tartalmaz (örököl):

- GetType() metódus, megadja az adott osztály típusának nevét (az osztály nevét és a névtérrel amelybe az osztály elhelyezésre került).
- ToString() metódus, amely megadja string alakban az adott példányt.
- Equals(x) metódus, mely megadja, hogy az adott példány és az 'x' példány egyenlő-e.
- GetHashCode() egy numerikus értéket ad meg (int), amely a példányra jellemző.

### 17.1. GetType()

Nézzünk egy példát. A GetType függvény alkalmazhatósága speciális eseteket leszámítva inkább csak nyomkövetési, naplózási feladatokra korlátozott. Készítsünk egy függvényt, mely kiírja a paraméterül kapott típus nevét:

```
static void tipusa(Object p)
{
    Console.WriteLine(p.GetType());
}
```

Próbáljuk ki a függvényt néhány egyszerű típusra:

```
int a=2;
string b="hello";
List<int> l = new List<int>();
tipusa(a);
tipusa(b);
tipusa(l);
```

A kiírások az alábbiak lesznek:

```
System.Int32
System.String
System.Collections.Generic.List`1[System.Int32]
```

A GetType() függvény felhasználására további példákat a 29.3 fejezetben fogunk látni.

## 17.2. ToString()

A ToString() alapvetően arra a célra szolgál, hogy a példányunkat stringgé tudjuk alakítani. Erre leggyakrabban a képernyőre írás során van szükség. A Console.Write() és WriteLine() függvények minden, paraméterül kapott értéket ilyen módon írnak ki a képernyőre – meghívják a ToString() metódusát, és a kapott string-et helyettesítik be a kiírandó szövegre. Az egyszerűbb számtípusok esetén a stringgé alakítás a 10-es számrendszerbeli alak meghatározásán alapul (számjegyek alakja). A logikai értékeknél a 'true' vagy 'false' szavakra alakításon stb.

```
int a = 12;
Console.WriteLine(a);
```

Valójában így olvasandó:

```
Console.WriteLine(a.ToString());
```

Ugyanaz az eset, amikor beillesztjük a kiírásba: ugyanígy a ToString() metódus hívódik meg. A helyben beszúrt kifejezések kiszámítódnak, a kapott érték típusának megfelelő ToString() metódus segít a kifejezés értékének kiírásában:

```
int a = 12;
int b = 20;
Console.WriteLine("{0}+{1}={2}", a, b, a+b);
```

Valójában:

```
Console.WriteLine("{0}+{1}={2}", a.ToString(), b.ToString(),
                  (a + b).ToString());
```

Saját osztályaink esetében a `ToString()` metódus felüldefiniálható (override), mivel a `ToString()` virtuális metódus:

```
class kacska
{
    public string nev;
    public kacska(string nev)
    {
        this.nev = nev;
    }

    public override string ToString()
    {
        return String.Format("{0} kacska", nev);
    }
}
```

```
static void Main(string[] args)
{
    kacska d = new kacska("donald");
    Console.WriteLine("a [{0}] furodni megy", d);
}
```

Ebben az esetben a 'd' kiírásakor valójában a 'd.ToString()' kiírása történik meg:

```
a [donald kacska] furodni megy
```

### 17.3. Equals()

Egyenlőséget kétféleképpen vizsgálhatunk a programunkban. Egyrészt az egyenlő operátor (==) használatával, a másrészt az Equals() metódus segítségével. A legtöbb esetben ezek egyformán működnek, mert az egyenlő operátor gyakran éppen az Equals() metódusra vezet vissza a működését:

```
if (a == b) Console.WriteLine("egyenlők");
else Console.WriteLine("nem egyenlők");

if (a.Equals(b)) Console.WriteLine("egyenlők");
else Console.WriteLine("nem egyenlők");
```

Az Equals() metódus is virtuális, az egyes osztályok felüldefiniálhatják. A referencia típusok esetén azonban az == operátor minden esetben azt vizsgálja, hogy a két példányváltozóban tárolt memóriacím egyenlő-e<sup>22</sup>, míg az Equals() működhet más összefüggések vizsgálata alapján is:

```
kacsa d = new kacsa("donald");
kacsa p = new kacsa("donald");
if (d == p) Console.WriteLine("ugyanaz a kacsa");
else Console.WriteLine("különböző kacsák");

if (d.Equals(p)) Console.WriteLine("ugyanaz a kacsa");
else Console.WriteLine("különböző kacsák");
```

Ez mindkét esetben „különböző” eredményt ad, mivel az Equals() alapértelmezett módon szintén az == eredményét használja fel az egyenlőség vizsgálatára. Ha azonban felüldefiniáljuk valamely osztályban (pl. két kacsa egyforma, ha ugyanaz a nevük):

```
class kacsa
{
    public string nev;
    // ....

    public override bool Equals(object obj)
    {
        if (this == obj) return true;

        if (obj is kacsa && (obj as kacsa).nev == this.nev)
            return true;
        else return false;
    }
}
```

Ekkor az d==p vizsgálat szerint a kacsák különbözőek, de az Equals vizsgálat szerint a két kacsa egyenlőnek tekinthető. Így van lehetőségünk a programunkban a saját osztályainkra alternatív egyenlőségvizsgálatot definiálni.

<sup>22</sup> Hacsak felül nem definiáljuk.

## 17.4. GetHashCode()

A GetHashCode() egy hash értéket generál az adott példányhoz. A Hash érték egy „kivonat” a példányról, mely a példány kulcsfontosságú jellemzőiből keletkezik. Előállításának gyorsnak kell lennie, mivel általában sebességoptimalizációs szempontból szokták használni. Például ha két példány egyenlőségét vizsgáljuk, akkor első lépésként összehasonlíthatjuk a két példány hash értékét. Ha ezek nem egyenlők, akkor a két példány biztosan nem egyforma. Ha a két hash érték egyenlő, akkor *lehetséges*, hogy a két példány egyforma. Mivel a hash értékek akkor is egyenlők lehetnek, ha a két példány nem egyforma; ezért alkalmasak lehetnek arra is, hogy az egyforma hash értékű példányokat csoportosítva nagy mennyiségű példány esetén is gyorsan szűkíthessük a keresést.

Bevezethetjük azt, hogy pl. a hash kódot a kacsza osztályban úgy képezzük, hogy szorozzuk a kacsza életkorát (hónapokban) a súlyával. Mivel a súly a tört érték (kilogrammban), így átszámoljuk dekagrammá, és úgy kerekítjük egész számra:

```
class kacsza
{
    protected double sulya;
    protected int eletkora;

    public override int GetHashCode()
    {
        return (int)(sulya * 100 * eletkora);
    }
}
```

Lehetséges, hogy egymástól különböző kacsza példányok esetén is kijöhet ugyanaz a hash érték. De emellett az is elképzelhető, hogy ha tudjuk, hogy egy 20 hónapos 3,5 kilós kacsát keresünk egy 10 000 elemű listában, ezen összehasonlítás alapján csak kevés példány egyezik, és a részletes (időigényes) összehasonlítást már csak erre a néhány példányra kell alkalmazni. Így jóval gyorsabb keresést tudunk megvalósítani.

## 17.5. Boxing - Unboxing

Az Object minden 'class' kulcsszóval képzett objektumosztály őse. Ha nem is közvetlen, de közvetett őse. Ezen típusok mind a referencia típuscsaládba tartoznak, a példányok elsődleges memóriaigénye 4 byte (memóriacímet tárolnak), míg a másodlagos terület memóriaigényét a new kulcsszó számolja ki és foglalja le a példányosításkor. Vizsgáljuk meg az alábbi értékadást:

```
kacsza d = new kacsza("donald");
object o = d;
```

Az object típusú 'o' példány memóriaigénye 4 byte, ahogy a 'd' változóé is. Az 'o = d' értékadás során a 'd'-ben tárolt memóriacím átkerül az 'o' változóba. Valójában tetszőleges példányváltozók között végrehajtható lenne az értékadó utasítás fizikailag, mivel mind-egyikük 4 byte, és mindegyikük csak egy memóriacímet tárol:

```
kacsa d = new kacsa("donald");  
F15Tomcat x = d;
```

Nyilván egy F15Tomcat típusú példányba egy kacsa memóriacímét elhelyezni fizikailag lehetséges, de valószínűtlen, hogy ezek után az 'x.repulj()' metódus az elvártaknak megfelelően működne. A fordítóprogram felügyeli ezeket az értékadásokat, hogy azok megfeleljenek a típuskompatibilitási szabályoknak.

Ugyanakkor ha az Object minden típus őse, akkor mi a helyzet az érték típusokkal (bool, double, int stb.)?

Az érték típusok nem a 'class', hanem a 'struct' kulcsszóval vannak képezve. A 'struct' (rekord) típusok változói nem 4 byte-osak, nem memóriacímet tárolnak, hanem közvetlenül magát az értéket.

A struct kulcsszóval képzett osztályoknak nem jelölhetjük meg az ősét (a struct esetén nincs értelmezve a származtatás elv), ősük alapértelmezett módon a System.ValueType osztály, akinek az őse az Object. Emiatt a struct típusú példányoknak is megvannak az előbb felsorolt metódusai (ToString, GetType stb.). A ValueType-ból nem lehet másképpen gyerekosztályt készíteni (fordító letiltotta ezt a lehetőséget), csakis a struct kulcsszó segítségével.

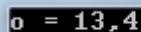
Mivel az érték típusok őse is az Object, úgy az alábbi értékadás is típushelyes:

```
double d = 13.4;  
object o = d;  
Console.WriteLine("o = {0}", o);
```

Az 'o = d' esetén a jobb oldal típusa (legyen az bármi) kompatibilis a bal oldal típusával (object), jelen esetünkben a double típusú 'd' változó is szerepelhet az értékadás jobb oldalán. Próbáljuk meg elképzelni mi történik ezen értékadás során, mivel a 'd' memóriaigénye 8 byte, az 'o' változó memóriaigénye pedig 4 byte<sup>23</sup>. Nyilván nem lehetséges a 8 byte-on tárolt tört számot átmásolni a 4 byte területre. Azonfelül, hogy a double karakterisztika-mantissza értéket később hiba lenne memóriacímként értelmezni. A fenti kód ezzel együtt helyes, működik, és a végén megjelenik az „o = 13,4” kiírás a képernyőn. Ennek legfőbb oka, hogy a kiírás során meghívásra került az 'o.ToString()' metódus megfelelően override-olt változata. Mivel az 'o' dinamikus típusa jelen esetben double lesz, a double ToString-je fogja előállítani a kiírandó értéket.

---

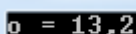
<sup>23</sup> Szándékos a double típus használata a példában. Ha 'd' típusa int lenne, ami szintén 4 byte-os, akkor a következőkben ismertetettek felüli működési ötletek is előállhatnának.



Most térjünk vissza magára az értékadásra. Két értelmezési lehetőség van. Az első a természetesebb gondolat: tároljuk el az 'o' változóban a 'd' változó memóriacímét. Ha ez így lenne, akkor mit kellene az alábbi kódnak kiírni a képernyőre?

```
double d = 13.2;
object o = d;
d = 16.5;
Console.WriteLine("o = {0}", o);
```

Ha az 'o' eltárolta a 'd' memóriacímét, és később a 'd'-ben változik az érték, akkor az utolsó kiírásnál a 'd' legutolsó, aktuális értékének, a 16,5-nek kell megjelennie.



Úgy tűnik, nem ez történik, hiszen a kiírás szerint az eredeti 13,2 érték jelenik meg. Keressünk új elméletet.

Azt a műveletet, melynek során egy referencia típusú változóba (ez lényegében az Object típus lehet csak) egy value type változó értékét helyezzük el, **boxing**-nak nevezzük. A boxing (dobozolás) lényege, hogy az értékről másolat készül a memóriában, és ezen másolat memóriacíme kerül az object-be. Vagyis az 'o = d' során a 'd' double változó értékéről (13,2) egy másolat készül (ami újabb 8 byte), és az értékadás során átmásolja (elementi) oda a 'd' aktuális értékét. Ezután a 'd' értékének megváltoztatása már semmilyen módon nem fogja befolyásolni az 'o'-ban tárolt tört értéket.

A Write során az 'o.ToString()' metódus kerül meghívásra, de van mód az 'o'-ban tárolt tört érték visszanyerésére is. Ez nem ennyire egyszerű:

```
double d = 13.2;
object o = d;
// ...
double x = o;
```

Az 'x = o' nem típushelyes. Gondoljuk végig: az 'o'-ban lényegében bármilyen típusú érték lehet, miért hinné el a fordító, hogy double van éppen benne? De mivel a típuskompatibilitás amúgy is egyirányú, és a bal oldal típusa double, amivel nem kompatibilis az ősosztály Object típusa, így az értékadás szintaktikailag hibás. Nincs nagy baj, csak alkalmazni kell típuskényszerítést. Az 'as' operátor jelenleg nem jöhet szóba, mivel az nem alkalmazható value type esetében. Csak a hagyományos zárójelezős módszer marad:

```
double d = 13.2;
object o = d;
// ...
double x = (double)o;
```



Ez a művelet, mikor visszanyerjük az Object típusú változóba elhelyezett value type értéket, az **unboxing**, a boxing ellentétes művelete. Ennek során az 'o' változóban szereplő másolati memóriacímről visszamásolódik a tárolt 8 byte-os double érték a fogadó oldali 'x' változóba.

## 17.6. Object lista

Vizsgáljuk meg ilyen szemszögből a List<Object> típusú listákat. Ha egy ilyen listát készítünk, a listába helyezett elemek bármilyen típusúak lehetnek, hiszen a listaelemeknek az Object típussal kell kompatibilisnek lenniük. Ugyanazon listához adhatunk hozzá stringeket, kacsákat, double számokat, bármit. Amikor referencia típusú elemeket adunk hozzá, akkor a memóriacímek kerülnek be a listába. Amikor value type értékeket adunk hozzá, akkor minden esetben boxing hajtodik végre, és a másolati érték memóriacíme adódik hozzá a listához.

Ellenben a List<double> eleve magukat a double értékeket tárolja. Szintén másolati formában, de legalább a berakáskor nem kerül plusz 4 byte-ba a memóriacím tárolása a 8 byte-os érték tárolásán felül. Vagyis berakáskor kimarad a boxing művelet.

```
List<Object> lo = new List<object>();
List<double> ld = new List<double>();
double d = 13.4;

lo.Add( d ); // boxing, lassú, 12 byte-ba kerül
ld.Add( d ); // nincs boxing, gyors, 8 byte-ba kerül
```

Amikor visszavesszük az elemet a listából, további problémák vannak az Object típusú listával. Hasonlóan a fentiekhez, típuskényszerítés kell, és unboxing történik:

```
double x = (double)lo[0]; // típuskényszerítés kell, unboxing, lassú
double z = ld[0]; // nem kell típuskényszerítés, nincs unboxing, gyors
```

A System névtér tartalmaz egy ArrayList nevű listát, amely lényegében megfelel a List<Object>-nek. A C# nyelv és a .NET Framework korai verzióiban ugyanis még nem létezett a List<double> és az egyéb típusos lista fogalma, csak ez az egyetlen lehetőség volt a lista képzésére. Az ArrayList ilyen szempontból az újabb verziók esetén már nem annyira hasznos lépés.

Még két megjegyzés a listakezeléssel kapcsolatosan. Amennyiben vegyes listát kell alkalmaznunk (Object alapú listát), úgy a ügyeljünk arra, hogy típuskényszerítéssel nyerjük vissza az elemeket, ne konvertálással. Hasonlítsuk össze az alábbi három módszert:

```
double h = (double)lo[0];
double k = Convert.ToDouble(lo[0]);
double m = double.Parse(lo[0].ToString());
```

Ha már Object típusú listát használunk, és tudjuk, hogy a 0. elem az double, akkor használjuk a 'h' változónál bemutatott módszert. Ugyan unboxing történik, de az még mindig a gyorsabb módszer. A 'k' esetén egy Object típusú értéket adunk át a

ToDouble()-nek, amely a kapott paraméterre azonnal meghívja a ToString()-t, majd a kapott string-et visszaalakítja double-lé. Az 'm' esetén ez jobban látszik, mivel a Parse() nem fogad el Object paramétert, ott már átadáskor meg kell hívni a ToString-et.

Bemutatunk egy hozzávetőleges sebességmérést. A fenti lépést egy sok (pl. 100 000 000) lépésszámú ciklusba foglaltuk, és lemértük az időigényét, hogy legyen értelme az elem kiolvasásának, összegezzük a számokat (bár ez némi plusz művelet, de mivel mindhárom esetben alkalmazni fogjuk, egyforma mértékben növeli meg az össz futási időt):

```
DateTime start = DateTime.Now;
double ossz = 0;
for (int i = 0; i < 100000; i++)
{
    double h = (double)lo[0];
    ossz = ossz + h;
}
DateTime stop = DateTime.Now;
Console.WriteLine("össz futási idő: {0}", stop - start);
Console.ReadLine();
```

A mérés szerint<sup>24</sup>:

'h' módszer	2,28 sec	100%
'k' módszer	3,77 sec	165%
'm' módszer	71,77 sec	3147%
List<double>	1,67 sec	60%

Az eredmény azt mutatja, hogy a Convert.ToDouble jó, valószínűleg az 'is' operátorral megvizsgálja, hogy double-t kapott, és alkalmasan konvertálja a 'h' módszerhez hasonlóan. Ekkor a plusz 'is' operátor használata, és a függvényhívás, paraméterátadás okozhatja a megnövekedett futási időt. A harmadik módszer tűnik a leglassúbb megoldásnak, a stringgé alakítás és visszaalakítás rendkívül időigényes. Ugyanakkor a List<double> használata esetén még típuskonverzióra sincs szükség, a futási idő meggyőzően a leggyorsabb.

Amennyiben a listáról csak bizonyos típusú elemeket kívánunk kiválogatni, alkalmas lehet rá a foreach ciklus is, de nem a következő módon:

```
List<Object> lo = new List<object>();
// ... lista feltöltése ...
//
double ossz = 0;
foreach (double d in lo)
{
    ossz = ossz + d;
}
```

A foreach nem hagy ki elemeket a lista feldolgozása során. Ha a következő listaelem nem double kompatibilis, akkor futási hibát fogunk kapni, amikor ezt a listaelemet a d válto-

<sup>24</sup> A mérési módszert megadtuk, nyilván nem mindegy, hogy milyen operációs rendszerünk van, milyen egyéb szoftverek futnak a gépen, processzor, memória, és .NET Framework függő időket kapunk. A továbbiakban közölt mérési eredmények csak tájékoztató jellegűek, érdemes őket saját gépen megismételni.

zóba próbálja majd berakni. Természetesen ha biztosak vagyunk benne, hogy minden elem double, akkor akár a fenti példában leírtak szerint is használhatjuk a ciklust. Ekkor felmerülhet a kérdés: miért nem List<double>-t használtunk eleve. Mérjük is ki:

```
List<Object> lo = new List<object>();
for (int i = 0; i < 1000; i++) lo.Add(1.2);
//
DateTime start = DateTime.Now;
double ossz = 0;
for (int i = 0; i < 1000000; i++)
{
    foreach (double d in lo)
        ossz = ossz + d;
}
DateTime stop = DateTime.Now;
Console.WriteLine("ossz futasi ido: {0}", stop - start);
Console.ReadLine();
```

A feldolgozás és mérés azonban érdekes eredményt ad. A List<Object> esetén a futási idő 14,48 sec volt, míg a List<Double> esetén a vártnál több, 16,69 sec. futási időt vett igénybe a fenti 1000000 iterációs számú ciklus lefutása. Ezzel együtt ne feledjük, hogy a List<Object> esetén minden listaelem plusz 4 byte tárolást igényel a List<double> lista használatával szemben!

Amennyiben nem minden elem szükséges a listáról, az 'is' operátor segítségével válogathatjuk ki a listaelemeket. A példában megszámoljuk hány 'kacsa' példány szerepel az 'lo' listán:

```
List<Object> lo = new List<Object>();
// ... lista feltöltése ...
//
int db = 0;
// foreach(kacsa k in lo) nem mehet ám!!!
foreach (object o in lo)
{
    if (o is kacsa)
        db++;
}
```

Ha a kacsa példányokkal műveletet is kívánunk végezni, ne feledjük az 'o' ciklusváltozó jelenleg 'object' típusú, vagyis hiába állapítjuk meg, hogy az 'o' kacsa típusú-e, a kacsa metódusait, mezőit nem érhetjük el csak típuskényszerítés után (mivel a pont operátor működésének alapja a statikus típus, nem a dinamikus):

```
foreach (object o in lo)
{
    if (o is kacsa)
        (o as kacsa).hapogj();
}
```

Ez természetesen akkor is igaz, ha a lista típusa nem Object, hanem tetszőleges „ős” osztály típus (jelen példában a kacsa valamely őse).

## 17.7. Object paraméter

Előfordulat, hogy a metódus, amelyet fejlesztünk, egy vagy több paramétert vár, ami lényegében „bármi” lehet. Ilyen a Console osztály Write és WriteLine metódusai. Próbáljunk egy hasonló kiíró metódust fejleszteni:

```
static void Kiiras(Object p)
{
    // ...
}
```

Ekkor természetesen a hívás helyén átadhatunk stringet, double-t, kacsát, bármit:

```
Kiiras( "valami szöveg" );
Kiiras( 12.5 );
kacsa d = new kacsa( "donald" );
Kiiras( d );
```

Az Object paraméter mellett erre természetesen van lehetőségünk, de a double átadása-kor boxing kerül végrehajtásra. Elkerülhetjük, ha célirányosan elkészítjük a függvény-nek ilyen típusú paramétert fogadó változatát is:

```
static void Kiiras(Double p)
{
    // ...
}
```

Ha megnézzük, a Console.Write és WriteLine függvényeknek van is speciális paraméterezéű változata a leggyakoribb value type paramétertípusokra.

Gondoljuk végig azt is, hogyan lehet tetszőleges számú, tetszőleges típusú paramétert átadni – átvenni. Tetszőleges paraméterek alatt azt értjük, hogy a paraméterek egymás-tól is eltérhetnek (pl. 1 double után 1 string, majd egy kacsa, 2 int, újra egy kacsa stb.). Ha nem tudjuk előre a paraméterek számár, használhatjuk a 'params' kulcsszót a formális paraméterlistában:

```
static void Kiiras(params Object[] t)
{
    // ...
}
```

Ekkor a függvény belsejében a 't' valójában egy vektor lesz, jelen esetben Object alaptípusból, elemszáma attól függ, hány paramétert adtunk át híváskor:

```
kacsa d = new kacsa("donald");
Kiiras("valami szöveg", 12.5, true, d);
```

A params kulcsszó azt jelenti, hogy a hívás helyén nem ugyanezen típusból (Object) álló vektor kerül átadásra, hanem a vektor elemeit adjuk át, így a vektort a paraméterek átvételekor kell létrehozni. A vektorra a függvény belsejében az indexeivel hivatkozhatunk:

t.Length	⇒	4	
t[0]	⇒	string	"valami szöveg"
t[1]	⇒	double	12.5
t[2]	⇒	bool	true
t[3]	⇒	kacsa	d

Ha ebből a 't' vektorból formázott stringet kívánunk készíteni, több úton is elindulhatunk. Az első a StringBuilder osztály használata. Ez lényegében egy stringekből álló lista, melyhez elemeket adhatunk hozzá. Nagy előnye, hogy amikor minden elemet hozzáadtunk, akkor az elemekből egyetlen string-et tud képezni, amelybe a listaelemeket összefűzi:

```
static void Kiiras(params Object[] t)
{
    StringBuilder b = new StringBuilder();
    foreach (object x in t)
        b.Append(x);
    string eredm = b.ToString();
}
```

A StringBuilder példány Append() metódusával lehet elemeket hozzáadni. Megfelel a listák Add() metódusának. A végén az egészet stringgé alakíthatjuk a ToString() segítségével. Nyilvánvaló, hogy a StringBuilder osztály override-olta a ToString() metódust, generálván az összefűzés eredményét.

Más úton is megoldhatjuk a stringgé alakítást, ha tudjuk előre a 't' elemszámát, pl. a String osztály Format metódusa (osztálysintű) segítségével. A metódus úgy viselkedik, mint a Console.WriteLine, de az eredményt nem írja ki a képernyőre, hanem a formázott string lesz a visszatérési érték:

```
static void Kiiras(params Object[] t)
{
    string eredm = String.Format("{0} {1} {2} {3}", t[0], t[1], t[2], t[3]);
}
```

Itt természetesen tudnunk kell, hogy hány elemű a vektor, amelyet kaptunk. Segít rajtunk, hogy a String.Format is params vektorban kéri az értékeket, így akár egyben is átadható:

```
static void Kiiras(params Object[] t)
{
    string eredm = String.Format("{0} {1} {2} {3}", t);
}
```

Természetesen azt is megkövetelhetjük, hogy a formátum string-et adja át paraméterként a hívó fél. Lényegében ezt kéri a Console.WriteLine is, az első paraméter a string, a további paraméterek a beillesztendő értékek, amelyekre a sorszámukkal hivatkozunk. Ez esetben a Kiiras() függvény első paramétere kötelezően string, a további paraméterek tetszőlegesek. Ekkor a String.Format is egyszerűbben alkalmazható:

```
static void Kiiras(string formatum, params Object[] t)
{
    string eredm = String.Format(formatum, t);
}
```

Ha így építjük fel a függvényünket, a hívás helyén kell helyesen paraméterezni:

```
kacsa d = new kacsa("donald");
Kiiras("{0} {1} {2}", 12.5, true, d);
```

Ekkor a függvény első paramétere lesz a formátum string, a 't' vektor három elemű lesz, rendre a 12,5; a true, és a 'd' értékeket fogja tartalmazni.

A 'params' kulcsszóval nemcsak Object paraméter vektor építhető fel!

```
static int Maximum(params int[] t)
{
    if (t == null || t.Length == 0) return int.MaxValue;
    var m = t[0];
    foreach (int x in t)
        if (x > m) m = x;
    return m;
}
```

A függvény tetszőleges számú int típusú paramétert elfogad. A vektorból kiválasztja a legnagyobb számot, és azt adja meg futási eredményként. Csak int-eket fogad paraméterként, mert ezt adtuk meg a 'params' kulcsszó után a vektor alaptípusaként.

## 18. Abstract osztályok

---

Nagyon egyértelműnek tűnik az a gondolat, hogy amikor egy programozó belekezd egy osztály írásába, akkor már minden információ a rendelkezésére áll, minden metódust meg fog tudni írni. Ez gyakran igaz is.

Azonban előfordul olyan helyzet, amikor az osztály fejlesztésekor már tudni lehet, hogy szükség van/lesz egy adott metódusra, de a metódus tényleges megírására nincs lehetőség. Jelentkezik ugyanis valamilyen részfeladat, melyet végre kellene hajtani. Meg tudjuk tervezni a feladatot elvégző metódus nevét, meg tudjuk határozni a metódus paraméterezését, de nem tudjuk megírni a törzsét! Miért tudjuk mindezt? Leggyakrabban azért, mert egy másik metódus törzsét írjuk éppen, amikor is elérkezünk ehhez a részfeladathoz, és rájövünk, hogy nem tudjuk megoldani. A metódusunk törzsének írását azonban folytatni kellene azon a ponton, mintha kész lenne ez a bizonyos részfeladat. Kiemeljük tehát a megadott részt egy külön metódusba, és visszatérve a fő metódus írásához – meghívjuk a kiemelt metódust.

Nézzünk egy példát! Egy élővilággal kapcsolatos szimulációt szeretnénk írni. Egy zárt területen (birtokon, erdőben) állatok élnek. Az állatok itt élnek, esznek, mozognak stb. Sokféle állatunk lesz, mindegyik picit más-más módon mozog, eszik, szaporodik. Egyik növényevő, másik ragadozó. Szeretnénk ennek tudatában elkészíteni egy alaposztályt „állat” névvel, specializálni „növényevő” és „ragadozó” irányba, végül elkészíteni az egyes konkrét állatokat. Lesz egy támogató „terület” osztályunk, aki majd tárolja, hogy melyik állat hol áll, segít megadni hol vannak üres területek, ahova vándorolhat egy állat, vagy hol vannak saját fajtájuk egyedei, esetleg ha ragadozó kérdezi hol vannak zsákmányállatok.

A teljesség igénye nélkül nézzük meg mondjuk a szaporodást. A szaporodásnak minden állat esetén ismert (egyszerűsített) feltételrendszere:

- nőtény,
  - életkorban megfelelő állat (nem túl fiatal, nem túl öreg),
  - megfelelő idő eltelt az előző szaporodás óta,
  - az anyaállat megfelelően táplált (egészséges),
  - egy ellenkező nemű megfelelő életkorú társ a közelben,
  - az új egyedek (gyerekállatok) számára van elég szabad terület az anyaállat környezetében (kezdőpozíció).
-

Próbáljuk megtervezni milyen mezők kellene, és milyen metódusok. Bevezetünk először is egy segédosztályt, amelyben koordinátákat lehet tárolni (saját helyünk koordinátái a területen belül, üres mezők koordinátái, zsákmányállatok koordinátái stb.).

```
class koord
{
    public int x;
    public int y;
    public koord(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

A gyerekosztályban a koordinátákon felül azt is el lehet tárolni, hogy mely állat van azon a területen.

```
class elo : koord
{
    public allat eloleny = null;
    public elo(int x, int y, allat eloleny)
        : base(x, y)
    {
        this.eloleny = eloleny;
    }
}
```

Az állat osztályba az alábbi mezőket vesszük fel adattárolásra:

```
enum neme { nosteny, him, egyeb }
class allat
{
    protected int eletkora; // hónapokban
    protected neme nem; // neme
    protected int utolso_szaporodas; // hanyadik honapban volt
    protected int joltaplalt; // skála [0..10], 0=nem, ..., 10=maximum
    protected koord kpos; // a területen belüli koordináták
}
```

Az állat szaporodási metódusát virtuális metódusként írjuk meg, hátha a gyerekosztályokban valami alapvetően változni fog, és kompletten cserélni kívánják a tartalmát:

```
public virtual void szaporodas()
{
    // #1: nosteny-e
    if (this.nem != neme.nosteny) return;
    // #2: életkorban megfelelo allat-e
    if ( ??? )
}
```



Az első probléma: hogyan kezeljük az életkori intervallumot? Ez az intervallum minden állatfajnál más. Az elefánt pl. 12 évesen válik ivaréretté, míg a vörös rókák 10 hónapon már azok. Nem tárolhatjuk konstansban az ivarérettség kezdőértékét. Melyik értéket tegyük bele? Hogyan módosítsuk más jellemzőjű állatok esetén? Hasonló okokból nem tárolhatjuk osztályszintű mezőben sem, hisz az elefántoknál beírnánk, hogy 144 hónap, a vörös rókák esetén beírnánk a 10-et, de mivel az osztályszintű mezőből csak egy van, ezzel felülnánk az elefántok 144-es értékét is.

Tárolhatnánk az ivarérettség kezdő és befejező értékét példányszintű mezőben is, de akkor a vörös róka példányok esetén ennyiszer kerülne be a 10-es érték (int, 4 byte), minden példány esetén külön-külön foglalva a memóriát. Nem túl jó gondolat.

Készíthetnénk erre a célra egy ellenőrző függvényt, amely megadja logikai értéként, hogy a példány ivarérett-e. Ebbe a függvénybe literálként kerülne be az adott állattípus esetén a minimális és maximális érték:

```
protected virtual bool ivarerett()
{
    if (10 <= this.eletkora) return true;
    else return false;
}
```

Feltéve, hogy megvan nekünk a függvény – folytatni tudjuk a megkezdett szaporodási függvényünket:

```
public virtual void szaporodas()
{
    // #1: nosteny-e
    if (this.nem != neme.nosteny) return;
    // #2: életkorban megfelelo allat-e
    if ( ivarerett()==false ) return;
}
```

Igen ám, de mit tartalmazzon az 'ivarerett()' függvény az 'allat' osztály szintjén? Ez annyira kezdeti tudású osztály, hogy semmit sem tud az ivarérettségről, hisz nem tudja milyen állattá válik később. Vizsgáljunk meg három megoldást!

**#1 megoldás:** Írjuk meg a metódust, tegyük bele valami fiktív adatot:

```
class allat
{
    protected virtual bool ivarerett()
    {
        if (0 <= this.eletkora) return true;
        else return false;
    }

    public virtual void szaporodas()
    {
        // #1: nosteny-e
        if (this.nem != neme.nosteny) return;
        // #2: életkorban megfelelo allat-e
        if ( ivarerett()==false ) return;
    }
}
```

A megoldás két szempontból is előnytelen:

- A virtual metódusokat nem kötelező felülírni a gyerekosztályok szintjén. Tehát a gyerekosztály programozója akár el is felejtetheti felüldefiniálni override segítségével, de ez nem fog „kiderülni”, hiszen úgy tűnik van egy működőképes változat ebből a függvényből. A szimuláció le fog futni, de gyanúsan szaporák lesznek bizonyos állatok. A futás elemzéséből kiderülhet, hogy a kicsinyek is azonnal szaporodni kezdenek, ennek okát kutatva jöhetünk rá, elmulasztottuk felüldefiniálni a függvényt!
- A kód ugyan most két soros, de valójában teljesen felesleges, hiszen a sorsa az, hogy minden gyerekosztály felüldefiniálja. Ennek ellenére a változat lefordításra kerül, belekerül az .exe-be, foglalja a helyet a memóriában, holott elvileg sosem kerül meghívásra.

**#2 megoldás:** Írjuk meg a metódust, de valójában hagyjuk üresen a törzsét. Nem könnyű jelen esetben, mivel a függvénynek egy bool értéket kell visszaadni. Adjunk vissza false-t:

```
protected virtual bool ivarerett()
{
    return false;
}
```

Közepesen jó megoldás, hiszen megint van egy függvénytörzsünk, aminek nincs tényleges haszna. Most az a szerencsénk, hogy találtunk egy olyan visszatérési értéket, ami eléggé árulkodó, ha egy gyerekosztály elfelejtené felüldefiniálni, az az állattípus hamar eltűnne a szimulációból, mivel sosem szaporodna. De ez újfent azt jelentené, hogy abban a hiszemben élénk egy ideig, hogy minden rendben van, a program elkészült, fut. Csak a működés során tapasztalt rendellenesség okainak elemzése mutatná ki a hibát.

**#3 megoldás:** Írjuk meg a metódust, de a törzsébe csak egy kivétel feldobást tartalmazzon!

```
protected virtual bool ivarerett()
{
    throw new NotImplementedException("ivarerett()");
}
```

Egy picit javult a helyzet, hiszen amikor a kód a függvényt meghívja, akkor azonnal exception-t kapunk, futási hibát, amely nagy valószínűséggel leállítja a programot. Persze ez még mindig azt jelenti, hogy a programunk fordítása sikeres, indítható (azt hiszünk minden rendben van), csak a hamarosan jelentkező futási hibák alapján értesülünk arról, hogy mégis így. De legalább nagyon gyorsan kiderül: mi nincs rendben, mit kell tennünk, hogy a kód teljesen elkészüljön. A függvény törzse rövid, ez egy aránylag jó megoldás.

Az előző három megoldás mindegyike úgy működött, hogy a fordító nem jelzett fordítási hibát, csak futás közben derült ki, hogy a kódunk még nincs teljesen kész. Ennek egyetlen oka van: a virtuális metódust nem kötelező felüldefiniálni, így a fordító nem jelzett hibát azon gyerekosztályok esetén, amelyek ezt nem tették meg.

A legjobb megoldás a problémára egyértelműen jelezni a fordítónak, hogy ez nemcsak egy egyszerű virtuális metódus, hanem egy olyan, amelyet kötelező felüldefiniálni. Az ilyen metódusokat az 'abstract' kulcsszóval kell megjelölni. További előnye az abstract metódusnak, hogy az ilyen metódus esetén nem kell törzset sem kidolgozni:

```
class allat
{
    protected abstract bool ivarerett();
```

Vegyük észre az alábbiakat:

- az 'abstract' mellett már nem szerepeltetjük a 'virtual' kulcsszót is, mivel az abstract jelentése magába foglalja a virtual jelentéstartalmat is,
- az abstract metódust nem követi törzs ( { ... } közötti kódrész), hanem azonnal lezárásra kerül a pontosvessző segítségével.

Még egy alapvető információ: amennyiben egy osztályban szerepelnek abstract metódusok, magát az osztályt is meg kell jelölnünk az abstract jelzővel (absztrakt osztály) – különben szintaktikai hibát kapunk. Az alábbi ábrán a problémát jelzi a fordító. A „*allat.ivarerett() is abstract but is contained in a non-abstract class 'allat'*” fordítása: „*az allat osztály ivarerett() metódusa abstract, de egy nem-absztrakt osztály tartalmazza*”:

```
class allat
{
    protected abstract bool ivarerett();
    public virtual void szaporodas()
    {
    }
```

'szimulacio.allat.ivarerett()' is abstract but it is contained in non-abstract class 'szimulacio.allat'

A megoldás tehát, hogy az osztályt is megjelöljük az abstract jelzővel:

```
abstract class allat
{
    protected abstract bool ivarerett();
```

Ekkor több dolgot is összegyűjtöttünk. Az abstract metódusokat kötelező felüldefiniálni a gyerekosztályban. Ha nem tesszük meg, szintaktikai (fordítási) hibát kapunk. Másképpen fogalmazva: a fordító figyelmeztet minket, hogy elfelejtettük felülírni a metódust!

```
class ragadozo : allat
{
    'szimulacio.ragadozo' does not implement inherited abstract member 'szimulacio.allat.ivarerett()'
```

A hibát az osztály nevével jelzi. A „ragadozo does not implement inherited abstract member `allat.ivarerett()`” fordítása: „a ragadozó osztály nem implementálta az állat osztálytól örökölt absztrakt `ivarerett()` metódust”.

A gyerekosztály fejlesztője tehát figyelmeztetést kapott, hogy ne feledkezzen el a metódus felüldefiniálásáról. Sajnos, a 'ragadozo' osztály nem tudja megírni a metódust, hisz itt még mindig nem ismert az ivarérettségre vonatkozó adatok. Mivel a fordító makacs, felül kell definiálnunk a metódust. Mit tegyünk a törzsébe? Megint végigmehetünk a #1..#3 megoldásokon, ugyanazokkal a problémákkal fogunk küzdeni. Ráadásul elkövetünk egy nagy hibát! Ugyanis ha felüldefiniáljuk az abstract metódust (override kulcsszóval), akkor megszűnik a kényszer. A gyerekosztály (vörös róka osztály) már megint nem fog figyelmeztetést kapni, hogy el ne felejtse felülírni a metódust. Az override-ot ugyanis megint csak nem kötelező felüldefiniálni. Ne válasszuk ezt a módszert!

Mivel a 'ragadozo' osztály tartalmaz absztrakt metódust (örökölte), miért is ne jelölnénk meg őt is az 'abstract' kulcsszóval?

```
abstract class ragadozo : allat
{
```

A fordító azonnal tudomásul veszi, hogy bár nem írjuk felül, de nem is fogjuk. Mivel az osztály is megjelölésre került, a továbbiakban szintaktikailag hibátlanak tekinti a kódot! Jöhet a gyerekosztály:

```
class vorosroka : ragadozo
{
    'szimulacio.vorosroka' does not implement inherited abstract member 'szimulacio.allat.ivarerett()'
}
```

Ugyanaz a hibaüzenet, ugyanaz az ok. Meg kell írni a metódust, vagy az osztályt kell megjelölni abstract-tal. Mivel most meg tudjuk írni a metódust, válasszuk végre ezt az opciót:

```
class vorosroka : ragadozo
{
    protected override bool ivarerett()
    {
        if (this.eletkora >= 10) return true;
        else return false;
    }
}
```

Az abstract metódusokat override segítségével kell felülírni. Mivel a felülírás megtörtént, az osztályt már nem kell megjelölni abstract jelzővel a továbbiakban.

Ha alaposabban megvizsgáljuk az `ivarerett()` metódust, valójában lehetne csak olvasható (csak `get` részt tartalmazó) property is. Lehet egy property absztrakt? Lehet!

```
abstract class allat
{
    protected abstract bool ivarerett { get; }
```

Mint láthatjuk, absztrakt property esetén meg kell adni a property típusát (`bool`), és, hogy mely részeit kell majd a gyerekosztálynak kidolgozni. Jelen esetben csak a `get` részt követeljük meg. Ha mindkét részre szükség lesz, `'... { get; set; }'` formában tudjuk beírni a forráskódba.

```
abstract class ragadozo : allat
{
```

A ragadozó osztály, mivel nem írta meg az absztrakt property-t (tartalmaz absztrakt elemet), így szintén megjelöli magát `abstract` kulcsszóval.

```
class vorosroka : ragadozo
{
    protected override bool ivarerett
    {
        get
        {
            if (this.eletkora >= 10) return true;
            else return false;
        }
    }
}
```

A vörös róka osztály `override` segítségével felülírta a property-t, így neki már nem kell saját magát `abstract` jelzővel ellátnia.

Felmerül a kérdés: van-e lehetőség nemcsak a `get`, de a `set` részt is kidolgozni? Próbáljuk meg – de sajnos a fordító hibát jelez:

```
protected override bool ivarerett
{
    get
    {
        if (this.eletkora >= 10) return true;
        else return false;
    }
    set
    {
        this._ivarerett = this.value;
    }
}
```

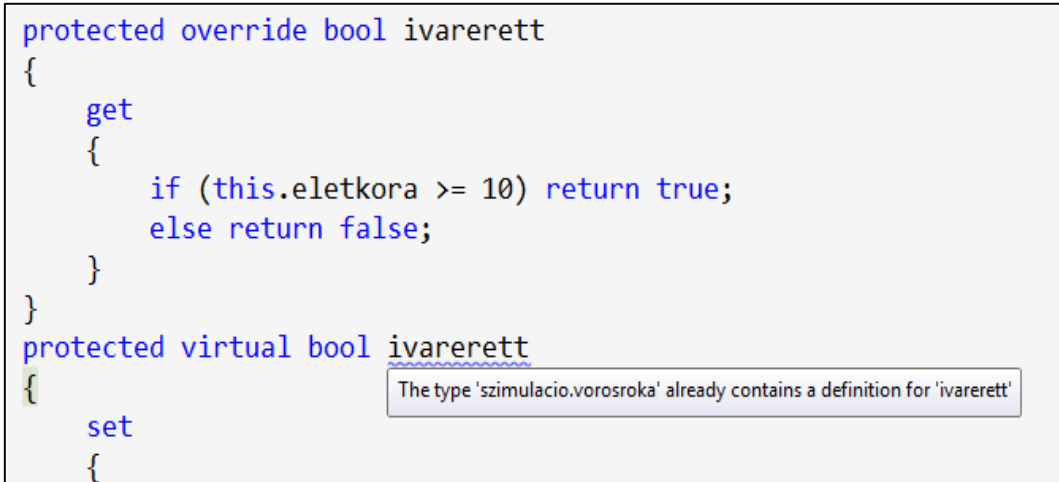
'szimulacio.vorosroka.ivarerett.set': cannot override because 'szimulacio.allat.ivarerett' does not have an overridable set accessor

Az üzenet *„ivarerett.set cannot override because allat.ivarerett does not have an overridable set accessor”* fordítása: *„a set nem lehet override, hiszen nincs mit overrideolni, az eredeti osztályban nem volt jelen”*. Igen, a probléma egészen egyszerű. A következő fejezetben tárgyalni fogjuk a VMT táblát, és ki fog derülni, hogy amennyiben az őssztályban nincs jelen egy `'virtual'` metódus (most `abstract` kulcsszó alakban), akkor nem

kerül be a bejegyzés a VMT táblába, így az override nem fogja tudta a táblázatbeli sorát felülírni.

Nincs más lehetőségünk, ketté kell választanunk a property-t:

```
protected override bool ivarerett
{
    get
    {
        if (this.eletkora >= 10) return true;
        else return false;
    }
}
protected virtual bool ivarerett
{
    set
    {
```



A hibaüzenet szerint nem lehet. A „*the type vorosroka already contains a definition for 'ivarerett'*” fordítása: „*a vorosroka típus már tartalmaz egy ivarerett tagot*”.

Utolsó megjegyzéseink:

- Konstruktor nem lehet absztrakt, mivel virtual sem lehet. Ugyanis a gyerekosztály nem tudná ugyanazon néven felülírni, mert a gyerekosztálybeli konstruktorok a gyerekosztály nevével egyeznek meg,
- Mező nem lehet absztrakt, mivel a mező definiálásakor meg kell adni a mező típusát, nevét, nincs olyan jellemző, amit ne tudnánk megadni.
- Destruktor sem lehet absztrakt, a destruktorként a Finalize() metódus override-ja. Emiatt a destruktorként írás nem kötelező.

## 19. VMT és DMT

---

Az OOP esetén nem beszélhetünk klasszikus értelemben vett függvényhívásról, mivel minden függvény valamely objektumosztály része, így metódus a neve. A metódusok hívása során három esetet különböztetünk meg:

- **Ha a metódus osztályszintű,** akkor a metódus neve előtt szerepel az osztály neve is. Amennyiben ebben az osztályban ilyen nevű metódusból több is szerepel, akkor az aktuális paraméterlista egyértelműen azonosítja, hogy melyik metódus hívása történjék meg. Ha más osztályokban is szerepel ugyanilyen nevű metódus, az most nem okoz semmilyen problémát, mivel a metódus neve előtt szerepel, melyik osztálybeli változatot kívánjuk meghívni. Vagyis teljesen egyértelműen kerül azonosításra a metódus. Ezért mindig korai kötést alkalmazunk.
- **Ha a metódus példányszintű, de nem virtuális,** akkor a híváskor a metódus-név előtt szerepel egy példány neve is. A példányváltozó deklarálásakor megadtuk a típusát (statikus típus). A konkrétan megjelölt objektumosztály által tartalmazott metódus kerül meghívásra. A tartalmazás azt jelenti, hogy örökölte vagy ő maga fejlesztette ki. Mivel nem virtuális a metódus, így vagy örökölte, vagy 'new' segítségével definiálta felül. Mindesetre az adott osztályból kiindulva az öröklési fában felfele haladva a gyökérelemig egy keresés hajtható végre, hogy melyik ponton van az adott nevű és paraméterezésű metódusból a legutolsó felüldefiniált változat. Ez kerül meghívásra. Mivel a kiinduló osztály típusa ismert a fordító által, a keresést fordítási időben végre tudja hajtani. Minden esetben korai kötést kell alkalmazni.
- **Ha a metódus példányszintű, és virtuális,** akkor a probléma lényege, hogy a példány statikus típusára a fordító nem alapozhat. A típuskompatibilitás miatt a statikus típust úgy kell értelmezni: „legalább ilyen típusú”. Az aktuális érték ez a típus vagy valamely gyerekosztályából kerül ki. A fordító nem képes eldönteni, hogy melyik gyerekosztály példánya kerül majd be értékként konkrétan, így nem tudja eldönteni azt sem, melyik metódusváltozat lesz a legfrissebb a konkrét esetben. Itt (információhiány miatt) késői kötést kell alkalmazni.

A korai kötés esetén a fordító a metódushívás minden aspektusát képes felismerni fordítási időben. A fordító által generált (gépi) kódban pontosan meg tudja jelölni, hogy a futást a program melyik pontját kell folytatni, hol van az a függvény belépési pontja, ahova át kell adni a vezérlést. Gépi kódban a 'call' utasítással lehet megjelölni. A 'call' paramétere szokásosan a memória azon bájtyjának címe, ahol a hívott függvény (lefordított utasításainak) belépési pontja van.

A késői kötés esetén a pont a memóriában nem azonosítható fordítási időben. Ugyanakkor ez egy függvényhívás, egy utasítás, melyhez a fordítónak gépi kódot kell rendelnie. A 'call' gépi kódú utasítást nem rendelheti hozzá, de valamit akkor is generálnia kell. Milyen kód generálódik a késői kötés esetén a hívás helyére?

---

### 19.1. A VMT segédtablázat

Hogyan működik a fordítóprogram? A forráskódot olvassa, és részekre bontja, kisebb részekből indul ki, de folyamatosan nagyobb fordítási egységeket épít. A tényleges kódgenerálásnak akkor lát neki, amikor már a nagyobb szintaktikai egység is előállt. Ilyen nagyobb blokk az Osztály (class). Tudnia kell a class nevét, kezdetét és végét, milyen mezők, property-k, metódusok vannak benne, mielőtt azok részleteibe belekezdené. Ez azt is jelenti, hogy egyszerre egy osztály forrásával dolgozik, amíg azzal nem végzett, másik osztály megértésébe nem kezd bele.

Amikor egy osztály forráskódját összegyűjtötte, épít egy táblázatot<sup>25</sup>, melynek neve **Virtual Method Table**, virtuális metódus táblázat, rövidítve **VMT**. Ez egy speciális feladatú táblázat, és csakis a virtual kulcsszóval megjelölt metódusok (és property-k) szerepelnek benne. Most koncentráljunk a metódusok esetére, az alapján a property-k kezelése is megérthető.

A VMT táblázat a fordító a forráskód olvasásával egyidőben, fordítási időben építi fel. Ekkor a táblázat felépítése:

- melyik osztályban szerepel ez a metódus,
- a metódus neve és paraméterezése,
- mi a metódus memóriacíme.

A metódus paraméterezése azért kell, mert egyforma nevű, de különböző paraméterezésű metódusból több is lehet, a metódus neve önmagában nem elég. Az egyszerűsítés kedvéért tekintsünk el ettől a jellemzőtől. A memóriacímet pedig a fordító határozza meg a kódgenerálási fázisban, most a konkrét értékek helyett oda csak egy jelzés kerüljön.

A táblázat az alábbi algoritmussal készül:

- vegyük a VMT tábla kiinduló állapotát (legyen induláskor üres, később majd visszatérünk rá)
- ha 'virtual' vagy 'abstract' kulcsszóval találkozunk (ebben az osztályban bevezetett új metódus), akkor adjunk a táblázat végéhez egy új bejegyzést róla,
- ha 'override'-dal találkozunk, keressük meg a bejegyzés sorát, és írjuk át az első oszlopot (a tartalmazó osztály nevét) erre az osztályra.

---

<sup>25</sup> Valójában sok táblázatot épít, de most számunkra csak ez az egy lesz érdekes.



Következzék egy egyszerűsített példa: a metódusok neve és paraméterezése esetünkben nem is annyira lényeges:

```
class proba1
{
    public void A() { /* ... */ }
    public virtual void B() { /* ... */ }
    public virtual void E() { /* ... */ }
    public void D() { /* ... */ }
}
```

Az ehhez tartozó VMT tábla az alábbiak szerint néz ki:

'proba1' osztály VMT		
proba1	B	[ proba1.B mem.címe ]
proba1	E	[ proba1.E mem.címe ]

Mivel esetünkben a memóriacímbe szerepel az osztály neve, ezért az első oszlopot a továbbiakban elhagyjuk:

'proba1' osztály VMT	
B	[ proba1.B mem.címe ]
E	[ proba1.E mem.címe ]

A VMT táblában nem szerepel az A() és D() metódusról semmilyen információ, mivel azok nem virtuális metódusok. Összesen két bejegyzés van a táblázatban, mert csak két virtuális metódusunk van. Mindkettő teljesen új, ezért – bár a VMT tábla kiinduló állapota üres volt – két új sorral bővítettük. Készítsük el egy gyerekosztályát ennek az osztálynak:

```
abstract class proba2 : proba1
{
    public abstract void C();
    public override void E() { base.E(); }
    public virtual void F() { /* ... */ }
}
```

A 'proba2' osztály első lépésben örököl mindent az ősosztálytól, a virtuális metódusokat is. A VMT tábla esetében azt jelenti, hogy átveszi az ősosztály VMT tábláját (ez lesz az induló állapota). A továbbiakban a fenti „algoritmus” szerint járunk el: módosítjuk a táblázat sorait, új sorokkal egészítjük ki:

'proba2' osztály VMT	
B	[ proba1.B mem.címe ]
E	[ <b>proba2</b> .E mem.címe ]
C	-
F	[ proba2.F mem.címe ]

Az absztrakt C() metódus esetén új sort adunk a táblázat végéhez, de a memóriacímet nem tudjuk kitölteni, mivel a C metódusnak nincs törzse. Az override miatt a táblázat adott sorát módosítjuk, a virtuális F() metódust ismételten a táblázat végére szúrjuk be.

Mi a különbség, ha nem ebben a sorrendben szerepelnek a 'proba2' belsejében a metódusok? Lássuk:

```
abstract class proba2 : proba1
{
    public override void E() { base.E(); }
    public virtual void F() { /* ... */ }
    public abstract void C();
}
```

Ezen sorrend esetén a táblázat utolsó 2 bejegyzése esetleg felcserélődik, de a táblázat első 2 bejegyzése nem változik, és az így kapott VMT is 4 bejegyzéses. A táblázat lényege nem változott tehát meg:

'proba2' osztály VMT	
B	[ proba1.B mem.címe ]
E	[ proba2.E mem.címe ]
F	[ proba2.F mem.címe ]
C	-

Ha virtuális property-t adunk hozzá, akkor is hasonlóan jár el a fordító (a második táblázatból indulunk ki):

```
abstract class proba3 : proba2
{
    public abstract int H { get; }
    public override void C() { }
    public virtual double G
    {
        get { return 0; }
        set { }
    }
}
```

'proba3' osztály VMT	
B	[ proba1.B mem.címe ]
E	[ proba2.E mem.címe ]
F	[ proba2.F mem.címe ]
C	[ proba3.C mem.címe ]
H.get	-
G.get	[ proba3.G.get mem.címe ]
G.set	[ proba3.G.set mem.címe ]

A get és a set rész más-más bejegyzésként kerül be, hiszen ők külön-külön is hívhatóak. A H property get része absztrakt, így nincs memóriacím. A C metódust felülírtuk, így ott már van memóriacím.

Vegyük észre az alábbiakat:

- A gyerekosztály VMT táblája mindig legalább annyi sort tartalmaz, mint az őszosztályé, mivel a gyerekosztály VMT-je úgy készül, hogy lemásoljuk az őszosztály VMT tábláját.
- A táblázat ezen első sorai minden esetben ugyanazon metódusra vonatkoznak, legfeljebb ha override volt ebben a gyerekosztályban, akkor cserélődik a memóriacíme.
- A táblázat új bejegyzésekkel bővíthet.
- Absztrakt osztály esetén a táblázatban van olyan bejegyzés, ahol a memóriacím nem kitölthető (nincs törzs).

Most térjünk vissza a késői kötésre! Tegyük fel, hogy valamely 'p' példányunk esetén meghívjuk az 'E()' metódust.

```
static void E_call(proba2 p)
{
    p.E();
}
```

Esetünkben a hívás helyén dől el, hogy milyen osztálybeli példányt adunk át ezen függvénynek, így a 'p' dinamikus típusát a fordító a fordításkor nem ismerheti. Késői kötés használ, hiszen látja, hogy a 'proba2' osztály esetében (statikus típus) az E() metódus virtuális.

Mit tehet a fordító? Milyen kódot generál az 'p.E();' helyére?

1. határozd meg a 'p' dinamikus típusát,
2. keresd ki ezen osztályhoz tartozó VMT táblában az 'E' metódus bejegyzését,
3. nézd meg milyen metóduscím tartozik oda,
4. ugorj erre a metódusra, ezt kell meghívni.

Lássuk a gyakorlatban (nem elfeledvén, hogy a proba2 és a proba3 valójában absztrakt osztály, belőlük példányosítani nem lehet; de jelen vizsgálat során most engedjük meg gondolatban, mivel az E() metódust kell meghívni, mert mindhárom osztály esetén hívható):

```
proba1 a = new proba1();
proba2 b = new proba2();
proba2 c = new proba3();
proba3 d = new proba3();
E_call(a);
E_call(b);
E_call(c);
E_call(d);
```

Az 'E\_call(a)' esetén a dinamikus típus 'proba1' lesz, az ottani VMT táblában ez az 'E' metódus a második bejegyzés, a táblázat a 'proba1.E()' függvény belépési pontját tartalmazza, így a 'proba1.E()' függvény hívódik meg.

Az 'E\_call(b)' esetén a dinamikus típus 'proba2' lesz, az ottani VMT táblában ez az 'E' metódus a második bejegyzés, a táblázat a 'proba2.E()' függvény belépési pontját tartalmazza, így a 'proba2.E()' függvény hívódik meg. Rendben van, hiszen a 'b' példány számára az 'E' metódusból ez a legújabb verzió.

Az 'F\_call(c)' esetén a dinamikus típus 'proba3' lesz (bár a 'c' statikus típusa a proba2, de ez most lényegtelen). Az ottani VMT táblában ez az 'E' metódus a második bejegyzés, a táblázat a 'proba2.E()' függvény belépési pontját tartalmazza (mivel a proba3 nem definiálta felül az E() metódust). Így a 'proba2.E()' függvény hívódik meg.

Ehhez hasonlóan az 'E\_call(d)' esetén is (a 'd' dinamikus típusa szintén proba3), így a 'proba2.E()' függvény hívódik meg.

Két dolgot kell már csak megértenünk:

- A példányok dinamikus típusát valójában nem határozzuk meg, helyette a példányhoz tartozó VMT tábla memóriacímét eltároljuk a példány egy speciális mezőjében (ezzel a példány memóriaigénye 4 byte-tal nő meg). Nevezzük ezt a mezőt a továbbiakban VMT-nek.
- A példányhoz tartozó VMT táblában nem kell megkeresni az adott metódus sorát, hiszen mindegy mi a példány dinamikus típusa, a szóban forgó metódus minden esetben ugyanabban a sorban lesz. Ennek oka, hogy a gyerekosztályok VMT táblája úgy kezdődik, hogy lemásoljuk az ősz osztály VMT tábláját. Ha az E() metódus a második sorban volt az ősz osztályban, akkor minden gyerekosztályában is a második sorban lesz.

Hogy a hívandó metódus a VMT tábla hányadik sorában lesz, már a fordító a fordítási fázisban is tudni fogja. Mivel látja a 'p' példány statikus típusát (proba2), és, hogy a proba2 osztály VMT táblájában az 'E' metódus a második sorban van, ezt az információt felhasználhatja a fordításkor és kódgeneráláskor.

Mivel nem kell keresni a metódus sorát a táblázatban, a táblázat első oszlopa már futás közben elhagyható. A VMT tábla futás közben egy oszlopos, csak a memóriacímeket tartalmazza.

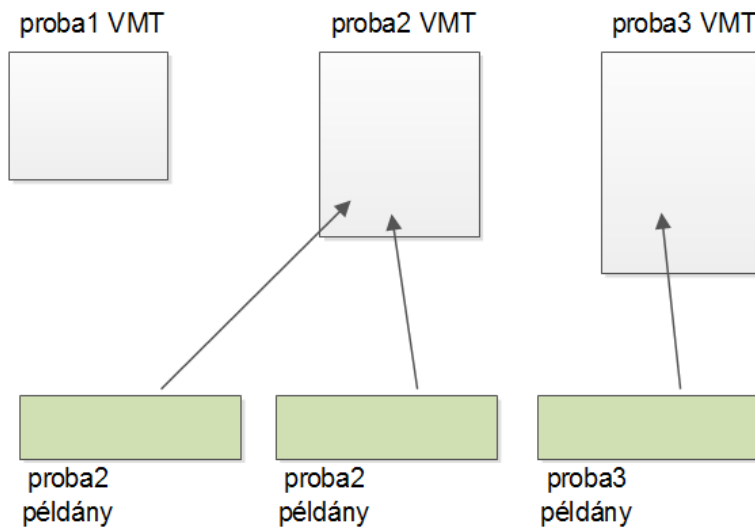
```
static void E_call(proba2 p)
{
    p.E();
}
```

A 'p.E()' híváshoz tartozó késői kötés generált kód a késői kötés esetén pedig az alábbi:

```
olvasd ki a p.VMT[2] memóriacímét
ugorj erre a memóriacímre
```

Néhány fontos tudnivaló:

- A memóriacímek 4 byte-osak (mindegy, hogy adat vagy függvény belépési pontja, a memóriacímek 4 byte-osak<sup>26</sup>),
- tehát a VMT táblák helyfoglalása pontosan annyszor 4 byte, ahány bejegyzés van bennük.
- A VMT táblák a futás közben a fentieknek megfelelő mennyiségű memóriát kötnek le.
- Az osztály VMT táblája akkor is a memóriában van, ha az osztályból nem készült még példány.
- Egy adott osztály VMT táblája csak egyszer kerül be a memóriába, minden (az adott osztályból készült) példány ugyanezen VMT táblára mutat.
- A példány VMT mezőjének címét a konstruktor helyezi el a példány VMT mezőbe, tehát ez a VMT mező értéke a példányosításkor automatikusan kitöltésre kerül.



## 19.2. A DMT segédtablázat

A VMT táblázat készítése egyszerű, segítségével a késői kötés megvalósítása is egyszerű és gyors. Egy észrevételünk azonban lehet: ha egy gyerekosztályban nem override-olunk metódusokat, a gyerekosztály VMT-je akkor is tartalmazza azokat a bejegyzéseket változatlan formában, amelyeket az ősosztály táblázat. Mivel úgy kezdjük a gyerekosztály VMT tábláját készíteni, hogy lemásoljuk az ősosztályét, szélsőséges esetben (nincs override, nincs virtual, abstract a gyerekosztályban) a gyerek VMT táblája akár 100% mértékben megegyezhet az ős VMT táblájával. Így feleslegesnek tűnik az eljárás, és memóriapazarlónak.

<sup>26</sup> 32 bites processzor architektúra esetén, a 64 bites kódnál a memóriacímek 8 byte-osak!

Bemutatunk egy lehetséges alternatív módszert, megoldást. Az alternatív módszer neve **Dinamikus Metódus Tábla** (*Dynamic Method Table*). Készítésekor az lesz a fő szempontunk, hogy amennyiben a táblázatunk valamely bejegyzése nem módosulna a gyerekosztály esetében, úgy azzal nem terheljük a gyerekosztály táblázatát. Csak azok a bejegyzések kerülnek be, amelyek megváltoztak az őszosztályéhoz képest (felülírás, új bejegyzés). Az az elképzelésünk, hogy így kevesebb memóriába kerül a táblázat.

Vegyük észre, hogy egy DMT tábla viszont nem teljes, nem tartalmaz az összes virtuális metódushoz bejegyzést. Vagyis a késői kötéshez itt egyetlen DMT tábla vizsgálata nem lesz elég. Végtelen esetben egy gyerekosztály DMT táblája akár üres is lehet, ha nem módosított egyetlen virtuális metódust sem, és újakat sem vezetett be.

Ezért minden DMT tábla tartalmaz egy linket az ősének a DMT táblájára. Ha egy metódus a gyerek táblájában nincs benne, akkor az ősében kell majd keresni. Ha ott sincs, akkor annak az ősében. Ennek megfelelően az előző fejezetbeli osztályok esetén a DMT táblák az alábbi módon néznek ki:

'proba1' osztály DMT	
	[ link az Object.DMT-re ]
B	[ proba1.B mem.címe ]
E	[ proba1.E mem.címe ]

'proba2' osztály DMT	
	[ link a proba1.DMT-re ]
E	[ proba2.E mem.címe ]
C	-
F	[ proba2.F mem.címe ]

'proba3' osztály DMT	
	[ link a proba2.DMT-re ]
C	[ proba3.C mem.címe ]
H.get	-
G.get	[ proba3.G.get mem.címe ]
G.set	[ proba3.G.set mem.címe ]

Milyen kód generálható egy késői kötés feloldására, ha a DMT táblát használjuk? Nem annyira egyszerű kód, mint a VMT esetén. A DMT táblák esetén nem igaz az a gondolat, hogy pl. az 'E()' metódus mindig a második bejegyzés. Még csak az sem igaz, hogy az 'E()' metódus garantáltan benne van a DMT táblában. Az 'E()' metódus ha benne is van a táblázatban, akkor is szinte bármelyik sor lehet. Ezért itt nem elég a metódusok memóriacímét tárolni, tárolni kell a sorokhoz azt is, hogy az adott sor melyik metódushoz tartozik. Hogyan tegyük ezt meg?

Ne tároljuk a metódus nevét, mert a nevek hosszúak. A string-ek összehasonlítása pedig nagyon lassú folyamat. Helyette minden virtuális metódusnak adjunk valamilyen sorszámot. Kezdjük módszeresen az 0-ás sorszámmal, és ha új metódust vezet be a programozó (virtual vagy abstract kulcsszó) akkor az új sorszámot kap. Az override esetén pedig meg kell keresni, mely sorszámú metódust módosítjuk.

A sorszáмок tárolása is helyigényes. Mennyi helyet adjunk neki? Ha 1 byte-ot szánunk rá, akkor a sorszáмок csak 0..255 közöttiek lehetnek. Ha 2 byte-t, akkor 0..65535 közöttiek, 4 byte esetén 0..4 milliárd. Az első választás, az 1 byte kevésnek tűnik. A 2 byte az elbizonytalanít: egy nagyobb projekt esetén nagyon sok metódus lehet, úgy érezzük, lehet, hogy kevés lesz a 60 000-es korlát. A 4 byte-os nyilván elég egy nagyon nagy projekthez is. Válasszuk most a továbbiakban a 2 byte-os esetet az egyszerűség kedvéért. Ekkor a DMT tábla minden bejegyzése  $2 + 4 = 6$  byte-os (a metódus sorszáma 2 byte, a memóriacíme pedig még mindig 4 byte). Ezen felül minden DMT tábla tartalmazza az ősosztály DMT táblájának címét, az még plusz 4 byte.

Ennek megfelelően:

- a proba1 VMT táblája 2 bejegyzés = 8 byte,  
DMT táblája 2 bejegyzés  $4 + 2 \times (2 + 4) = 16$  byte,
- a proba2 VMT táblája 4 soros = 16 byte,  
DMT táblája 3 soros  $4 + 3 \times (2 + 4) = 22$  byte,
- a proba3 VMT táblája 7 soros = 28 byte, a  
DMT táblája 4 soros,  $4 + 4 \times (2 + 4) = 28$  byte.

Láthatjuk tehát, hogy a DMT módszerével nem feltétlenül használunk el kevesebb memóriát. Nyilván példafüggő, jelen példákban aránylag sokszor vezettünk be új virtuális metódusokat, és sokszor írtuk őket felül. Természetesen a DMT nagyságrendekkel kevesebb memóriát „fogyaszt” ha ilyen szempontokból egy gyerekosztály keveset változik.

Lássuk a második szempontot, a futási sebességet. A VMT használata esetén a késői kötés két lépéses volt: olvassuk ki a táblázat előre megadott sorában lévő értéket (1 memóriaolvasási művelet), és máris ugorhattunk a megfelelő függvényre. A DMT esetén nem tudni, futás közben, melyik sorban van az információ, sőt, még azt sem, hogy egyáltalán benne van-e a táblázatban.

A táblázat tartalmazza a metódusok azonosító sorszámát. A fordító tudja fordításkor, amikor a késői kötésre kell kódot generálnia, hogy mi a szóban forgó metódus azonosítója (jelöljük M-mel a sorszámot). Ezért olyan kódot generál, amely:

- keresd meg az aktuális példány DMT táblázatában az M sorszámú metódust,
- ha nem találod, lépj az ősosztály DMT-jére, és folytasd az első lépéstől.

A kereséshez alapvetően kétféle algoritmust használhatunk. Az első a szekvenciális keresés, amikor elindulunk a táblázat első sorától, szépen sorban haladunk a táblázat vége felé, és közben nézzük, hogy megtaláltuk-e a keresett sorszámot. Ez nyilván elég lassú módszer.

A második módszer a bináris keresés. Ehhez szükséges, hogy a táblázat sorba legyen rendezve a metódusok sorszáma szerint. Ez megoldható, a fordítóprogramnak van ideje így fordítási időben rendezni a DMT táblát.

A keresés a felezéses módszer szerint a középső táblabejegyzéssel kezdődik. Ha nem találtuk meg a keresett sort, akkor a sorszám alapján vagy az alsó felében, vagy a felső felében folytatjuk a keresést. Szintén felezzük a még nem ellenőrzött táblázatrészt, és újra megtudjuk, melyik részben kell folytatni a tevékenységet. Addig tudjuk ismételni, míg vagy meg nem találjuk a keresett bejegyzést, vagy el nem fogyott a felezgetés során a táblázat – ekkor levonhatjuk a következtetést, hogy a táblázat nem tartalmazza a keresett metódus sorát.

A bináris keresés jól használható tehát, de emellett is érezhető, hogy a keresés jóval több időt vesz igénybe, mint a VMT esetében. Sőt, változó mennyiségű időt, hiszen nagyon szerencsés helyzetben az első felezés során akár meg is találhatjuk a keresendő metódust, ellenkező esetben sok táblázatot kell visszafele haladva végignéznünk amíg megtaláljuk a hívandó metódus sorát.

Összefoglalva:

- A VMT annak ellenére, hogy elég terjedelmes, nem feltétlenül foglal le több memóriát mint a DMT.
- A DMT általában akkor foglalhat le kevesebb memóriát mint a VMT, ha kevés módosítás történik az osztályokban.
- A DMT szinte biztosan jóval lassúbb működést garantál futás közben.

Mindkét módszerrel megoldható tehát a késői kötés. Ezzel együtt a legtöbb programozási nyelv a VMT táblás módszert használja, mivel annak futási ideje kiszámíthatóan gyors, a memóriapazarlás pedig kevésbé fontos szempont manapság, amikor 4 GB RAM 5 ezer forint alatt van.

Egyes programozási nyelvek<sup>27</sup> azonban meghagyják a programozónak a döntést, melyik módszer szerint kívánja valamely metódusát kezelni. Választhat a VMT és a DMT táblabeli kezelési mód között. Ekkor:

- Ha a tervezés során az az érzés, hogy a metódust ritkán fogják a gyerekosztályok felüldefiniálni, érdemesebb DMT-be rakni. Ugyanis ha a gyerekosztályok állandóan felüldefiniálják azt, akkor az minden DMT-be be fog kerülni 6 byte helyfoglalással, míg a VMT táblában csak 4 byte lenne.
- Ha a metódust gyakran hívják meg (pl. ciklus belsejében), akkor érdemesebb a VMT-be rakni, hiszen az garantálja a lehetőségekhez képest a maximális futási sebességet<sup>28</sup>. Ha a metódust ritkán hívják, akkor választható a DMT módszer is.

---

<sup>27</sup> Ilyen nyelv pl. a Borland Delphi 7 (2002), ahol a 'dynamic' és a 'virtual' szavak segítségével lehet a választást leírni. A gyerekosztályok már minden esetben az 'override'-ot használták, mivel a döntés később már nem módosítható, nem változtatható meg.

<sup>28</sup> A legnagyobb futási sebességet egyébként a korai kötés adja.



## 20. Partial Class

---

Mikor csapatban dolgoznak az OOP programozók, többféleképpen is el tudják osztani egymás között a munkát. Egyrészt a tervezők elkészítik azon objektumosztályok terveit, amelyekből a végén a teljes program összeállítható lesz. Meghatározzák, milyen metódusok legyenek, milyen paraméterrel és feladattal. Generálhatják az osztályok vázlatos kódját, amiben már benne vannak a metódusok, de egyelőre üres törzsszel. A fejlesztők elosztják egymás között ki, melyik objektumosztályt kezdi el fejleszteni, hogy az üres törzsek kóddal töltődjenek fel.

De mi van akkor, ha van egy hatalmas objektumosztály, sok-sok metódussal, propertyvel, feladattal. Tud ezen több programozó is dolgozni egyszerre?

A válasz általában nem, legalábbis nem könnyű. Az osztály forráskódja ugyanis végső soron egy text file, és kevés fejlesztésköz engedi azt meg, hogy ugyanazon text file-on több editor program is dolgozzon.

Megoldható oly módon, hogy az egyik programozó dolgozik az ősosztályon, a metódusok nagy részét abstract-tá minősíti, a második programozó pedig a gyerekosztályokon dolgozik, csak az abstract metódusokat dolgozza ki. Így technikalilag két osztály van, két forráskód. Nyilván a program többi része a gyerekosztállyal fog dolgozni, hisz ő már nem abstract, ő már minden metódust kidolgozva tartalmaz. Probléma, hogy az abstract metódusok mindig virtuálisak is, így tehát a korai kötésű nem virtuális metódusokat nem lehetne a gyerekosztály programozójára bízni.

Hogy egyszerűsítsék a folyamatot, a C# nyelvben bevezettek egy új fogalmat, mely nagyon könnyen megérthető. Valamely osztály kódját kettévághatjuk (vagy akár még több felé is), a részeket elhelyezhetjük más-más forráskódba is akár. A lényeg, hogy értesítsük a fordítót erről a tényről, azért, hogy számítson erre, és össze tudja forrasztani a teljes forráskódot a részekből. Ehhez a 'class'-t meg kell jelöni a 'partial' (részleges) kulcsszóval:

forraskod1.cs:

```
namespace elolenyek.haziallatok
{
    partial class kutya : allat
    {
        protected double sulya;
    }
}
```

forraskod2.cs:

```
namespace elolenyek.haziallatok
{
    partial class kutya : allat
    {
        public void sulyBeallit(double sulya)
        {
            this.sulya = sulya;
        }
    }
}
```

---

Ekkor két programozó is tud dolgozni ugyanazon az osztályon, egyikük az egyik forráskódon, másikuk a másik forráskódon. Nincs szükség ehhez ősosztály-gyerekosztály viszonyt kialakítani, abstract metódusokat alkalmazni.

*Megjegyzés:* elvileg előfordulhat, hogy az osztály mindkét része ugyanabban a forráskódban van. A gyakorlati életben kevésbé jellemző eset, de elvileg nem hiba.

Másik tipikus alkalmazása a partial class-nak, amikor az osztályon egy programozó dolgozik – meg egy másik program. Az osztály bizonyos részei, elsősorban a mezők ismeretében a property-k, a konstruktorok, de sok esetben bizonyos feladatú metódusok is generálhatóak egy szoftveres eszköz segítségével. Az OOP tervezők gyakran használnak UML tervező eszközöket az objektumosztály tervezéséhez. Ezek grafikus felületű eszközök, nyomon követhető az osztályok egymással való kapcsolata is, a példányok interakciói (melyik példány melyik másik példány milyen metódusait hívja meg stb.).

Az ilyen felületeken bevitt terv információk alapján generálhatóak az osztály forráskódjának számos része. Generálhatóak a mezők (név, védelmi szint), és property-eket is létrejönnek hozzá. A szoftver által írt helyes C# forráskód az osztály ezen részét tartalmazza, partial class-ként kerül be a teljes projekt forráskódjába. Az osztály nem generálható részét a programozó írja egy másik forráskódban, szintén partial classként. Ez azért nagyon jó, mert ha az UML felületen módosítanak egy részletet, az képes újra generálni a saját kódrészét anélkül, hogy a programozó által beleírt kódot összezavarná.

Nagyon fontos két információ:

- Ahhoz, hogy a fordító ténylegesen össze tudja fésülni a két forráskódrészt, az osztálynak mindkét helyen ugyanabba a névtérbe kell esnie.
- Ha az osztálynak ősosztálya is van, elég csak az egyik helyen feltüntetni azt. Ha mindkét helyen feltüntetjük – értelemszerűen egyformát kell írunk, különben szintaktikai hibának tekinti a fordító.

forraskod1.cs:

```
namespace elolenyek.haziallatok
{
    // itt fel van tüntetve az ősosztály
    partial class kutya : allat
    {
        protected double sulya;
    }
}
```

forraskod2.cs:

```
namespace elolenyek.haziallatok
{
    // itt már nincs feltüntetve az ősosztály
    partial class kutya
    {
        public void sulyBeallit(double sulya)
        {
            this.sulya = sulya;
        }
    }
}
```

## 21. Destruktorok

---

A konstruktornak neveztük a példány létrehozásakor először lefutó metódust. Néhány szintaktikai megkötéstől és feladatuktól eltekintve tulajdonképpen hagyományos metódusok. Gyakran merül fel igény azonban olyan metódusokra is, amik a példány *életének megszűnésekor*, lezárásképpen futnak. Ezek a metódusok a példány által elkezdett tevékenységet zárják le. Ilyen eset lehet, ha az operációs rendszert is bevonjuk a példány tevékenységébe. Leggyakoribb esetek:

- A példány nyomtatásba kezd, az operációs rendszernek szól a kezdéskor, aki a nyomtatási sorhoz adja a folyamatot, de csak akkor kezdődik el a nyomtatás, ha a példány jelzi a folyamat sikeres végét is (befejezés), vagy megszakítását.
- A példány fájl nyitott meg írásra: az operációs rendszer ekkor más folyamatok részére nem engedélyezi ugyanezen fájl megnyitását<sup>29</sup>.
- A példány hálózati portot nyitott, hogy más számítógépekről kapcsolódást és adatokat fogadhasson.
- A példány más számítógép valamely portjára csatlakozott, hogy a két program adatokat cserélhessen egymással.
- A példány nagy mennyiségű memóriát allokált, nem példányosítás, hanem operációs rendszerhívás révén<sup>30</sup>.
- A példány külső adatforráshoz (pl. SQL szerver) csatlakozott, ott esetleg tranzakciós folyamatba kezdett, és jelezni kell a tranzakció végét vagy a csatlakozás lezárását.

A bezárást végző metódus korábban nem volt elkülönítve az általános célú metódusoktól. Csak a programozó tudta, hogy a *Destroy()* vagy *Finish()* nevű metódusának valójában ez a feladata. Hívása is manuálisan (explicit módon) történt: amikor a program futása során szükség volt rá, a hívását beépítették a megfelelő ponton a forráskódba. Az explicit hívás gyakran okozott problémát:

- Túl korán hívták meg, a folyamatot a metódus lezárta, de mivel maga a példány tovább létezett, a program egy későbbi pontján úgy használta mintha a folyamat még nem szakadt volna meg (ez futási hibát okoz).
- Túl későn hívták meg, a folyamatot már korábban le lehetett volna zárni. (Az optimális időpontbeli lezárás azért fontos, hogy az erőforrásokkal az operációs rendszer minél hatékonyabban tudjon gazdálkoni.)
- Egyáltalán nem hívták meg: a program egyszerűen elfelejtette a lezárást meghívni, pl. mert egy elágazás olyan ágán volt a lezárás, amelyre nem került rá a vezérlés.

<sup>29</sup> A legáltalánosabb eset, bár létezik olyan megnyitás is, amikor ezt megengedjük.

<sup>30</sup> Ha példányosítással foglal le memóriát, akkor a GC fel fogja szabadítani, így nem lenne szükség a befejezés jelzésére

---

Ezek közül a túl korai lezárás a legveszélyesebb. Ugyanis jellemzően futási hibát okoz, a program egy későbbi, távoli pontján (ahol még mindig használnánk). E ponttól visszafejteni, hogy mikor is történt a lezárás ami a hiba valódi oka – sokszor nem könnyű feladat, de mindig nagyon időigényes.

A hibáknak sokszor az az oka, hogy a példányváltozók igazából csak a példány memóriacímét tárolják. A memóriacímek sok változóba bekerülhetnek egyszerű értékadások segítségével, listákhoz adhatjuk hozzá, vektorokba tárolhatjuk el, metódushíváskor paraméterként adhatjuk át. Bármelyik változón keresztül lezárhatjuk a folyamatot, az magára a példányra vonatkozik, így a többi memóriacímen keresztül sem tudjuk később továbblépni a folyamattal. Ez is jelentősen megnehezíti a hibakeresést.

Miközben a módszer (explicit hívás) megmaradt, a programozók jelentős része igényelt erre a célra automatizmust. Ennek igénye először a memória foglalás-felszabadítás kapcsán merült fel. Az első időkben a nagyobb méretű összetett változóknak (tömbök, rekordok, listák, sorok, veremk stb.) memóriafoglalása külön történt, ahogy a felszabadítása is. A tapasztalat szerint a programozók általában nem foglalni felejtették el az objektumokat, hanem felszabadítani. Ennek során képződött az a jelenség, amit **memóriaszivárgásnak** (*memory leak*) neveztek el. Ez azt jelentette, hogy ha elindítottuk a programot, akkor induláskor még sok memóriánk volt, de ahogy a program futott, a memória egyre fogyott-fogyott, mintha egy lyukon (léken) keresztül elszivárgott volna. Hogy melyik program felelős ezért, hamar kideríthető, ugyanis egyszerű módon megvizsgálható melyik program mennyi memóriát köt le. A legtöbbet lekötő program volt a felelős (általában). Ha a programból kiléptünk, akkor hirtelen megint sok szabad memória lett, ugyanis az operációs rendszer a program bezárásakor (többé-kevésbé) felszabadítja az összes, a program által lekötött, de a futása során fel nem szabadított memóriát is.

A programok gyakori újraindításával tehát a memóriaszivárgás kezelhető, de nyilván nem tekinthető megoldásnak. Vannak programok (pl. szervereken futó szolgáltatások), amelyek újraindítása a szerver újraindításával együtt szokott megtörténni, illetve gyakori újraindításuk szolgáltatási zavarokhoz vezetne (gondoljunk pl. egy web szerverre). A problémát nyilván a hibás programozás okozza, így azt a program javításával kell megoldani.

A két probléma összekapcsolható, együtt kezelhető. Amikor a példányhoz tartozó memória felszabadítható (a példány megszűnik), akkor legkésőbb az általa nyitva hagyott folyamatok is lezárhatóak.

A memóriakezelésre a **Garbage Collecting** technikát fejlesztették ki. Erről később még lesz szó, de már most is ismerjük a lényegét: egy mechanizmus felfedezi mely memóriaterületek szabadíthatóak fel, és intézkedik azok tényleges felszabadításáról vagy újrahasznosításától. A GC a közvetlen felszabadítás előtt még megteszi azt is, hogy meghívja a példány egy kitüntetett szerepkörű metódusát. A metódus megvizsgálhatja, hogy a folyamat még mindig aktív-e, és intézkedhet annak megszakításáról. Ha ez a metódus végzett – a GC a példányhoz kötődő memóriát felszabadítja.

Hogy a GC ezt megtehesse, tudnia kell pontosan melyik az a metódus, amit meg kell hívnia. Nem lehet több ilyen, a GC nem fog válogatni közöttük. Ennek a metódusnak nem lehet paramétere.

A metódus, amelyet a GC a fentiekben ismertetett módon kezel, a **destruktor**. A destruktor tehát egy speciális feladatú metódus, speciális szintaktikával íródott (hogy a GC megismerje), s mivel implicit módon a rendszer hívja, nem lehet paramétere sem.

A destruktorra vonatkozó speciális szintaktikai szabályok:

- neve egyezik az osztály nevével,
- a név előtt egy hullámjel (~) szerepel,
- nincs védelmi szint megjelölve (private),
- nincs visszatérési típusa (void sincs megjelölve).

Szemantikai szempontból:

- a destruktor lefutása gyors, mert a GC-nek a lefutás idejéig várakoznia kell, ami nem célszerű,
- ellenőrzi, hogy van-e a példány működésével kapcsolatosan valami bezárandó, megszakítandó lépés,
- ha igen, kezdeményezi a megszakítását (lezárást).

Állandó és ismétlődő megfogalmazási hiba, hogy a destruktor feladata a memória felszabadítása. Mint láthattuk, a memória felszabadítása valójában a GC feladata. Persze a mondat lehet igaz – amennyiben ténylegesen azért írtuk, hogy a példányhoz tartozó extra memóriát felszabadítsuk.

Meg kell értenünk, hogy itt valójában két „világ” van. Az egyik az operációs rendszer világa (hideg és sötét, kegyetlen, procedurális világ). Az operációs rendszerünk programozási modellje a Win32, mely sok ezer függvényhívásból áll, kedvenc fogalma a „handle”, és a szereti használni a 16 bites hexadecimális konstansokat, amiket „flag”-eknek hív, és amikből szintén több ezernyi van. Ha itt kívánunk memóriát foglalni, akkor használhatjuk pl. a Marshal osztály AllocHGlobal() metódusát<sup>31</sup>, amelynek paramétere a lefoglalandó terület nagysága bájtokban. A másik világ a .NET Framework világa (világos, kedves, barátságos), ami OOP alapú. Itt a 'new' segítségével foglalunk memóriát, amely önállóan kalkulálja ki a szükséges memóriát.

Fontos tudnivaló, a GC csak és kizárólag a 'new'-val lefoglalt memóriaterületekkel foglalkozik, azokat szabadítja fel. Amelyeket a Marshal osztály segítségével foglalunk le, azokat nem. Ha tehát ilyet teszünk, a destruktor belsejében szereplő memóriefelszabadító függvényhívás, a Marshal.FreeHGlobal(...) segítségével végezzük el. Ekkor elmondhatjuk, hogy a destruktorunk feladata a memória felszabadítása. De ne feledjük: a destruktor sosem a példányhoz kötődő memóriát szabadítja fel.

<sup>31</sup> Ez nagyjából megfelel a C nyelvi szokásos malloc függvénynek.

```
class externalRecord
{
    protected IntPtr extraMem;

    // konstruktor
    public externalRecord()
    {
        // 200 byte foglalás
        extraMem = Marshal.AllocHGlobal(200);
    }

    // destruktor
    ~externalRecord()
    {
        // ha meg nincs felszabadítva
        if (extraMem != IntPtr.Zero)
            Marshal.FreeHGlobal(extraMem);
    }
}
```

### 21.1. Ha nem írunk destruktor

A destruktor fogalma már a C# nyelv tervezése előtt létezett. Emiatt a C# nyelvbe is bekerült ez a konstrukció. Ugyanakkor érdekesség, hogy valójában nincs destruktor a háttérben. Helyette az Object ősétől örökölt virtuális **Finalize()** metódust írjuk felül, amikor destruktor írunk az osztályunkba. Vagyis a fenti kód olyan, mintha az alábbi írtuk volna:

```
class externalRecord
{
    // ...
    // destruktor
    public override void Finalize()
    {
        // ha meg nincs felszabadítva
        if (extraMem != IntPtr.Zero)
            Marshal.FreeHGlobal(extraMem);
    }
}
```

Ebből következik, hogy ha nem írunk destruktor egy osztályhoz, akkor is „van” destruktor, hiszen a GC valójában a Finalize()-t hívja, amiből pedig legalább egyet örököl az Object-től.

### 21.2. Mikor ne írjunk destruktor?

Van még egy nagyon fontos tudnivaló a GC működésével kapcsolatban. A GC egy időben nem egyetlen példánnyal foglalkozik. Az idejének nagy részét a felszabadítható példányok felfedezésével tölti, és összegyűjti a felfedezett példányok memóriacímeit. Ezen felül, amikor egy példány felszabadíthatóvá válik, általában nagyon sok más példány is

felszabadítható válik vele egyidőben. Különösen igaz ez a lista példányokra, amelyek sok más példány memóriacímét tárolják. Ha magát a lista memóriacímét elveszítjük, általában a benne tárolt példányok is felszabadíthatókká válnak. Nagyon gyakori eset az is, hogy a példányunk a mezőiben más példányok memóriacímeit tárolja (amennyiben az adott mező típusa a referencia típuscsaládba sorolható).

Amikor a GC felszabadítási állapotba lép, akkorra már általában összegyűjtött sok példány memóriacímét. A GC szemszögéből nézve ezek mindegyik egyforma, mindegyik felszabadításra vár. Nincs közöttük sorrendiség, nincsenek rendezettségi szempontok. Másképp fogalmazva: a GC szabadon dönt ezeket milyen sorrendben szabadítja fel, a viselkedése ilyen szempontból nem determinisztikus, nem kiszámítható.

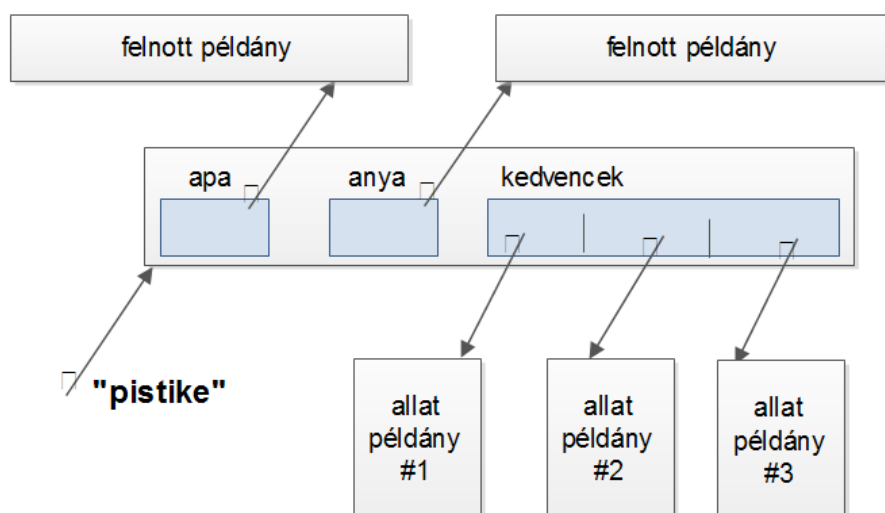
Vegyük az alábbi elképzelt esetet: egy gyerek osztályban tároljuk a szülők (apa, anya) referenciáit, valamint egy listában a kedvenc állatainak (kutya, cicák) címeit is:

```
class gyermek
{
    protected felnőtt apa;
    protected felnőtt anya;
    protected List<allatok> kedvencek = new List<allatok>();

    ~gyermek()
    {
        Console.WriteLine("búcsúsom {0} mama drága", anya.neve);
    }
}
```

Azt gondolnánk, hogy a fenti kód helyes, nincs vele semmi gond. Elképzelhető, hogyha lefuttatjuk, még működik is, kiíródik pl. "búcsúsom Jolán mama drága". Más esetekben azonban nem íródik ki. Semmi sem íródik ki.

Vizsgáljuk meg, hogyan néz ki a memória:



Amikor elveszítjük Pistike gyermek memóriacímét, elképzelhető, hogy az apa és az anya objektumpéldányokat is elveszítjük, ha azok memóriacímeit máshol már nem tároljuk. Elveszíthetjük vele egyidőben sok állatka memóriacímét is. Két lehetőség van:

- a GC előbb a gyermek példányt szabadítja fel, s csak utána az „anya” mezőbeli példányt, vagy
- a GC előbb szabadítja fel az „anya” mezőben tárolt felnőtt példányt, s csak utána lát neki a gyermek példány felszabadításának.

Az első viselkedés számunkra természetesebbnek tűnik, hiszen a gyermek példány memóriacímét tényleg nem ismeri már a program, de az anyára még van hivatkozás a gyermekből. De a GC-nek ez már mindegy, a felesleges példányok egymás közti viszonyát nem vizsgálja. Nem is vizsgálhatja, hiszen elképzelhető lenne, hogy az anya példányban még van hivatkozás a gyermekekre:

```
class felnőtt
{
    protected List<gyermek> gyerekek = new List<gyermek>();
    // ...
}
```

Ekkor a GC könnyen bajba kerülhetne, hiszen nem tudna döntenit milyen sorrendben haladjon. Legegyszerűbb: nem is foglalkozik ezekkel a belső hivatkozásokkal. Tehát ami nekünk természetes és érthető gondolat – az sajnos nem a valóság most. De a GC kétségtelenül sorrendben halad a megsemmisítések során, így előfordulhat az az eset, hogy előbb veszi sorba a gyermek példányt, és csak utána a felnőtt példányt. Ez esetben a destruktor belsejében szereplő `Console.WriteLine` még hibátlanul lefut, el tudja érni az `'anya.neve'` mezőt.

Második esetben azonban mire a gyermek sorra következik, a felnőtt példány már felszabadításra került. A gyermek `'anya'` mezőjében még van egy memóriacím (ahol korábban a felnőtt példány volt), de az már nem érvényes (*invalid*). Ekkor a `Console.WriteLine`-beli hivatkozás a `'neve'` mezőre futási hibát generál. A GC észleli, ennek ellenére befejezi amibe belekezdett: felszabadítja a példányhoz kötött memóriát.

Fontos szabály: a példány mezőiben tárolt referenciák a destruktor belsejében már nem feltétlenül érhetők el, elképzelhető, hogy a hozzájuk tartozó memóriacímen tárolt példány már korábban feldolgozásra került a GC által. Csak az érték típusú (*value type*) mezőkre hivatkozhatunk a destruktor belsejében.

Felmerülhet bennünk a gondolat egy, az előzőhöz hasonló napló író osztály készítésekor a destruktorban bezárni a fájlt. Tegyük fel, hogy a példány a konstruktorában megadott nevű fájlt nyitja meg mint naplófájlt, biztosítja a lehetőséget az írásra, egy metódust a



állomány bezárásához, de amennyiben azt elfelejtenék meghívni, úgy a destruktorban szeretné bezárni a fájlt.

```
class fileLog
{
    protected StreamWriter w = null;
    public fileLog(string filenev)
    {
        this.w = new StreamWriter(filenev, true, Encoding.UTF8);
    }

    public void writeln(string esemeny, params object[] parms)
    {
        if (w!=null)
            this.w.WriteLine(esemeny, parms);
    }

    public void bezar()
    {
        if (this.w != null)
            this.w.Close();
        this.w = null;
    }

    // destruktor
    ~fileLog()
    {
        bezar();
    }
}
```

Ugyanakkor gondoljuk végig: ha a 'fileLog' példányunk memóriacímét elveszíti a program, ugyanakkor veszik el a StreamWriter példány is, hiába szerepel a 'w' mezőben a memóriacíme. Hogy milyen sorrendben kerülnek a GC által feldolgozásra, az nem determinisztikus. Vagyis elképzelhető, hogy a 'w' mezőbeli memóriacím nem 'null', de az a memóriacím mégis érvénytelen. Ekkor a destruktor, aki meghívja a 'bezar()' metódust, ott mégis futási hibát fog okozni, hisz a 'w.Close()' már nem végrehajtható kód.

Nem lényeges mi történik azokkal a példányokkal, amelyek a mi példányunkhoz vannak csatolva. Kénytelenek magukról gondoskodni, hiszen a fő példány destruktorában már nem tud rájuk határozni, nem tudja értesíteni őket, hogy ő maga megszűnik, így a csatolt példányokra sem lesz már többet szükség.

### 21.3. Mikor írjunk destruktort?

Az eddigi ismeretek alapján a címbeli kérdésre már könnyen lehet választ adni. Szinte sosem lesz szükségünk destruktork írására. Amíg a feladatokat más .NET Framework példányok segítségével oldjuk meg, addig nem:

- ha portot akarunk nyitni, példányosítsuk meg a `TcpListener` osztályt,
- ha kapcsolódni akarunk más számítógépekhez, példányosítsuk meg a `TcpClient` osztályt,
- ha fájlt akarunk olvasni, használjuk a `StreamReader` osztály valamely példányát,
- ha fájlt akarunk írni, példányosítsunk a `StreamWriter` osztályból,
- ha nyomtatni szeretnénk, a `PrintDocument` class példányra szükségünk.

*Megjegyzés:* a fenti feladatokat általában más módon is megoldhatjuk, más osztályok segítségével, a fentiek csak példák voltak.

Vagyis a legtöbb probléma a C# nyelven a .NET Framework valamely osztályának példányosításával megoldható. Mint láthattuk, más példányokra a destruktorkban nem lehet hivatkozni, hiszen a memóriacímük lehetséges, hogy már nem használható. Ezért destruktork írása nemcsak hogy nem érdemes, de általában nem is lehetséges.

Kivételt képez ha nem a .NET Framework valamely példányát vesszük használatba, hanem külső, nem managed kódokat használunk, pl. a Win32 környezet függvényeit. Ilyenkor gyakran kell tárolnunk a példányunk mezőiben olyan memóriacímeket vagy `handle`<sup>32</sup> értékeket (leggyakrabban az `IntPtr` osztály példányainak segítségével), melyeket a GC nem kezel, így felszabadításuk nem történik meg automatikusan. Ekkor érdemes a destruktorkban a lefoglalt memóriát felszabadítani, a `handle`-kre meghívni a megfelelő `Close()` függvényt.

---

<sup>32</sup> A „handle” fogalmának nem igazán létezik jó magyar fordítása – talán a „kezelő” megfelelő lehet. Ez egy numerikus érték, mely az operációs rendszerben egy erőforrást azonosít. A `handle`-t azért adja át az operációs rendszer a programnak, hogy később hivatkozhasson az adott erőforrásra. Ezért sokban hasonlít a memóriacímekre, bár a `handle` értéke maga nem tekinthető memóriacímnek, inkább hasonlítható egy tömbindexre. Úgy képzelhető el, hogy a tömböt magát az operációs rendszer tárolja, így maga az indexérték a program számára valójában semmitmondó, de az operációs rendszerbeli (Win32) függvényeknek átadva ők hozzá tudnak férni ehhez a tömbhöz, és tudnak dolgozni az adott erőforrással.

## 22. Generic

---

Vizsgáljuk meg alaposan az általános célú összetett adattípusainkat. A legegyszerűbb a vektor. Minden vektor egyforma abban az értelemben, hogy mindegyiknek van `.Length` propertyje, amely megmondja hány elemű a vektor. Mindegyik vektor 0-tól indexeli az elemeit, mindegyik vektorban lehetőségünk van adott indexű elem értékét megadni vagy lekérdezni. Miben különböznek a vektorok? Az elem adattípusában! Az elemek adattípusát a vektor definiálásakor meg kell adni:

```
int[] tt = new int[20];
```

Hasonló a listák esete: minden listában van `.Add(...)` metódus, a lista meg tudja mondani saját elemszámát a `.Count` property segítségével, le lehet kérdezni tetszőleges sorszámu elem értékét stb. Miben különböznek ezek a listák egymástól? A lista elemtípusában!

Hogyan definiáljunk listát? A lista elemtípusa érdektelen a `.Count` szempontjából, de nem érdektelen az `.Add(...)` metódus, az indexelő szempontjából:

```
class sajátLista
{
    public abstract void Add(??? t);
    public abstract ??? this[int i] { get; set; }
}
```

Ez az a probléma, ahol az absztrakt sem segít, hiszen a ??? helyére konkrét típusnevet kell beírni még akkor is, ha sem magát az `Add(...)` törzset, sem az indexelő törzset megírni nem tudjuk csak a konkrét típus esetén.

De még az absztrakt esetén is meg kell adni a konkrét paraméterezést, a típusok neveit, hiszen az override kapcsán azt már módosítani nem szabad. Emiatt valójában nem tudunk oda beírni semmit, hiszen ha beírunk bármit is (pl. `int`), akkor ebből már sosem lesz `double` lista.

A megoldást a **generic** típus adja. A generic (általános) típus egy olyan osztály, amelyről sok mindent tudunk, lehet, hogy minden metódusát és property-jét meg tudnánk írni – csak egy dolgot nem tudunk: milyen típussal kell, hogy dolgozzon. Ekkor ezt a típust az osztály paramétereként jelöljük meg. Az osztály paramétereit az osztály deklarálásakor az osztály neve után adjuk meg, nem gömbölyű zárójelben ahogy általában a metódusok paraméterezését adjuk meg, nem is szögletesben (az az indexelő paraméterezése), hanem „kacsacsőrök” között:

```
class sajátLista<T>
{
    /* ... */
}
```

A fenti azt jelöli, hogy a 'sajatLista' osztálynak van egy típusparamétere, melyet a továbbiakban T-nek nevezünk. Hogy a T típus pontosan milyen típust jelöl, az majd csak példányosításakor fog kiderülni:

```
sajatLista <double> ll = new sajatLista <double>();
```

Ebben az esetben a 'T' konkrétan a 'double' típus lesz (T = double). Térjünk vissza az osztály megírására:

```
class sajatLista<T>
{
    public abstract void Add( T t);
    public abstract T this[int i] { get; set; }
}
```

A 'T' típusra a készülő osztályban tetszőleges helyen hivatkozhatunk. Szerepelhet mint

- metódus paraméterének típusa (mint az Add(...) esetén),
- vagy mint visszatérési típus (mint az indexelő esetén),
- mint mező vagy konstans típusa,
- de akár metódus törzsében is szerepelhet, a ilyen típusú változót szeretnénk deklarálni.

Az osztály ilyenkor már szintaktikailag teljes és hibátlan lehet (holott a T konkrét értéke nem ismert fordításkor). Ugyanakkor működő kódot csak akkor lehet majd generálni, ha a T konkrét típusa (példányosításakor) kiderül.

Hogyan oldjuk meg a fentiekhez hasonló problémát olyan programozási nyelveken, ahol a generic típus nem létező fogalom? Hasonlóan. Megírjuk az osztályt egy konkrét típusal:

```
class sajatLista_double
{
    public abstract void Add(double t);
    public abstract double this[int i] { get; set; }
}
```

Majd ha más típussal is működnie kell az osztálynak, akkor kijelöljük a forráskódot, copy-paste, és minden 'double' előfordulást kicserélünk a másik típusra (pl. 'kacsa'):

```
class sajatLista_kacsa
{
    public abstract void Add(kacsa t);
    public abstract kacsa this[int i] { get; set; }
}
```

Ha megint másik típus kellene (pl. `int`) akkor ugyanúgy járunk el, kijelölés, copy-paste, csere:

```
class sajátLista_int
{
    public abstract void Add(int t);
    public abstract int this[int i] { get; set; }
}
```

A módszer működni látszik (nem kell generic), de:

- a „csere” során hibát véthetünk. Például ha az utolsó változatot nézzük, az indexelő paramétere is `int`, akárcsak a visszatérési típusa. Ha ezt a változatot másoljuk le, majd próbálunk minden `'int'`-et kicserélni az új típusra (pl. `'diak'`) ügyelni kell, hogy nem minden `'int'`-et kell cserélni, nehogy ezt kapjuk:

```
public abstract diak this[diak i] { get; set; }
```

- az osztály forráskódját így birtokolnunk kell, hogy a copy-paste működhessen. Azt jelenti, hogy a kész osztály forráskódját is oda kell adnunk a többi programozónak, akik használni akarják, ami nem feltétlenül előny.
- Ha már sok másolatot készítettünk az eredeti osztályról, de abba bővíteni szeretnénk valamit, vagy hibát javítani, vagy újra meg kell ismételni a másolás-csere eljárásokat, vagy minden másolatba is át kell vezetni a módosításokat!

A fenti problémák mindegyikét megoldja a generic alkalmazása. Az osztály lefordítható (hisz szintaktikailag teljes és hibátalan), a lefordított változatot adhatjuk majd tovább a többi programozónak. Mivel a generic osztály forráskódja csak egyszer kerül megadásra, csak egy helyen kell bővíteni és hibát javítani.

Tudnunk kell: nemcsak egy paramétere lehet egy osztálynak, tetszőlegesen sok:

```
class sKetparamos< T,M >
{
    public M mezo;
    public sKetparamos(T a, M b)
    {
        this.mezo = b;
    }
    public M valami(T x)
    {
        M a = mezo;
        /* .. */
        return a;
    }
}
```

A példányosítás során kell megadnunk mindkét típust:

```
sKetparamos<int, double> f = new sKetparamos<int, double>(12, 44.3);
```

Ekkor a 'T' konkrétat 'int' lesz, az 'M' pedig a 'double', minden helyen behelyettesítésre kerül az osztály forráskódjában, és generálódik a konkrét (már minden szempontból ismert) osztály futásképes kódja.

Nemcsak osztály, akár metódus is lehet generic. A tipikus példa erre a csere (*swap*) metódus, mely generic módon megírva tetszőleges típusú változók között végrehajtja a cserét:

```
public void Swap<T>(ref T lhs, ref T rhs)
{
    T temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

Hívása során meg kell adni milyen típusú változók között kívánjuk a cserét végrehajtani. Ha int típust adunk meg, akkor természetesen a paramétereknek is int típusúnak kell lenniük:

```
int a = 1;
int b = 2;

Swap<int>(ref a, ref b);
```

Ravaszabb esetekben már hibát kaphatunk. Például szeretnénk megírni egy maximum keresést három paraméterre generic módon:

```
public T maximum<T>(T a, T b, T c)
{
    T x = a;
    if ( x<b ) x = b;
    if ( x<c ) x = c;
    return x;
}
```

Vegyük észre, hogy ha három double között szeretnénk majd használni a cserét, akkor a metódus is double értéket fog visszaadni, így a 'T'-t a metódus visszatérési típusaként is felhasználjuk. Kell egy segédváltozó, az 'x', melynek típusa szintén 'T' lesz. Sajnos a fenti kód fordítása nem lesz sikeres, mert a fordító nem hiszi el, hogy minden 'T' típus esetén végrehajtható lesz majd a függvény törzse, ugyanis nem minden típus esetén van összehasonlító operátor (< operátor) amit alkalmazhat. Például mi történne, ha lenne három 'kacsa' típusú változónk, k1, k2, k3 néven, és az alábbi módon hívnánk meg ezt a metódust:

```
kacsa m = maximum<kacsa>(k1, k2, k3);
```

Hogyan lehet két kacsát összehasonlítani, hogy melyik a nagyobb? Nem lehetetlen, dönthetünk például úgy, hogy a kacsák súlya lesz az összehasonlítás alapja. De ezen a ponton be kell lássuk, hogy a fordító nem lehet biztos abban, hogy mi ezt megtettük. Nem lehet biztos abban sem, hogy az 'x' és 'b' változó értéke összehasonlítható a < operátor segítségével.

Valójában azt is le kellene írunk a metódusnál, hogy „a metódus T paraméterrel dolgozik, de T csak olyan lehet, amelyre van kidolgozott < operátor”. Ha pontosan ezt nem is, de hasonló leírhatunk a 'where' (ahol) megszorítással:

```
public T maximum<T>(T a, T b, T c) where T: IComparable
{
    T x = a;
    if (x.CompareTo(b) < 0) x = b;
    if (x.CompareTo(c) < 0) x = c;
    return x;
}
```

Itt azt írtuk le, hogy a T típusról megköveteljük, hogy tegyen eleget az IComparable interface követelményeinek (az interface-ekről a 23. fejezetben lesz szó), amely jelen esetben azt jelenti, hogy garantáltan lesz neki 'CompareTo(...)' metódusa. Ezt ki is használjuk a metódus törzsében. A CompareTo olyan metódus, amely megadja a két elem esetén hogy az azok milyen módon viszonyul egymáshoz (pl. 'a.CompareTo(b)' alakban). A CompareTo -1-t ad vissza, ha a<b, 0-t, ha a==b, és +1-t ad meg, ha a>b) Eképpen a CompareTo(...) pótolja a hiányzó összehasonlító operátort.

Legismertebb generic osztályok a System.Collections.Generic névtérben vannak:

- maga a lista osztály, mely a List<T> módon került megírásra.
- a kétparaméteres Dictionary<T,M>, amelyről az Indexelő fejezetben már írtunk,
- a Stack<T> amely egy univerzális verem,
- a Queue<T> amely egy univerzális sor adatszerkezet,
- LinkedList<T> amely nagyon hasonló a List<T>-hez, de a lista belső reprezentációja láncolt lista.

## 23. Interface

---

Induljunk ki a következő problémából: egy lista objektumot készítünk, amelyre objektumpéldányok kerülnek fel a program futása során. Amikor a felhasználó a programunkban a „mentés” gombra kattint, az objektumokat menteni kell (ezek változtak meg a program kezelése során). Mindaddig, míg a felhasználó nem dönt a mentésről, nem kell menteni. Ezért is történik úgy a programunkban, hogy amikor valami változik, akkor azt felrakjuk a listára.

A lista kezelését meg kellene oldanunk. Ennek során listaelemeket lehet hozzáadni a listánkhoz (ügyelni kell az egyediségre), majd egy ciklusban sorra elővenni. Mivel nem tudjuk, hogyan kell menteni az adott objektumot, megkérjük őt hogy mentse le ő saját magát – meghívjuk a `Save()` metódusát.

```
class mentesiLista
{
    protected List<object> l = new List<object>();
    public void Add(Object p)
    {
        if (l.Contains(p) == false)
            l.Add(p);
    }
    public void SaveAll()
    {
        foreach (Object p in l)
            p.Save();
        l.Clear();
    }
}
```

Két probléma van:

- az `Add(...)` esetén az `Object` mint paramétertípus nem megfelelő, hiszen nem fogadhatunk el bármit, csak olyan objektumot, amelynek van `Save()` metódusa,
- a `SaveAll()` metódusbeli `foreach`-nek is ugyanez a megkötése: az `Object` típusú 'p'-nek nincs `Save()` metódusa, ott szintaktikai hibát jelez a fordító.

Kezdjük az első problémával. Szeretnénk elérni, hogy az `Object` típus helyett olyan típust szerepeltessünk, amely tartalmaz `Save()` függvényt. Készítsünk ilyen osztályt:

```
class saveObj
{
    public virtual void Save()
    {
    }
}
```



Azt kérjük majd, hogy minden osztály, amelyet a listára akarnak rakni, származzon ebből a `saveObj` osztályból. Így biztosan lesz `Save()` metódusa, és a típuskompatibilitás miatt ezt a `saveObj` osztályt használhatjuk paramétertípusként is:

```
class mentesiLista
{
    protected List<saveObj> l = new List<saveObj>();
    public void Add(saveObj p)
    {
        if (l.Contains(p) == false)
            l.Add(p);
    }
    public void SaveAll()
    {
        foreach (saveObj p in l)
            p.Save();
        l.Clear();
    }
}
```

Az elképzelés tökéletes, a megvalósítással vannak problémák. Az elsőt magunk is észrevehetjük: értelmetlen a `'saveObj'` osztálybeli `Save()` metódusunk, hiszen nincs törzse (nem tudjuk megírni), és a gyerekosztályok hajlamosak lesznek elfeledkezni a metódus felüldefiniálásáról. Itt megint érdemes absztrakt osztályt használni, de a `'mentesiLista'` osztály emiatt semmilyen módon nem fog változni:

```
abstract class saveObj
{
    public abstract void Save();
}
```

Sokkal nagyobb probléma azonban, hogy erős megkötés a C# nyelvben, hogy egy osztálynak csak egy őse lehet. Ez azt jelenti, hogy a programban használt objektumok fejlesztésekor ezt meg kell adni kiinduló ősosztálynak. Ennek ismerete azonban a kezdetkor (ezen időpontban) egyáltalán nem biztosított. Tegyük fel, hogy a program mérnöki grafikus tervezőprogram, és a felhasználó által a sablonokból felrakott építőelemeket egy másik csapat programozta. Azok az objektumosztályok mit sem tudnak a mi `'saveObj'` osztályunkról, és nem is származnak belőle. Minden építőelem osztályban van `Save()` metódus, de az ősök között nincs ott a mi `'saveObj'` osztályunk.

Pontosítanunk kell a megfogalmazásunkat. Most az az elv, hogy „az `Add(...)` metódus paramétereként olyan példányt kell átadni, aki a `saveObj` osztállyal típuskompatibilis (gyerek-osztálya)”. Valójában mi azt szeretnénk volna leírni, hogy „az `Add(...)` metódusnak csak olyan példányt adjunk át, amelynek van `Save()` metódusa”.

Az ilyen jellegű leírások készítésére találták ki az interfaceket. Az interfacek:

- absztrakt osztályokhoz hasonlítanak: metódusok és property-k prototípusát<sup>33</sup> tartalmazzák csak, törzsét sosem,
- de nem számítanak osztálynak, vagyis a „csak egy őse lehet” elv alkalmazása esetén ők nem számítanak bele ebbe az „egy ős” fogalomba.

<sup>33</sup> A metódus prototípusa a fejrésze, vagyis a metódus neve, visszatérési típusa és a paramétereinek listája. A property prototípusa maga a property típusának neve, és, hogy get illetve set részek közül melyiket tartalmazza (esetleg mindkettőt).

Az interfacek pontosan azt írják le, amit a feladat elején megfogalmaztunk: „legyen neki Save() metódusa”. Az interface neve szokásosan nagy I betűvel kezdődik (hagyomány). Hasonlítsuk össze az absztrakt osztály és az interface definícióját:

```

abstract class saveObj
{
    public abstract void Save();
}

interface ISaveObj
{
    void Save();
}

```

Tehát a különbségek:

- 'class' helyett az 'interface' szót kell használni,
- nem kell a 'class' előtti 'abstract' jelzőt használni az 'interface' szó előtt,
- 'public' és egyéb védelmi szinteket nem kell (tilos) használni,
- a metódusok és property-k előtt nem kell az 'abstract' kulcsszó,
- míg az absztrakt osztály tartalmazhatott mezőket, addig az interface nem tartalmazhat,
- míg az absztrakt osztály tartalmazhatott kidolgozott metódusokat is (törzzsel együtt), esetleg kidolgozott property törzseket is, addig kidolgozott törzsek nem szerepelhetnek az interfaceben,
- az absztrakt osztályban lehetnek konstruktorok, destruktork, az interface-ben nem lehet,
- az absztrakt osztályban szerepelhetnek osztályszintű elemek (mezők, konstansok, metódusok stb.) addig az interface-ben ilyenek nem lehetnek.

Az interface tehát leír egyfajta követelménylistát, tartalma metódusok és property-k listája. Ugyanakkor az interface **egy típus** is, tehát alkalmas arra, hogy akár változókat deklaráljunk vele, paraméterek típusát adjuk meg, vagy vektor és lista alaptípusaként is szerepelhet:

```

class mentesiLista
{
    protected List<ISaveObj> l = new List<ISaveObj>();
    public void Add(ISaveObj p)
    {
        if (l.Contains(p) == false)
            l.Add(p);
    }
    public void SaveAll()
    {
        foreach (ISaveObj p in l)
            p.Save();
        l.Clear();
    }
}

```

Az interface mint típus ugyanakkor alkalmatlan arra, hogy példányosítsunk. A „cannot create an instance of the abstract class or interface ISaveObj” fordítása „absztrakt osztályból és interface-ből nem lehet példányosítani”.

```
ISaveObj p = new ISaveObj();
```

interface ektf.ISaveObj

Error:  
Cannot create an instance of the abstract class or interface 'ektf.ISaveObj'

Azonban az interface típus szerepelhet ősként valamely objektumosztálynál. Ekkor a típuskompatibilitás miatt az adott „gyerekosztály” kompatibilis lesz az interface típusal, és szerepelhet a fenti 'Add(...)' paramétereként.

```
class grafikusElem : grafikusAlap, ISaveObj
{
    /* ... */
}
```

Mivel az interface típus nem számít osztálynak, így szerepelhet az ősök listáján akár több interface is. Az adott „gyerekosztály” az összes interface-szel típuskompatibilis lesz.

Ennek persze ára van. Amennyiben egy osztály felveszi az ősei listájára valamely interface-t, úgy köteles az interface-ben definiált metódusokat és property-ket kidolgozni. A fordítóprogram ellenőrzi, és keményen büntet: szintaktikai hibásnak jelöli az osztályt és megtagadja a fordítást:

```
class grafikusElem : grafikusAlap, ISaveObj
{
    /* ... */
}
```

'ektf.grafikusElem' does not implement interface member 'ektf.ISaveObj.Save()'

Amikor egy interface-t felvesszünk az ősök listájára, és kidolgozzuk a követelményeket, azt mondjuk „implementáltuk az interface-t”. Ha valamely követelmény kidolgozatlan marad, akkor a következő üzenetet írja a fordító: „*grafikusElem does not implement interface member ISaveObj.Save()*”, vagyis „*a grafikusElem osztály nem tartalmazza a Save() elemét az ISaveObj interface-nek*”.

Több lehetőségünk is van ilyenkor:

- kidolgozzuk a Save() metódust,
- a grafikusElem osztály ősei listájáról eltávolítjuk az ISaveObj interface-t, ekkor nyilván abbamarad a hibaüzenet,
- absztrakt metódusként írjuk meg a Save() metódust.

Az utolsó lehetőség is érdekes. A fordítóprogram tudomásul veszi, hogy az osztály tartalmazza az interface által megkövetelt metódust (holott végül is nem). De ne feledjük, az absztrakt osztályokból nem lehet példányosítani, míg az összes absztrakt metódus kidolgozásra nem kerül. Így végül is a követelmény előbb-utóbb teljesül, a kötelezettség öröklődik.

A kérdés az, hogy a `Save()` metódus kidolgozása során mik a követelmények:

- az interface által előírt metódusok mindig **public** védelmi szinttel kell, hogy rendelkezzenek,
- lehetnek virtuálisak is, de nem követelmény.

Mivel az interface-ben nem szerepel sem az `abstract`, sem a `virtual` jelző a metódus előtt, így az osztályban, amikor a metódust kívánjuk megírni, nem szerepelhet előtte az `override`<sup>34</sup>:

```
class grafikusElem : grafikusAlap, ISaveObj
{
    public void Save ()
    {
        /* ... */
    }
}
```

Ugyanakkor előfordulhat az az eset is, hogy az őosztály már maga is tartalmazza a szóban forgó metódust, bár ő (valamiért) nem jelölte meg az ősei között az interface-t:

```
class grafikusAlap
{
    public void Save ()
    {
        /* ... */
    }
}
```

Ekkor a gyerekosztály, mivel már tartalmazza az interface által kért metódust (mert örökölte), neki csak annyi a dolga, hogy feltünteti az interface-t az ősei között, és élvezi a típuskompatibilitás által nyújtott előnyöket:

```
class grafikusElem : grafikusAlap, ISaveObj
{
    // örököltük a "public void Save()" -t
}
```

Amennyiben az őosztályban azonban a `Save()` `virtual` (vagy `absztrakt`) volt, és a gyerekosztály szeretné felülírni, úgy természetesen használhatja az `'override'` kulcsszót. De fontos tudnunk, hogy ez esetben sem az interface miatt kell az `'override'`, hanem az egyéb körülmények miatt!

```
class grafikusAlap
{
    public virtual void Save ()
    {
        /* ... */
    }
}
```

---

<sup>34</sup> Hacsak... példa később!

```
class grafikusElem : grafikusAlap, ISaveObj
{
    public override void Save()
    {
        /* ... */
    }
}
```

Bárhogyan is, de tegyük fel, hogy osztályunk sikeresen implementálta az ISaveObj interface-t. Elkészült a mentésekkel foglalkozó lista is. Ekkor már össze tudjuk kapcsolni a kettőt:

```
mentesiLista m = new mentesiLista();
// egy elem hozzáadása
grafikusElem p = new grafikusElem();
m.Add( p );
// további elemek hozzáadása
// ...
// majd minden mentése
m.SaveAll();
```

Az interface-ek alkalmasak tehát arra, hogy olyan objektumosztályok közötti összefüggéseket (egyformaságokat) írjanak le, amelyek az egyformaságot nem az öröklődési láncnak köszönhetik. Az, hogy az 'F15Tomcat' osztálynak van 'felszallas()' metódusa, azt mondjuk örökölte valamelyik ősétől (pl. a 'vadaszrepulo' osztálytól), míg a 'kacsa' osztálynak is van 'felszallas()' metódusa, de kicsi az esélye, hogy ő is a 'vadaszrepulo' osztálytól örökölte. Egy interface mégis leírhatja ezt az egyformaságot:

```
interface IFelszallas
{
    void felszallas();
}
```

Amennyiben mindkét említett osztály implementálja az interface-t, úgy az alábbi metódus számára mindkettő átadható:

```
static public void menekulj(IFelszallas p)
{
    p.felszallas();
}
```

A típuskompatibilitás miatt mindkét függvényhívás elfogadható:

```
F15Tomcat f = new F15Tomcat();
kacsa k = new kacsa();
//
menekulj(f);
menekulj(k);
```

Nézzünk egy példát az interface-es property-re is. Vegyünk megint egy listát, amely különböző súlyú tárgyakat képes tárolni (mint egy zsák). A listába csak olyan elemeket lehet elhelyezni, amelyeknek van súlya. A lista súlytároló kapacitása limitált, a konstruktorában lehet megadni. Definiáljuk előtte az interface-t:

```
interface ISulyossag
{
    double sulya { get; }
}
```

Majd a súlytárolós listát:

```
class sulyLista
{
    protected double maxSuly;
    protected double _aktSuly;
    protected List<ISulyossag> l = new List<ISulyossag>();
    public sulyLista(double maxSuly)
    {
        if (maxSuly <= 0)
            throw new ArgumentException("nem jo max suly");
        this.maxSuly = maxSuly;
        this._aktSuly = 0.0;
    }
    public void Add(ISulyossag a)
    {
        if (a.sulya + this._aktSuly < this.maxSuly)
        {
            l.Add(a);
            this._aktSuly += a.sulya;
        }
    }
    public double aktSuly
    {
        get { return _aktSuly; }
    }
}
```

Ekkor az Add(...) metódus csak olyan elemeket helyez el a listán, amelyeknek van lekérdezhető súlyuk (és csak ha a hozzáadandó elem súlyával nem lesz még túlsúlyos a lista). Azt használjuk ki, hogy ezeknek az elemeknek van 'sulya' property-je, aminek biztosan van 'get' része, tehát olvasható.

```
class kacska : haziallat, ISulyossag
{
    protected double _sulya;
    public double sulya
    {
        get { return _sulya; }
        set
        {
            if (value < 0 || value > 20.0)
                throw new ArgumentException("nem jo kacasuly");
            this._sulya = value;
        }
    }
    /* ... */
}
```

### 23.1. Generic interface-k

Az interface is lehet generic. Az előző példában a súly property, amelyet az interface előírt – kötelezően double. De mi van ha a súlyt inkább int-ként kívánjuk leírni?

```
interface ISulya<T>
{
    T suly { get; }
}
```

Ez esetben a kacska jelezheti, hogy ő azt a 'ISulya' interface-t implementálja, amelyik szerint a 'suly' property egész számként adja meg a súlyt:

```
class kacska : haziallat, ISulya<int>
{
    protected int _suly;
    public int suly
    {
        get { return _suly; }
    }
}
```

A 'sulyLista' generic-esen való megírása azonban már komoly problémákat vet fel:

```
class sulyLista<T>
{
    protected T maxSuly;
    protected T _aktSuly;
    protected List<ISulya<T>> l = new List<ISulya<T>>();
    public sulyLista(T maxSuly)
    {
        if (maxSuly<=0) throw new ArgumentException("nem jo max suly");
        this.maxSuly = maxSuly;
        this._aktSuly = 0;
    }
    public void Add(ISulya<T> a)
    {
        if (a.suly+this._aktSuly<this.maxSuly)
        {
            l.Add(a);
            this._aktSuly += a.suly;
        }
    }
    public T aktSuly
    {
        get { return _aktSuly; }
    }
}
```

A pirossal jelölt szintaktikai hibák okai, hogy a T típusról semmit sem tud a fordító. Nem tudja, hogy egy T típusú elem összehasonlítható-e 0-val, eldönthető-e hogy 0-nál kisebb-e vagy egyenlő-e vele. Egy T típusú \_aktSuly mezőben nem tudni, hogy lehet-e 0 kezdőértéket rakni. Nem tudni azt sem, hogy összeadható-e két T típusú elem, és megvizsgálható-e, hogy az összeg kisebb-e mint egy harmadik T típusú érték stb.

## 23.2. Interface-k öröklődése

Az interface-k között is van öröklődés, de interface őse csak másik interface lehet. E pillanattól kezdve minden teljesen hasonlóan működik, mint az osztályok esetében.

```
interface IRepules
{
    void felszallas();
    void leszallas();
}

interface IHarciRepules : IRepules
{
    void celzas(Object cel);
    void kiloves();
}
```

Nyilvánvaló, hogy az osztály, amelyik implementálni kívánja az IHarciRepules interface-t, annak bizony mind a négy metódust meg kell írnia. Ugyanakkor egyszerre lesz típuskompatibilis az IRepules interfésszel is.

## 23.3. IEnumerable és a foreach

A C# nyelven a foreach ciklusról tudjuk, hogy képes vektorokat és listákat használni, rendre felolvasva az elemeket. Valójában a foreach ciklus ennél univerzálisabb: bármivel képes együttműködni, amik implementálták az IEnumerable és IEnumerator interface-eket. Úgy kell elképzelni, hogy a bejárando objektumnak (lista, vektor, bármi) az IEnumerable interface-t kell implementálnia. Ez könnyű interface, a 'GetEnumerator()' függvényt kell csak megírni. A függvény által visszaadott objektumpéldánynak kell olyannak lennie, amely az IEnumerator interface minden függvényét tartalmazza (implementálta az IEnumerator interface-t). Az objektumot fogja a foreach dolgoztatni: a bejárás kezdetén elkéri tőle az első elemet, majd folyamatosan kéri a következőt.

Készítsünk el egy 'udvar' osztályt, amely egy  $20 \times 20$ -as mátrixban tartalmazza, hogy a házunk négyzet alakú udvarát ha egységnyi területnégyzetekre osztjuk, melyik területen éppen milyen háziállat tartózkodik. A területre mindenféle háziállat bekerülhet (kacsák, nyulak, tyúkok, malacok stb.), egy terület egység akár üres is lehet. Mindenféle támogató függvényeket készítünk, amivel adott területre el lehet helyezni állatokat, lekérdezni melyik típusból hány darab van, hol van a legközelebbi adott típusú állat stb. (Ezeket nem részletezzük.) A fő tároló elem egy 't' nevű protected mátrix lesz:

```
class udvarom : IEnumerable
{
    public haziallat[,] t = new haziallat[20, 20];
    /* ... tamogato metodusok ... */

    public IEnumerator GetEnumerator()
    {
        var p = new udvaromEnum(t, 20, 20);
        return p;
    }
}
```

Ha példány készül ebből az osztályból, és foreach-csel be akarjuk majd járni, a foreach első dolga lesz meghívni a GetEnumerator() függvényt, hogy megkapja a segítő példányt. A segítő példányt külön osztályba írtuk meg. Ennek a külön segítő példánynak át kell adnunk a  $20 \times 20$  mátrixunkat, megjelölve, hogy  $20 \times 20$  méretű (ugyanazt a példány is megtehetné, de



így egyszerűbb). Így az 'udvarom' példányok már elvileg alkalmasak a foreach ciklussal való együttműködésre:

```
udvarom u = new udvarom();
// feltöltés
foreach (haziallat p in u)
{
    /* ... */
}
```

Készítsük el a segédpéldány osztályát (amelyből a GetEnumerator() függvény fog példányosítani a foreach kérésére):

- Kell egy Reset() függvény, de ez csak a COM kompatibilitás miatt szerepel, a foreach nem használja.
- Kell egy MoveNext(), amelyet a foreach szisztematikusan meghív, hogy léptesse a bejárást. A MoveNext()-nek logikai true értéket kell megadni, ha sikerült (nem érte még el a végét). A foreach azzal kezdi, hogy meghívja a MoveNext()-et. Ha a bejárásbeli tároló (lista, vektor, mátrix) egyáltalán nem tartalmaz elemet, a MoveNext() azonnal false értékkel jelzi. Ekkor a foreach ciklusmagja egyszer sem fog lefutni (előltesztelő ciklus).
- Kell egy Current olvasható property, amely – miután a MoveNext() jelezte a true értékkel, hogy van még feldolgozható elem a tárolóban – kiolvassa ezt az elemet.

```
class udvaromEnum : IEnumerator
{
    protected haziallat[,] t;
    protected int N;
    protected int M;
    protected int i;
    protected int j;
    // .....
    public udvaromEnum(haziallat[,] t, int sor, int oszlop)
    {
        this.t = t;
        this.N = sor;
        this.M = oszlop;
        // kezdopos
        i = 0;
        j = -1;
    }
    // .....
    public void Reset()
    {
        throw new NotImplementedException();
    }
    // .....
    public bool MoveNext()
    {
        while (true)
        {
            j++;
            if (i >= N) return false;
            if (j >= M) { i++; j = -1; continue; }
            if (t[i, j] == null) continue;
            return true;
        }
    }
    // .....
    public object Current
    {
        get
        {
            return this.t[i, j];
        }
    }
}
```

Az `'udvaromEnum'` osztály konstruktora az átvett paramétereket elmenti a példányszintű mezőibe (a mátrixot, és annak méreteit), és az `'i'` és `'j'` mezőit inicializálja. Ezek mint ciklusváltozók fogják mutatni a bejárás aktuális pozícióját. Az `'i'` mutatja, hogy a mátrix hányadik soránál járunk, a `'j'` pedig, hogy ezen a soron belül hányadik oszlopban. Az `'i'` értékét 0-ra állítja a konstruktor, a `'j'` értékét pedig azért  $-1$ -re, mert a `MoveNext()` első dolga lesz megnövelni 1-gyel, így lesz a `MoveNext()` első vizsgált cellaeleme, a `t[0,0]` koordinátájú.

A `Reset()` metódus törzsére nincs szükség. (Csak egy kivételt dob fel, de a `foreach` nem használja ezt, így nem lényeges.)

A `MoveNext()`-et a `foreach` kezdetben is és minden feldolgozott elem után is meghívja. Abból kell kiindulni, hogy utoljára a `t[i,j]` elemet dolgozott fel a `foreach`, így a `MoveNext()` első lépése, hogy a következő elemre lép. Mivel soronként dolgozzuk fel a mátrixot, így először a `'j'` értékét növeljük. Ha túlléptük a sort jobbra (második `if`), akkor növeljük az `'i'` értékét, és a `'j'`-t visszaállítjuk  $-1$ -re. Ha már a sorokban is túlléptük a mátrixot (`'i'` értéke túl nagyra nőtt, első `if`), akkor nincs több feldolgozható elem a mátrixban. Ha elvileg jó az `'i'` és a `'j'` értéke is, de a cella üres (null értékű, harmadik `if`), akkor ez a cella még nem jó, keresni kell további cellát ami nem üres (tartalmaz háziállat példányt). Ha minden vizsgálaton megtörtént (`'i'` és `'j'` értéke mátrixon belüli, a cella nem üres), akkor a `MoveNext()` sikeresen megtalálta a következő feldolgozható elemet (`return true`).

Amennyiben a `MoveNext()` sikert jelzett, úgy a `foreach` meg fogja hívni a `Current` property-t, kiolvasva annak értékét. Mivel a `MoveNext()` az `'i'` és `'j'` értékeit állította be a következő feldolgozható elemre, itt már csak az a dolgunk, hogy visszaadjuk ezen cella értékét.

## 24. Nullable type

---

A null-képes típus koncepciója az, hogy az érték típusok (bool, int, double stb.) nem képesek felvenni speciális értéket, mely azt jelzi, hogy „még nem kapott értéket”. Például a double képes, létezik a Double.NaN konstans (NaN = not a number, nem szám), de bool esetén már ilyen speciális érték nincs. Ugyanakkor ha a bool értékünket adatbázisból olvassuk be, akkor ott gyakori eset, hogy a bool (MS-SQL bit típusa) érték tárolásánál megengedik a null értéket is az SQL táblában. Egy ilyen értéket beolvasni a memóriába nagyon kockázatos, beolvasáskor meg kell vizsgálni, hogy null értékű-e, s ha igen, akkor alapértelmezett, pl. false értékkel helyettesíteni. Sajnos ezzel el is takartuk azt a tényt, hogy a táblában ez nem false, hanem null (nem különböztethető meg később, hogy azért false az érték mert null volt, vagy azért false mert false volt a rekordban).

A nullable típusok tehát a szokásos értéken felül még a null értéket is képesek felvenni, ennyiben többek a hagyományos változóknál. Egyszerűen jelölhetők a programban: a típusdeklarációkor az alaptípus neve mögött egy kérdőjelet kell feltüntetni:

```
bool? b = false;
int? x = 0;
double? d = null;
char? c = '\x32';
```

Értelmetlen a kérdőjel módosítót referencia típus után helyezni, mivel azok eleve képesek null értéket tárolni:

```
string ss = null; // ez eleve jó
string? s = null; // ez szintaktikai hiba
```

Ez a kérdőjeles szintaktikai valójában a System.Nullable<T> generic alias neve. Vagyis a

```
double? d
```

az valójában a

```
System.Nullable<double> d
```

illetve, a 'using System;' miatt a

```
Nullable<double> d
```

deklaráció rövidített alakja.

---

A nullable típusra is alkalmazható a boxing művelet. Ez nagyon intelligens megoldás, ugyanis ha a nullable típusú változó éppen a 'null' értéket tárolja, akkor a boxing valójában nem történik meg, az object típusú változó egyenesen a null értéket veszi fel. (Mivel az object típusú változó referencia típusú, ő eleve képes null értéket tárolni):

```
bool? b = null;
object o = b;
```

Ez esetben az unboxing-ra ügyelni kell, könnyű hibát véteni. Helyesen ekkor is nullable típusra kell visszalakítani:

```
bool? m = (bool?)o;
```

Itt a nullable típusok környékén érdemes beszélni a C# nyelv ?? operátoráról is, ami egy bináris operátor, első operandusa egy olyan érték, amely akár null is lehet (felveheti a null értéket). Viselkedését tekintve ha az első operandusa null, akkor a kifejezés eredménye a második operandusbeli érték. Ha az első azonban nem null, akkor eredményül az első operandusbeli értéket adja meg (ilyenkor nem is fontos a második operandus értéke). Azért nagyon hasznos, mert a nullable típusú változóból ha át akarunk térni nem nullable típusú értékre, akkor azzal problémáink lennének:

```
bool? b = false;
bool x = b; // ez szintaktikai hibas
```

Az 'x' nem veheti fel általában a 'b'-beli értéket, mivel az 'x' nem képes a 'null' érték tárolására, pedig az a 'b'-nek lehetséges értéke. Emiatt típuskényszerítést kellene alkalmazni. Jelen esetben az 'as' operátoros típuskényszerítés nem jöhet szóba, mivel value type-okról van szó:

```
bool? b = false;
bool x = (bool)b; // ez futási hibát okozhat
```

Ha azonban a 'b' értéke ténylegesen null, akkor ez a típuskényszerítés futás közbeni hibát (exception) válthat ki. Helyes megoldás if-fel ellenőrizni és kezelni az esetet:

```
bool? b = false;
bool x;
if (b != null) x = (bool)b;
else x = false; // alapértelmezett
```

Vagy ugyanez kicsit rövidebben:

```
bool? b = false;
bool x = false; // alapértelmezett
if (b != null) x = (bool)b;
```

De leírható a `?:` operátorral is:

```
bool? b = false;  
bool x = b != null ? (bool)b : false;
```

Legegyszerűbben pedig a `??` operátor segítségével:

```
bool? b = false;  
bool x = b ?? false; // b vagy false
```

A nullable típus esetén a szokásos memóriaigényen felül szükség van egy bool típusú mezőre is, amely mutatja, hogy a változó értéke null érték-e vagy sem. Tehát egy hagyományos 'double' változó helyigénye 8 byte, a 'double?' helyigénye 8 + 1 byte. Mivel azonban a 32 bites fordítók előszeretettel 4 byte-os kiigazítással kezelik a memóriát, valójában ennek helyigénye 8 + 1 byte, de a utána kihagy a memóriában 3 byte-ot a fordító a következő változó helyfoglalása előtt – tehát valójában 12 byte-ot fogyaszt el egy ilyen változó. Ezzel együtt a nullable típusú változó továbbra is value type-nak tekintendő.

## 25. Kivételkezelés

---

A kivételkezelés rendkívül fontos témakör, a megoldás módja miatt mindenképpen erősen kötődik az OOP témakörhöz.

A programokat szinte a kezdetektől függvényekből építették fel. Az elején nem volt ez triviális gondolat, mert az első programozási nyelvek nem ismerték a függvény fogalmát. Később azonban alapvető módszerré vált a függvényekből való építkezés, éppen ezért a programozási nyelvek is szintaktikai szintre emelték ennek használatát – támogatták a függvények készítését, nevet adhattunk neki, paramétereket definiálhattunk, visszatérési értéket használhattunk stb.

A függvények önálló feladattal rendelkező programrészek. Minden függvénynek van feladata, melynek elvégzésére felkéri a program adott pontja, ahonnan a függvényt meghívjuk. Ha a feladat pontosításra szorul, paraméterek átadásával tesszük mindezt.

Mi történik, ha a meghívott függvény képtelen a felkért feladat elvégzésére? Számtalan oka lehet, de a leggyakoribb a hibás aktuális paraméterértékek megadása. Másképp fogalmazva: a feladat, amelyre felkérték, az adott paraméterértékek mellett nem végrehajtható. Vizsgáljunk meg két konkrét esetet:

- A `Sqrt(..)` függvény feladata, hogy az aktuális paraméterértékként kapott szám négyzetgyökét számolja ki. A paraméter tört szám is lehet, vagyis a paraméter típusa `double`. Az `Sqrt` függvény nem képes ellátni a feladatát, ha a paraméterként kapott számérték negatív (a négyzetgyök nem kiszámítható).
- A `FileCreate(...)` függvény feladata, hogy a paraméterként megadott nevű fájl létrehozza a lemezen. A fájl neve egy `string`, akár teljes elérési útvonalat is tartalmazhat. A `FileCreate` függvény nem képes ellátni a feladatot, ha a megadott nevű fájl nem hozható létre a lemezen.

Meg kell egyeznünk, a hibás esetek kezelésében. Ha a függvény nem képes végrehajtani a feladatot, valahogy jeleznie kell. Az alábbi eseteket képzelhetjük el:

- a függvény hibaüzenetet ír a képernyőre,
- a függvény hangjelzést ad, naplófájlba ír (log), vagy más módon jelzi a hibát,
- a függvény egy globális változóba rakja a hibakódot,
- a függvény speciális értéket ad vissza.

Vegyük sorra az eseteket, hogy megértsük melyiket miért nem tekintjük jó megoldásnak. Az első esetben a függvény hibaüzenetet ír ki:

```
static double Sqrt(double x)
{
    if (x < 0) Console.WriteLine("nem kiszamithato a gyoke");
    // ... folytatás
}
```

Két problémát vet fel ez a megoldás. Az egyik, hogy a függvényünk visszatérési típusa `double`, és a fordítóprogram elvár tőlünk egy visszatérési értéket attól függetlenül, hogy mit írunk a képernyőre. Mit adjunk vissza ebben az esetben?

```
static double Sqrt(double x)
{
    if (x < 0)
    {
        Console.WriteLine("nem kiszamithato a gyoke");
        return 0;
    }
    // ... folytatás
}
```

Gondoljuk át: 0 valóban jó visszatérési érték? A meghívó kód valóban tudni fogja, hogy baj történt? Nem találhatnánk ennél alkalmasabb visszatérési értéket? A `-1` pl. jobbnak tűnik, ugyanis a 0 előfordulhat hibamentes esetben is, a 0.0 négyzetgyöke ugyanis pont 0, vagyis ennek visszaadott értéke vagy hibát jelez, vagy azt, hogy ez a helyes megoldás. A `-1` nem lehet helyes megoldás, nincs olyan szám akinek az `Sqrt` függvénynek ezt kellene adnia megoldásképpen. Persze jelen esetben még jobb megoldást is találhatunk:

```
return Double.NaN;
```

A `NaN` (Not-A-Number) egy `double` konstans, melyet akkor használunk, amikor „nincs szám”. Az érték kiválóan képes jelezni, hogy a paraméterként átadott számnak nincs négyzetgyöke (pl. mert negatív szám). Ez lehet akkor a megoldás?

Megjegyzés: a megoldás elvárja a függvényt meghívó kódtól, hogy ellenőrizze le mi volt a válaszuk. Valahogy így:

```
Console.Write("kérek egy számot:");
double x = double.Parse( Console.ReadLine() );
double b = Sqrt( x );
if (Double.IsNaN(b)) Console.WriteLine("nincs gyoke");
else Console.WriteLine("gyoke={0}",b);
```

Szimpatikus és megfelelő megoldásnak tűnik, ha hajlamosak vagyunk szemet hunyni pár apróság fölött:

- a program futtatásakor valójában már kiíródik egyszer, hogy „nem kiszámítható a gyök” (a függvény írja), mielőtt kiírná a főprogram, hogy „nincs gyöke”,
- mivel mi írtuk a függvényt és a főprogramot is, emlékszünk és tudjuk, hogy `NaN` értéket ad meg problémás esetben a függvény,
- a függvény nem ültethető át (újra fel nem használható) más programjainkban.

Amikor hibaüzenetet írunk ki a képernyőre, mindig gondoljuk át mi ezzel a célunk. A hibakiírás kinek és miben fog majd segíteni? Ha ez egy nagy feladatkörrel rendelkező program része, amely rengeteg adattal dolgozik, órákig számol, és a 24. percben megjelenik a „nem kiszámítható a gyök” kiírás, az valóban fog segíteni bármiben is? Ki fogja elolvasni az üzenetet? A felhasználó? Számára ad-e valami információt azzal kapcsolatosan, hogy mit tegyen a következő percekben? Megérti ő a kiírást? Ha felveszi a telefont

és beolvassa a hibakiírást a helpdesk munkatársnak, aki átadja a programozónak hibajavításra, ki fog-e derülni bármi is azzal kapcsolatban, hogy hol és mi volt a hiba? Egyáltalán: milyen nyelven írjuk a hibaüzeneteket?

Nagyobb projektek esetén szokásosabb megoldás program futása során naplófájlt készíteni, amely tartalmazza a program működése közbeni fontosabb mérföldkövek elérésének időpontját, a program adott állapotát. A hiba detektálásakor kiíródik pontosan melyik függvényben történt a hiba észlelése, esetleg mik voltak az előzmények (hívási lánc a Main függvényből kiindulva), az aktuális paraméterek értékei, és egyéb, a hiba esetleges reprodukálásához szükséges információk, amelybe akár a gép típusa, az operációs rendszer típusa, egyéb információk is beletartozhatnak. Az egysoros hibakiíró üzenet nyilván nem tartalmazhat ennyi információt, ezért használata a legtöbb esetben haszontalan.

A függvényen belüli hibakiírással tehát sok baj van. Gyorsan segíthetünk rajta: kikommenteljük a függvényből a hibajelzést. Igen, de ekkor mi fog történni ha a felhasználó kód ennél jelentősebben bonyolultabb. Például egy kifejezésben akarjuk a függvényt felhasználni?

```
double b = Math.PI * Sqrt( x ) / 2;
```

Ha kikommenteljük a függvénybeli hibakiírást, akkor nem fog kiderülni, hogy egyáltalán valami baj történt. A kifejezés kiértékelése során a NaN-nal való műveletvégzés következményeképp a 'b' értéke is NaN lesz. Tegyük fel, hogy a kifejezés még bonyolultabb:

```
double b = Math.PI * ( Sqrt( x ) + Sqrt( y ) ) / 2;
```

Egy ilyen kifejezésben, ha a 'b' értékét NaN-nak találjuk, honnan fogjuk tudni eldönteni mi volt az oka? Bármelyik (vagy mindkettő) eredménye amennyiben NaN, úgy az egész kifejezésünk eredménye NaN lesz. Mielőtt bekezdénénk a kifejezés kiszámításába, számoltassuk ki külön-külön az értékeket és ellenőrizzük le az eredményeket?

```
double b;  
double b1 = Sqrt(x);  
if (Double.IsNaN(b1)) Console.WriteLine("nincs gyöke x-nek");  
else  
{  
    double b2 = Sqrt(y);  
    if (Double.IsNaN(b2)) Console.WriteLine("nincs gyöke y-nak");  
    else b = Math.PI * ( b1+ b2 ) / 2;  
}
```

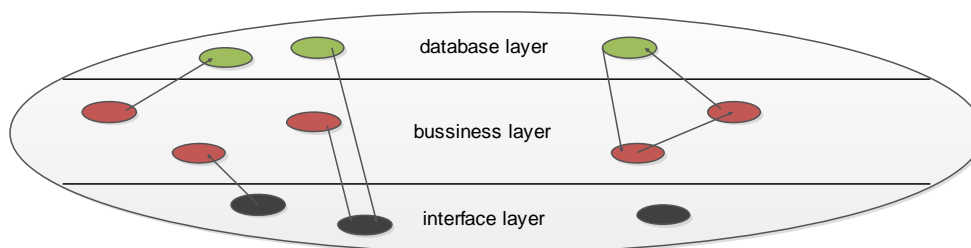
Máris túlbonyolódott a hívó kód. A másik probléma, hogy a hívó kódnak kell ellenőrizni, hogy a függvény el tudta-e végezni a feladatot. Ugyanis erről a hívó kód hajlamos megfeledkezni. Senki nem ír úgy programot, hogy minden utasítást, minden függvényhívást körberak if-ekkel. Ez nemcsak nagyon bonyolítja (főként egymásba ágyazás esetén) a hívó kódot, de jelentősen le is lassítja a futást. Ugyanakkor ha nem teszi meg a hívó kód,



akkor a probléma még nagyobbá duzzadhat, mivel a számolási folyamat nem áll le, és sok tucat sorral távolabb fog majd fatális hibát okozni. Ilyenkor nagyon nehéz visszanyomozni, hol kezdődött az a folyamat, amely a súlyos hibáig vezetett.

A függvény belsejébe elhelyezett hibakiírás persze sokat segít, mert legalább a képernyőn láthatjuk, hogy volt gyökvonási hibánk. Ugyanakkor meg kell értenünk: ilyet professzionális programozó – a három rétegű alkalmazásfejlesztési modell elvéből kifolyólag – nem ír bele a függvénybe. Az említett elv szerint a programok felbonthatóak három különböző feladatú részre:

- Felhasználói interface (*user interface, presentation logic*), melynek feladata a program és a felhasználó közötti kapcsolattartás. Ennek során a felhasználó kezeli a programot, folyamatokat indít és állít le, adatokat visz be, a számolási eredményeket olvassa el.
- Alkalmazáslogika (*bussiness/application logic*), amelyek a bevitt adatok alapján a tényleges számítási feladatokat ellátják. Általában ez a program „magva”, a know-how lényegi részét tartalmazza. Például egy mérnöki tervező program esetén itt vannak kódolva azon részek, amelyek ellenőrzik, hogy a tervezett alkatrészek együtt működőképes szerkezetet alkotnak-e stb.
- Adatkezelő réteg (*database managing*), mely háttértárra menti, tárolja, visszaolvassa, törli azokat az adatokat, amelyekkel a program dolgozik. Jellemzően SQL szerverekkel oldjuk meg manapság, de más megoldások (pl. XML file) is léteznek.



Az egyes rétegekbe (layerek) a programunk függvényeit kell besorolni. A helyes módszer az, ha egy függvény egyértelműen besorolható pontosan egy rétegbe (az elv kiterjesztése szerint a komplett objektumosztályok is pontosan egyetlen rétegbe kell, hogy besorolhatóak legyenek).

A négyzetgyök kiszámító függvényünk a tevékenységének fő célja szerint a középső, alkalmazáslogikai rétegbe sorolandó. Egy ilyen függvény nem írhat a képernyőre, még hibüzenetet sem. Miért? Mert ha más környezetbe (pl. Windows-os grafikus felületű programba vagy web-es alkalmazásba, esetleg mobiltelefonos alkalmazásba) kívánjuk áttelepíteni, itt a `Console.WriteLine` egyszerűen nem működik. Jobb ha nem is teszünk ilyet bele.

Elv: amely függvény számol, az nem jeleníti meg a számolás eredményét vagy sikertelenségét. Amely függvény megjelenít valamit, az nem számol.

A három rétegű architektúrában a három réteg fizikailag három „programot” is jelent. Webes alkalmazás esetén az adatbáziskezelő SQL szerver az egyik résztvevő, az alkalmazás logika fut a szerveren (web szerver), míg az adatmegjelenítést végző program a böngésző, aki a kliens gépen fut.

Ennek értelmében az Sqrt függvényen belüli hibaüzenet kiírása sérti a szabályt, ezért nem javasolt a használata.

A visszatérési értékkel is gond van. Választhattuk volna a  $-1$ -et is, hasonlóan a NaN-hoz. De akár a Double.NegativeInfinity értéket is (és persze akár a  $-2$ -t, vagy a  $-1,5$ -öt is stb.). Ha mi írjuk a függvényt és a felhasználói kódot is, akkor ez nem gond. Ha különböző programozók, akkor hogyan tájékozódik a felhasználói programot készítő arról, hogy mi a speciális hibát jelző visszatérési érték? És ha már a felhasználói kód elkészült, de mi módosítjuk a függvényünkben (más döntést hozunk utólag), honnan fogja tudni, hogy neki is módosítania kell a hibakezelő értéket?

Foglaljuk össze a megoldás problémáit:

- a három rétegű modell miatt nem javasolt hibakiírásokat tenni a függvényekbe,
- a hibaüzenetek nagyobb projekt esetén egyébként is használhatatlanok, nem értelmezhetőek,
- a függvényeknek hiba esetén is vissza kell adni értéket, amelynek ilyenkor valamilyen szélsőséges, speciális értéknek kell lennie,
- gyakori eset, hogy több értékből is választhatunk, ilyenkor a hívó kód bizonytalan lehet melyikhez is kellene használnia,
- utólagos módosítással nagyon sok kárt okozhatunk, ha a hívó kód nem követi le a módosításainkat,
- a hívó kódnak minden esetben érdemes ellenőrizni, hogy speciális hibát jelző visszatérési értéket kapott-e, különben kritikus (fatális) hibát is okozhat, de csak jóval távolabbi ponton, melyből a hiba igazi oka nehezen visszakereshető,
- az ellenőrzések lassítják és bonyolítják a hívó oldali működést.

Sokan kezelik a problémahalmazt oly módon, hogy megelőzik. Ekkor *mielőtt* meghívják a szóban forgó függvényt, leellenőrzik, hogy az végrehajtható-e:

```
Console.WriteLine("kérek egy számot:");
double x = double.Parse( Console.ReadLine() );
if (x<0) Console.WriteLine("nem lesz gyöke x-nek");
else
{
    int b = Math.PI * Sqrt( x ) / 2;
    /* ... */
}
```

Vajon a függvény törzsének megírásakor kihagyhatjuk ugyanezt az ellenőrzést a paraméter értékére vonatkozóan? Nyilvánvalóan nem: nem bízhatunk meg a külvilágban annyira, hogy feltételezzük: biztosan olyan számot ad át nekünk a gyök kiszámítására, amelyet előtte helyesen és hibátlanul leellenőrzött. Máris érezhetünk egy problémát: a

paraméter értékét a külvilág is leellenőrzi és a függvény törzse is. A dupla ellenőrzés kedvezőtlenül hat a kód futási sebességére. Emellett felvet még egy problémát is: biztos, hogy a külvilág jól és mindenre kiterjedően ellenőrzi le a hibalehetőségeket? Gondoljuk végig a másik példánkon, a fájl létrehozó függvényen. Mik lehetnek a hibák, ami miatt nem sikerülhet a fájl létrehozása:

- ilyen nevű fájl már van,
- a fájlnev hibás felépítésű, pl. nem megengedett karakter is szerepel benne,
- a fájlnev túl hosszú,
- az alkönyvtár amiben létrehoznánk a fájlt, nem létezik (vagy a szülő könyvtárra sem),
- nincs ilyen meghajtó sem, ahol az alkönyvtárat létrehoznánk,
- a meghajtó, amin létrehoznánk csak olvasható meghajtó (pl. cd lemez),
- a meghajtó betelt, nem hozható létre rajta több fájl (pl. USB pen),
- a meghajtó hálózati meghajtó, és nincs rajta fájl létrehozási jogunk,
- a meghajtó hálózati, de jelenleg nem elérhető.

Biztos, hogy a programunk minden egyes pontján, ahonnan fájlt akarunk létrehozni ezen függvény segítségével, az összes hibalehetőségre előellenőrzést kívánunk végrehajtani (amelyet a függvény törzsében ráadásul amúgy is meg fogunk ismételni)? Nyilván nem.

Összefoglalásképp:

- A függvény maga ellenőrizze le, hogy a kapott paraméterek alapján a feladata végrehajtható-e, ő fogja mindenkinél pontosabban tudni leellenőrizni, és a külvilágban amúgy sem bízhat meg.
- A külvilág, ne próbálja átvállalni ta függvénytől az ellenőrzés szerepét.
- A külvilág ne előellenőrizze le a lehetséges hibákat, az dupla ellenőrzést fog jelenteni, és lassú programfutást.

A fentiek értelmében tehát leszögezhetjük:

- a függvény végezze az ellenőrzést,
- ha problémát észlel, lehetőleg jelezze a hívó kódnak,
- ne a felhasználónak jelezze a hibát pl. hibaüzenet formájában.

A legutolsó a legfontosabb: ha a függvény nem volt képes elvégezni a feladatát, azt nem a felhasználó felé kell jeleznie, aki nem tudja milyen függvények vannak a kódban, milyen sorrendben hívódnak meg, melyiknek mi a feladata – jelezze a hibát az őt hívó kód felé, hogy az őt hívó kód dönthessen arról, hogy a részfeladat hiányában képes-e tovább folytatni a saját (magasabb szintű) feladatát.

Mindezt úgy kell megoldani, hogy a hívó kódnak ne kelljen összes függvényhívás esetén if-ek segítségével ellenőrizni, hogy rendben van-e minden. Szeretünk optimisták lenni: a dolgok általában rendben vannak. De szeretnénk tudni, ha mégis valamiféle kivételes helyzetbe kerülünk.

## 25.1. A kivétel feldobása

Tegyük fel tehát, írjuk az `Sqrt` függvényünket, beleírtuk az `if`-et, leellenőriztük a paraméter értékét, és látjuk, hogy baj van, a négyzetgyök nem kiszámítható. Hibaüzenetet nem írhatunk ki, értéket nem szeretnénk visszaadni (még extrém értéket sem), aggódunk a minket hívó kódért, nehogy elhitessük vele, hogy minden rendben van. Ekkor mit tegyünk?

Egy mechanizmus siet a segítségünkre. Úgy kell elképzelni, hogy a programunk induláskor normál állapotban van. Ekkor az történik, amit idáig megismertünk: az utasításaink a leírt sorrendben kerülnek végrehajtásra: függvényeket hívunk, visszatérnek, a kifejezéseink kiértékelődnek, a dolgok haladnak a rendes kerékvágásban. Van a programnak egy másik állapota is: a hiba állapot. Ha hiba állapotba lépünk, akkor a normál állapot utasításai nem kerülnek végrehajtásra. A ciklusoknak azonnal vége, az elágazásokat átugorja a program, a kifejezések kiértékelése abbamarad. A program hívási lánc visszabontja magát a `Main` függvény felé, miközben a függvényekből a végrehajtás a `return`-höz hasonlóan visszatér a hívás helyére (bár visszatérési érték nélkül). Ha ebből az állapotból nem zökkenünk vissza a normál állapotba, akkor a visszatérések során visszajutunk a `Main` függvény hívási pontjába, de mivel még mindig hiba állapotban vagyunk, a `Main` függvény maradék utasításai sem hajtódnak végre, és így a `Main` függvény is befejeződik. Ekkor viszont programnak vége van.

Ha tehát hiba állapotba billentjük a programot, az könnyen okozza a program (nem azonnali, de gyors) leállítását. Hogyan kell hiba állapotba billenteni a programot? A **throw** utasítás segítségével.

A `throw` utasítás önmagában is állhat bizonyos körülmények között (melyekről később lesz szó), de első formájában paramétert igényel. A paramétere szükségszerűen egy objektumpéldány. Ez a példány mezőiben hordozza a hibával kapcsolatos információkat, melyek a hívó kód számára segítenek megérteni a hiba pontos okát. Nagyon egyszerű esetben egy hibakód (egész szám), bonyolultabb esetben számtalan kitöltött mezőt is jelenthet.

A `C#` nyelvben van megkötés arra, hogy milyen típusú objektumosztály példányai alkalmazhatók erre a feladatra, más nyelvekben nincs. A `C#` nyelvben az `Exception` osztállyal típuskompatibilis példányokat szabad csak a `throw` paramétereként feltüntetni. Mely osztályok kompatibilisek? Maga az `Exception` osztály, és gyerekosztályai. Utóbbiakból elég sok van, sőt, magunk is készíthetünk sajátokat. Egy egyszerű (bár nem legegyszerűbb) példa:

```
static double Sqrt(double x)
{
    if (x < 0)
    {
        Exception e = new Exception();
        throw e;
    }
    // ... folytatás
    return double.NaN;
}
```

Az `Exception` osztály három publikus konstruktorral rendelkezik. Az első paraméter nélküli, ekkor csak azt jelzi, hiba történt (az ok leírása nélkül). A második esetben egy string-ben megadhatunk hibaszöveget is (hasonlóan mint a hibakiírás esetén). A harmadikban pedig még egy beágyazott kivételt példányt is megadhatunk, de ezt egyelőre hagyjuk. Leggyakrabban használt a string paraméteres változat:

```
Exception e = new Exception("A számnak nincs négyzetgyöke");
throw e;
```

Nem szoktunk egyébként kódokban találkozni a `throw` utasítás ezen kivitelezésével. Gondoljuk végig mi lesz az 'e' példány sorsa. Van az a mondás: a `return` után nincs élet a függvényben. Nos, a `throw` után nincs élet a programban. A 'throw' hatására a program átbillen hiba állapotba, az 'e' példányunkra vonatkozóan utasítást már nem fogunk kiadni, semmilyen utasítást nem fog végrehajtani a program (egyelőre legalábbis maradjunk ennyiben). A fenti két lépéses hibajelzés akkor tekinthető hasznosnak, ha a konstruktor futtatása után még a hibajelzésen, az 'e' példányon finomítunk:

```
static double Sqrt(double x)
{
    if (x < 0)
    {
        Exception e = new Exception("A számnak nincs négyzetgyöke");
        e.Data.Add("paraméter értéke x=", x);
        e.Source = "Sqrt func";
        throw e;
    }
    // ... folytatás
    return double.NaN;
}
```

Ennek hiányában egyszerűbb alakban használjuk: a példányosítást és a hiba állapotba billentést egyetlen kifejezésbe foglaljuk:

```
throw new Exception("A számnak nincs négyzetgyöke");
```

Ebben az esetben a frissen elkészült `Exception` példány nem kerül lementésre semmiféle 'e' változóba. A `throw` áthelyezi a programot hiba állapotba, és csatolja a példányt.

Vegyük észre, hogy ezen az ágon a függvényünk már nem igényel `return` utasítást, nem kell gondolkodnunk, hogy `-1`-et, vagy `NaN`-t adjunk vissza a hívási helyre. A hívási helyre lényegében a hibaleíró `Exception` példányt adjuk vissza. Mindenesetre a `throw` hatására (ez esetben) a függvény azonnal visszatér a hívás helyére. Lássuk, mi történik a hívás helyén. Nézzük mi történik az alábbi kódban:

```
double b = Sqrt( x );
if (Double.IsNaN(b)) Console.WriteLine("nincs gyöke");
else Console.WriteLine("gyöke={0}", b);
```

Ha a négyzetgyök nem kiszámítható, és az Sqrt függvény a throw segítségével hiba állapotba tér át, akkor a 'double b = ...' utasítást sem fejezik be (normál állapotú utasítás), sem az őt követő if-eket nem hajtja végre a program. Bátran írhatjuk a következőt is akár:

```
double b = Sqrt( x );  
Console.WriteLine("gyöke={0}",b);
```

A WriteLine csak akkor fog végrehajtódni, ha az Sqrt nem jelzett hibát, nem került a program hiba állapotba. Ellenkező esetben ez az utasítás is kimarad. Hasonlóan, nyugodtan írhatjuk a gyökvonást kifejezésbe is:

```
double b = Math.PI * ( Sqrt( x ) + Sqrt( y ) ) / 2;  
Console.WriteLine("eredmény={0}",b);
```

Ha bármelyik Sqrt hívás hibát jelez, a program szintén nem fogja végrehajtani a kiírást, mivel az normál állapotban végrehajtandó utasítás: **és ehhez nem kellett a hívó oldalon if-et használni!**

A második félmondat azért fontos, mert erről könnyen elfeledkezhetünk. Nem szeretünk lépten-nyomon if-eket írni, sokszor feleslegesek, kioptimalizálhatóak, néha elrontjuk a feltételeket. Az if-ekkel sok a baj. A throw használata mellett nincs rá szükség, hiszen ha baj van, akkor a program már nem fut tovább a hívó oldalon se, ehhez neki nem kell már if-eket leírnia. Sőt, a hívó oldal az optimista kódolási elv szerint úgy írja az utasításokat, mintha hiba nem fordulhatna elő. Ha tényleg nem fordul elő egy hiba sem, akkor a kód valóban lefut. Ha azonban hiba történik valamelyik ponton, a hívó oldali kód automatikusan nem fut tovább a hibaállapot miatt.

## 25.2. A hiba oka

A kérdés a következő: ezzel a módszerrel, hogyan tudjuk jelezni a különböző hibaokokat a hívó felé? Hogyan fogja megkülönböztetni a hívó oldal pl. a FileCreate() hívásának siker telenségét okozó hiba okát?

Több módon is megközelíthetjük a problémát, de egyiket sem fogjuk a későbbiekben javasolni. Az egyik legegyszerűbbnek tűnő, hogy az üzenet szövegét használjuk fel:

```
throw new Exception("hibás karakter van a filenévben");  
throw new Exception("nem létező alkönyvtár");  
throw new Exception("hálózati meghajtó hiba");
```

Ha 'e'-vel jelöljük majd később a hibaleíró példányt, akkor if-ek segítségével megvizsgálhatjuk. A példányban a konstruktor paramétereként megadott string hibaüzenet szövege a 'Message' mezőben tárolódik:

```
if (e.Message=="hálózati meghajtó hiba") ....;
```

Nyilvánvaló, hogy a módszer nem megfelelő. Okai:

- a hibaüzenet szövegének módosítása két helyen is meg kell történnjen: a kivétel feldobásának helyén, és az ellenőrzés helyén (pl. a szövegezés átfogalmazása, pontosítása, vagy más nyelven megadása esetén),
- a string-ek összehasonlítása lassú művelet,
- a hívó kódnak ismernie kell a hibaüzenet pontos szövegét fejlesztéskor.

Felhasználhatjuk ezen felül a Data mezőt is, amelyben tetszőleges mennyiségű adatot, információt tudunk tárolni. De ez sem bevett módszer. Helyette a feldobott kivétel típusát szokás felhasználni, ez a javasolt módszer.

Ugyanis nemcsak az Exception osztály példányai használhatóak a kivétel feldobásakor (throw), hanem a gyerekosztályai is. A gyerekosztályok reprezentálják az egyes hibatípusokat, másképpen: a hiba okát. A hibakezelés szempontjából olyan az Exception osztály, mint a típuskompatibilitás szempontjából az Object. Jelenleg őt tekintjük a gyökerelemnek. A .NET előre definiált hibatípusai (a teljesség igénye nélkül) az alábbiak:

```
Exception
  ApplicationException
  SystemException
    ArgumentException
      ArgumentNullException
      ArgumentOutOfRangeException
    FormatException
    IOException
      DriveNotFoundException
      FileNotFoundException
      PathTooLongException
      PipeException
    NotImplementedException
    NullReferenceException
    OutOfMemoryException
    PrintSystemException
      PrintCommitAttributesException
      PrintingNotSupportedException
      PrintJobException
      PrintQueueException
      PrintServerException
    StackOverflowException
```

A fenti listában az indentálás (beljebb kezdés) a gyerekosztályok jele. Vagyis pl. az 'ArgumentNullException' és az 'ArgumentException', akinek őse a 'SystemException', akinek őse maga az 'Exception' osztály.

Az Exception osztályok elnevezésével kapcsolatosan nem kötelező, hogy a név vége '...Exception' legyen, de hagyomány. A .NET Framework több tucat kivételosztályt tartalmaz, és továbbiakat is lehet definiálni. Amikor hibát kívánunk jelezni kivétel feldobásával, akkor ki kell választanunk a megfelelő típusú objektumosztályt, és a megfelelő példányát csatolni a throw utasítás segítségével:

```
throw new ArgumentNullException("a paraméter értéke null");  
throw new PathTooLongException ("a file neve túl hosszú");  
throw new OutOfMemoryException ("elfogyott a memória ");
```

### 25.3. A hiba kezelése

Ha programunkat hiba állapotba billentjük a throw segítségével, akkor az átlépi a normál állapotú utasításokat, a függvények blokkjának végére érve visszatér a hívás helyére, és ezt addig folytatja, amíg el nem éri a Main függvényből kiinduló kezdeti hívási pontot. A Main-ben lévő utasításokat is átlépi, de a Main függvény végét elérve a program futása leáll (futás közbeni hibával a program terminál).

A folyamatot megállíthatjuk a try-catch utasításpár segítségével. A try-nak saját utasításblokkja van, melynek belsejében, ha kivétel feldobása következik be (akár magában a blokkban, akár a blokkon belül hívott függvények belsejében), akkor a catch (elkap) elkapja a throw segítségével feldobott kivételt, és a program visszaáll a normál működésbe. Vázlatosan:

```
try  
{  
    // normál módú  
    // utasítások  
    // amelyek akár  
    // kivétel feldobását  
    // is okozhatják  
}  
catch  
{  
    // hiba állapotban lefutó  
    // utasítások  
    // akkor futnak le ha kivétel  
    // dobódott fel a try-ban  
}  
// az itteni utasítások már  
// minden esetben lefutnak  
// mivel a catch mindig leszedi  
// a hiba állapotot
```

Fogunk még finomítani ezen a vázlatos felépítésen, de alapvetően erről van szó. A try belsejébe írjuk az „optimista” kódot, amiben bízunk, hogy le fog futni hibátlanul, úgy gondoljuk az esetek legnagyobb részében le is fog, ezért nem kívánjuk teletűzdelni ellenőrzésekkel. Ha így történik, akkor a catch blokk mintha nem is létezne, nem fut le egyetlen utasítása se. Ellenben ha baj van (exception keletkezik) a try belsejében, akkor a try blokkbeli utasítások végrehajtása félbeszakad, és a catch rész kerül végrehajtásra. A catch részben van a „B” terv, ami vagy megpróbálja javítani, korrigálni a hibát, vagy



más módon kiszámolni amit ki kell, illetve végső esetben értesíteni a felhasználót a hibáról.

A try belsejébe függvényhívásokat is elhelyezhetünk, olyanokat, amelyek a három rétegű fejlesztési elvek miatt nem kommunikálhatnak a felhasználóval, ezért problémájukat Exception feldobásával jelzik. A hívó oldal elkaphatja a feldobott kivételt, és ő már másik rétegbe tartozóként a hibaüzenetet jelezheti a felhasználó felé:

```
try
{
    Console.WriteLine("kérek egy számot:");
    double x = double.Parse(Console.ReadLine());
    double b = Math.PI * Sqrt(x) / 2;
}
catch
{
    Console.WriteLine("Hiba történt a feldolgozás során");
}
double c = b*2;
```

Gondoljuk végig mi történhet ebben a try blokkban:

- a Console.WriteLine lényegében nem okozhat hibát,
- a double.Parse igenis okozhat hibát, ha a billentyűzetről beírt szám nem értelmezhető tört számként,
- a Sqrt() függvényhívása is okozhat hibát, ha az „x” értéke alapján a gyök nem kiszámítható.

Mind a Parse() mind a Sqrt() függvények az application logic rétegbe esnek, nem kommunikálhatnak a felhasználó felé, problémáikat kivétel feldobásával jelzik. Ha nem volt baj, akkor a bekérés sikeres, a gyökvonás sikeres, a 'b' értéke kiszámításra kerül. Ha bármelyik hibát okozott volna (bekérés, gyökszámítás) akkor a hibaüzenet megjelenik a képernyőn.

Nem teljesen hibátlan a kód, mivel a 'b' változó try blokkján belül került deklarálásra, így a catch utáni részen a 'b' változó hatáskörön kívüli, nem elérhető. Egyébként sem logikus a kód, mert ha a try-ban mégis hiba keletkezne, akkor a 'b' nem kap értéket, így a 'c' kifejezés sem kiszámítható. Javítsunk rajta:

```
double b;
try
{
    Console.WriteLine("kérek egy számot:");
    double x = double.Parse(Console.ReadLine());
    b = Math.PI * Sqrt(x) / 2;
}
catch
{
    Console.WriteLine("Hiba történt a feldolgozás során");
}
double c = b*2;
```

Most a 'b' deklarációját a try blokk elé, kívülre vittük, s így hatásköre módosult, jelentősen bővült. De a kód még mindig hibás, ha a try belsejében hiba történik, akkor a 'b' nem

kap értéket, így a 'c' kiszámítása nem lehetséges (nem tudni milyen értéket fog megszorzni kettővel). Javítsunk még egyet rajta:

```
double b;
try
{
    Console.WriteLine("kérek egy számot:");
    double x = double.Parse(Console.ReadLine());
    b = Math.PI * Sqrt(x) / 2;
}
catch
{
    Console.WriteLine("Hiba történt a feldolgozás során");
    b = 0;
}
double c = b*2;
```

A catch blokkon belül is adunk a 'b'-nek értéket, egyéb megoldás hiányában 0-t. (Ez a „B” terv.) Ekkor a 'c' értéke feltétlenül kiszámítható, hiszen a 'b'-nek mindenképpen van értéke, ha minden rendben van akkor a beírt szám gyöke alapján a kifejezésbeli érték, probléma esetén a 0.

Pusztán ennyi ismeret segítségével megoldható az alábbi programozási probléma: *kérjünk be 5 db egész számot billentyűzetről, és helyezzük el őket egy listába. A program ügyeljen arra, hogy ha a felhasználó nem értelmezhető számot írna be, akkor jelezzen hibát.*

```
List<int> l = new List<int>();
while (l.Count < 5)
{
    try
    {
        Console.WriteLine("kérek egy egész számot:");
        int x = int.Parse(Console.ReadLine());
        l.Add(x);
    }
    catch
    {
        Console.WriteLine("Ez nem egész szám, hiba történt");
    }
}
```

Az 'l.Add(x)' csak akkor kerül végrehajtásra, ha a Parse(..) nem jelzett hibát. Ellenkező esetben a hibaüzenet megjelenik, és ebben a ciklusmenetben az 'l' lista nem bővül új elemmel. A ciklus addig fut, amíg az 'l' lista 5 eleműre nem bővül, tehát 5 sikeres számbeírás nem történik meg. Ha nem tennénk try-ba a bekérést, akkor:

```
List<int> l = new List<int>();
while (l.Count < 5)
{
    Console.WriteLine("kérek egy egész számot:");
    int x = int.Parse(Console.ReadLine());
    l.Add(x);
}
```

Ez esetben az egész számként nem értelmezhető érték beírásakor a Parse(...) kivétellel jelezné a bajt, a program hiba állapotba billenne, és catch hiányában elérné a Main() függvény végét és futási hibával véglegesen leállna.

Térjünk vissza az Sqrt függvényes problémához. Beláttuk, hogy akár a Parse(), akár a Sqrt() okozhat hibát a try blokkon belül. Honnan tudhatjuk meg melyik okozta?

```
double b;
try
{
    Console.WriteLine("kérek egy számot:");
    try
    {
        double x = double.Parse(Console.ReadLine());
    }
    catch
    {
        Console.WriteLine("A parse hibát okozott");
    }
    b = Math.PI * Sqrt( x ) / 2;
}
catch
{
    Console.WriteLine("Hiba történt a gyökvonás során");
    b = 0;
}
double c = b*2;
```

Elemezzük ki a megoldást (leszámítva, hogy az 'x' a try blokkon belüli deklarációja akadályozza, hogy az Sqrt(..)-nek átadhassuk paraméterként). A try blokkok egymásba ágyazása révén általában elkülöníthetőek a feldobott kivételek helyei. Ám ha ugyanabban a kifejezésben szerepel mindkét lehetséges hibaforrás, úgy már ez a módszer nem működne:

```
double b = Math.PI * double.Parse(Console.ReadLine()) * Sqrt( x ) / 2;
```

## 25.4. A hiba okának felderítése

Hogyan válasszuk szét ilyen esetekben, hogy melyik kifejezés-rész generálta a hibát? Nos, ideje felhasználni azt az információt, hogy a különböző hibatípusokat különböző kivételosztályok jelzik! Az `Sqrt()`-ről tudjuk, hogy az alap, `Exception` osztálybeli példány feldobásával jelzi a problémát. A `Parse()`-ről kis utánajárással felderíthető, hogy amennyiben a string nem szám formátumú, úgy `FormatException`-t dob fel.

### ▲ Exceptions

Exception	Condition
<code>ArgumentNullException</code>	<code>s</code> is <b>null</b> .
<code>FormatException</code>	<code>s</code> does not represent a number in a valid format.
<code>OverflowException</code>	<code>s</code> represents a number that is less than <code>MinValue</code> or greater than <code>MaxValue</code> .

Amennyiben nemcsak az érdekel bennünket, hogy kivétel feldobása bekövetkezett-e, de az is, hogy konkrétan mi a hiba oka (milyen kivetelpéldány van csatolva), akkor a `catch` kulcsszót fel kell paraméterezni:

```
try
{
    // utasítások
}
catch (Exception e)
{
    // hiba kezelése
    // a hibaleíró példány az „e”-ben
}
```

A `catch` paramétere hasonló, mint a függvények paraméterezése. Ha hivatkozni szeretnénk a `catch` törzsében a csatolt hibaleíró példányhoz, nevet kell neki adni, pl. `'e'`, és meg kell adni a típusát (`Exception`). Az `'e'` példányváltozó értéke a `throw` utasítás paramétereként korábban megadott, feldobott hibaleíró példány lesz:

```
try
{
    Exception p = new Exception("baj van");
    throw p;
}
catch (Exception e) // e = p
{
    // hiba kezelése
}
```

Milyen típusú példányok kerülhetnek be az `'e'` változóba? Erre a típuskompatibilitás adja meg a választ. Mivel az `'e'` statikus típusa `Exception`, ezért konkrétan minden `Exception` gyerekosztálybeli típus értékét is felveheti. Mivel a `throw` paramétere csakis ilyen `Exception` gyerekosztály lehet, így lényegében bármilyen feldobott hiba esetén az `'e'` képes azt eltárolni.

```
try
{
    double b = Math.PI * double.Parse(Console.ReadLine()) * Sqrt(x) / 2;
}
catch (Exception e) // e = p
{
    if (e is FormatException) Console.WriteLine("A parse okozta");
    else if (e is Exception) Console.WriteLine("Az Sqrt okozta");
}
```

Az 'is' operátor segítségével megvizsgálhatjuk, hogy az 'e' példánynak mi a dinamikus típusa, s így el tudjuk dönteni melyik részkifejezés dobta fel a kivételt. Persze vegyük észre, hogy az 'e is Exception' vizsgálatnak valójában nincs értelme, mivel az biztosan 'true' értéket fog adni.

A módszer – bár működik – komoly kérdéseket vet fel. Mi történik ugyanis, ha a Parse esetén tudjuk mi a hibakezelés módja, de az Sqrt hibája esetén nem? A Parse hibáját el szeretnénk kapni, mert tudjuk mi a teendő ezen hiba esetén, de az Sqrt hibát át szeretnénk engedni magunkon. Ha elkapjuk mindkét feldobott kivételt, akkor már késő – a program visszaállt normál üzemmódba, hiszen a hibát lekezeltek.

## 25.5. A kivétel újrafeldobása

Tüneti kezelésként a catch blokk belsejében dobunk fel hibát. Ugyanazt, amit elraktunk.

```
try
{
    double b = Math.PI * double.Parse(Console.ReadLine()) * Sqrt(x) / 2;
}
catch (Exception e) // e = p
{
    if (e is FormatException) Console.WriteLine("A parse okozta");
    else throw e; // elkapott kivétel újrafeldobása
}
```

Az elkapott 'e' példányt újra feldobjuk a throw-val. Ezt szabad, a catch belsejében is lehetkezhetsz kivétel. Mivel a catch már leszedte a korábban feldobott kivételt, és a program már visszaállt normál állapotba – szabad újra visszabillenteni hibaállapotba, akár ugyanazon kivétel újbóli feldobásával (visszadobás).

Ezt egyszerűbben is el tudjuk érni, a paraméter nélküli throw segítségével:

```
try
{
    double b = Math.PI * double.Parse(Console.ReadLine()) * Sqrt(x) / 2;
}
catch (Exception e) // e = p
{
    if (e is FormatException) Console.WriteLine("A parse okozta");
    else throw; // az elkapott kivétel újrafeldobása
}
```

A paraméter nélküli throw csak a catch törzsén belül szerepelhet, ekkor az elkapott kivételt dobja fel újra. A kivétel eredeti feldobását mindenképpen paraméterezni kell, mivel hibaállapotba való átbillentéshez csatolni kell a hibaleíró is.

Természesen van hozzá jogunk, hogy ne az eredeti hibát dobjuk vissza, hanem újat készítsünk, és azt dobjuk fel:

```
try
{
    double b = Math.PI * double.Parse(Console.ReadLine()) * Sqrt(x) / 2;
}
catch (Exception e) // e = p
{
    if (e is FormatException)
        throw new Exception("hiba volt a parse-ben");
    else throw; // az eredeti hiba visszadobása
}
```

## 25.6. A kivételek szétválogatása

Ha az a problémánk, hogy elkaptunk egy kivételt, amelyet nem szerettünk volna elkapni, és emiatt újra fel kell dobnunk – akkor valószínűleg rossz a kódunk. Ugyanis a kivétel elkapása és újrafeldobása sok idő, lassítja a program futását. Helyesebb magatartás el sem kapni a számunkra nem érdekes hibát. Helyesebb kód:

```
try
{
    double b = Math.PI * double.Parse(Console.ReadLine()) * Sqrt(x) / 2;
}
catch (FormatException e) // e = p
{
    Console.WriteLine("A parse okozta");
}
```

A catch ügyesebb paraméterezésével elérhetjük, hogy a catch ág csak a `FormatException` (vagy valamely gyerekosztályának) feldobására reagáljon. A `Sqrt` függvény egyszerű `Exception` példányt dob fel, azt a típuskompatibilitási szabályok értelmében nem lehet a fentiek szerint deklarált `e` változónak értékül adni. Efajta hibatípus esetén nem kapjuk el a hibát, így nem is kell újra feldobni. Ráadásul kimaradt az `if-es` vizsgálat, hogy az elkapott hibatípus `FormatException`-e, ami jelen esetben garantálja a catch paramétereként megadott típusnév.

Egy `try`-hoz több `catch` is tartozhat, amennyiben azok más-más típusú paraméterrel rendelkeznek:

```

try
{
    // kód
}
catch (FormatException e)
{
    /* ... */
}
catch (NullReferenceException e)
{
    /* ... */
}
catch (InsufficientMemoryException e)
{
    /* ... */
}

```

Ez esetben ez hasonló szerkezetű, mint egy switch, vagyis a catch ágak elágazási ágak. Értelmezése az IS operátor alapján érthető meg:

```

catch (FormatException e)
    // ha a feldobott kivétel kompatibilis a FormatException-nal
catch (NullReferenceException e)
    // ha kompatibilis a NullreferenceException-nal
catch (InsufficientMemoryException e)
    // ha kompatibilis az InsufficientMemoryException-nal

```

Abban is hasonlít a switch esetére, hogy a sok catch ág közül maximum az egyik futhat le. Sorrendben értékelődnek ki, így fontos lesz a sorrend is. Nézzük az alábbi (hibás) példát:

```

try
{
    // kód
}
catch (Exception e)
{
    /* ... */
}
catch (FormatException e)
{
    /* ... */
}

```

Amennyiben ténylegesen FormatException kivételt kerülne feldobásra a try-ban, az első ág kiválasztható lenne, mivel az (is) kompatibilis az Exception őssosztállyal. Emiatt valószínűleg a második ág sosem választódna ki. A helyes sorrend:

```

try
{
    // kód
}
catch (FormatException e)
{
    /* ... */
}
catch (Exception e)
{
    /* ... */
}

```

Vagyis a „bővebb” kivételágakat, az ősosztályokat sorrendben hátrébb kell rakni, hogy ne akadályozzák a gyerekosztályok által kezelhető eseteket. Amennyiben van `Exception` águnk is, úgy azt a legvégére érdemes elhelyezni.

## 25.7. Saját kivételek

Mint láttuk, a feldobott kivétel típusa kulcsfontosságú, segít elhatárolni egymástól az egyes hibatípusokat, így magát a hiba okát is. A .NET Framework eleve számos kivétel-osztályt tartalmaz, melyek segítségével a hiba oka általában leírható.

Ha azonban speciális hibaleíró osztályt kívánunk definiálni saját felépítéssel, mezőkkel, metódusokkal – megtehetjük. Saját kivételosztály definiálására van lehetőség, melyhez saját konstruktorokat is írhatunk, a hiba számunkra fontos körülményeinek tisztázását a paraméterek segítségével írhatnánk le.

A saját kivételosztály definiálása pontosan ugyanúgy történik, mint bármely más osztály származtatása. Mivel csak akkor tudjuk a saját osztályunkbeli példányt a `throw` utasítás paramétereként feltüntetni, ha a típusa kompatibilis az `Exception`-nel, valójában csak annyi speciális szabálya van a saját kivételosztály készítésének, hogy őseként egy már létező kivételosztályt kell választani.

Más okból is készítünk saját kivételosztályt, például amikor egyszerűen egy saját típust szeretnénk létrehozni, hogy a `catch` ágakban jól elkülönített módon jelezhessük a speciális hibaokot. Ekkor az ősként választott kivételosztályt lényegében nem is egészítjük ki újabb mezőkkel, metódusokkal, csak egyszerűen létrehozuk a gyerekosztályt. Természetesen számos konstruktor írására lesz szükség, hogy átadhassuk az ősosztály konstruktorának a paramétereket.

```
class NullHibaExcep : NullReferenceException
{
    public NullHibaExcep(string msg) : base(msg)
    {
    }
    public NullHibaExcep() : base()
    {
    }
}
```

Ez az egyszerűbb eset, amikor csak létrehozunk egy saját kivételosztályt két konstruktorral, melyek az ősosztály hasonló paraméterezésű konstruktorainak párjai. A paramétereket csak átengedi magán, átadja az ős konstruktoroknak. Ennek fényében az osztály az ősosztálynak egy (gyenge) másolata, kevesebb konstruktorral, mint az eredeti osztály. Ugyanakkor külön típusnak számít, és saját `catch` ágat lehet rá definiálni:



```

try
{
    // kód amely kivételt is dobhat
}
catch (NullHibaExcep e)
{
    // ha throw new NullHibaExcep(...) volt
}
catch (Exception e)
{
    // ha egyéb hiba volt
}

```

Ezzel teljesen analóg módon lehet olyan saját kivételosztályt is készíteni, amely tényleges bővítés, új mezőket és metódusokat is tartalmaz:

```

class KonvertHibaException : FormatException
{
    public string ertek; // amit nem sikerult konvertalni
    public KonvertHibaException(string msg, string ertek) : base(msg)
    {
        this.ertek = ertek;
    }
    public KonvertHibaException(string ertek): base("parse error")
    {
        this.ertek = ertek;
    }
    public KonvertHibaException(): base()
    {
        this.ertek = null;
    }
    public void naplobaIr()
    {
        string msg = String.Format("parse konvertalasi hiba "+
                                   "tortent, eredeti ertek={0}",this.ertek);
        log.messageWrite(msg);
    }
}

```

## 25.8. Finally

A finally blokk használata leggyakrabban erőforrás-használathoz köthető. Erőforrásnak tekintünk bármit, amiből kevés van, tehát elfogyhat, illetve megszerzéséért harc folyik. Ez lehet szén, olaj, emberi erőforrás, bármi. Informatikai viszonylatban erőforrásnak tekinthetjük a memóriát, a hálózati kapcsolatot, a nyomtatót, az adatbázis kapcsolatot stb.

Az erőforrással való munka során első lépés minden esetben az erőforrás megszerzése. A megszerzés gyakran nem egyszerű, várakozni kell míg hozzánk kerül az erőforrás használatának joga. Az erőforrás elsődleges tulajdonosa általában az operációs rendszer, tőle kell kérni a használatot. A használatot igyekezzünk minél rövidebb idő alatt befejezni, és visszaadni a használati jogot, hogy a többi várakozó is hozzájuthasson.

A lépések tehát:

- 1: erőforrás igénylése és megszerzése,
- 2: erőforrás használata,
- 3: erőforrás visszaadása, felszabadítása.

A feladat nehézséget az adja, hogy amennyiben sikeresen megszereztük az erőforrást, azt mindenképpen fel kell szabadítani a végén – akkor is, ha a használat során kivételt kaptunk (hibát), és akkor is ha minden rendben zajlott. Ugyanakkor, ha kivétel történne (és azt nem akarjuk kezelni), visszaadnánk a hívó oldalra, hogy tájékozódhasson a siker-telenségről.

Lássuk a példát (vázlatosan):

```
nyomtato p = oprendszer.nyomtatoElker();
try
{
    // "p" nyomtató használata
    // hiba is történhet
}
catch (Exception e)
{
    // nem csinálunk semmit
}
p.felszabadit();
throw e; // visszadobjuk a kivételt
```

Ebben a megoldási vázban több hiba is van:

- a catch ág lényegében üres, mivel nem kívánjuk kezelni a hibát, csak azért kapjuk el, hogy az 'e' változónkba bekerüljön a hiba,
- viszont ez az 'e' változó a catch ágon kívül már nem elérhető, tehát a végén a 'throw e' nem tudja azt visszadobni (abban a sorban nincs 'e' változó),
- amikor nem volt probléma a 'p' nyomtató használata során, nincs kivétel, nincs mit visszadobnunk.

Finomítsunk az elképzelésünkön:

```
nyomtato p = oprendszer.nyomtatoElker();
try
{
    // "p" nyomtató használata
    // hiba is történhet
}
catch (Exception e)
{
    p.felszabadit();
    throw e; // visszadobjuk a kivételt
}
p.felszabadit();
```

Most a fenti problémák mindegyike megoldódott, de az erőforrás felszabadításának utasítása kétszer szerepel a kódban. Egyszer a catch belsejében, mielőtt visszadobnánk a kivételt, másrészt a catch után is szerepel, mivel ha nem volt hiba, akkor nem fut le a catch, ekkor is fel kell szabadítani az erőforrást.

Ha az erőforrás felszabadítása nem egyetlen függvényhívás, hanem összetettebb folyamat, a kétszeri szerepeltetése a kódban már problémás. Kifejezetten ezen problémára alkalmazható a **finally** konstrukció:

```
nyomtato p = oprendszer.nyomtatoElker();  
try  
{  
    // "p" nyomtató használata  
    // hiba is történhet  
}  
finally  
{  
    p.felszabadit();  
}
```

A **finally** is a **try** kulcsszóval együtt szerepel. Ez – függetlenül attól, hogy a **try** belsejében keletkezett-e kivétel – mindenképpen lefut. Ennek értelmében tehát a 'p' felszabadítása mindenképpen bekövetkezik, ha a **try** blokkba beléptünk. Ugyanakkor a **catch**-cselel-  
lenkezőleg, a **finally** az esetleges bekövetkezett kivételt nem kezeli le, hanem a **finally** blokk végén azt visszadobja. Kivétel keletkezése esetén tehát a:

```
finally  
{  
    p.felszabadit();  
}
```

megfelel a

```
catch (Exception e)  
{  
    p.felszabadit();  
    throw e; // visszadobjuk a kivételt  
}
```

működésnek.

Gyakori kérdés (hiba), hogy az erőforrás lefoglalását is beleveszik a **try** belsejébe. Ez azonban helytelen: ha maga az erőforrás lefoglalása dobja a kivétel (hibát), és ez benne van a **try**-ban, akkor a **finally** le fog futni, benne az erőforrás felszabadítási utasítással. De mivel az erőforrás nincs lefoglalva, elképzelhető, hogy éppen a felszabadítása dob fel kivételt. A feladat az, hogy a keletkező kivételt ne „nyeljük el”, hanem adjuk vissza a hívó kódnak, de az új kivétel el fogja azt fedni, és az eredeti el fog tűnni. Ha azonban az erőforrás lefoglalása nincs a **try**-ban, és kivétel dob fel, akkor nem fut le a **finally**, és az eredeti kivétel kezeletlenül és azonnal megy vissza a hívó kódhoz. Ekkor nem kerül sor a (szükségtelen) erőforrás felszabadítására.

## 25.9. A kivétel keletkezése hibát okoz

A kérdés amire érdemes megkeresni a választ: mi történik ha a try... finally... páros esetén a try belsejében kivétel keletkezik, ezért átlépünk a finally-ra, de az ottani utasítások végrehajtása során is keletkezik (egy másik) kivétel?

```
try
{
    try
    {
        try { throw new NullReferenceException("nullref"); }
        finally { throw new IndexOutOfRangeException("indexoutof"); }
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
catch (Exception f)
{
    Console.WriteLine(f.Message);
}
```

A program futás közben csak az 'indexoutof' szót írja ki. Pedig először a 'nullref' üzenetet hordozó NullReferenceException kivétel dobódik fel. Ez kiváltja a finally blokk lefutását, amely feldobja az IndexOutOfRangeException kivételt. A finally végén (ha volt) a kivételt vissza kell dobni. De úgy tűnik, az új kivétel lecseréli (felülírja) a feldobott null ref. exceptiont.

Ebből a példából sok dologra fény derül. Egyszerre egy kivétel lehet aktív (feldobott). A később feldobott példány felülírja a tárolt aktív kivételt. Ezért ha a finallyban kivétel keletkezik, akkor innentől csak ez az egy lesz az aktív. Hasonló viselkedést tapasztalunk, ha a legbelső try-hoz nem finally blokkot, hanem catch blokkot csatolunk, és másik kivételt dobunk fel benne. Ezzel kevesebb a probléma, hiszen a catch eleve leszedi az aktív kivételt. Ekkor sokkal érdekesebb kérdés az, hogy ha több catch ág is van, egy korai catch ág belsejében feldobott kivételt egy rákövetkező catch ág képes-e elkapni:

```
try
{
    try
    {
        throw new IndexOutOfRangeException("indexout");
    }
    catch (IndexOutOfRangeException)
    {
        throw new OutOfMemoryException("outofmem");
    }
    catch (Exception e)
    {
        Console.WriteLine("belso {0}", e.Message);
    }
}
catch (Exception f)
{
    Console.WriteLine("kulso {0}", f.Message);
}
```

Lehet olyan elképzelésünk, hogy az 'IndexOutOfRangeException' catch ágban feldobott OutOfMemory kivételt a rákövetkező 'Exception e' ág akár el is kaphatná, típusa szerint is megfelelő erre a célra. Ugyanakkor ha lefuttatjuk a programot, a kiírásból rájöhetünk, hogy a 'külső' ág kapta el, nem a belső. Leszűrhetjük a tapasztalatot: egy try (több) catch ága úgy működik, mint a switch: legfeljebb egy ága választható ki. Ha a kiválasztott catch ágban újabb kivétel keletkezik, azt a további ágai már nem kezelhetik le, ehhez külső catch szükséges.

## 25.10. Try... catch... finally...

Egy hármas kombinációt alkothatunk a try – catch – finally segítségével:

```
try
{
    // 1-es blokk
}
catch (IndexOutOfRangeException)
{
    // 2-es blokk
}
catch (Exception)
{
    // 3-as blokk
}
finally
{
    // 4-es blokk
}
```

Ebben a felállásban (több catch ág is lehet, ne feledjük) a működés a korábbiak alapján kitalálható. A catch-ek – ha képesek – elkapják a hibát, és megszüntethetik. A finally mindenképpen lefut, akkor is, ha a try-ban eleve nem is képződött hiba, akkor is ha keletkezett, de a catch-ek elkapták, akkor is ha nem kapták el. Sőt, akkor is lefut a finally, ha a try belsejében kivétel keletkezik, azt valamely catch ág elkapja, de ugyanazt a (vagy egy másik) kivételt visszadobja.

A try... finally... catch... nem létezik.

## 25.11. Egymásba ágyazás

A try... catch... finally.... blokkok tetszőleges mélységben egymásba ágyazhatók. A viselkedés kiszámítható, ha nem felejtkezünk el a következő szabályokról:

- a catch elkapja azokat a kivételeket, amely a hozzá tartozó try blokkban keletkeznek,
- de amelyek nem a hozzá tartozó try-ban keletkeznek, azokat nem kapja el,
- a finally mindenképpen lefut, ha volt kivétel ha nem (de szintén csak a hozzá tartozó try blokk belsejére vonatkozik ez a dolog),
- ha a try előtt keletkezett kivétel, akkor a finally nem fut le.

```
// |  
// | 0-as blokk  
// |  
try  
{  
    // |  
    // | 1-es blokk  
    // |  
    try  
    {  
        // |  
        // | 2-es blokk  
        // |  
    }  
    catch (IndexOutOfRangeException)  
    {  
        // |  
        // | 3-as blokk  
        // |  
    }  
    catch (Exception)  
    {  
        // |  
        // | 4-es blokk  
        // |  
    }  
    finally  
    {  
        // |  
        // | 5-os blokk  
        // |  
    }  
    // |  
    // | 6-os blokk  
    // |  
}  
catch (Exception f)  
{  
    // |  
    // | 7-es blokk  
    // |  
}  
// |  
// | 8-es blokk  
// |
```

1. Mi történik, ha a 0-s blokkban keletkezik kivétel?  
A 0-s blokk maradék utasításai már nem hajtódnak végre, s mivel a 0-s blokk

nincs try-ban, további utasítások nem kerülnek végrehajtásra, a program ezen a ponton visszatér a hívó kódhoz a kivétellel.

2. Mi történik, ha először az 1-es blokkban keletkezik kivétel?

Az 1-es blokk maradék utasításai nem hajtódnak végre, az 1-es blokk try-ához tartozó catch, a 7-es blokk hajtódik végre. Itt kezelésre kerül a kivétel. A 7-es blokk után, mivel a kivétel kezelődik, a 8-as blokk is végrehajtódik. Ha a 7-es blokkban a kivétel kezelése közben újból kivétel váltódna ki, akkor a 7-es blokk maradék utasításai már nem hajtódnak végre, a 8-as blokk sem hajtódik végre, a program visszatér a hívó oldalra a 7-es blokkban keletkezett kivétellel.

3. Mi történik, ha a 2-es blokkban keletkezik kivétel?

Ha `IndexOutOfRangeException`, akkor a 3-as, különben a 4-es blokk hajtódik végre. A megfelelő catch ág után a finally 5-ös blokkja is végrehajtódik. A kivétel lekezelése miatt nincs már hiba állapot, így a 6-os blokk is végrehajtódik, végül a 8-as blokk is.

4. Mi történik, ha a 6-as blokkban keletkezik hiba?

A 6-os blokk try-ban van, így a hozzá tartozó catch, a 7-es blokk hajtódik végre. A catch kezeli a kivételt, a hiba állapot megszűnik, így a 8-as blokk is végrehajtódik.

5. Mikor hajtódik végre a 8-as blokk?

Ha nem történt sehol hiba, illetve ha a hiba nem a 0-s blokkban keletkezett, de a megfelelő catch ág a keletkezett kivételt lekezelte.

Hasonló kérdéseket lehetne még fogalmazni, a válaszok a korábban ismerttetett működés végiggondolása után egyszerűen megfogalmazhatóak, illetve kis tesztprogram (`Console.WriteLine` és `throw` felhasználásával) könnyen konstruálható, így a működés akár le is ellenőrizhető.

## 26. Operátorok

---

Az operátorok a programozás szükséges elemei. Segítségükkel kifejezéseket alkothatunk, amelyek abban segítenek, hogy újabb értékeket számolhassunk ki a már meglévők segítségével. A C# nyelv, mint minden más magasszintű programozási nyelv, számos operátort ismer (pl. az összeadás, kivonás, egyenlőségvizsgálat stb.). Ezek az alapszintű operátorok bizonyos típusok között vannak értelmezve. Ugyanazon operátor jel működése (szemantika) attól függ, hogy milyen típusú adatokra alkalmazzuk. Például az összeadás operátor értelmezve van számok között is, ekkor numerikus összeadás jelentésű. Stringek között string összefűzés, míg karakterek között szintén numerikus összeadást jelez (a karakter unicode kódjainak összeadását jelenti).

A procedurális nyelvekben létezett néhány összetett adattípus, rekord, tömb, esetleg lista. Az objektumorientált programozás lényege azonban az, hogy ezen túllépjünk. A típusok, amelyeket létrehozunk, összetett adattípusok saját műveletekkel. A saját típusok nem képesek egy ponton továbblépni: ha műveleteket kívánunk végezni két vagy több saját típusú adaton, akkor ahhoz metódusokat, függvényeket kell alkalmaznunk. Az operátorok nem alkalmazhatóak az adatokra, hiszen a fordítóprogram nem tudja milyen kódot generáljon az adott esetre.

Ugyanakkor az operátorok és a függvények kapcsolata sokkal szorosabb mint azt elsőre gondolnánk:

```
int a = 12;  
int b = 23;  
int c = a + b;
```

Az 'a' és 'b' értékek összeadását nemcsak operátorok, hanem metódusok segítségével is értelmezhetjük:

```
int c = osszead(a,b);
```

Eképpen az összeadás operátor (és bármely más operátor) működése helyettesíthető egy, az adott típusra értelmezett globális függvény hívásával:

```
static int osszead(int a, int b)  
{  
    return ...;  
}  
  
static string osszead(string a, string b)  
{  
    return ...;  
}  
  
static int osszead(char a, char b)  
{  
    return ...;  
}
```



A `+` operátor használatakor a fordítóprogram egyszerűen helyettesíti azt a megfelelő paraméterezésű függvény hívásával. Ha adott paraméterezésű változat nincs a függvényből, akkor az operátor nincs értelmezve ezen típusok esetén, így fordítási hibát lehet adni. Ha a fordító belső működési szabálya szerint a `+` operátort mindig az 'összead' függvény hívására cseréli le, és csak a fenti paraméterezésű változatok léteznek, akkor két bool adat közé írt `+` operátort nem tud a fordító függvényhívásra átfordítani<sup>35</sup>, és így fordítási hibát jelez a fordító.

Ha szeretnénk a saját típusunkra is alkalmazni operátorokat, ki kell tudnunk használni ezt a fajta megközelítést. Függvényeket fogunk írni, melyek speciális névadási szintaktikával rendelkeznek, ezért a fordítóprogram operátorok alakjában is fel fogja a rejtett függvényhívást ismerni. Ezen túl is sok megkötésnek kell még eleget tennünk, melyekkel sorban meg ismerkedünk a fejezet további részeiben.

A C# nyelvi fordító háromfajta operátort ismer fel és kezel:

- egyoperandusú operátor, mint pl. a `++` operátor,
- kétoperandusú operátor, mint pl. az összeadás,
- típuskonverziós (típuskényszerítő) operátorok.

A saját operátoraink írását vissza kell tudni vezetni valamelyik itt felsorolt operátor viselkedésére. (Mindegyik típusra további speciális szabályok vonatkoznak.)

Más programozási nyelveken is van lehetőség az operátorok értelmezési körét bővíteni. Egyes nyelveken lehetőség van új operátor jelölést is fejleszteni (új operátorok bevezetése) az ezekre alkalmas jelek kombinációjával. Más nyelveken csak a nyelv által eleve adott, valamilyen jelentéssel létező operátoroknak lehet új jelentést kölcsönözni. Továbbá egyes nyelveken az új operátor bevezetésekor megadható annak asszociativitása, prioritása is, míg más nyelveken (elsősorban ahol csak már eleve létező operátorok újraértelmezése támogatott) ezeken a tulajdonságokon nem lehet változtatni.

A C# nyelv e szempontból a szigorú nyelvek közé tartozik. Csak eleve létező operátorok jelentését szabad bővíteni, ezeken belül sem mindegyikét. Az operátorok asszociativitása és prioritása nem módosítható.

Az operátor függvények metódusok, hiszen a C# nyelv tiszta OOP nyelv, függvények írására csak osztályon belül van lehetőség. Az operátor függvények elhelyezése során erre vonatkozó szabályok is vannak; ezen függvények elhelyezése, a tartalmazó osztály kiválasztása fontos!

## 26.1. Egyoperandusú operátorok fejlesztése

A C# nyelv legfontosabb unáris (egyoperandusú) operátorai: `+`, `-`, `!`, `~`, `++`, `--`.

Ha egy `X` unáris operátort szeretnénk egy `T` típusú értékre alkalmazni, akkor a fordítóprogram ellenőrzi, hogy van-e 'operator `X(T a)`' függvény a `T` típus deklarációjában, vagy valamely ősosztályának deklarációjában.

<sup>35</sup> A fordítóprogram saját döntése, hogy valóban függvényhívásra fordítja le az operátort, vagy a függvény törzsében szereplő kevés utasítást helyben fejt ki (inline függvény). Mivel a függvényhívás extra mennyiségű memóriát és időt igényel, az egyszerűbb esetekben a fordító inkább a függvénytörzs kifejtését választja.

Tegyük fel, hogy van egy saját 'Datum' típusunk, amely képes egy adott dátum értékét tárolni. Döntésünk értelmében érdekes módszert választunk a dátum tárolására: kitűzünk egy adott dátum értéket, önkényes választásunk az 1970.01.01 dátumra esik. Minden más dátumot ezen start ponthoz viszonyítunk: eltároljuk, hogy ezen naphoz képest hány nap telt el. Vagyis az 1970.02.10 dátumot úgy tároljuk el, hogy valójában csak a 41 értéket tároljuk, jelezvén, hogy a mi dátumunk, a február 10 az 41 napnyi távolságra van a kezdődátumtól.

Ezen dátumtárolási módszernek vannak előnyei, és hátrányai is. Előnye, hogy gyorsan eldönthető két dátumról melyik a korábbi vagy későbbi, könnyen meghatározható két dátum esetén, hány nap különbség van közöttük. Hátránya, hogy adott érték esetén (pl. 4519 nap) nem könnyű (bár nem lehetetlen) megmondani, pontosan hányadik év hanyadik hónap melyik napja. Ez utóbbival az áttekinthetőség miatt nem fogunk foglalkozni.

```
class datum
{
    protected int _nap;
    public int nap
    {
        get { return this._nap; }
    }
    public datum(int n)
    {
        this._nap = n;
    }
    public datum(string sDatum)
    {
        // valahogy kiszamoljuk sDatum hanyadik nap
        // ez kerül tárolásra
        this._nap = ...;
    }
}
```

Szeretnénk egy ilyen datum típusú értékre operátort alkalmazni. Például a ++ operátornak van értelme, jelölje ez a művelet a „lépj a következő napi dátumra”:

```
datum d = new datum("2012.01.20");
d++;
```

Hogy a fordító felismerje, hogy a 'd++' kifejezéshez milyen kódot kell generálnia, el kell készítenünk a 'datum' típusra értelmezett '++' operátor függvényt:

```
class datum
{
    public static datum operator ++(datum x)
    {
        return new datum(x.nap + 1);
    }
}
```

Ekkor ha a kódunkban valahol szerepel a 'd++' kifejezés (ahol 'd' egy 'datum' típusú változó), úgy a fordító a kifejezést 'datum.operator++(d)'-re cseréli le:

```
// ezt írjuk
d++;
// de ez történik
d = datum.operator++(d);
```

Vagyis meghívódik a megadott típusnak megfelelő operátor függvény, és az általa előállított új értékkel felülíródik a 'd' korábbi értéke. Vegyük észre, a fordító nem tudja, hogy ennek hatására valóban növeledött-e a d-beli érték, mivel e körülmények között erről nem tud meggyőződni.

Rögzítsük a szabályokat, amelyek az unáris operátorok írásához szükségesek:

- Az unáris operátor függvényt ugyanabba a T típus osztályába kell beleírni, amelyre vonatkozik, vagyis ha egy dátum típusú értékre alkalmazott operátort fejlesztünk azt a 'class dátum'-ba kell belehelyezni.
- Az unáris operátor függvények static és public jelzőűek.
- A függvénynek csakis egy paramétere lehet (hiszen unáris operátorról van szó).
- A paraméter típusának meg kell egyeznie az őt tartalmazó osztály típusával (vagyis az első szabály újra: a T típusra alkalmazandó operátort a class T -be kell helyezni).
- A függvény visszatérési típusának is ugyanannak a típusnak kell lennie (ezen operátorok nem változtathatnak a típuson).

Az unáris operátor általános alakja:

```
public static <típus> operator <operator>( <típus> <pname> )
{
    ...
}
```

Vitassuk meg az operátor függvény törzsének megvalósítását! A feladat, hogy kapunk egy dátumot, és az eggyel nagyobb értéket kell visszaadnunk visszatérési értéként. Alapvetően kétféleképpen érhetjük el:

```
public static datum operator ++(datum x)
{
    return new datum(x.nap + 1);
}
```

Vagy:

```
public static datum operator ++(datum x)
{
    x._nap++;
    return x;
}
```

Az első esetben új dátumot készítettünk, és azt adtuk meg visszatérési értéként. Második esetben a paraméterként kapott dátum mezőjét módosítottuk, és visszaadtuk lényegében a paraméterünket, amit kaptunk. Mi a különbség a két megvalósítás között?

Ne feledjük, hogy a 'class' szóval képzett típusok a referencia típuscsaládba tartoznak. Vagyis a függvény paramétere valójában egy memóriacím, és amit a függvény visszaad az is egy memóriacím. Vizsgáljuk meg az operátor függvény kétféle megvalósítását az alábbi kód esetén:

```
datum d = new datum("1970.01.10");  
datum p = d;  
d++;  
Console.WriteLine(p.nap);
```

Az első lépésben elkészítjük a 'd' változót a memóriában. A 'nap' mező értéke 10 lesz, hiszen 10 napnyi távolságra vagyunk a start ponttól. A 'd' tárolja a memóriacímet. A második lépésben erről a memóriacímről másolatot készítünk a 'p' változóban, a 'p'-beli 'nap' mező értéke is 10. Megnöveljük a 'd' értékét 1-gyel a ++ operátor segítségével. Ez új dátum példányt készít, melynek memóriacímét adja meg, s eképp a 'd' változó új értéket kap, az új példány memóriacímét kapja meg. A 'p' változó (ami még mindig a régi példány memóriacímét őrzi) dátuma nem változott meg, a 'p' értéke lényegében továbbra is '1970.01.10' míg a 'd' értéke az új '1970.01.11' lesz.

Második esetben azonban nem ez fog történni. Az első lépésben létrehozuk a dátum példányt, memóriacímét a 'd' változóba helyezzük. Másolatot készítünk a memóriacímről a 'p' változóba is. Majd növeljük a 'd'-beli dátumértéket 1-gyel. Ez a megoldás nem hoz létre új példányt, hanem a paraméterként kapott példányon hajtja végre a módosítást. Így a végrehajtás végén a 'd'-ben ugyanaz a memóriacím marad, bár a dátum értéke növeledött 1-gyel. Ez még nem lenne baj, azonban a 'p'-beli dátum is növeledött 1-gyel, holott nem kértük.

A tanulság az, hogy mindkét megvalósítás működik. Csak rajtunk múlik, hogy melyiket választjuk. Ha nem zavar bennünket az utóbbi működés, választhatjuk a második megoldást is. A 'class'-ok esetén azonban jellemzőbb az első megoldás, mivel a felhasználó esetleg nem érti mitől módosult egy másik változóbeli érték is, amikor arra nem alkalmazott semmilyen operátort.

## 26.2. Kétooperandusú operátorok fejlesztése

Nemcsak egyoperandusú, hanem kétooperandusú operátor fejlesztésére is van lehetőség. Ezek az alábbiak lehetnek: + - \* / % & | ^ << >> == != > < >= <=

A kétooperandusú operátorok két (akár különböző) típusok között szerepelhetnek mint műveleti jelek. Ezért a hozzá tartozó operátor függvény kétparaméteres. A korábbi példát fejlesztve értelmezzük a dátumok és egész számok összeadását:

```
datum d = new datum("1970.01.10");  
datum p = d + 10;
```

Dátum és egész szám összeadásán azt értjük, hogy képezzük azt a dátumot, amely meghatározott nappal későbbi.

```
class datum
{
    public static datum operator +(datum d, int n)
    {
        return new datum(d.nap + n);
    }
}
```

Az operátor függvény kétparaméteres, egyik (első) paramétere 'datum', a második paramétere 'int' típusú. Eredménye egy újabb 'datum' példány, vagyis a korábban felírt kifejezés végrehajtható és típushelyes.

A kétparaméteres (bináris) operátorok készítésére vonatkozó szabályok:

- ha egy bináris operátor függvény két paraméterének típusai T1 és T2, akkor a függvényt vagy a T1, vagy a T2 típusba kell helyezni (vagyis a fenti dátum+int függvényt írhattuk volna a dátum, de akár az int típus osztályába is).
- a bináris operátor függvények static és public jelzőjűek,
- a függvénynek pontosan két paraméterrel kell rendelkezzen (hiszen bináris operátorrról van szó),
- a függvény visszatérési típusának nem kell megegyeznie a paraméterek típusainak egyikével sem (a logikai összehasonlító operátorok pl. bool-lal térnek vissza, miközben a paramétereik típusa nem bool).

A bináris operátor általános alakja:

```
public static <típus3> operator <operator>(
    <típus1> <pname1>, <típus2> <pname2> )
{
    ...
}
```

Példaképpen írjuk meg a két dátumot összehasonlító operátort is:

```
public static bool operator <(datum a, datum b)
{
    return a.nap < b.nap;
}
```

Amennyiben a kódot le akarjuk fordítani, hibaüzenetet kapunk. Ugyanis a fordító felismeri, hogy most már tudnánk ilyen feltételvizsgálatot felírni:

```
datum d = new datum("1970.01.10");
datum p = new datum("1970.01.12");
if ( p < d ) ...
```

De nem tudnánk pl. ilyet felírni:

```
if ( d > p ) ...
```

Holott ha eldönthető, hogy 'p' kisebb-e mint 'd', akkor az is eldönthető, hogy 'd' nagyobb-e mint 'p'! Ezért *'the operator < requires a matching operator > to also be defined'* hibaüzenetet adja, vagyis az *< operátor megírása esetén kötelezővé teszi a párját, a > operátor megírását is:*

```
public static bool operator >(datum a, datum b)
{
    return a.nap < b.nap;
}
```

Hasonló párban szerepelnek az == és != operátorok, valamint a <= és a >= operátorok is. Ezen párok vagy mindegyikét, vagy egyikét sem írjuk meg. Az == és != operátorok felülírásakor további javaslat, hogy írjuk felül a GetHashCode() és Equals() metódusokat is!

```
public static bool operator ==(datum a, datum b)
{
    return a.nap == b.nap;
}
public static bool operator !=(datum a, datum b)
{
    return a.nap != b.nap;
}
```

Felmerül még egy kérdés: definiáltuk a 'datum' + 'int' esetet, de mi a helyzet ha fordítva írjuk fel a kifejezést?

```
datum d = new datum("1970.01.10");
datum p = d + 10;
datum k = 10 + d;
```

Érdekes, mert bár matematikából tanultuk, hogy a + operátor kommutatív (a két operandus a kifejezésben felcserélhető, és ez nem befolyásolja a végeredményt), de a C# nyelvi fordító ezt nem veszi tudomásul. Az 'int + datum' kifejezést nem fogadja be, jelzi, hogy ilyen paraméterezésű (paramétersorrendű) operátor függvény nincs!

Bináris operátorokra, utolsó példaként nézzük meg, hogyan kell értelmezni két dátum különbségét. Jelentse ez a továbbiakban a két dátum közötti napok távolságát:

```
public static int operator -(datum a, datum b)
{
    return a.nap - b.nap;
}
```

### 26.3. Típuskényszerítő operátorok fejlesztése

A harmadik típusú operátorokról első pillanatban nem is gondolnánk, hogy operátorok. A szokásos példa a double – int konverzió:

```
int a = 10;
double b = a; // implicit
int c = (int)b; // explicit
```

A 'b=a' értékadás során a jobb oldali int típusú érték futás közben double ábrázolási formára konvertálódik, és eltárolódik a 'b' változóba. Ez automatikus művelet, ezt hívjuk implicit típuskonverziónak. A harmadik utasításban a 'c = (int)b' esetén a jobb oldali double értéket explicit módon alakítjuk int típusúvá és int ábrázolási formátumúvá. A típuskonverziós lépéseket az implicit és explicit operátorok végzik.

Ilyen konverziós operátorokat könnyedén fejleszthetünk a saját típusainkhoz:

```
public static explicit operator string(datum d)
{
    return String.Format( ... ); // átgondolandó
}
```

Ez az operátor függvény a datum ⇒ string explicit konverziót támogatja. E pillanattól kezdve írható ilyen felhasználói kód:

```
datum d = new datum("2012.12.02");
string f = (string)d;
```

Ha nem 'explicit' hanem 'implicit' kulcsszót használtuk volna a függvény megírásakor, akkor az 'f' változóba a 'd' értékét közvetlenül is belerakhattuk volna: 'string f = d;' hisz nem szükséges a konverziót kiírni.

Az alábbi kód alapján akár a string paraméterű konstruktort is kihagyhatjuk, létrehozhatunk string értékből dátumot közvetlenebb módon is:

```
class datum
{
    public static explicit operator datum(string s)
    {
        return new datum(s);
    }
}
```

Mivel ezek után létezik string  $\Rightarrow$  datum explicit típuskonverzió, így hibátlan az alábbi kód:

```
datum d = (datum) "2012.12.02";
```

Az implicit típuskonverziós operátorok írása – bár lényegében ugyanolyan egyszerű mint az explicit társaié – mindig tervezési döntés. Az implicit operátorok esetén előfordulhat, hogy véletlenül írtuk le az adott értékadást, de mivel épp létezik implicit konverziós operátor „véletlenül” a hibás kód is működőképes:

```
public static implicit operator int(datum d)
{
    return d.nap;
}
```

Ha létezik implicit dátum  $\Rightarrow$  int konverzió, könnyű egész számmá alakítani a dátumot:

```
datum d = (datum) "2012.12.02";
int n = d; // implicit
```

A sok példa után lássuk a szabályokat:

- A konverziós operátorok egy paramétert fogadnak, és visszatérési típusuk minden esetben különböző ettől a típustól.
- Ha a paraméter típusa A, a visszatérési típus B, akkor a konverziós operátor elhelyezhető akár az A akár a B típust leíró osztályba.
- A típuskonverziós operátor függvények static és public jelzőjűek.
- A függvény tartalmazza vagy az 'explicit' vagy 'implicit' jelzőt is, attól függően, hogy milyen típusú konverziót akarunk leírni.

A konverziós operátor általános alakja:

```
public static <implicit|explicit> operator <típus2>(<típus1> <pname>)
{
    ...
}
```



## 26.4. Záró problémák

Két problémát mutatunk be. C# nyelv esetén, a stringek egyenlőségének és nem egyenlőségének vizsgálata operátorokkal van megoldva, ellenben a kisebb, kisebb-egyenlő és egyéb vizsgálatok nem. Értelmezhető a stringek közötti ilyen jellegű vizsgálat (a kisebb string az, amelyik „előrébb” van az ABC rendben), a telefonkönyv is betűrendbe van rendezve, a számítógépek esetén pedig rendelkezésre áll a UNICODE táblázat, minden karakterhez számkód (sorszám) van rendelve, amely segíti az ilyen jellegű vizsgálatokat. Ugyanakkor nem írható fel a stringek sorrendi vizsgálata operátorok segítségével:

```
string a = "alma";
string b = "falat";
if ( a < b ) ...
```

Sajnos a string típushoz az összehasonlító operátorokat nem írták meg a redmondi fiúk. Helyette ajánlják a String osztály statikus Compare metódusát, mely int típusú értékkel jelzi az A és B string összehasonlításának eredményét:

+1 (pozitív), ha  $A > B$ ,

0, ha  $A == B$ ,

-1 (negatív), ha  $A < B$ .

Ez alapján könnyű lett volna megírni a szükséges operátorokat:

```
public static bool operator <(string a, string b)
{
    return String.Compare(a, b) < 0; // a<b ha negatív
}
```

Tegyük fel, hogy mi már értjük az operátorok írásának titkait, szívesen megírnánk azt a 6-8 operátor függvényt. Sajnos nem tudjuk, ugyanis a szabályok értelmében a függvényeket a 'string' osztályba kell írni, amelynek a forráskódja nem áll rendelkezésünkre. Itt ez a történet véget ér (bár a kiterjesztő metódusok fejezet nyújt némi reményt, de pont operátorfüggvényekre nem alkalmazható az ott megadott módszer).

Másik probléma, amelyet megemlítünk, a konverziós és egyéb operátorok bősége esetén léphet fel. Korábban megírtuk a dátum  $\Rightarrow$  int implicit operátort, valamint a dátum + int összeadást végző operátort. Vizsgáljuk meg a következő kódot:

```
datum d = new datum(12);
datum b = d + 10;           // (1)-es kifejezés
int c = d + 10;             // (2)-es kifejezés
```

Az (1)-es kifejezés is kétértelmű, de a (2)-essel még több a baj. Az (1)-es esetén a 'd+10' jelentheti azt is, hogy adjuk össze a kifejelesztett operátor segítségével a dátumot és az intet, eredményül dátumot fogunk kapni. De jelentheti azt is, hogy a 'd'-t implicit módon

konvertáljuk át int-té, és végezzünk numerikus összeadást a két int érték között. A (1)-es esetén ez hibára vezetne, mivel a kapott numerikus int értéket vissza kellene alakítani dátum típusra implicit módon, de ilyen operátort nem fejlesztettünk ki. A (2)-es esetén azonban eldönthetetlen, hogy 'dátum+int' összeadás után implicit int-re konvertálás eredménye kerül majd a 'c' változóba, vagy 'dátum' implicit int konverzió utáni numerikus összeadás eredménye kerül a 'c'-be. Ez a példa is arra világít rá, hogy az implicit konverziós lehetőségek gyakrabban okoznak semmint oldják meg a problémát. Ha csak explicit konverzió létezne dátum  $\Rightarrow$  int irányban, akkor a fenti kifejezések egyértelműek lennének, míg a második viselkedést az explicit típuskonverziós operátor alkalmazása kényszerítené ki:

```
datum d = new datum(12);  
datum b = d + 10; // datum + int összeadó operátor  
datum c = (datum)((int)d + 10); // int + int, eredmény visszaalakítva  
int e = (int)d + 10; // int + int, eredmény kész
```

## 26.5. Extensible methods

Az extensible methods (kiterjesztő metódus) az öröklődésen keresztüli továbbfejlesztés egy speciális esetére nyújt megoldást. Gyakori az az eset, amikor egy létező osztályunk létező metódusait nem felülírni (módosítani) akarjuk, mindössze új metódussal szeretnénk kiegészíteni azt. A származtatás során a gyerekosztályt egyszerű és természetes ilyen módon kiegészíteni új metódusokkal, de a C# nyelv 3.0-ás verziójától kezdve speciálisan erre a feladatra alternatív módszert is ad.

Hogy egy már létező osztályba utólag (kívülről), az eredeti osztály forráskódjának módosítása nélkül adjunk új metódusokat, helyezzük el a metódust egy statikus osztályba. A statikus osztályok (*static class*) olyan osztályok, amelyek csak statikus metódusokat és mezőket tartalmaznak, példányosítás belőlük nem lehetséges. A statikus osztályokba nem szabad példányszintű konstruktort írni, és ezt a fordító sem teszi meg, nem ír bele alapértelmezett konstruktort.

A kiterjesztő metódusok megírása statikus metódus írását jelenti, de ezen metódusok mindig – furcsa módon – példányszintűek lesznek. A statikus metódus első paramétere ezért minden esetben a példány lesz, ennek típusa a példány típusa lesz. A paraméter típusa tehát egyúttal azt az osztályt is definiálja, amelybe a metódus utólag beillesztésre kerül.

A statikus osztályt névtérbe kell helyezni, és nem szabad beágyazni más osztálydefinícióba, és nem lehet generikus sem. Példaképpen tegyük fel, hogy van egy kör osztályunk, a kör középpontjának koordinátaival, sugarával (adatelemek), valamint vannak metódusok a kör kerület és terület kiszámítására. Egészítsük ki a kör osztályt egy logikai értékkel visszatérő metódussal, mely megadja, hogy a kör középpontja valamely (x vagy y) tengelyre esik-e:

```
namespace Kiegészitesek
{
    public static class kiegészit
    {
        public static bool tengelyes(this kor k)
        {
            if (k.x == 0 || k.y == 0) return true;
            else return false;
        }
    }
}
```

A kód érdekessége az, hogy ez egy static metódus, az első paraméterénél szerepel a 'kor' típus neve, de a 'kor' előtt a 'this' kulcsszó is szerepel. Használata csak akkor lehetséges, ha a statikus osztály névtérét a using-gal megnyitjuk:

```
using Kiegészitesek;
```

Utána már a kódkiegészítés is jelzi, hogy a 'kor' példánynak van ilyen metódusa, és a fordító is elfogadja:

```
kor k = new kor();  
if ( k.tengelyes() ) ...
```

A metódusoknak paramétereik is lehetnek, melyet az speciális szerepkörű első paramétert követően kell feltüntetni:

```
namespace Kiegészitesek  
{  
    public static class kiegészit  
    {  
        public static void mozgat(this kor k, int x, int y)  
        {  
            k.x += x;  
            k.y += y;  
        }  
    }  
}
```

Használata:

```
kor k = new kor();  
// ...  
k.mozgat(10,30);
```

Sajnos ezzel a módszerrel „utólag” nem lehet sem property-t, sem operátor függvényt be-  
rakni az adott osztályba.

## 27. Szerelvények

---

Nagyobb projektek esetén mindenképpen szükséges a megírandó kód valamiféle elosztása, szétoztása, részekre bontása. Ennek több oka is lehet, pl. mert több programozó dolgozik a projekten, mindegyikük szeretné a saját részét a többiekétől elkülönült módon fejleszteni, fordítani, tesztelni. Másik ok lehet, hogy maga a feladat indokolja a feladat részekre bontását. Általános fejlesztési elv a háromrétegű (*multitier, three-tier architecture*) alkalmazásmodell, melynek során a program tevékenységeit három különböző rétegbe soroljuk be, és ezen rétegbeli funkcionálisokat külön fejlesztjük és teszteljük le. A három réteg egyik elképzelés szerint:

- data layer (adatkezelő réteg): melybe a program és a külső adatforrások (jellemzően SQL szerverek, webservicek) közötti kommunikáció sorolható (program-program adatcsere),
- application layer / application logic (alkalmazáslogika): melybe a program fő tevékenységei tartoznak bele, amely tevékenységek az adott alkalmazásra jellemzőek (ha ez egy számlázó program, akkor ide tartozik pl. az év végi nyereség kalkulációja stb.),
- user interface (felhasználói felület): a program interaktivitásai, felhasználó input (billentyűzet, egér) kezelése, a program által megjelenítendő üzenetek kiírása stb.

Ezenfelül előkerül az is, hogy bizonyos tevékenységek kódjai más-más alkalmazásokban újrafelhasználhatóak, így hát érdemes általánosan megírni azokat, és olyan formában tárolni, hogy minél kényelmesebben lehessen azt egy újabb projekt indulásakor elővenni és a projekt részévé tenni. Ha ezen kód a későbbiekben módosulna (hibajavítás, optimalizálás, funkciók bővülése), akkor szintén sokat jelentene, ha az érintett projektekben könnyen és gyorsan lehetne az új változatot beemelni.

Felmerülő probléma, hogy egyes esetekben nem komplett alkalmazás megírása a feladat, hanem csak egy jól körülhatárolt részfeladat elkészítése. Például készítsünk kódot, amely nagy sebességgel kódol/dekódol videó fájlokat, titkosít, tömörít adatot stb. Egy-egy ilyen részfeladat elkészítése nagy szakmai felkészültséget igényelhet az adott területen, és az elkészült függvénygyűjtemény önmagában is termék. A forráskód publikálása ilyenkor nem szerencsés, hiszen a know-how gyakran drágább, mint maga a megírt kód. Hogyan adjuk közre a kész termékünket, ha a forráskód publikálása nem megfelelő megoldás? Hogyan tudják az általunk írt kódrészletet más alkalmazásokba beemelni és használni?

A problémák egyelőre nem állnak szoros kapcsolatban az OOP-vel, hiszen akár Assembly szintű (második generációs) programozási nyelvekkel kapcsolatosan is megfogalmazhatóak. Ezért először vizsgáljuk meg a kérdést és a lehetséges megoldásokat egyéb nyelveken, akár nem OOP környezetben is.

---

Amikor a programozó elkészíti a forráskódot, első lépésben a fordító (compiler) fogja azt értelmezni<sup>36</sup>. A fordító működésének eredménye a tárgykód, amely a forráskódba írt utasításaink gépi kódú változatát<sup>37</sup> tartalmazza. Ez a tárgykód még nem futtatható. Az eset hasonlít arra, mikor az író már megírta a könyvet, kezében a lapok, de az mégsem könyv. A programunk végső formáját a szerkesztő (linker) program állítja elő, akit gyakran kifelejtünk, hisz az ő tevékenysége bár fontos, de nem látványos. (Ellentétben a compilerrel, aki gyakran utasítja vissza a forráskódunkat szintaktikai hibákra hivatkozva.)

Az egyes tárgykódok közötti kapcsolatokat (egyik tárgykódból másik tárgykódbeli függvényt hívunk meg) a linker ellenőrzi le, és oldja fel. Ezért ezt a módszert **statikus szerkesztésnek** (*static linking*) nevezzük.

A compiler által előállított tárgykód érdekes, mivel tartalmazza az általunk megírt kódot, de az mégsem a forráskódunk. A forráskódban mindennek szép, beszédes neve van (változók, függvények), kommentek vannak, jól tördelt, magas szintű vezérlési szerkezeteket használunk (ciklusok, elágazások). A tárgykódban nincsenek kommentek, az általunk használt azonosítóneveket a fordító semmitmondó kódnevekre fordítja le (melyek gyakran egyszerű sorszámok), és a szép vezérlési szerkezeteinket nehezen értelmezhető, ugró utasításokra fordítja át. A tárgykódú változat tehát egyfajta szempontból jóval biztonságosabb változata a kódunknak, mint az eredeti forráskód. Elolvasása és megértése jóval több idő és energia az idegenek számára, a know-how ellopása tehát jócskán problémás.

A tárgykód alapján az eredeti kód, a know-how visszafejtése tehát lehetséges, de körülményes. A folyamatot **reverse engineering** (visszamodellezés, visszafejtés) nevezzük. Ennek során az okok (miért éppen így oldottuk meg a problémát), a körülmények (a tervezés, a tesztelés) ritkán válnak ismertté, csak maga a megoldás technikai részletei derülnek ki. A visszafejtést a legtöbb ország törvényei tiltják, és rendkívül sok súlyos pénzbírsággal fenyegető per alapját képezik.

Ráadásul míg az eredeti fejlesztő rendelkezik az eredeti forráskóddal, addig a visszafejtő csak egy, attól minőségileg jóval gyengébb változattal. A további módosítások, fejlesztések, hibajavítások sokkal könnyebben kivitelezhetőek az eredeti fejlesztő által, mint a visszafejtő által birtokolt példány esetén.

Ha nem cél a visszafejtés megakadályozása, a forráskód csak „házon belül” mozog, akkor is hasznos a tárgykód formájú változat, hiszen az egyes projektekbe nem kell a kódrész forráskódját beemelni (berakni valamely mappába, a sok hasonló nevű forráskód fájl közé, vagy esetleg a függvény forráskódját egy hosszabb forráskód közepébe fizikailag be-

---

<sup>36</sup> Maga a fordító is több fázisban működik, első lépésben a lexker szétbontja a szöveget elemi egységekre (tokenek, szimbólumok), majd a parser fogja a szimbólumokat szintaktikailag elemezni (sorrendiség, típus-helyesség), végül a kódgenerálási fázisban (mely gyakran a parser működésébe épített, vele párhuzamosan fut) készül el a generált kód. Ezt esetleg még egy optimalizálási fázis módosítja, így készül el a végső változat.

<sup>37</sup> Nem feltétlenül a processzor gépi kódja, C# esetén pl. a virtuális gép gépi kódja. Általánosan mondhatjuk, hogy egyszerűen egy másik programozási nyelvre fordítja át, melynek absztrakciós szintje jellemzően alacsonyabb. Például egyes C, C++ fordítótól kérhetünk assembly nyelvi kimenetet is.

másolni), hiszen ez esetben minden fordításkor a fordítónak el kell olvasnia és értelmeznie azt, majd kódot kell rá generálnia újra és újra. Helyette elég ha csak a linker találkozik vele, és építi be a végső, futtatható változatba. A kódrész verziójának változásakor egyszerűen újrafordítjuk, és a kapott tárgykódokat adjuk át az érintett projektekbe.

Azonban a statikus linkelés módszerének vannak bizonyos problémái:

- Amennyiben a függvényeink módosulnak, újrafordítjuk, bemásoljuk az érintett projektekbe, utána azokat is újra kell fordítani, és cserélni a futtatási környezetében (a felhasználók számítógépein).
- Nehéz követni, hogy mely projekteket érint, nem szabad kihagyni egyet sem, mindenütt cserélni kell.
- A programok nagy méretűek lesznek, a tárgykódokat a linkernek be kell építeni a futtatható (.exe) állományba.

Létezik a **dinamikus linkelés** módszere is (*dynamic linking*). Ennek során a futtatható állományba nem kerül be fizikailag minden, a futásához szükséges függvény gépi kódú változata. A futtatható állomány (.exe) éppen ezért lényegében önmagában nem is futásképes, hiányos. Azonban kiegészítésképpen további, a linker által előállított „csomagok” is rendelkezésre állnak, melyekbe a hiányzó függvények gépi kódú változata kerül. Úgy kell elképzelnünk, mintha a teljes, végső, mindent tartalmazó, futásképes .exe állományt darabokra szedtük volna, mint amikor egy képet puzzle darabokra tördelünk. Együtt még mindig kiadják a teljes és futásképes változatot, de külön-külön életképtelenek.

Ha egy teljes, egész, futásképes .exe állományt darabolunk, lesz pontosan egy darabka, amelybe a főprogram (Main függvény) esik, és lesznek darabok amelyekben nem lesz főprogram. A kitüntetett darabkát a linker továbbra is .exe kiterjesztéssel menti a lemezre, mivel őt kell majd indítani a felhasználónak, de a további darabokat más kiterjesztéssel állítja elő. A további darabok kiterjesztése Windows operációs rendszerű platformon **.DLL** (*dynamically linked library*)<sup>38</sup>.

A **szerelvény** (*assembly*) elnevezés tehát arra utal, hogy a vonat áll egy mozdonyból (.exe), és kocsikból (.dll), úgy együtt adják ki a teljes egészet. A linker amikor valamely egységet szerkeszt, vagy állít elő, akkor tudnia kell annak típusát (.dll vagy .exe). Amennyiben DLL-t kell neki előállítania, nem számít hibának, ha nincs benne Main függvény (jellemzően nincs is). Azonban a futtatható kód (.exe) előállításakor annak belépési pontját is ki kell töltenie, tehát abban Main függvénynek szerepelnie kell.

<sup>38</sup> Egyéb kiterjesztések is előfordulhatnak, pl. „drv”, „fon” stb.

## 27.1. A Windows DLL

A DLL technológia támogatása a Windows-os platformon az első verzióktól kezdve szerepel. Maga a Windows operációs rendszer is számtalan DLL fájlból áll, melyek együttműködése adja az operációs rendszer működését. A Windows operációs rendszer DLL-jeiben lévő függvényeket nemcsak a Windows DLL-jei hívhatják meg, hanem tetszőleges felhasználói program is. A DLL-ek ilyen szempontból nem biztonságosak.

A linker képes a futtatható állományt (.exe) úgy is hibátlanak minősíteni és előállítani a futtatható kódot, hogy nincs benne tárolva a futáshoz szükséges minden függvény. A hiányzó függvények esetén azonban ismernie kell:

- az adott függvény milyen nevű külső fájlban (dll) szerepel,
- a dll belsejében milyen néven, azonosítóval van a függvény<sup>39</sup>.

Ezeket az információkat a szerkesztőnek bele kell mentenie a futtatható programba. Amikor az operációs rendszer egy ilyen program indítására készül, kiolvassa a futáshoz szükséges DLL-ek neveit, és ellenőrzi, hogy azok is rendelkezésre állnak-e, és valóban szerepel-e bennük az adott azonosítójú függvény. Amennyiben ezen DLL-ek is további DLL-ekre hivatkoznak, azokat is ellenőrzi, hogy megvannak-e a gépen. Ha egy is hiányzik, akkor a program indítására nincs lehetőség, az operációs rendszer megtagadja az indítást.

Ha van egy DLL fájlunk, honnan tudhatjuk meg milyen nevű függvények szerepelnek benne? Ha mi készítettük a DLL fájlt, akkor természetesen tudjuk a függvények neveit, paraméterezését, tevékenységüket. Ha nem mi írtuk, akkor szükséges, hogy rendelkezünk valamiféle dokumentációval. A C, C++ nyelveken nagy segítségünkre van maga a nyelv is, mivel itt szokás ún. header (fejléc) fájlokat készíteni, amelyek kódot nem, de a függvények fejrészének leírását tartalmazzák. A fájlokat nem külön dokumentálási célok miatt készítik a programozók, hanem szokásos és szükséges részei a fejlesztési folyamatnak, tehát eleve rendelkezésre állnak. A DLL-ekhez csatolhatják a header (.h kiterjesztésű) fájlokat is, így a szükséges információk egy része rendelkezésre áll.

Ha semmilyen dokumentációnk nincs, akkor sem teljesen reménytelen a helyzet, mivel a DLL fájlokban a bennük szereplő függvények nevei listaszerűen szerepelnek. A lista mindenképpen szükséges a DLL belsejében, hiszen a program indulásakor az operációs rendszer ez alapján ellenőrzi, hogy a szükséges függvények valóban rendelkezésre állnak-e. A lista egyszerű segédprogramokkal kinyerhető, bár ez a tevékenység már súrolja a reverse engineering határát.

Vegyük észre azonban, hogy a DLL-ben nincs tárolva a függvények paramétereinek száma és típusa. A C nyelvi header fájlok tartalmazzák, de annak hiányában csak a függvények neveit ismerhetjük meg a segédprogram segítségével. A függvény paraméte-

---

<sup>39</sup> az a függvénynév amellyel mi hivatkozunk rá a programunkban, és az a „függvénynév” amely a DLL-ben szerepel – eltérő lehet.



reinek megfejtése már ténylegesen a függvény törzsének megértésével válna lehetővé, de ez már valóban reverse engineering lépés, amely általában illegális.

Windows platformon egy idegen DLL használatba vétele dokumentáció hiányában egyáltalán nem könnyű feladat, gyakran időben és energiában többbe kerül, mint a kívánt funkciók újrakódolása.

Magukról a Windows operációs rendszer DLL-jeiről, azok függvényeiről, a függvények neveiről, a paraméterek számáról, típusáról, a függvény által végzett tevékenységről a Microsoft ad ki dokumentációt, melyet Windows API-nak (WinAPI, Windows Application Programming Interface) nevezünk. Ennek több változata is van, mivel a különböző Windows operációs rendszerek eltérnek egymástól, illetve ugyanazon Windows operációs rendszer 32 bites és 64 bites kiadásai is különböznek egymástól. A Windows '95 verzióban a WinAPI több mint 9000 függvényt, körülbelül 29 000 konstanst, és nagyjából 4800 típusdefiníciót írt le<sup>40</sup>.

*Megjegyzés:* a Linux operációs rendszer alatt a DLL-ek .so (shared object) néven ismeretek. Ezen a platformon a Windows-os programok nem indulnak el, ennek számos oka van, többek között az, hogy nincsenek meg azok a Windows DLL-ek, amelyek a programok futásához szükségesek. A Linux világában létező project a WinE, amelynek célja éppen ezen DLL-ek megalkotása, hogy a Windows-os programok indíthatóak legyenek Linuxon is.

## 27.2. A DLL pokol

A DLL-ek segítségével számos probléma megoldható. A DLL önálló fordítási és szerkesztési egység. A program többi egységétől függetlenül, külön fájlként van jelen a lemezen. Ha a DLL valamely függvényén módosítunk, újra tudjuk fordítani, és az érintett projektekben egyszerűen fájl szinten felülírjuk az új DLL változattal. Ha a programjaink különböző telephelyeken, ügyfeleknél futnak, elég csak ezt a DLL-t elküldeni cserére. Ha a programunk automatikusan frissíti magát, egy nem túl nagy méretű fájl kell csak letöltenie és felülírnia, ami a hálózati erőforrások használatának szempontjából is jobb. A DLL még mindig lefordított kód, tehát a DLL alapján a know-how visszafejtése, az algoritmusok dekódolása hasonlóan nehéz mint a tárgykódok esetén.

Elképzeltető az az eset, amikor ugyanazon cég több alkalmazása is fut ugyanazon számítógépen. Az alkalmazások mindegyike (vagy többjük) használja a cég egyik DLL-jét (közös szolgáltatások függvényei). Egy ilyen verziócserekor ugyanazon a gépen több példányban is szerepelhet tehát ugyanazon DLL verziója, mindegyik cseréjét el kell végezni.

Persze megoldható, hogy a DLL-ek egy külön gyűjtőmappában vannak, és az alkalmazások közösen használják ugyanazon DLL fájlt. Ekkor a cserére csak egyszer van szükség,

---

<sup>40</sup> Forrás: <http://www.spinellis.gr/pubs/jrnl/1997-CSI-WinApi/html/win.html>, *A Critique of the Windows Application Programming Interface*, Diomidis Spinellis, University of the Aegean

hiszen a DLL csak egy példányban létezik a lemezen. Hasonló elv mentén megoldható, hogy egyik alkalmazás egy másik alkalmazás DLL-jét használja fel. Ez gyakori, ha az alkalmazásfejlesztők megállapodnak egymással, átadják a szükséges dokumentációkat, s így egy olyan alkalmazáscsoportot fejlesztenek ki, melyek funkcionalitása és együttműködése magasabb szintű mint a versenytársaiké.

Jellemzővé vált emiatt az alkalmazások telepítésekor a DLL-ek megosztása. Egyik hely volt erre a C:\Program Files\Common Files alkönyvtár, illetve a C:\Windows\System32 alkönyvtár, ahova az alkalmazások telepítéskor megosztott DLL-jeiket bemásolhatták. Szükségtelen volt ez, ha az alkalmazások felismerték egymást a lemezen, kiolvashatták, hogy a partner alkalmazás milyen alkönyvtárba települt fel, és onnan direkt módon elérhették és hivatkozhatták a DLL-eket. Erre támogatásul a korábban a Windows központi INI fájlt (win.ini), illetve később ennek robosztusabb változatát, a Windows Registry-t lehetett használni.

A **DLL HELL** (dll pokol) néven ismert jelenség ebből az elképzelésből fakadó gyakorlati problémák összességére utal:

- Valamely alkalmazás bemásol a közös mappába egy DLL-t, felülírva egy ott már korábban, más alkalmazás által telepített DLL-t. Ennek következtében a korábban telepített, jól működő alkalmazásaink nem indulnak el, vagy hibásan működnek. A hiba oka nehezen deríthető ki, mivel a telepítés menete a felhasználó által jellemzően ismeretlen, és nem lát közvetlen összefüggést a két esemény (új alkalmazás telepítése, régiék meghibásodása) között.
- Az alkalmazások eltávolítása (uninstall) az általuk megosztott DLL-ek törlését is jelenti, melynek következtében a tőle függő más alkalmazások működése megváltozik, meghibásodik.
- Az alkalmazások eltávolításkor a megosztott DLL-eket fent hagyják a lemezen. A DLL-ek később már nehezen azonosíthatóak, nehezen határozhatóak meg, melyik alkalmazáshoz tartoztak eredetileg, és mikor törölhetők le a lemezről.
- Valamely alkalmazás frissítése során az általa megosztott DLL-ek is frissülnek, de az ezeket használó más alkalmazások nem feltétlenül kompatibilisek az új DLL-el, így azok működésében zavarok keletkeznek, meghibásodnak.
- Ennek megoldásképp a megosztott DLL-ek korábbi verzióit is megtartották, és az új verziókat is. Ez problémás, mert az operációs rendszer nem kezelte a DLL-ek verzióinformációit, a programok indulásakor csak az adott nevű DLL-ek létezését ellenőrizte, így a különböző verziószámú változatnak igazából ugyanaz a fizikai fájlneve kellett, hogy legyen, ami miatt a verziók kezelése rendkívül nehézkes volt.

### 27.3. A .NET C# DLL

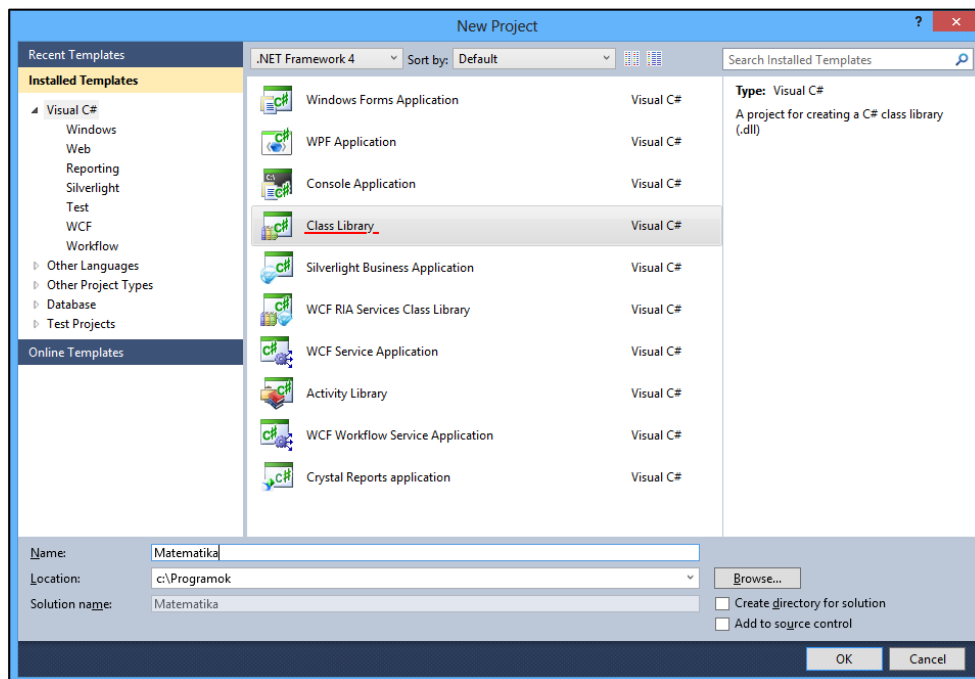
A .NET alatt is van lehetőségünk DLL készítésére. A .NET DLL kiterjesztésében, és funkciójában is ugyanezt a szerepet tölti be, mint az eddig ismertett DLL-ek, de vannak alapvető különbségek.

Először is az, hogy bár a kiterjesztés itt is DLL lesz, de a fájlok belső szerkezete és felépítése alapvetően eltér. Legjelentősebb különbségek:

- A .NET DLL nem gépi kódú, hanem a futtató rendszer virtuális gépi kódjára fordított változat.
- Nem egyszerűen csak függvények vannak bennük, hanem teljesen OOP-s támogatást kaptak, vagyis komplett osztályok, védelmi szint információk, és minden más is helyet kap bennük.
- A DLL készítő cég digitális azonosítója, a verziószám szerves részét képezi a DLL-nek, így a DLL azonosításánál ezek az információk is megadhatóak.
- A DLL betöltésének folyamatát nem maga az operációs rendszer, hanem a futtató rendszer végzi, aki egy más DLL megosztási mechanizmust is ismer, melyet GAC-nak nevezünk (lásd később).

### 27.4. A DLL készítésének és felhasználásának lépései

Amennyiben DLL-t szeretnénk a VS-sel készíteni, projekt típusként a **Class Library**-t kell választani (File / New Project / Class Library):



A Class Library esetén nem a szokásos főprogrammal rendelkező mintát kapjuk kiinduló forráskódnak, hanem egy jóval egyszerűbbet:

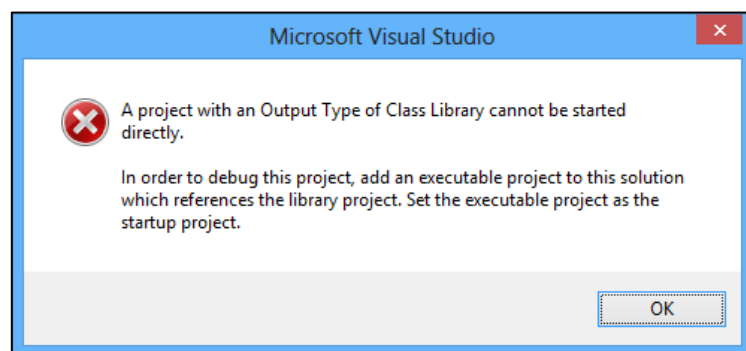
```
namespace Matematika
{
    public class Class1
    {
    }
}
```

A névtérben egy 'public' módosítóval ellátott osztály szerepel csak, és nincs Main függvényünk. Ez normális, hiszen a DLL-ek nem tartalmaznak főprogramot. A class szó előtt a public módosító fontos lesz később.

Első lépésként írjunk egy egyszerű függvényt. A függvény paramétereiként egy string-et kap, mely egy egész számot ír le (pl. bekérték Console.ReadLine() segítségével). Ha sikerül parse segítségével kiemelni a stringből, akkor a függvény ezen értéket adja meg válaszképp. Ha nem sikerül, akkor a második paraméterként megadott alapértelmezett szám lesz a függvény eredménye:

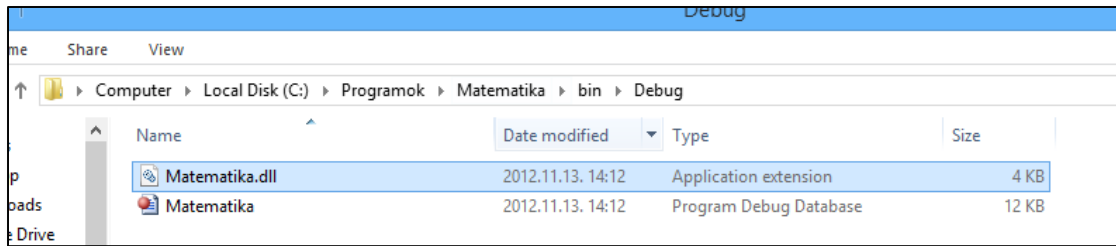
```
public static int IParse(string ertek, int defErtek)
{
    try
    {
        return Int32.Parse(ertek);
    }
    catch { }
    return defErtek;
}
```

Ez az a függvény, amelyet sok alkalmazásunkban fel kívánunk használni. Fordítsuk le a DLL-t. A szokásos zöld nyíl ikonra kattintás kellemetlen problémával jár, az ugyanis a programindítás ikonja, márpedig egy DLL-t a főprogram hiányában nem lehet elindítani („a project with an output type of class library cannot be started directly” = „class library típusú project nem indítható direktben”):

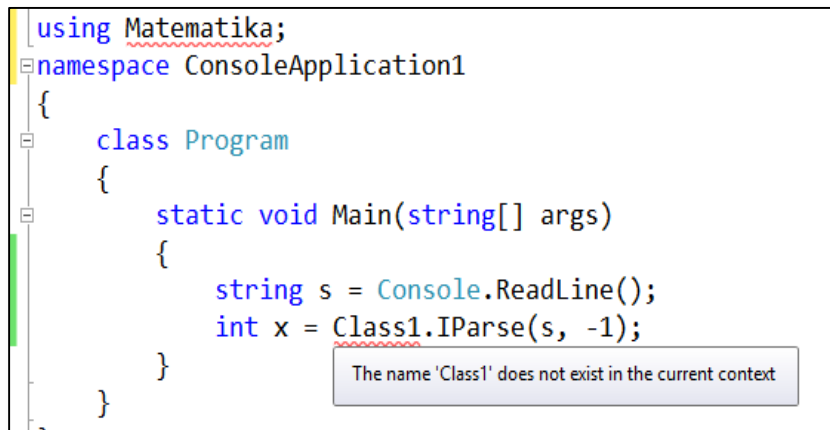


A DLL fordításához ezért inkább a Build / Build solution, vagy a Build / Rebuild solution menüpontokat szoktuk használni. A menüpontok csak fordítást és kódgenerálást végeznek, indításra itt nem kerül sor. Ugyanakkor a szintaktikai hibákat ellenőrzi és kijelzi a rendszer.

A fordítás után ha megnézzük az alkönyvtárszerkezetet, a szokásos Bin/Debug alkönyvtárban most nem .exe, hanem .dll kiterjesztésű fájlt fogunk találni:

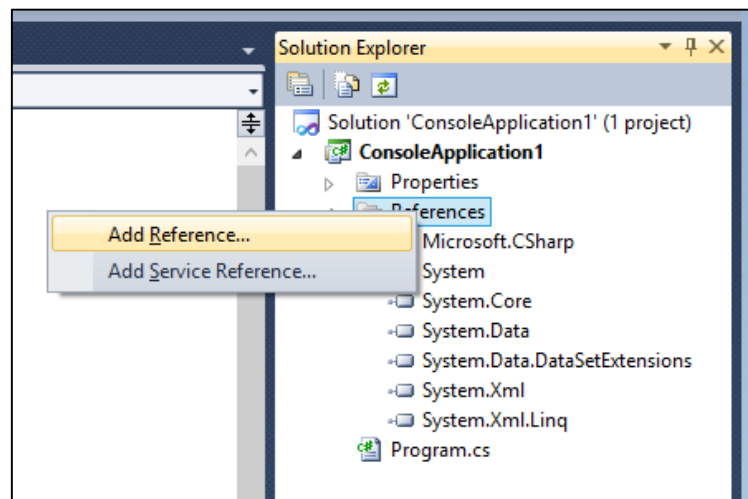


Szükséges egy másik project, pl. egy konzolos alkalmazás, amely tartalmaz főprogramot. Itt választhatunk, hogy elindítjuk a VS-t még egy példányban, vagy most ezt a projektet bezárjuk és újat kezdünk, esetleg a Solution Explorer segítségével újabb projektet adunk ugyanezen solution-höz. Válasszuk az egyszerűség kedvéért az elsőt, indítsunk el még egy VS-t, készítsünk el egy konzolos programot, amely meghívja ezt a függvényt:

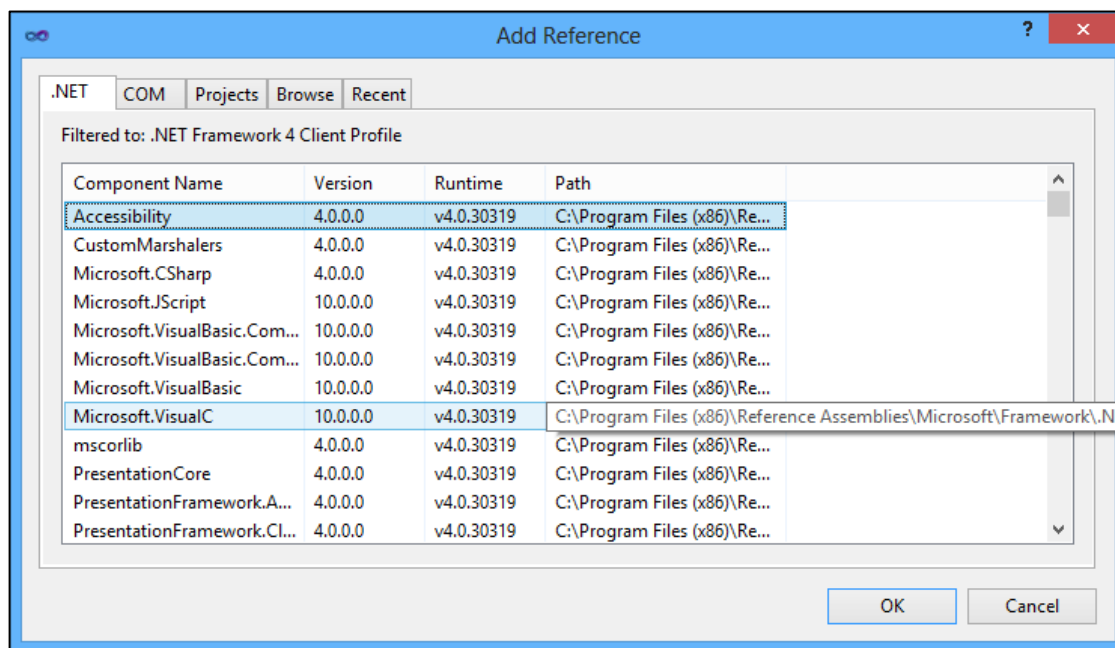


A hibaüzenet arra utal, hogy a fordító nem ismerte fel az osztálynevet. A névtér using-gal való megnyitása is hibás. A probléma az, hogy konzolos projekt írásakor a fordító nem tud még a DLL létezéséről, nem ismeri a nevét, nem tudja melyikre gondolunk. Nyilván nem fogja kitalálni a gondolatunkat és átnézni a lemezünket, keresve, hogy van-e valahol egy DLL, amiben van olyan osztály. Nekünk kell azonosítani és a projekthez adni a DLL-t manuálisan.

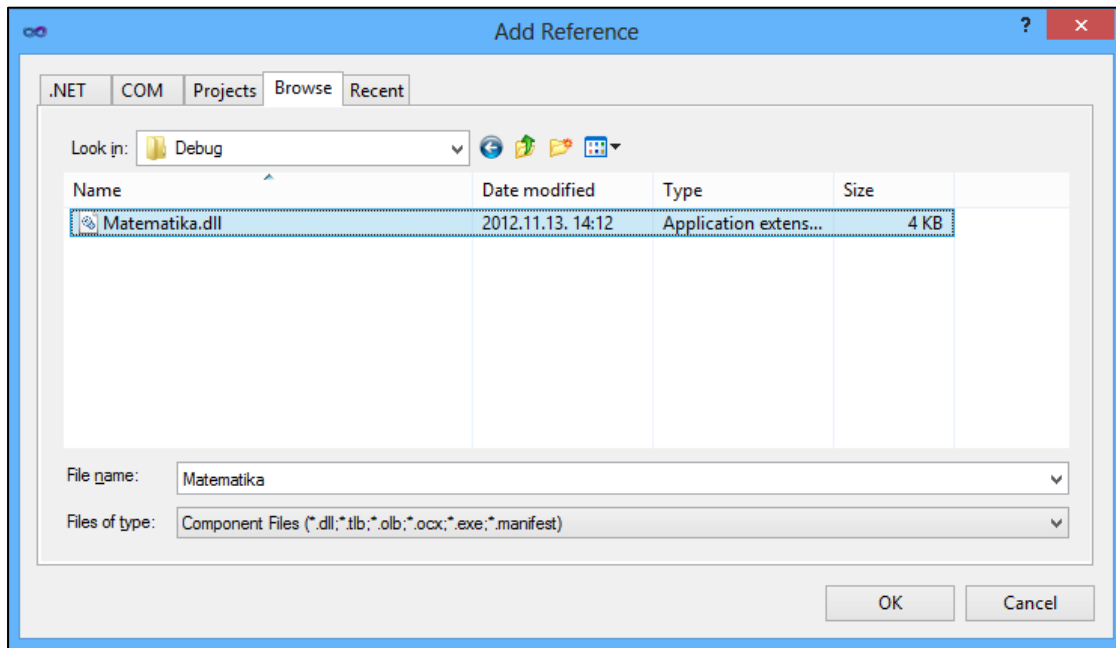
A konzolos projektünk Solution Explorer részén szerepel egy References ág, melyet megnyitva megnézhetjük, milyen külső DLL-eket használ a programunk. Ezen DLL-ek listájára kell felvenni az új DLL-t. Kattintsunk a References bejegyzésre a jobb egérgombbal, és válasszuk ki az Add Reference menüpontot:



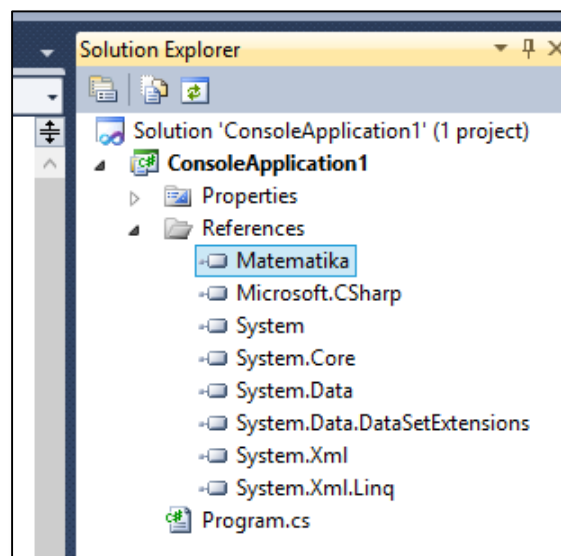
Egy elég komoly, összetett párbeszédablak bukkan fel, amelynek segítségével azonosíthatjuk a hozzáadni kívánt DLL-t:



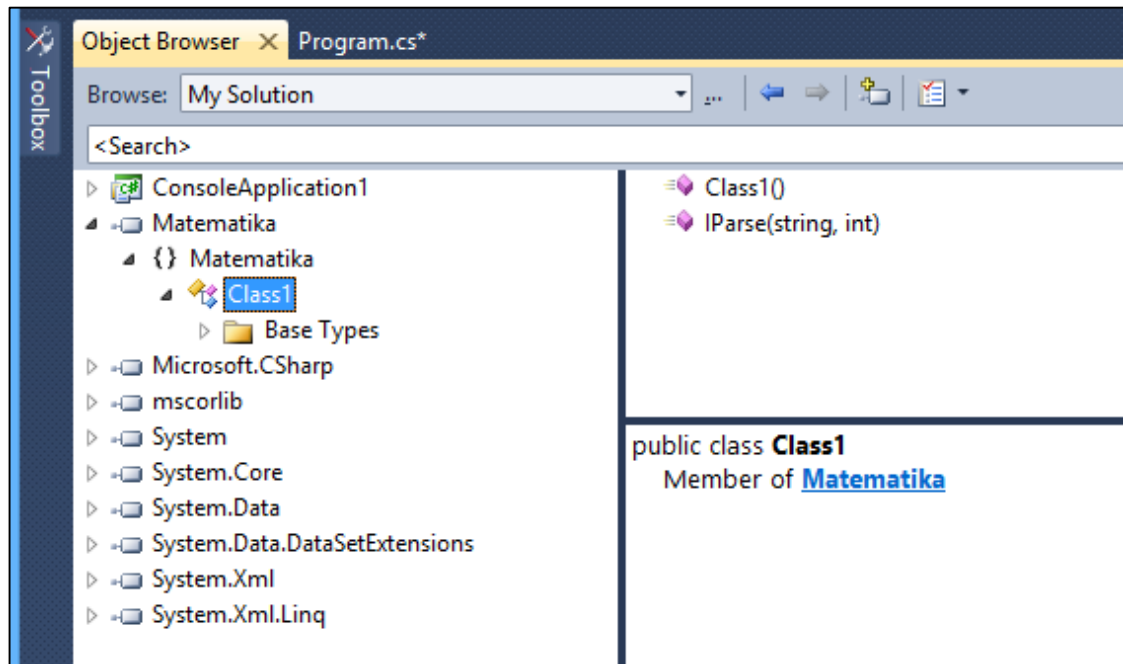
A .NET fülön a GAC-ban (lásd később) tárolt DLL-ek kerülnek listázásra, a COM fülön a rendszerbe a régebbi Windows-os technikát (Component Object Model) támogató regisztrált DLL-ek, a Projects fülön pedig az ugyanezen solution-ben szereplő DLL-ek kerülnek listázásra. Nekünk a Browse fül kell, ahol a diszken lévő alkönyvtárakban keresve azonosíthatjuk a DLL-t amelyet hozzá kívánunk adni. Keressük ki a C:\Programok\Matematika\Bin\Debug alkönyvtárat (ahova a DLL generálódott), és kattintsunk a DLL-re:



Ha jól csináltuk, a DLL felkerül a References listára, amelyet a Solution Explorer-ben látnunk is kell:



Ha nem mi írtuk a DLL-t, és nem rendelkezünk dokumentációval, akkor nem tudhatjuk, hogy ebben a DLL-ben milyen osztályok, és azokban milyen függvények szerepelnek. Kattinthatunk azonban a DLL-re jobb egérgombbal, és választhatjuk a View in Object Browser menüpontot:



Néhány kattintással az Object Browser ablakban láthatóvá válik, hogy ebben a DLL-ben van egy Matematika névtér (a bal oldali fában {} ikon jelzi a névtér), illetve jobb alul a „member of Matematika” is jelzi, hogy a Matematika névtér tagja. Láthatjuk, hogy ebben a névtérben van egy Class1 nevű osztály, melyben két függvény van. Egyik a konstruktor (mi nem írtunk konstruktort, de az alapértelmezett konstruktor generálódott számára), illetve az IParse függvény, melynek egyik paramétere egy string, a másik pedig int. Ez máris sokkal több, mint amit egy nem .NET-es DLL-ről ki tudnánk deríteni.

Ami nincs itt, hogy mi a szerepe ennek a függvénynek, mit csinál, és mi az értelme az egyes paramétereknek. Ezen segíthetünk, ha visszalépünk a DLL projektre, és dokumentációs megjegyzést írunk a függvényhez:

```

/// <summary>
/// A függvény hasonlóan az int.Parse-hez, stringből
/// képez számot. Ha nem sikerül, akkor az alapértelmezett
/// értéket adja vissza.
/// </summary>
/// <param name="ertek">a string amiből a számot ki kell nyerni</param>
/// <param name="defErtek">alapértelmezett visszatérési
/// érték hiba esetén</param>
/// <returns>a stringből kinyert szám vagy az alapértelmezett érték</returns>
public static int IParse(string ertek, int defErtek)
{
    try
    {
        return Int32.Parse(ertek);
    }
    catch { }
    return defErtek;
}

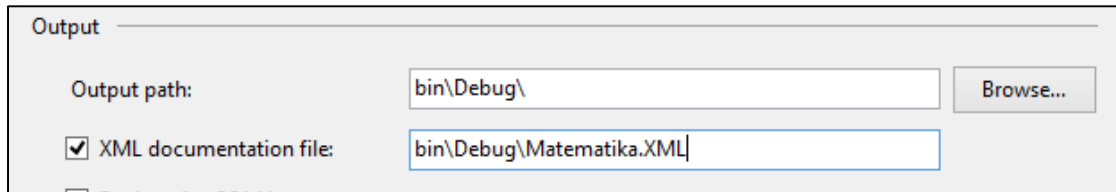
```

A dokumentációs megjegyzések (komment) három darab perjel segítségével írhatóak le. Egy függvény esetén le kell írni az összefoglalást, összegzést (summary), vagyis, hogy mi

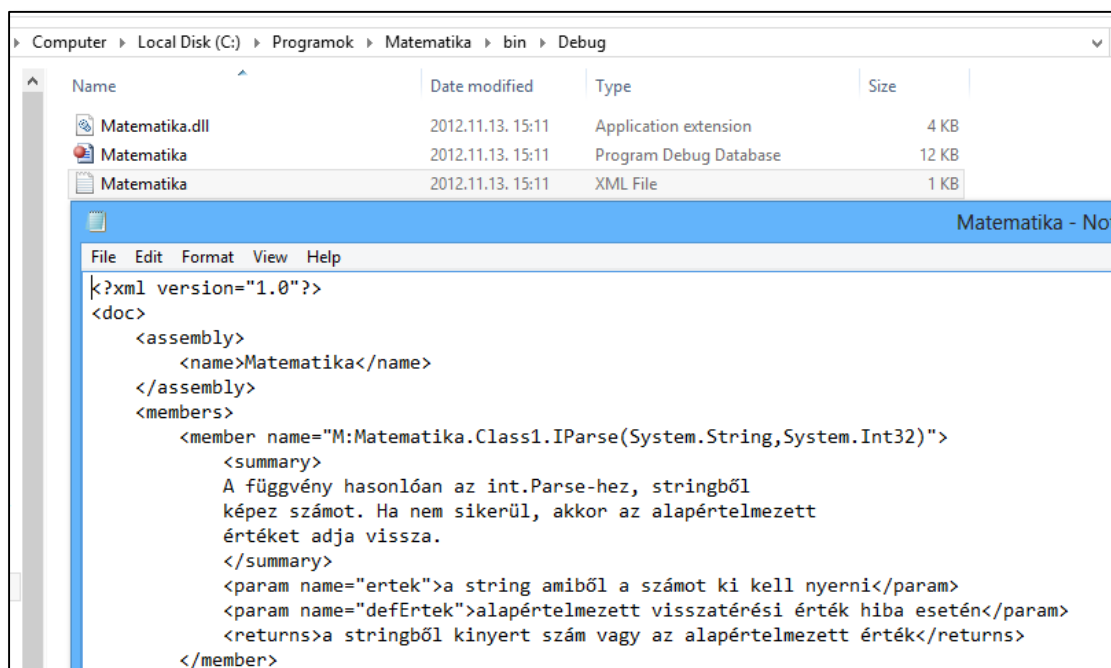


a függvény feladat, mit fog csinálni. A paramétereiről (param) egyesével le kell írni mi a szerepük, a lehetséges értékeit, valamint függvény esetén le kell írni a visszatérési érték (returns), hogyan képődik.

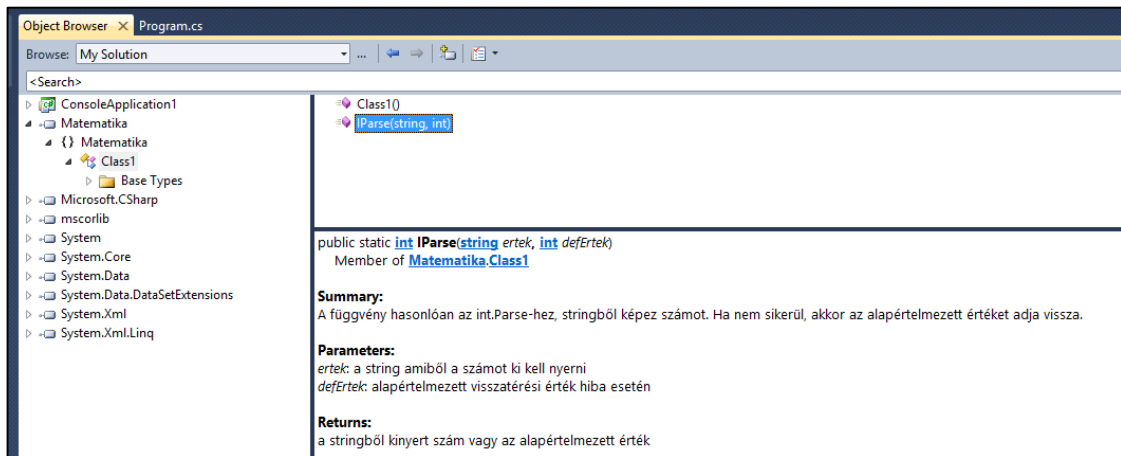
A dokumentációs megjegyzés forráskódba írása önmagában még nem elég. A fordítót utasítani kell, hogy ezeket a kommenteket gyűjtse ki egy xml fájlba, és helyezze el a DLL mellé a mappába. A Solution Explorerben a projekt nevére kattintva (jobb klikk) válasszuk ki az Options menüpontot, majd a Build részen alul pipáljuk ki az XML Documentation File jelölőnégyzetet:



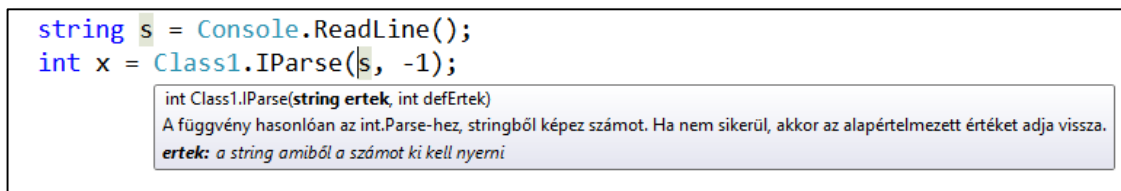
Az újrafordítás után (Rebuild Solution) a DLL mellett egy XML fájl is megjelenik. A kigyűjtött kommentek tartalma:



A konzolos projekthez visszatérve, az IParse függvényt kiválasztva megjelenik a komment az Object Browser-ben (bár ehhez frissülnie kell a DLL-nek, mely történhet sikeres fordításkor, de a VS korábbi verzióiban gyakran arra volt szükség, hogy az érintett DLL-t eltávolítsuk a referencia listáról, majd újra hozzáadjuk):



Ezen felül a forráskódban a függvényre hivatkozás során is meg fog jelenni a megfelelő segítség:



## 27.5. A DLL és a GAC

A GAC – Global Assembly Cache egy mappa, ahol az alkalmazások megoszthatják egymással a DLL-eket. A mappa helye lehet pl. a

„C:\Windows\Microsoft.NET\assembly\GAC\_32”

alkönyvtár, de framework verzió, windows verzió esetén eltérő is lehet. A GAC mappában DLL-eket tárolunk, de csak digitálisan aláírt DLL-ek kerülhetnek ide. A digitális aláírás a DLL készítő céget azonosítja, így ha két különböző cég is készítené ugyanazon nevű DLL-t, mindkettő békésen elfér a GAC-ban, mivel a rendszer meg tudja őket különböztetni egymástól. Másrészről ugyanazon cég ugyanazon DLL-jéből is képes tárolni több különböző verziót. A GAC tehát ezen szempontok mentén megbízhatóan működik.

A GAC-ba saját DLL-t a gacutil.exe segítségével helyezhetünk el. Ez egy parancssori eszköz, command ablakból szoktuk használni. DLL telepítéskor (install) a /i kapcsoló segítségével lehet DLL-t a GAC-ba másolni:

```
gacutil /i sajátom.dll
```

A gacutil ellenőrzi, hogy a DLL digitális alá van-e írva, illetve kiolvassa a verziószámot, majd a DLL-t telepíti a GAC-ba. A továbbiakban a Visual Studio-ban az Add Reference menüpontban meg fogjuk találni ezt a DLL-t, és hozzáadhatjuk a projekthez.

## 27.6. A DLL és az OOP

A DLL-ek készítése tehát egyszerű lépés a VS segítségével. Vizsgáljuk meg, hogy nyelvi szempontból mikre kell ügyelni!

Egy DLL több osztálydefiníciót is tartalmazhat, amelyek közül néhány publikus, a külső alkalmazás számára készült; míg más osztályok esetleg csak belső használatra készültek. Ezek vagy titkos osztályok (részét képezik a féltve őrzött know-how-nak), vagy egyszerűen csak segédosztályok, melyeket nem szükséges nyilvánosságra hozni.

A külső alkalmazásból elérhető osztályoknak 'public class' jelzővel kell rendelkezniük, míg a belső titkos vagy segédosztályok egyszerűen 'class'-ok, mindenféle jelző nélkül (ami lényegében megegyezik a 'private class' fogalmával, holott ez a jelző nem írható le a class elé). A DLL-ben értelemszerűen kell legalább egy 'public class'-nak lennie<sup>41</sup>, különben a külső alkalmazás felé használhatatlan lesz.

A public class a külső alkalmazásban ugyanúgy jelen van, mintha azt ott helyben írták volna. Ha ugyanazon a gépen van a DLL forráskódja is, és debug-ként került a DLL fordításra, akkor a főprogram nyomkövetésekor a VS képes belemenni a DLL forráskódjába is, valamint a DLL-be is rakhatunk töréspontot, a külső alkalmazás futtatásakor a VS a törésponton meg fog állni. Ez segíti a DLL tesztelését.

A public class sok függvényt tartalmazhat. Közülök a public és protected jelzőjűek a külső alkalmazásban is elérhetnek (a public direktben, a protected csak akkor, ha gyerekosztályt készítünk a dll belsejében lévő public class-ból). Ugyanakor mi történik, ha egy publikus függvényünk paramétere egy olyan típusú adat, amely típus szintén a DLL-ben kerül deklarálásra, de nem public?

```
// nem public !!!
enum munkanapok { hetfo, kedd, szerda, csutortok, pentek }

public static void naploIr(munkanapok nap, string uzenet)
{
    // ...
}
```

Inconsistent accessibility: parameter type 'Matematika.Class1.munkanapok' is less accessible than method 'Mat

Inkonzisztens elérhetőség! A naploIr() függvény nem lehet publikus, a külvilág ugyanis nem tudná meghívni, ha a külvilág nem tudja használni a munkanapok típust! Ha egy metódus elérhető a külső alkalmazás felé, akkor a paramétereinek típusa, a visszatérési érték típusa is feltétlenül publikus. Érdekes, hogy az esetleg benne feldobott saját kivétel típusa nem kell, hogy elérhető legyen a külvilág felé, ugyanakkor ha nem érhető

<sup>41</sup> Mivel a névtérben szerepelhet osztályon kívül még interface, struct, enum, delegate is, a pontos megfogalmazás szerint ezek közül legalább egyiknek public-nak kell lennie, különben a DLL nem oszt meg semmit az alkalmazással.

el, akkor a külvilág nem tud rá ilyen típusú catch ágat definiálni, csak a legközelebbi elérhető őszosztály típusára.

### 27.7. A DLL kiegészítő védelmi szintjei

Ha a DLL-ben van egy nem publikus osztály, attól a benne lévő függvények, mezők elérhetősége akár publikus is lehet. Ez kényelmes, mivel az elérhetőség a DLL belsejében lévő kód számára teljesen nyitott (megbízható kódterület, ugyanaz a fejlesztő fogja használni ezeket a mezőket és a függvényeket, aki az osztályt is készítette), míg a külvilág felé ezek nem látszódnak, a teljes osztály és tartalma private-szerűen védett.

Ugyanakkor a publikus osztályok mezőire és függvényeire meggondoltan adjunk ki publikus hozzáférést, ha a külső kódot egy másik fejlesztő fogja írni. Ha levesszük ezek elérhetőségét private-re, akkor a DLL belsejében is kényelmetlenségünk lesz, holott nem saját magunk előtt kívánjuk óvni a mezőket és függvényeket. Mi a megoldás?

Az egyes mezőkhöz és függvényekhez a szokásos public / protected / private védelmi szinteken kívül két új védelmi szintet is használhatunk:

- protected internal,
- private internal.

Az internal (belső) védelmi szint a DLL belsejre vonatkozik, egyenértékű a public védelmi szinttel. Egy 'private internal' védelmi szintű mező tehát a DLL szemszögéből nézve publikus, a külső alkalmazás felé private. A 'protected internal' pedig belülre publikus, kívülre protected. A harmadik lehetséges páros, a 'public internal' nem létezik, mivel az a belülre is publikus, kívülre is publikus változat lenne, de ezt az egyszerű 'public' is leírja.

## 28. Callback

---

A callback egy olyan technológia, amely szintén a programozás történetének kezdete óta rendelkezésre áll, és bizonyította létjogosultságát. Utolsó fejezetek egyikében beszélünk róla, mert hatásmechanizmusa az OOP elveivel ellentétes, sőt, a legtöbb elvet ki lehet kerülni segítségével. Ugyanakkor a benne rejlő elvek nagyszerűsége és hatékonysága miatt mindenképpen érdemes megismerni a programozás ezen módját.

A történet a pointerekkel kezdődött. A pointer egy olyan adattípus, amely a C# referencia típusával rokon. Egy pointerben egy memóriacímet lehet tárolni. Ez a típus az alacsony szintű programozási nyelveken került bevezetésre és használatra, de a magasszintű nyelvek igyekeznek kiküszöbölni, mivel segítségével a memóriát ugyan nagyon hatékonyan használhatjuk, de a fordítónk szinte minden ellenőrzési mechanizmusát kikerüljük, így az előállt kód bár gyorsan fut, de csak a programozó ügyességén múlik, hogy helyesen működik-e vagy sem.

Egy futó program tipikusan három területen használja ki a számítógép memóriáját. Egyik területre (kódterület) töltődik be a program kódja (a függvények törzsei stb.). A második terület az adatterület, ahol a változóink kerülnek tárolásra (ez a GC fenntartása). A harmadik a verem, ahova a függvények paraméterei, a lokális változók, és a függvények visszatérési címei kerülnek be, ez a terület számunkra most nem érdekes.

A közönséges adatpointerek olyan változók, amelyek azokat a memóriacímeket képesek kezelni és tárolni, amelyek az adatterületre mutatnak, tehát valamely változónkban tárolt adatra. Valójában minden változó egyfajta pointer, ami az adatterületen mutatja a tárolt adat memóriacímét. Tipikus eset, hogy a pointert ráállítjuk egy tömb első elemére, majd a pointer értékét (a memóriacímet) növelve a tömb további elemeire is rá tudunk mutatni, ki tudjuk olvasni az ottani értéket, meg tudjuk azt változtatni. Egyszerű és hatékony módja annak, hogy egy ciklus belsejében, a tömb minden elemét megvizsgáljuk.

A pointerek egy teljesen más típusa, amikor a memóriacím nem egy adatra mutat, hanem a kódterület egy pontjára, egy függvény belépési pontjára. Ezeket a pointereket függvénypointernek nevezzük. Egy ilyen pointeren keresztül az általa mutatott függvény elindítható.

A C nyelvben minden függvénynév valójában változónak fogható fel, amely az általa képviselt függvény memóriacímét tartalmazza. Ezen a nyelven a függvény indító operátor hiányában, csak a függvény nevét leírva annak memóriacímét kapjuk meg (a változó értékét). Ha a függvény neve mögött szerepeltetjük a gömbölyű zárójeleket, akkor azzal jelezzük, hogy a függvényt el szeretnénk indítani.

A C# nyelvben is megoldható olyan változók deklarálása és használata, amelyek értéke egy C# függvény (metódus) memóriacíme. A legnehezebb rész ebben a változó típusának létrehozása.

---

A nehézséget a biztonság okozza. Egy ilyen változóban elvileg, technikailag bármilyen függvény memóriacíme eltárolható, mivel a memóriacímek egyszerűen 4 byte-os<sup>42</sup> számértékeknek foghatóak fel, tehát akár egy int típusú változó is képes lenne rá. Gyakorlatilag azonban a fordítónak tisztában kell lenni nemcsak azzal, hogy a változóban egy függvény memóriacíme került tárolásra, de azzal is, hogy milyen paraméterezésű és visszatérési értékkel rendelkező függvényről van szó.

Ezt a típust, a szóban forgó változó típusát a **delegate** kulcsszó segítségével lehet elkészíteni. Tételezzük fel, hogy van egy függvényünk, amely két int paramétert vár, és egy int-tel tér vissza:

```
static int osszead(int a, int b)
{
    return a + b;
}
```

Ha egy ilyen függvény memóriacímét szeretnék eltárolni egy változóban, akkor a típust az alábbiak szerint írjuk le:

```
delegate int fv_osszead(int x, int y);
```

A típusunk neve 'fv\_osszead' lesz. A további információk amik még szerepelnek ebben a sorban, azok a hívandó függvényünk paraméterezését és visszatérési értékének típusát írják le. Tehát a típusnév (fv\_osszead) előtt álló 'int' a függvény típusa, a mögötte álló rész pedig a függvény paraméterezése. Valamiért meg kell adni a függvény paraméterezésekor a paraméterek nevét is, de a névválasztás itt valójában teljesen lényegtelen, mivel a függvény törzsét már értelemszerűen nem kell megadni, a sort (utasítást) a pontosvessző zárja le.

Nagyon fontos megértenünk, hogy ezzel a sorral még csak egy típus nevét hoztuk létre. A 'fv\_osszead' olyan típust ír le, amely a két int paraméterű, int-tel visszatérő függvényekre vonatkozik. Az ilyen típusú változóba csakis ilyen jellemzőjű függvények memóriacíme tárolható el!

Hogy ezt megtehesük, szükségünk van egy olyan típusú változóra:

```
fv_osszead fv1 = null;
```

Az fv1 változó tehát helyigénye 4 byte, értéke kezdetben (a kód szerint) null, vagyis még nincs benne memóriacímm. Tekintve, hogy van ilyen jellemzőjű függvényünk (két int a paramétere, visszatérési típusa is int), a változónkba tudunk helyezni tényleges értéket is:

```
fv1 = osszead;
```

---

<sup>42</sup> 32 bites processzor architektúra esetén.

Vagy rövidebben írva:

```
fv_osszead fv1 = osszead;
```

Vegyük észre, hogy a szóban forgó függvény nevét ('osszead') most az értékadás jobb oldalán úgy használtuk, hogy nem tettük ki a függvény hívó operátort (két gömbölyű zárójel). Ha így írtuk volna, az több szempontból is hibás lenne:

```
fv_osszead fv1 = osszead(); // HIBÁS
```

Először is a függvényünk vár két paramétert, két int-et. Nem adtuk át. Pótoljuk a hiányt:

```
fv_osszead fv1 = osszead(12,30); // HIBÁS
```

A függvényünk hivatkozása sokat javult, de ekkor azt írtuk le, hogy „indítsd el a függvény ezekkel a paraméterekkel, és a visszatérési értékét helyezd el a fv1 változóba”. Azonban a függvény visszatérési értéke int típusú, azt nem lehet elhelyezni egy 'fv\_osszead' típusú változóba! Ezért nem érdemes így használni. A függvényünk memóriacímének lekérdezéséhez használjuk fel a függvényünk nevét, (csak a nevét), és ne indítsuk el a függvényt az operátorral!

A kérdés: mire jó ez? Ha a változónkba bekerül a függvény memóriacíme, akkor a változón keresztül el tudjuk indítani a szóban forgó függvényt, teljesen hasonló szintaktikával, mint hagyományosan. Az alábbi két sor egyenértékű:

```
int c = osszead(12, 30);
int d = fv1(12, 30);
```

A két sor hasonlósága onnan ered, hogy az 'osszead' is valójában változónak tekinthető, melyben eredendően benne van a függvény memóriacíme, akárcsak az 'fv1' változó, aki-ben szintén benne van ugyanezen függvény memóriacíme. Eddig még nem derült ki, mire jó, hogy kétféleképpen is meg tudjuk hívni a függvényt, úgy érezhetjük: erre a feladatra elég lenne egy módszer is! Ráadásul az első sor esetén pontosan tudjuk melyik függvény fog elindulni, míg a második esetén pusztán az adott sort látva nem derül ki, visszafele kell megkeresni a kódban melyik függvény memóriacímét tettük az 'fv1' változóba bele.

A „mire jó ez” kérdésre a válasz: pl. alkalmazhatjuk hatékonyság növelésére, de a kód elegánsabbá tételére is, valamint a kódunk működésében elképesztő rugalmasságot is elérhetünk ha kellően ügyesek és kreatívak vagyunk. Valamint eldobhatjuk az öröklődés elvét is, mivel azt tudjuk ezzel a módszerrel pótolni<sup>43</sup>!

<sup>43</sup> Elég sok esetben, példák később lesznek.

## 28.1. Alkalmazáslogika és felhasználói felület

Vizsgáljuk meg a háromrétegű alkalmazásfejlesztés javaslatait. Az elv szerint az application logic rétegbe eső függvények nem kommunikálnak a felhasználóval, azt a user interface rétegbeli függvények teszik. Fejlesszünk olyan programot, amely bekér egy pozitív egész számot (pl. 100), majd generálja és kiírja a prímszámokat eddig a számig. Ha ezt az elvet teljesen komolyan vesszük, akkor az alábbi megoldás lehet csak helyes: a prímszámokat generáló függvény nem írhatja ki a megtalált prímszámokat (mert ő az application logic), a kiíró függvény meg nem generálhat prímszámokat (mert ő a user interface), tehát a generált prímszámokat helyezzük el egy listában, adjuk vissza, majd adjuk át a kiíró függvénynek. Így nem sérül az elv, egyik függvény sem esik egyidőben két rétegbe! A user interface függvényei az alábbiak:

```
static int bekeres()
{
    while (true)
    {
        Console.WriteLine("meddig generáljam a prímszámokat?");
        int n = int.Parse(Console.ReadLine());
        if (n > 1) return n;
    }
}

static void kiir_prim(int a)
{
    Console.WriteLine("ujabb primszamra bukkantam: {0}", a);
}

static void kiiras(List<int> L)
{
    foreach (int x in L)
        kiir_prim(x);
}
```

A prímszámok generálását az alábbi függvények végzik:

```
static bool prime(int a)
{
    if (a%2==0) return false;
    int h = (int)Math.Sqrt(a);
    for (int i = 3; i < h; i = i + 2)
    {
        if (a % i == 0) return false;
    }
    return true;
}

static List<int> primszamok(int n)
{
    List<int> ret = new List<int>();
    for (int i = 2; i <= n; i++)
    {
        if (prime(i)) ret.Add(i);
    }
    return ret;
}
```



A működést vezérlő főprogram pedig:

```
static void Main(string[] args)
{
    int n = bekeres();
    List<int> primek = primeszamok(n);
    kiiras(primek);
    Console.ReadLine();
}
```

Mint láttuk a megoldás teljes, elegáns, a háromrétegű alkalmazásfejlesztés előírásait maradéktalanul betartja! Egy probléma van csak: a hatékonyság! A megoldásunk első menetében a megtalált prímszámokat listába gyűjtjük, a listaműveletek növelik a futási időt, és a lista tárolása terheli a memóriát. A prímszámok kiírását külön ciklus végzi, az összesített futási idő elég gyenge, a memóriaigény pedig nagy. A háromrétegű alkalmazásfejlesztés elvének betartása nélkül a prímszámkereső függvény egyszerűsíthető lenne:

```
static void primeszamok(int n)
{
    for (int i = 2; i <= n; i++)
    {
        if (prime(i))
            Console.WriteLine("ujabb primeszamra bukkantam: {0}", i);
    }
}
```

Mint a megoldásban is látjuk, ha integráljuk a két működést egyetlen függvényben, akkor bár a függvény egyidőben két réteg feladatait is ellátja, két rétegbe is beleesik, de a hatékonysága megnő. Hogyan tudnánk megtartani az előző elegáns megoldást, és a hatékonyságot is?

A válasz a függvény pointerekben rejlik<sup>44</sup>. Első lépésként elkészítjük a prímszám kiíró függvényhez tartozó típust. A függvény egy int-et kap paraméterként (a kiírandó prímszám értékét), és void-ot ad vissza. Tehát:

```
delegate void fv_primkiiras(int x);
```

Ha tudunk ilyen típusú változót deklarálni, akkor tudunk ilyen típusú függvény paramétert is készíteni. Adjunk egy paramétert a prískeresőhöz:

```
static void primeszamok(int n, fv_primkiiras kiiras)
{
    for (int i = 2; i <= n; i++)
    {
        if (prime(i)) ... ?
    }
}
```

<sup>44</sup> Megoldható lenne OOP stílusban is, interface és egy objektum segítségével, de el kell ismerni ez a megoldás egyszerűbb és áttekinthetőbb.

Ezen függvény meghívásakor tehát át kell adnunk egy határszámot ( $n$ ), ameddig a príme-  
ket keressük, és egy olyan függvény memóriacímét, ami egy int paramétert vár, és  
void visszatérésű. Van ilyen függvényünk; a 'kiir\_prim'. Töröltük a hatékonyságot rontó  
listakezelést, hogy a sebesség és memóriakihasználás szempontjából megfelelő legyen a  
megoldás. A főprogramból az alábbi módon kell meghívni a függvényt:

```
int n = bekeres();
primszamok(n, kiir_prim);
```

A második paraméterként a függvény memóriacímét kell megadni, vagyis megint nem az  
a megoldás, hogy el is indítjuk a függvényt, nem tesszük ki a függvény indító operátort,  
hanem csak a függvény nevét adjuk meg, amivel valójában a memóriacímét adjuk át.

Visszatérve a 'primszamok' függvényre, amikor találunk egy prímszámot, meghívjuk a  
'kiiras' függvényt, csináljon az bármit is az általunk talált prímszámmal. Mivel addigra  
már van benne memóriacím, rajta keresztül el tudjuk indítani a szóban forgó függvényt:

```
static void primszamok(int n, fv_primkiiras kiiras)
{
    for (int i = 2; i <= n; i++)
    {
        if (prime(i)) kiiras( i );
    }
}
```

A függvénypointerek segítségével tehát kiemeltük egy, az application layer-be eső függ-  
vényből ezt a funkcionalitást, miközben a hatékonyság sem romlott!

## 28.2. Dönts egyszer – használd sokszor

Másik példa, melyen keresztül bemutatatható a függvénypointerek rugalmassága legyen a  
következő: a programunk futásával kapcsolatos eseményekről naplóbejegyzéseket készít.  
A naplóbejegyzéseket tárolhatjuk sql adatbázisban, xml fájlban, egyszerű text fájlban.  
Mindhárom változatot implementáltuk, mindhárom függvény szerepel az alkalmazá-  
sunkban. A felhasználó a programunk telepítésekor választja ki, melyik változatot sze-  
retné használni (esetleg azt is választhatja, hogy egyáltalán ne készítsünk naplófájlt).  
Ezt a telepítéskor egy konfigurációs fájlba menti el a program, melyet induláskor kiol-  
vas. A választást egy int típusú változóba menti el, a 0 jelenti, hogy nem kér naplózást, 1  
jelenti az sql, 2 az xml, 3 a text fájl választást. Az egyes függvények nevei naploIrSql,  
naploIrXml, naploIrTxt, mindegyik paraméterezése egyetlen string, a kiírandó naplóbe-  
jegyzés.

A programunk működése közben több ponton kíván a naplóba írni. Nyilván nem megol-  
dás, hogy minden egyes ilyen tevékenység esetén az alábbi kódrészt kell bemásolni:

```
string uzenet = "kalkulacio indul";
if (naploValasztas == 1) naploIrSql(uzenet);
else if (naploValasztas == 2) naploIrXml(uzenet);
else if (naploValasztas == 3) naploIrTxt(uzenet);
```

Többek között azért sem, mert ha kidolgozunk egy negyedik naplóbaíró módszert is, akkor a program több pontján kell beilleszteni azt az elágazási ágot is (és ez akkor is igaz, ha if-ek helyett switch-eket írunk). Célszerű helyette egy központi üzenetíró függvényt definiálni, amely kiválasztja a megfelelő kiíró függvényt. Az alkalmazás többi pontjairól ezt a központi függvényt kell megfelelő paraméterezéssel meghívni:

```
static void naploIr(string uzenet)
{
    if (naploValasztas == 1) naploIrSql(uzenet);
    else if (naploValasztas == 2) naploIrXml(uzenet);
    else if (naploValasztas == 3) naploIrTxt(uzenet);
}
```

Ez már megfelelőbb megoldás, eléggé központosított elosztási rendszer, újabb elágazások rendszerbe illesztése, a karbantarás már egyszerű. Hatékonyságával vannak gondok, mivel minden egyes naplóbaírás során végigmegyünk az if-ek vizsgálatain újra és újra. Csináljuk ezt ügyesebben függvénypontterek segítségével! Első lépésként definiáljunk egy, a naplóbaíró konkrét függvényváltozatokra illeszthető típust:

```
delegate void fv_naplobaIras(string m);
```

Továbbá definiáljunk egy (akár) statikus változót, amelyben majd a kiválasztott függvény memóriacímét fogjuk tárolni:

```
static fv_naplobaIras naploIr = null;
```

Miután beolvastuk a konfigurációs fájlt, mielőtt a program érdemi része elindulna, válasszuk ki (egyszer), melyik függvényt kell majd használni a program futása során:

```
if (naploValasztas == 1) naploIr = naploIrSql;
else if (naploValasztas == 2) naploIr = naploIrXml;
else if (naploValasztas == 3) naploIr = naploIrTxt;
```

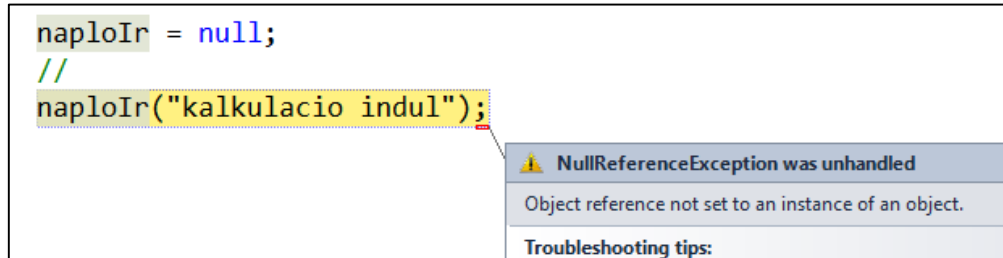
Ezzel lényegében készen is vagyunk. A program minden további pontján, ha naplóbeli üzenetet kívánunk rögzíteni, akkor a függvényponttert kell használnunk:

```
naploIr("kalkulacio indul");
```

Vegyük észre, hogy a megoldás továbbra is jól karbantartható, ha újabb naplóbaíró lehetőséget fejlesztünk ki, az új függvény memóriacímét is betehetjük a változónkba. A hatékonyság azonban jelentősen javult, mivel a naplóba írás során nem kell minden egyes híváskor az if-eken átjutni, valójában közvetlenül a megfelelő függvényt hívjuk meg további késlekedések nélkül.

### 28.3. Nem kitöltött függvénypointerek

Gondolnunk kell arra az eshetőségre is, amikor a függvénypointerbe még nem helyeztünk el egyetlen létező függvényünk memóriacímét sem. Ekkor a változónk értéke a szokásos üres memóriacím, a null konstans. Egy ilyen null érték esetén a függvény meghívása értelemszerűen hibát okoz, kivételt dob fel:



A probléma sajnos elegánsan nem kezelhető. Megoldható hogy biztosak lehessünk abban, hogy ilyen nem fordul elő (pl. létrehozhatunk egy alapértelmezett, akár üres törzsű naplóíró függvényt, és egyéb választás hiányában ebbe a függvénybe irányítjuk a naplóíró tevékenységet).

```

static void naploIrSemmi(string msg)
{
}

```

Majd:

```

if (naploValasztas == 1) naploIr = naploIrSql;
else if (naploValasztas == 2) naploIr = naploIrXml;
else if (naploValasztas == 3) naploIr = naploIrTxt;
else naploIr = naploIrSemmi; // különben a semmit sem tevőbe !

```

Ezek után nem kell foglalkoznunk valójában a null érték problémával. Ha valamiért nem kivitelezhető, akkor a hívás előtt érdemes ellenőrizni a hívás várható sikerességét:

```

if (naploIr!=null) naploIr("kalkulacio indul");

```

Természetesen nem elegáns, nem is hatékony, de egyelőre nincs más lehetőségünk.

### 28.4. Példányszintű függvények

Az előző példákban osztályszintű függvényekre alkalmaztuk a függvénypointereket. Ez egyszerűbb és könnyebben megérthető művelet, de valójában nincs annak semmi akadálya, hogy példányszintű függvény memóriacímét tároljuk el. Az egyetlen ami itt érdekes lehet: hogyan hivatkozzunk a példányszintű függvényre, hogyan azonosítsuk őt?

Készítsünk el egy naplóbaíró függvényt példányszinten:

```
class Email
{
    public string email_cim;
    public void elkuld(string msg)
    {
        // TODO:
        // az uzenetet email formajaban
        // küldjük el a fejlesztonek
    }
}
```

A változónk (naploIr) eleve képes tárolni ilyen címet, így őt nem kell módosítani. Azonban hogy példányszintű metódus címét tárolhassuk egy függvénypointer változóban, először is szükségünk van egy példányra!

```
Email p = new Email("bugreport@sw-house.company.hu");
naploIr = p.elkuld;
```

Mint látjuk, a példányszintű metódus címének tárolása lényegében ugyanaz a lépés, csak itt a függvény neve előtt szerepeltetni kell a példány nevét is! Ennek oka (mint korábban leírtuk), hogy a példányszintű metódusoknak a példány is a paraméterük (this néven fogadja a függvény), így a rendszer valójában nemcsak a függvény címét tárolja el, hanem a példány elérhetőségét (referenciáját) is. A függvénypointeren keresztül meghíváskor pedig a szokásos paramétereken túl a példányt is átadja, különben hibát kapnánk.

*Megjegyzés:* a GC ekkor a példányt nem szüntetheti meg, nem törölheti ki a memóriából, mivel a függvénypointer indirekt módon hivatkozik a példányra, tárolja annak címét.

## 28.5. Függvénylista kezelése

Ahol egyetlen függvény memóriacímét tárolhatjuk, ott tárolhatunk többet is – listába szervezve. Tegyük fel, azt szeretnénk elérni, hogy a program üzenetek tárolásra kerüljenek SQL adatbázisba, valamint a fejlesztő SMS-ben kapjon értesítést, ha valamely (esetleg hibás) programtevékenység történt:

```
static List<fv_naplobaIras> naploIr = new List<fv_naplobaIras>();
```

Ezért egy listába felvesszük mindkét függvényünk memóracímét:

```
naploIr.Add( naploIrSql );
naploIr.Add( naploIrSms );
```

A programunk adott pontján a listában tárolt (tetszőleges számú) függvény meghívható sorban egymás után egy egyszerű ciklus segítségével:

```
foreach( fv_naploIras f in naploIr )  
    f("hiba: varatlan programleallas, lemez betelt");
```

Itt előnyt kövcsolhatunk abból, hogy a lista minden esetben létezik, akkor is ha nulla elemű. Tehát a null értékkel nem kell foglalkoznunk külön. Ha a lista üres (nulla elemű), a foreach ciklus ezt az esetet is kezeli, tehát nem kapunk sem hibát, sem kivételt.

## 28.6. Eseménylista

Olyan gyakori az a feladat, amikor nem egy, hanem több függvény meghívása szükséges adott esemény bekövetkeztekor, hogy a C# nyelv saját konstrukciót tartalmaz a megoldására, amit **eseménykezelőnek** (*event*) nevezzük.

Hagyományos esetben az alábbi módon deklaráltuk a függvénypointer változónkat:

```
static fv_naploIras naploIr;
```

Egy ilyen változó esetén egy időben csak egyetlen függvény memóriacímét tudtuk tárolni és meghívni:

```
naploIr = naploIrSql;  
naploIr("hiba: varatlan programleallas, lemez betelt");
```

Ha ismerjük és használjuk az 'event' kulcsszót, akkor a változó deklaráció még nem bonyolódik nagyon:

```
static event fv_naploIras naploIr;
```

Egy ilyen 'event' módosítóval ellátott változó egy időben nem egy, hanem tetszőlegesen sok függvény memóriacímének tárolására képes – tehát a háttérben ő valójában egy lista. A függvények memóriacímét nem egyszerű értékadással kell elhelyezni ebben a változóban, hiszen nemcsak egy címet képes tárolni. A függvény címének „hozzáadását” úgy nevezzük: feliratkozás az eseményre. Tehát a lista nem túl elegáns Add() műveletének meghívása helyett a += operátort kell használnunk:

```
naploIr += naploIrSql;  
naploIr += naploIrSms;
```

Az eseményre (event) bármennyi függvény feliratkozhat. A függvények meghívása azonban sokkal egyszerűbb szintaktikával történik, nem kell ciklust alkalmaznunk:

```
naploIr("hiba: varatlan programleallas, lemez betelt");
```

A fenti szintaktika kicsit zavaró, mert egyetlen függvényhívásnak tűnik. De mivel a 'naploIr' változó nem egyszerű változó, hanem 'event' módosítójú, ez az egyetlen függvényhívás valójában minden feliratkozott függvény meghívását kiváltja, a feliratkozás sorrendjében.

## 28.7. Származtatás másként

A függvénypointereknek további kreatív alkalmazása révén metódusok felüldefiniálását tudjuk helyettesíteni. Vegyük példaként egy szimulációs problémát: állataink vannak, akik tudnak futni, repülni, úszni:

```
abstract class Allat
{
    abstract public void fut();
    abstract public void uszik();
    abstract public void repul();
}
```

Konkrét állatok fejlesztése esetén a konkrét függvények elkészülnek. Egy házikacsa pl. jól úszik, nagyon lassan fut, azonban nem tud repülni. A gepárd kiválóan fut, elfogadhatóan úszik, de szintén nem tud repülni. A sas nem tud úszni, nem fut jól, de annál jobban repül. További állatok esetén további kombinációk képzelhetők el. Hogyan építsük fel a fejlesztési fa szerkezetet, hogy minél kevesebb kódot kelljen megírni!? Ha a sas az őse a kacsának, akkor a kacska is öröklí a jó repülő képességet. Fordítva a sas örökölné az úszást. Nyilván ne válasszuk a gepárdot a kacska őseinek (se fordítva). Ha a három osztályt egymástól teljesen függetlenül fejlesztjük ki, akkor viszont senki sem örököl semmit a másiktól, minden metódust külön meg kell írunk.

Ahelyett, hogy tovább törnénk a fejünket, fejlesszük ki a szükséges metódusokat (ebben a példában osztályszintű függvények segítségével):

```
class Allat
{
    public static void fut_benan() { /* ... */ }
    public static void fut_kivaloan() { /* ... */ }
    public static void uszik_sehogy() { /* ... */ }
    public static void uszik_jol() { /* ... */ }
    public static void uszik_kivaloan() { /* ... */ }
    public static void repul_sehogy() { /* ... */ }
    public static void repul_jol() { /* ... */ }
    public static void repul_kivaloan() { /* ... */ }
}
```

Készítsük el a fenti függvények (egységes) delegate típusát. A fenti függvények mindegyike void visszatérésű, paraméter nélküli:

```
delegate void fv_mozogj();
```

Egészítsük ki az 'Allat' osztályt három mezővel, melyek mindegyike függvénypointer:

```
class Allat
{
    // ... függvények ...
    public fv_mozogj fut = null;
    public fv_mozogj uszik = null;
    public fv_mozogj repul = null;
}
```

A konkrét állatok létrehozásakor válogassuk össze a megfelelő metódusokat:

```
Allat kacska = new Allat();
kacska.fut = Allat.fut_benan;
kacska.uszik = Allat.uszik_kivaloan;
kacska.repul = Allat.repul_sehogy;
```

Másik állat esetén más összeállítást készítünk:

```
Allat cica = new Allat();
cica.fut = Allat.fut_kivaloan;
cica.uszik = Allat.uszik_sehogy;
cica.repul = Allat.repul_sehogy;
```

Egy ügyes konstruktor sokat segíthet ebben a feladatban

```
class Allat
{
    // .. constructor ...
    public Allat(fv_mozogj fut, fv_mozogj usz, fv_mozogj rep)
    {
        this.fut = fut;
        this.uszik = usz;
        this.repul = rep;
    }
}
```

Így már sokkal egyszerűbb az állatok készítése:

```
Allat krokodil = new Allat(
    Allat.fut_benan,
    Allat.uszik_kivaloan,
    Allat.repul_sehogy );
```



Összegzésként elmondhatjuk, hogy megúsztuk a származtatást, gyerekosztályok készítését, a késői kötést (vele a VMT tábla extra memóriaigényét), miközben a kapott példányok használata továbbra is elegáns és hatékony:

```
cica.fut();
krokodil.uszik();
sas.repul();
```

Ha mégis ragaszkodnánk a származtatáshoz, a különböző állat típusokhoz, a „látszat kedvéért” készíthetünk gyerekosztályokat is:

```
class Macska : Allat
{
    public Macska()
    {
        this.fut = Allat.fut_kivaloan;
        this.uszik = Allat.uszik_sehogyz;
        this.repul = Allat.uszik_sehogyz;
    }
}
```

Ez esetben a macska példányok készítése annyit jelent, hogy a mezőket a macskákra jellemző függvények értékeivel töltjük fel, ami annyit jelent, hogy minden macska példány egyformán jól fut, egyformán nem tud úszni, sem repülni. Ugyanakkor ha lesz egy különleges macskánk, aki valamiért megtanul úszni, akkor csak neki át tudjuk állítani az 'uszik' mezőjét más úszási függvényre. A szokásos származtatási és gyerekosztály módszerekkel ehhez kellene a macska osztály speciális 'uszniTudoMacska' gyerekosztályát is elkészíteni, ahol override-oljuk a macska örökölt „nem tudok úszni” metódusát.

*Megjegyzés:* vegyük észre, hogy ez az átállítás akár futás közben is megtörténhet. Tehát míg a cirmos cicánk alapvetően egy hagyományos macska példány, aki kezdetben nem tudott úszni, de később megtanult, akkor a példány eldobása és újra elkészítése helyett egyszerűen átállítjuk ezen mező értékét:

```
Macska cirmos = new Macska();
// cirmos ekkor még nem tud úszni
// ...
// de aztán megtanul
cirmos.uszik = Allat.uszik_jol;
```

## 28.8. TIE osztályok

Az előző részben leírt 'Allat' és 'Macska' osztályok közös jellemzője, hogy tartalmazznak függvénypointer típusú mezőket, melyek konkrét függvényekre kerülnek beállításra. Ráadásul ezekben az osztályokban szerepelhetnének még a szokásos OOP elemek is, mezők, hagyományos metódusok, virtuális és korai kötésűek, property-k stb.

A TIE osztályok olyan osztályok, amelyek nem tartalmaznak kidolgozott metódusokat, minden metódusa valójában függvénytípusú. Ilyen értelemben a 'Macska' osztály is egy ilyen TIE osztály.

A TIE osztályok példányai készítésükkor még nem működőképesek, hiszen a mezőiket fel kell tölteni konkrét, létező függvények memóriacímeivel. Ezen függvények mindegyike szerepelhet más-más osztályokban, vegyesen osztályszintű és példányszintű függvények is. A TIE példány célja mindössze az, hogy a megfelelő függvények összeválogatásának egy gyűjteménye lehessen. A TIE példányon keresztül ezeket a függvényeket meg is tudjuk hívni, egyszerű szintaktikával, mintha azok valójában a példány metódusai lennének. Ennek az az értelme, hogy a példányon keresztül egy ilyen központosított módon tároljuk és használjuk a függvényeket. A függvények összeválogatását vezérelheti konfigurációs fájl is.

A TIE példánybeli függvények memóriacímét ugyanakkor futás közben akár ki is cserélhetjük valami más függvény címére, miközben a program kódjai ebből mit sem vesznek észre, továbbra is ugyanúgy használják a példány metódusait.

Tegyük fel, hogy a programunk ismerősök elérhetőségeit (contact list) kezeli. Az ismerősök adatait rögzítés után azonnal menti egy SQL adatbázisba.

```
class contact
{
    /* ... mezők és metódusok ... */
}
delegate void fv_mentes(contact c);
```

A TIE osztályunk egyetlen függvénytípusú tárol:

```
class tarolas
{
    public static fv_mentes mentes;
}
```

A kódunk induláskor beállítja a mentést az adatbázisba mentést végző függvényre.

```
tarolas.mentes = ContactToSql.tarolas; // beállít egy függvényre
```

A program felhasználói felületén beírhatjuk az ismerősünk adatait, majd kattinthatunk a tárolás gombra. Ekkor meghívjuk a beállított SQL adatbázisba mentő függvényt a függvénytípusú ponteron keresztül:

```
public void btnTarolasClick(Object sender)
{
    contact c = new contact();
    // a 'c' mezőinek feltöltése
    // ..
    // tárolás kérése
    tarolas.mentes(c);
}
```

Az SQL-be mentő függvény hiba esetén átállíthatja a pointer értékét egy tartalék mentő mechanizmusra:

```
class ContactToSql
{
    public static void tarolas(contact c)
    {
        try
        {
            // sql szerverre csatlakozás
            // insert rekord
        }
        catch
        {
            // tartalék mentési mechanizmus
            tarolas.mentes = ContactToXml.tarolas;
            // és meghívása, hogy ez a contact is mentődjön
            tarolas.mentes(c) ;
        }
    }
}
```

A tárolás gombra kattintás kódja változatlan marad. Észre sem veszi, hogy átállították a mentést adatbázisról XML fájlba mentésre. Neki mindegy, mentés van, mentődjön valahova, nem az ő dolga hova. Mint ahogy az sem, hogy ha később megjavul az SQL szerver (pl. újraindítás alatt volt, pár perc múlva újra működőképes lesz), akkor a program ezt észlelve a mentést visszaállítja újra az sql szerverre mentő függvényre.

## 29. Reflection

---

Ahogy korábban ismertettük, a.NET alatt íródott DLL-eket „assembly”-knek, szerelvényeknek nevezzük. Ilyen szerelvényeket a program fejlesztése közben csatolhatunk a projectünkhöz az 'Add Reference' menüpont segítségével, és a fejlesztés közben az ezekben fellelhető publikus osztályokat felhasználhatjuk, azokból példányosíthatunk, azokat továbbfejleszthetjük stb. Amikor a programunk elindul, az operációs rendszer ezen szerelvényeket megkeresi és betölti a memóriába. Bármelyik hiánya esetén a programunk futását még annak tényleges elindulása előtt leállítja, ellehetetleníti.

Sokkal problémásabb a helyzet akkor, ha olyan assembly-t kívánunk a programban használni, amelyet a fejlesztés során nem csatoltunk be. Először is értelemszerűen az abban lévő osztályokra a fejlesztés során nem hivatkozhatunk semmilyen módon, hiszen a fordító nem tudná őket azonosítóit felismerni. Másrészt ezen assembly-k hiányában a programunk elindul, hiszen nem szükséges feltétel a futáshoz az assembly-k jelenléte, hiszen nem regisztrált részei a programunknak.

Miért tennénk ilyet, hogy olyan assembly-kre hivatkozunk, melyeket nem csatoltunk hozzá a programhoz? Többek között azért, mert a fejlesztéskor ezek még nem voltak készen, vagy nem álltak rendelkezésünkre, illetve nem akarjuk ilyen erős kötéssé alakítani. Erre legjobb példa lehet a közismert WinAMP program, amelyhez számtalan kiegészítés (plugin) készül független fejlesztőktől, és amelyek különféle szolgáltatásokat adnak hozzá a lejátszóhoz. Maga az mp3 file lejátszása is egy dinamikusan cserélhető plugin segítségével történik, nem is szólva a vizuális effektekről.

Nyilván a plugin fejlesztése során bizonyos szabályokat be kell tartani, mivel ennek hiányában a program vázát alkotó alkalmazás (.exe) nem képes felismerni a plugin-t rejtő dll tartalmát. Megállapodás lehet az, hogy a plugin-nek tartalmaznia kell egy bizonyos nevű osztályt, abban adott nevű és paraméterezésű metódust. Amennyiben megvan, úgy a programváz futás közben ezen assembly-t képes betölteni a memóriába, megkeresni benne a megállapodás szerint kialakított függvényt, és meghívni azt amikor annak szolgáltatására szükség van. Sajnos a programváz sem állapodhat meg, hogy az adott nevű függvény valóban azt a szolgáltatást fogja-e nyújtani, amit elvárnak tőle; ezért ez bizalmi kérdés a két kód között.

## 29.1. Assembly betöltése dinamikusan

Az assembly-k dinamikus kezelését a System.Reflection névtérben lévő osztályok, és azok metódusai végzik. Az alábbiakban megadunk egy olyan függvényt, amely egy assembly-t tölt be a memóriába annak fájlneve ismeretében:

```
using System.Reflection;
using System.IO;

class PluginManager
{
    public static string pluginRootDir = @"C:\myProgram\myPlugins";

    public static Assembly LoadAssembly(string dllName)
    {
        // ha a dllName név vége nem .dll, akkor kiegészítjük
        if (Path.GetExtension(dllName).ToUpper() != "DLL") dllName += ".dll";

        // elébe írjuk a plugin-eket tartalmazó alkönyvtár nevét
        // (abszolút file név készítése)
        string dllFileName = Path.Combine(pluginRootDir, dllName);

        // a dll file betöltése, referencia elkészítése
        // ha a dll file betöltése nem sikerül, az 'a' értéke marad 'null'
        Assembly a = null;
        try { a = Assembly.LoadFile(dllFileName); }
        catch {}

        // visszatérési érték
        return a;
    }
}
```

Amennyiben nem határozzuk meg a Assembly.LoadFile metódusnak a dll fájlunk teljes elérési útvonalas nevét, úgy ő a megadott nevű dll fájlát valamennyi heuristicus szerint próbálja megkeresni. Keresi a futó program aktuális alkönyvtárában (.exe), valamint a rendszer GAC-ban (Global Assembly Cache). Ezt elkerülendő, a fenti esetben csak az adott alkönyvtárban (pluginRootDir) lévő fájlok közül próbál betölteni, melynek értékét konfigurációs fájlból is beolvashatjuk.

## 29.2. Saját assemblyre hivatkozás

A futó .exe kiterjesztésű fájl is szerelvénynek minősül a .NET felfogás szerint. Egy olyan szerelvénynek, melynek kiterjesztése nem .dll hanem .exe, és van benne Main() függvény is. Mindez nem mond ellent az assembly lényegének. Nem szükséges az assembly-ket feltétlenül .dll kiterjesztéssel ellátni, és maga a fájl betöltése a memóriában nem jelenti annak elindítását sem, így lényegtelen szempont, hogy van-e benne Main() függvény, vagy sem. Ennek megfelelően tehát az előző fejezetben említett eljárással nemcsak tényleges .dll-ek, hanem akár .exe-k is betölthetők. Természetesen itt csak .NET kompatibilis dll-ekről, exe-kről van szó. Ezen a módszerrel nem lehetséges natív (Win32)-es dll-ek, sem exe-k betöltése.

Tehát maga a futó program is szerelvénynek minősül, csak őt már nem kell betölteni még egyszer a memóriába, mivel ő már betöltődött (fut). A saját assembly-re az alábbi módon kereshetünk referenciát (GetExecutingAssembly függvény):

```
using System.Reflection;

class PluginManager
{
    public static Assembly myExe()
    {
        return Assembly.GetExecutingAssembly();
    }
}
```

Mint látjuk, mindkét módszerrel, egységesen egy Assembly típusú objektum-példányhoz juthatunk, mellyel a továbbiakban már egységesen dolgozhatunk.

### 29.3. Egy osztály megkeresése egy szerelvény belsejében

Amennyiben van egy Assembly osztálybeli példányunk, úgy lehetőség van annak belsejében egy tetszőleges osztály megkeresésére. Ismernünk kell, hogy az adott osztály milyen nevű névtérben helyezkedik el, és ismernünk kell az osztály nevét. A DLL írásakor ismertetett módon ezen osztálynak public láthatóságúnak kell lennie (public class), hiszen csak ezek érhetőek el a DLL területén kívül.

A keresett osztályra a referenciát egy System.Type objektumosztálybeli példány fogja hordozni. A Type példányon keresztül juthatunk hozzá további információkhoz az adott, általa képviselt osztályról, például listaszerűen lekérdezhetjük annak milyen metódusai vannak stb.

```
Assembly dll = PluginManager.LoadAssembly("akarmilyenPlugin.dll");
Type t = dll.GetType("enNevterem.enOsztalyom", false, true);
```

A dll által képviselt assembly-ben a GetType függvénnyel lehet keresni. Meg kell adni, milyen névtérbeli milyen osztályt keresünk. A névtér és az osztály neve itt bele van égetve a kódba, de fejlettebb esetekben azt valamilyen konfigurációs állomány is tartalmazhatja (pl. xml vagy ini).

A GetType második paramétere, a false, azt írja le, hogy amennyiben a fenti nevű osztály nem létezne a dll belsejében, úgy nem kérünk Exception-t, hanem helyette a GetType térjen vissza null értékkel. Ha true lenne, úgy a nem létező, nem megtalálható osztály esetén a GetType egy, az okot leíró megfelelő Exception feldobásával reagálna.

A GetType harmadik paramétere, a true, azt jelöli, hogy a keresés kisbetű-nagybetű elterésre érzéketlen módon hajtódjon végre. A C és a C# szintaktika kisbetű-nagybetű érzékeny egyéb esetekben, de a keresés során ettől eltérhetünk (mint a fenti esetben). Ez a névtér és az osztály nevére vonatkozó engedmény.

A fenti példában a 't' változóba vagy null érték; vagy egy, a keresett, megadott nevű osztályról szóló információt leíró Type példány referenciája kerül. A keresést a dinamikusan betöltött DLL belsejében, vagy a futó exe (a programunk) belsejében deklarált osztályokra is lefuttathatjuk.

## 29.4. Egy osztály metódusának megkeresése

Amennyiben rendelkezésünkre áll egy Type példány (pl. a GetType segítségével), úgy azon keresztül az adott osztályról további információkat kérhetünk le. Amennyiben metódusokat keresünk, úgy a MethodInfo osztálybeli példányokat kapunk keresési eredményként:

```
MethodInfo creator = t.GetMethod("Create",
    BindingFlags.Public | BindingFlags.Static);
```

A fenti kód szerint a 't' típusleíró példány ('enNevterem.enOsztyalom' osztály) Create nevű metódusát keressük. Az overloading szabály alapján ebből akár több is lehet. A metódus a fenti példa szerint legyen 'public' és 'static' jelzőjű, vagyis osztályszintű. A paraméterlistája üres kell egyen (mivel arra nem írtunk elő semmit).

```
MethodInfo dosome = t.GetMethod("DoSomething",
    BindingFlags.Public | BindingFlags.Static,
    null,
    new Type[] { typeof(int), typeof(string), typeof(int) },
    null);
```

Ebben a példában már pontosítunk a keresésen: olyan 'DoSomething' nevű függvényt keresünk, amely osztályszintű, publikus, és három paraméterű, rendre int, string és int típusokkal.

Teljesen hasonlóan kereshetünk mezőket (MemberInfo példányokat kapunk a .GetMember(...) függvény segítségével), konstruktorokat (ConstructorInfo példányokat kapunk, a .GetConstructor(...) függvényt kell használnunk), de akár property-eket is lekérdezhetünk. Lekérdezhetjük, hogy az adott típus milyen interface-ekkel kompatibilis, ki az őse stb.

## 29.5. Osztálysztíű metódus meghívása I.

Amennyiben a fenti GetMethod segítségével egy metódust megtaláltunk, úgy lehetőségünk van azt meghívni:

```
// a függvény akit meghívunk az alábbi szignatúrájú:  
//     public static void Create() { ... }  
  
creator.Invoke(null, null);
```

Az Invoke segítségével hívhatunk meg egy metódust. Az első paramétere (null) jelentése szerint ez nem példányszintű metódus (nem az, hiszen osztálysztíű a Create(...)). A második paramétere szerint (szintén null) nem vár paramétert.

Az Invoke egyébként meghívja a megadott függvényt, majd annak visszatérési értékét visszaadná. Ezzel most nem foglalkozunk, hiszen a függvényünk most void visszatérési típusú.

## 29.6. Osztálysztíű metódus meghívása II.

Amennyiben a metódus paramétert vár, úgy azt természetesen át kell neki adni:

```
// a függvény akit meghívunk az alábbi szignatúrájú:  
//     public static void DoSomething(int a, string r, int b) { ... }  
  
dosome.Invoke(null, new Object[] { 12, "Hello", 20 });
```

Mint látjuk, az Invoke első paramétere maradt null, hiszen továbbra is osztálysztíű metódust hívunk. A második paraméterében azonban a kereséskor is specifikált paraméterlistának megfelelően elhelyezünk egy int-et, egy string-et, majd még egy int-et.

## 29.7. Példányszintű konstruktor megkeresése és példányosítás

Ha a betöltött assembly-ben szereplő osztályt példányosítani szeretnénk (mivel egy példányszintű metódus meghívása a feladat), akkor első lépésként a konstruktort kell felderíteni. Ehhez egy ConstructorInfo példány megszerzése az első lépés:

```
// a példában keresett konstruktor (int,string) paraméterezésű  
ConstructorInfo c = t.GetConstructor(  
    new Type[] { typeof(int), typeof(string) } );
```



Amennyiben sikeresen felderítettünk egy példányszintű konstruktort, úgy azon keresztül létrehozhatunk példányt:

```
// az (int,string) paraméterezésű konstruktor hívása
Object x = c.Invoke(new Object[] { 12, "Hello" });
```

Az Invoke() egy általános Object típusú értéket ad vissza, de a generált 'x' példány valójában egy, az adott osztályból létrehozott példány. Az 'is' operátorral erről meg is győződhetnénk, de ezt most nem használhatjuk, mivel a típusnevet nem használhatjuk fel a kódukban.

## 29.8. Példányszintű metódus megkeresése és meghívása

Az 'x' példány statikus típusa Object, ennek ellenére az 'x' példány adott metódusát típuskényszerítés nélkül meg tudjuk hívni. A GetMethod() segítségével felderítjük a szokásos módon a metódust az osztály belsejében. Ekkor természetesen nem BindingFlags.Static jellegű metódust keresünk, hanem BindingFlags.Instance jellegűt (instance = példány):

```
MethodInfo calc = t.GetMethod("calcDivide",
    BindingFlags.Public | BindingFlags.Instance,
    null,
    new Type[] { typeof(int), typeof(int) },
    null);
```

A példányszintű metódus meghívásához továbbra is az Invoke() használható, de első paramétereként ekkor az előzőekben létrehozott példányt ('x') is meg kell adni. Amennyiben a függvény visszatérési értékkel is rendelkezik, akkor az Invoke() általános Object típusú visszatérési értékét a megfelelő típusra kell kényszeríteni:

```
// a meghívandó fv 'double'-val tér vissza,
// és 'fv(int,int)' paraméterezésű
double res = (double) calc.Invoke(x, new Object[] { 12, 20 });
```

Tehát ha ismerjük egy osztály nevét (string alakban) a Reflection segítségével, akkor meghívhatjuk annak osztálysintű függvényeit, képesek vagyunk példányt készíteni a konstruktora segítségével, majd elérni a példányszintű mezőket, meghívni a példányszintű függvényeket is. Erre általában nincs szükség, hiszen a kód írása során e tevékenységeket a szokásos módon, a fordítóprogram szigorú típusellenőrzése mellett közvetlenül is megtehetjük. Azonban ha a típus fordításkor nincs jelen, akkor a forráskódba sem szerepeltethetjük a nevét. A futás közben, dinamikusan betöltött DLL-ek esetén ez az út azonban járható, és szükséges is.

Még egy példát mutatunk, ahol a reflection segíteni tud, pedig nem dinamikusan betöltött típussal dolgozunk. Egy állatkerti szimulációban legyen egy közös kiindulási 'Allat' osztályunk, melybe az állatok alapvető funkcióit (eszik, alszik, szaporodik) szeretnénk implementálni. Ebből az ősosztályból fogjuk majd elkészíteni a specifikus alosztályokat, panda, tigris, kolibri stb., az állatkertünk lakóit.

```
abstract class Allat
{
    public abstract Allat szaporodik();
}
```

A szaporodik() metódusunknak nem tervezünk paramétert, visszatérése egy 'Allat' példány lesz (valójában bármilyen, ezzel a típussal kompatibilis, vagyis gyerekosztálybeli példány is lehet a visszatérési érték). Feladata előállítani az új jövevényt, egy példányt a megfelelő objektumosztályból. Ez alatt azt értjük, hogy ha egy panda példánynak hívjuk meg a szaporodik metódusát, akkor elvárjuk tőle, hogy egy új panda példányt készítsen. Éppen ezért nem tudjuk elkészíteni az ősosztályban, kénytelenek vagyunk a konkrét kódot a gyerekosztályban megírni:

```
class Panda : Allat
{
    public override Allat szaporodik()
    {
        return new Panda();
    }
}
```

Hasonló egysoros függvényt kell készíteni minden konkrétabb osztály (tigris, pingvin stb.) esetén is. Nem nagy munka, de szeretnénk, ezt a feladatot valahogy áttolni az ősosztályhoz.

Ehhez a példányszintű metódusok belsejében felhasználható 'this'-t tudjuk segítségül hívni, amely minden esetben az aktuális példány referenciáját hordozza, és dinamikus típusa mindig egyezik a külső kódban szereplő példány típusával. A reflection segítségével lekérhetjük a this típusleíró példányát, mely panda példány esetén a panda osztály leírását, egy tigris esetén a tigris osztály leírását adja meg. Mivel ez esetben rendelkezésre áll a 'this' példány, a típusinformációt nem a korábban bemutatott osztálysintű GetType függvény, hanem a példányokhoz is tartozó (az Object őstől örökölt) példányszintű GetType-t használjuk:

```
class Allat
{
    public Allat szaporodik()
    {
        Type t = this.GetType();
        ...
    }
}
```

Miután rendelkezésre áll a Type leíró példányunk, a további lépések a korábban leírtaknak megfelelően végrehajtható. Megkeressük az adott típushoz tartozó paraméter nélkü-

li konstruktort, meghívjuk, és a kapott (panda, tigris stb.) példányt adjuk meg visszatérési értéként:

```
public Allat szaporodik()
{
    Type t = this.GetType();
    ConstructorInfo c = t.GetConstructor(new Type[] { });
    return c.Invoke(null) as Allat;
}
```

Ennek helyes működését könnyedén ellenőrizhetjük. Definiáljunk egy nagyon egyszerű gyerekosztályt:

```
class Panda : Allat
{
}
```

Készítsünk egy egyszerű Main függvényt a teszteléshez:

```
Panda p = new Panda();
Allat x = p.szaporodik();
if (x is Panda) Console.WriteLine("ez egy panda");
else Console.WriteLine("hm... mi történetett");
```

A program futásakor megjelenik a várt „ez egy panda” kiírás, így a reflection segítségével készített példány valóban teljes értékű Panda osztálybeli példány. Természetesen megoldható lett volna a példány készítése akkor is, ha a Panda osztályban nemcsak paraméter nélküli konstruktor van, hiszen paraméteres konstruktor hívására is van lehetőség.

## 30. Zárszó

---

A jegyzet célja az volt, hogy bemutassa az Objektumorientált programozás technikáit, azok alkalmazhatóságát, viszonylag egyszerű példákon keresztül. Reméljük, a technikák mindegyikére megérthető, elfogadható alkalmazási területet tudtunk mutatni, mely meggyőzte a tisztelt Olvasót arról, hogy ezen megoldások ügyes és kreatív kézben biztonságos és elegáns kód írását teszik lehetővé.

Sok mindenről nem ejtettünk szót, sok mindenről nem kellő mélységben. Elképzelhető, hogy a példák nem minden esetben érenek el tudományos magasságokat. Mivel technikákat akartunk bemutatni, nem kerestünk olyan példákat, melyek esetén az alkalmazott algoritmusok megértése elvette volna az idő nagyobb részét, elfedte volna a mondanivaló lényegét. Nem írtuk meg az eljárások törzsét, legtöbb helyen kommentek formájában jeleztük, hogy mit is kellene annak a kódnak tudni, ami oda kerülne. Elismerjük, sokszor izgalmas lett volna az eljárások törzsének forráskódja is. Reméljük, ennek ellenére az Olvasó nem csalódottan teszi le a jegyzetet, hanem olyan érzéssel, hogy nem olyan misztikus ez az objektumokkal teli, kivételeket feldobó és elkapó, felüldefiníálható operátorokkal rendelkező világ, ahol a legtöbb dolognak kétfajta típusa is van, és szabad olyan metódusokat is írni, amelyeknek nincs is törzsük. Minden kedves Olvasónak sok sikert kívánunk ezen világ feldezéséhez!

```
throw new EndOfSemesterException("Vége.");
```

---



## Irodalomjegyzék

---

- 1) Illés Zoltán: Programozás C# nyelven, JEDLIK OKTATÁSI STÚDIÓ, Budapest, 2005
- 2) Sipos Marianna: Programozás élesben, Infokit, 2004, ISBN: 963 216 652 3
- 3) 2. Jeffrey Richter: CLR via C#, Fourth Edition, Microsoft Press, A Division of Microsoft Corporation, One Microsoft Way, Redmond, Washington 98052-6399
- 4) Bertrand Meyer: Object-Oriented Software Construction 2nd edition, ISE Inc.Santa Barbara (California)
- 5) Andrew Troelsen: Pro C# 5.0 and the .NET 4.5 Framework, Apress, 2012, ISBN: 978-1-4302-4233-8
- 6) Nyékiné Gaizler Judit (szerk): Programozási Nyelvek, Kiskapu Kft, 2003.



