

Magasszintű Programozási Nyelvek II. – Kidolgozott vizsgakérdések

A kérdések dr. Kovásznai Gergely és Hernyák Zoltán előadás fóliái alapján lettek kidolgozva. A szöveg részek, ábrák ezekből a diákból lettek átemelve teljes egészében vagy kissé átdolgozva. Amire esetleg itt sem találtam választ az a Hernyák Zoltán wiki oldalán található anyagrész alapján került kiegészítésre. A helyírási hibákért, elírásokért elnézést kérek! Mindenkinek jó tanulást!

2016

1. OOP története. OOP alapfogalmak és alapelvek.

- Az OOP nyelvek és a procedurális nyelvek viszonya.

OOP = Objektum-orientált programozás

- Adatmodell: rekordok = osztályok (class)
- Algoritmus:
eljárások, függvények = metódusok
- OOP célja: adatmodell+algoritmus szorosabb integrációja

Alan Kay fektette le az OOP alapelveket diplomamunkájában 1969-ben.

Xerox Palo Alto-i kutatóközpontja:

- forradalmi kísérleti fejlesztések: GUI, egér, hálózat
- SmallTalk OOP nyelv
- Xerox PARC személyi számítógép
- „inspirálta” az Apple-t (Macintosh) és a Microsoft-ot (Windows)

Meglévő eljárásorientált nyelveket bővítik OOP lehetőségekkel:

- pl. Pascal -> Object Pascal, Delphi
- pl. C -> C++

Új, tisztán OOP nyelveket terveznek

- pl. Java, C#

A támogató nyelvek használata nem szerencsés választás. Ugyanis mindkét elvek precíz ismeretét igénylik a programozótól. Ezen túl az elvek keveredése néha félreértelmezhető, nehezen érthető programkódot eredményez. Szerencsésebb választás a tisztán OOP elvek alkalmazásával történő programírás.

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

- OOP alapfogalmak: osztály, mező, metódus, property, konstruktor, példányosítás, objektum stb.

Osztály: típusdefiníció, mely mezőket és metódusokat tartalmaz

Mező: osztályban adattárolási feladata van, a változóknak felel meg.

Metódus: osztályban funkciókat testesít meg, eljárásoknak és függvényeknek felel meg.

Példányosítás: adott osztály típusú adathoz memóriefoglalása és inicializálás.

Objektum: adott osztály egy példánya.

Konstruktor: osztály speciális metódusa, mely egy új példány inicializálását végzi.

Destruktor: osztály speciális metódusa, mely az objektum memóriában való felszabadítását végzi.

Adatrejtés: annak szabályozása, hogy az osztály mely mezői hogyan legyenek a kívüllág számára elérhetőek (olvasható/írható).

- Az OOP három alapelve és azok jelentősége.

Egységbezárás (encapsulation):

- Az osztály egyetlen egységként tartalmazza az adatokat (=mezők) és az őket kezelő metódusokat.
- A mezőket csak a metódusokon keresztül változtathatjuk meg, ellenőrzött módon.
- Az osztály felelős a saját állapota konzisztenciájának fenntartásában.

Öröklődés (inheritance):

- Egy már definiált osztályból (=ősosztály) új, hasonló osztályok (=gyerekosztály) kifejlesztése.
- A gyerekosztály örökli az ősosztály mezőit és metódusait.
- Ezeket a programozó felül is definiálhatja.

Sokoldalúság (polymorphism):

- A származtatás során az ős osztályok metódusai képesek legyenek az új átdefiniált metódusok használatára újraírás nélkül is.
- Ezt virtuális metódusokon keresztül érhetjük el.
 - Egyes nyelvekben minden metódus virtuális (pl.: Java)
 - A metódusoknál külön jelezni kell, melyik a virtuális (pl.: Delphi, C#, C++)

2. Adatrejtés. Property.

- A mező fogalma.

Mező vagy más néven adattag. Feladata az osztályban való adattárolás, az objektum aktuális állapotát tárolja. Gyakorlatilag egy változónak felel meg.

- Védelmi szintek, azok kapcsolata a hatáskör fogalmával.

Alapvetően 3 védelmi szint van.

- **private:** csak az adott osztályon belül érhető el
- **protected:** csak az adott osztály, és a belőle származtatott gyerekosztályok érhetik el
- **public:** bárki számára elérhető

Ha az osztály egy DLL-ben van definiálva, akkor:

- **internal:** csak adott DLL-en belül elérhető
- **internal protected:** csak adott DLL-en belül vagy gyerekosztályaiból elérhető

A védelem gyakorlatilag hatáskört módosít. Sajnos nincs arra mód, hogy egy változó lekérdezhetőségét és megváltoztathatóságát külön szabályozzuk pontosan azért, mert a védelem hatáskört jelent. Ha nincs hatáskör nincs hozzáférés. Ha van hatáskör módosítható is.

Az alapértelmezett védelmi szint a **private**.

- A public mező kontroll nélkül írható, ezért használata kockázatos, NEM AJÁNLOTT!
- A private/protected mező használata JAVASOLT, amelyhez a hozzáférést public metódusokon keresztül biztosítjuk.

- Az adatrejtés értelme, jelentősége.

Az *egységbezárás* alproblémája, lényege, hogy az osztály kritikus mezőit elrejtjük a külső programrészek elől.

Cél: A külvilág *csak ellenőrzött módon* férjen hozzá a mezőkhöz, vagyis az objektum képes legyen adatokat tárolni, azokat a külvilág képes legyen lekérdezni és megváltoztatni, de csak ellenőrzött módon!

- Property fogalma. Property készítésének szintaktikai és szemantikai szabályai.

Más néven tulajdonság. Tulajdonképpen egy „virtuális” mező: nem igazi, külön adatmező mivel a memóriában nem foglalódik neki hely (helytakarékos).

Van/lehet „getter” és „setter”része:

- C#-ban **get** és **set** kulcsszavak
- Nem kötelező mindkét részt kidolgozni:
 - csak **get**: csak olvasható mező
 - csak **set**: csak írható mező
 - **get + set**: írható és olvasható
- A **set** részben a **value** kulcsszóval lehet hivatkozni a beállítandó értékre.

Előnye:

- hibakezelés (mező közvetlen eléréséhez képest)
- szebb, elegánsabb kód (a metódushíváshoz képest)

Hátránya:

- sebesség-megtévesztő: a programozó gyors elérésű (fizikai) mezőnek hiszi, holott lassú és bonyolult kód állhat mögötte

```
class Teglalap
{
    private int X1,X2;
    private int Y1,Y2;
    public int Szeles
    {
        get
        {
            return X2-X1;
        }
        set
        {
            X2 = X1 + value;
        }
    }
}
```

- Az elrejtett mező kívülről történő írási és olvasási lehetőségei, azok szokásos megoldása property-ken és metódusokon keresztül.

Property-ken keresztül: Egy virtuális mezőt hozunk létre, aminek a védelmi szintje public/protected és a property get és set tulajdonságával tudjuk olvasni és írni a mezőt. Ezt az előző részben már kifejtettem.

Metódusokon keresztül: Nekünk kell megírunk a **get** és **set** metódusokat, ahol a **get metódus** segítségével tudjuk olvasni, a **set metódussal** pedig írni a mezőben tárolt adatot.

```

class Hallgato
{
    private int eletkor;

    public void setEletkor(int ujErtek)
    {
        if (18<=ujErtek && ujErtek<=70)
            eletkor = ujErtek;
        else throw new Exception("Hibás érték!");
    }

    public int getEletkor()
    {
        return eletkor;
    }
}

```

```
h.setAge( -40 );
```

- Hasonlítsa össze a property-n keresztüli megoldást a metóduson keresztüli megoldással!

Nem minden OOP nyelv támogatja a property-k készítését.

- Amelyik nyelv nem támogatja ott: **set** és **get** metódusok készítése.
- Amelyik nyelv támogatja ott érdemes ezt a megoldást használni:
 - nem jelent többletmunkát a készítőjének,
 - a felhasználói oldalon elegánsabb,
 - jobban olvasható kód megírását teszi lehetővé.

Ugyanakkor a felhasználói oldalt a property könnyen megtévesztheti. A felhasználói oldalon ugyanis nem könnyű megkülönböztetni a property-t a ténylegesen létező mezőktől, mivel mindkettőt ugyanazzal a szintaxissal kell használnia. De amíg a ténylegesen létező mező olvasása és írása direkt módon, nagy sebességgel történik, addig a property írása és olvasása minden esetben függvényhívással jár. Ez mindenképpen lassúbb, mint a mező esetén.

3. Osztálysintű és példányszintű mezők. Konstansok.

- A különbség az osztálysintű és példányszintű mezők között, különös tekintettel az élettartami kérdésekre.

Az objektum aktuális *állapotát* tárolják. Kétfajta lehet:

Példányszintű:

- példányonként más-más értékű lehet
- annyi „darab” a memóriában, ahány példány
- elérése: **példánynév.mezőnév**

Osztályszintű:

- osztályra vonatkozó információ, közös a példányok között
- program futása alatt 1 „darab” végig a memóriában
- elérése: osztálynév.mezőnév

- A példányszintű mezők deklarálásának és felhasználásának szintaktikája osztályon belül és kívül.

A példány szintű mezők az alapértelmezett mezők, vagyis deklarálásuk során nem kell semmilyen kulcsszóval megjelölni azt. Vagyis a legfontosabb jellemzője a példány szintű mezőknek, hogy nem szerepel a deklarációjukban a static kulcsszó.

Osztályon belüli felhasználása:

```
class Hallgato
{
    private static uint hallgatoSzam = 0;
    private string neptunKod, nev;
    ...
    public Hallgato(string nev, string neptunKod) {
        hallgatoSzam++;
        ...
    }
};
```

Osztályon kívüli felhasználása:

A mezőre hivatkozás előtt az osztályból példányt kell készíteni.

```
class TAuto
{
    public int tankAktualisToltottsege;
    public string rendszam;
}

public static void Main()
{
    TAuto saját = new TAuto();
    saját.rendszam = "EWING-01";
    saját.tankAktualisToltottsege = 45;

    TAuto szomszed = new TAuto();
    szomszed.rendszam = "BIGBOSS-001";
    szomszed.tankAktualisToltottsege = 30;
}
```

- Az osztályszintű mezők deklarálásának és felhasználásának szintaktikája osztályon belül és kívül.

Az osztály-szintű mezők olyan jellegű adatokat hordoznak, amelyet elég egyetlen példányban tárolni, mivel ezen adat minden példány esetén amúgy is ugyanolyan értéket hordozna. Ez memóriatakarékos megoldás. Az ilyen jellegű mezőket a **static** kulcsszóval kell megjelölni.

Osztályon belüli felhasználása:

```
class Hallgato
{
    private static uint hallgatoSzam = 0;
    private string neptunKod, nev;
    ...
    public Hallgato(string nev, string neptunKod) {
        hallgatoSzam++;
        ...
    }
};
```

Osztályon kívüli felhasználása:

Az osztályon kívüli hivatkozáshoz nem szükséges példányt készíteni. A mezőt osztálynév.mezőnév szintaktikával direkt módon is megcímezhetjük.

```
class Mercedes_S500
{
    public static int tankMaximalisKapacitas;
    public int tankAktualisToltottsege;
    public string rendszam;
}
```

```
public static void Main()
{
    Mercedes_S500.tankMaximalisKapacitas = 80;
    ...
    Mercedes_S500 merci1 = new Mercedes_S500(45, „JKL881”);
}
```

- A konstans fogalma, deklarációja, hasonlósága az osztályszintű mezőkkel.

Inicializálás során kap értéket, amit később *nem lehet* megváltoztatni. Minden konstans egyúttal osztályszintű mező, ezért osztálynéven keresztül lehet rá hivatkozni: **osztálynév.konstansnév**

```
class Kor
{
    public const double Pi = 3.14;
    public double sugar;
    ...
}
```

```
Kor kor = new Kor();
kor.sugar = 12.4;
double kerulet = Kor.Pi * kor.sugar * kor.sugar;
```

4. Osztálysintű és példányszintű metódusok.

- A példányszintű metódusok létrehozásának és felhasználásának szintaktikai és szemantikai szabályai.

Meghívása példányon keresztül történhet. Hozzáfér az osztály példányszintű és osztálysintű mezőihöz is.

```
class TAllat
{
    public int xPos, yPos;
    public double Menj(int ujX, int ujY)
    {
        xPos = ujX;
        yPos = ujY;
    }
}
```

```
TAllat kacska = new TAllat();
kacska.Menj(10,20);

TAllat macska = new TAllat();
macska.Menj(30,20);
```

- Az osztálysintű metódusok létrehozásának és felhasználásának szintaktikai és szemantikai szabályai.

Meghívása osztálynéven keresztül történik. Nem férhet hozzá példányszintű mezőkhöz, csak osztálysintűekhez. A **static** kulcsszó használata kötelező.


```
class TAllat
{
    public int xPos, yPos;
    public double Menj(int ujX, int ujY)
    {
        xPos = ujX;
        yPos = ujY;
    }
}
```

```
TAllat kacska = new TAllat();
kacska.Menj(10,20);

TAllat macska = new TAllat();
macska.Menj(30,20);
```

- Ismertesse a metódusok belsejéből más metódusok meghívhatóságának problémakörét!

Osztályszintű metódusból hívhatunk másik metódust, de csak ha az is osztályszintű! Példányszintű metódus hívására nincs lehetőség ugyanis példányszintű metódus hívásához példány is kellene.

```
class TTermesztet
{
    static public string Evszak;
    static public int datum;
    public static void datumNovel()
    {
        datum ++;
        if (datum==60) // március 1
            evszakValt();
    }
}
```

Példányszintű metódusból meghívhatunk másik példányszintű metódust is, és osztályszintű metódust is.

```
class TKutya
{
    double sulya = 10;
    public void eszik()
    {
        sulya += 0.1; // 10 dekát hízik
        vakkantas();
    }
    public void vakkantas()
    {
        Console.WriteLine("vau-vau");
    }
}
```

- Ismertesse a metódusok belsejéből más osztály belsejében deklarált mezők elérhetőségének problémakörét!

Az osztályszintű metódus belsejében nem szabad hivatkozni semmilyen példányszintű mezőre, hiszen elképzelhető olyan eset is, amikor az adott mezőből még nincs is a memóriában. Másrészt, ha lenne is példányunk, a metódus meghívásakor nem kell a példányra hivatkozni, így a metódus sem tudná, hogy melyik példány mezőjét kell elérni. Az osztályszintű metódus belsejében ezért csak osztályszintű mezőkre, és konstansokra szabad hivatkozni.

Mivel egy példányszintű metódus meghívása során a hívás helyén azonosítjuk a példányt, így a metódus belsejében szabad példányszintű mezőket kezelnünk, hiszen a rendszer tudni fogja, melyik példány mezőit kívánjuk kezelni. A példányszintű metódus belsejében az osztályszintű mezők, metódusok természetesen elérhetőek, hiszen azok működtetéséhez példányra sincs szükség.

- Milyen feltételek mellett készítünk osztályszintű, és milyenek mellett példányszintű metódusokat?

Osztályszintű metódust alkalmazunk, ha:

- A metódus futásához minden információt paraméterként adunk át.

```
class Math
{
    public static double Sin( double degree )
    {
        ...
    }
}
```

- Az adott osztályból úgysem lesz példányosítva, mert úgyis csak 1 db példány lenne belőle. Ekkor az osztály minden mezője osztályszintű, és minden metódusa osztályszintű.

```
class Console
{
    public static double WriteLine(...)
    {
        ...
    }
}
```

- Az adott metódus nem hivatkozik példányszintű mezőre, sem metódusra. Csak az osztályszintű mezőkkel dolgozik.

```

class TTermesztet
{
    static public string Evszak;
    public static void evszakValt()
    {
        switch (Evszak)
        {
            "nyár": Evszak = "ősz";break;
            "ősz": Evszak = "tél";break;
            ...
        }
    }
}

```

Példányszintű metódust alkalmazunk, ha:

- Az adott osztályt példányosítani akarjuk.
- Egy metódus törzsében példányszintű mezőre szeretnénk hivatkozni vagy példányszintű metódust akarunk hívni.
- Ugyanazon a példányon akarunk különböző műveleteket végre hajtani, vagy metódusokat meghívni.

- A 'this' kulcsszó és a metódusok kapcsolata.

Egy példányszintű metódus belsejében a példány mezőinek elérése automatikus, egyszerűen a mező nevének beírásával történik.

```

class TVadAllat
{
    private int Eletkor = 0;
    public void Oregszik()
    {
        Eletkor++;
        ...
    }
}

```

Néha szükséges az objektum metódusaiban magára az *aktuális példányra* hivatkozni. Ezt a 'this' szóval tehetjük meg:

```

class TVadAllat
{
    public void Megeszik(TNovenyevő aldozat)
    {
        class TNovenyevő
        {
            public void Megtamad(TVadallat gyilkos)
            {
                if (tulFaradt) gyilkos.Megeszik( this );
                ...
            }
        }
    }
}

```

Valójában a 'this' egy láthatatlan plusz paraméter, amelyet a fordítóprogram automatikusan kezel. A this szócskát osztály szintű metódusban (static) nem használható fel, mert a 'this' a példányt azonosítja, de az osztályszintű metódus hívásához nem kell (és nincs is) példány!

5. Öröklődés szabályai és alkalmazása mezőkre.

- Az öröklődés szabályai.

Az öröklődés az objektum-osztályok továbbfejlesztésének lehetősége. Ennek során a származtatott osztály öröklí ősétől azok attribútumait, és metódusait, de ezeket bizonyos szabályok mellett újakkal egészíthet ki, és meg is változtathatja.

1. Az eredeti osztályt ősosztálynak nevezzük (szülő).
2. Az új, továbbfejlesztett osztályt származtatott osztálynak (gyerek).
3. Az ősosztály nevét kettőspont mögött kell megadnunk a származtatott osztály neve után.
4. Egy ősből több leszármaztatott osztályt is készíthetünk.
5. Amennyiben nem választunk őst (nem jelölünk ki explicit módon egyet sem), úgy a fordító ezt úgy értelmezi, hogy az **Object** nevű objektumosztályt választjuk ősnek.
6. Egy származtatott osztálynak legfeljebb egy szülője lehet (pl.: Pascal, Java, C#): öröklődési fa (legfelül a választott kiinduló osztály foglal helyet, az alatta lévő szinten a közvetlen leszármazott osztályok, majd annak a gyermek-osztályai, stb.)
7. A metódusok törzsét megváltoztathatjuk.
8. A mezők neveit, típusait általában nem változtathatjuk meg.
9. Új mezőkkel, és metódusokkal egészíthetjük ki az osztályt.

- A mezők öröklődési szabályai példányszintű mezők esetén, a különböző védelmi szintek esetén.

Példányszintű mezők öröklődésének szabályai:

- Minden mezőt öröklünk, még a private mezőket is.
- A private mezőkre nem tudunk az új osztályban közvetlenül hivatkozni, de közvetve igen (meghívunk egy örökölt metódust, amely a törzsében felhasználja a private mezőt).
- A gyerekosztályban bevezethetünk új mezőket.
- Nem szoktunk ugyanolyan nevű mezőt bevezetni, mint amelyet már örököltünk, mert az zavaró. A C# ezt megengedi, de megerősítést kér.
- Ha mégis megteesszük, akkor ez újbóli definiálásnak számít, és ennek során akár más mezőtípust is választhatunk. De ez csak 'elfedi' az örököltet, meg nem változtatja azt!

Különböző védelmi szintek esetén:

- **Private** mezőt eleve nem látja a gyerekosztály, ezért bevezethetünk ugyanolyan nevű mezőt.
- **Protected/public** mezőt viszont látja, ezért ugyanolyan nevű mező bevezetésénél használni kell a **new** kulcsszót! A new-val jelzi a programozó a fordítónak, hogy „tudja, mit csinál”.
- A mezők öröklődési szabályai osztályszintű mezők esetén, a különböző védelmi szintek esetén.

Osztályszintű mezők öröklődésének szabályai:

- Valójában nem öröklődés.
- Ugyanarra a fizikailag csak egy példányban létező mezőre már kétféleképpen is hivatkozhatunk.

```
class Elso
{
    public static int a;
}
```

```
class Masodik : Elso
{
    public static int b;
}
```

```
Elso.a = 1;
Masodik.a = 2;
Masodik.b = 3;

cw(Elso.a); // 2
cw(Masodik.a); // 2
cw(Masodik.b); // 3
```

- Kiküszöbölhető a **new** kulcsszó használatával.

```
class Masodik : Elso
{
    new public static int a;
}
```

```
Elso.a = 1;
Masodik.a = 2;

cw(Elso.a); // 1
cw(Masodik.a); // 2
```

- A mezők újradefiniálási lehetőségei a különböző védelmi szintek esetén.

Ilyen szituációban a hatáskörök elemzése nyújt választ a problémás esetekben.

```
class TElso
{
    private int a;
    protected int b;
    public int c;
}
```

```

class TMasodik:TElso
{
    public string a;
    public double b;
    public string c;
    public string d;
}

```

A fenti példában az ősz osztályunk mindhárom mezőjét örökli a gyermekosztály, ami egyező nevű mezők bevezetését szeretné elvégezni.

Az **a** mező esetén egyező nevű új mező létrehozásának nincs akadálya, hiszen ez a mező az ősz osztályban **private** hatáskörű. Ennek megfelelően hatásköre nem terjed túl a **TElso** osztály keretein, így a **TMasodik**-beli egyező nevű mezővel nem kerül ütközésbe.

A **b** nevű mező bevezetésével (és hasonlóan a **c** mezővel is) azonban problémák vannak. Ugyanis ezen örökölt mezők elérhetőek lennének a **TMasodik** osztály metódusaiban is, hiszen hatáskörük kiterjed a gyermek-osztályra is. Az újonnan bevezetett egyező nevű mezők hatásköre átfedi ezt a területet. Az ilyen esetek mindig problémásak, hiszen a mezőre hivatkozás nem lesz egyértelmű.

Az ilyen szituációkat a fordítóprogram érhető okokból nem szereti. A tapasztalatok szerint az ilyen esetek a programozókat is megzavarják. Ugyanakkor nagyon ritkán indokolt az az eset, hogy az újonnan bevezetendő mező neve meg kell egyezzen az örökölt mező nevével.

Mivel az egyező név pedig csak a bajt okozná, ezért a C# fordító egyszerűen nem fogadja el a fenti problémás esetet, és szintaktikai hibára hivatkozva megtagadja a **TMasodik** osztály kódjának lefordítását. Ezzel kényszeríti a programozót, hogy gondolja át a névválasztását, és inkább egy eltérő, egyedi nevet válasszon az újonnan bevezetett mezők esetén.

Ugyanakkor a programozónak van döntési joga ez ügyben. Dönthet úgy, hogy nem hajlandó másik nevet választani a saját mezőinek, inkább vállalja az ezzel járó kényelmetlenségeket, és nehézségeket. Ekkor meg kell jelölni a problémás mezőket a **new** kulcsszóval (jelentése új).

A **new** kulcsszót csak ilyen jellegű, problémás esetben szabad használni. A **new** kulcsszó felhívja a forráskódot böngésző programozók figyelmét arra, hogy ezen mezők újradeklarálásra kerültek, ezért óvatosan értelmezzék a kódot.

Ilyen megerősített esetekben a fordítóprogram tudomásul veszi a hatáskörök átfedésével kapcsolatos problémákat. Átfedés esetén mindig az az azonosító élvez elsőbbséget, amely később került deklarálásra, aki hozzánk közelebb van, vagyis elsősorban a saját, új mezőink.

6. Öröklődés szabályai és alkalmazása metódusokra.

- Az öröklődés szabályai.

Az öröklődés az objektum-osztályok továbbfejlesztésének lehetősége. Ennek során a származtatott osztály öröklí ősétől azok attribútumait, és metódusait, de ezeket bizonyos szabályok mellett újakkal egészíthet ki, és meg is változtathatja.

1. Az eredeti osztályt ősosztálynak nevezzük (szülő).
 2. Az új, továbbfejlesztett osztályt származtatott osztálynak (gyerek).
 3. Az ősosztály nevét kettőspont mögött kell megadnunk a származtatott osztály neve után.
 4. Egy ősből több leszármaztatott osztályt is készíthetünk.
 5. Amennyiben nem választunk őst (nem jelölünk ki explicit módon egyet sem), úgy a fordító ezt úgy értelmezi, hogy az **Object** nevű objektumosztályt választjuk ősnek.
 6. Egy származtatott osztálynak legfeljebb egy szülője lehet (pl.: Pascal, Java, C#): öröklődési fa (legfelül a választott kiinduló osztály foglal helyet, az alatta lévő szinten a közvetlen leszármazott osztályok, majd annak a gyermek-osztályai, stb.)
 7. A metódusok törzsét megváltoztathatjuk.
 8. A mezők neveit, típusait általában nem változtathatjuk meg.
 9. Új mezőkkel, és metódusokkal egészíthetjük ki az osztályt.
- A metódusok öröklődési szabályai példányszintű metódusok esetén, a különbözővédelmi szintek esetén.
 - A metódusok öröklődési szabályai osztályszintű metódusok esetén, a különböző védelmi szintek esetén.

Az öröklés során az örökölt metódusok sok esetben (egyszerűbb esetekben) úgy viselkednek, mintha azok eleve osztályunk részét képeznék. Ugyanúgy meghívhatóak, mintha eleve az osztályunkban lett volna a kód megírva.

Természetesen gond van a 'private' hatáskörű metódusokkal, amelyek nem hívhatóak meg a gyermekosztályunk példányán keresztül sem, illetve a gyermekosztályban bevezetett új metódusokból sem lehet őket meghívni, mivel hatáskörük nem terjed ki a gyermekosztályra.

- A metódusok újradefiniálási lehetőségei a különböző védelmi szintek esetén, a paraméterlista változása és nem változása esetén (overloading szabály, virtuális metódusok, egyszerű felüldefiniálás esete).

Az **overloading szabály** azt mondja ki, hogy ugyanazzal a névvel több metódus is létezhet, de a formális paraméterlistájuk egyértelműen különböző kell legyen. Ezen különbség azért fontos, hogy a hívás helyén feltüntetett aktuális paraméterlista alapján a fordítóprogram egyértelműen el tudja dönteni, melyik metódus-változatot kívánjuk meghívni.

```
class Hallgato
{
    public void Beiratkozas()
    {
        ...
    }
}
```

```
class Hallgato2 : Hallgato
{
    public void Beiratkozas(
        int ev, int felev )
    {
        ...
    }
}
```

```
Hallgato2 k = new Hallgato2();
k.Beiratkozas();
k.Beiratkozas(2006,1);
```

Az ugyanolyan nevű és paraméterezésű metódus „elfedi” az őszintább metódusát ekkor korai kötés kell alkalmazni, hogy a fordítónak jelezzük tudjuk mit csinálunk.

- **new** kulcsszó – Korai kötés: a fordító „összeköti” a metódus hivatkozását a fizikai metódussal.
- A metódushívások által hivatkozott metódusok **fordítási időben** lesznek kiválasztva.

Jó lenne, ha:

- az őszintább „érzékelné”, ha egy metódusát egy gyerekosztálya felüldefiniálja!
- a fordító nem „kötné össze” a metódushívásokat a fizikai metódusokkal!
- ha az „összekötés” csak **futási időben** történne meg!
- ha az alapértelmezett korai kötés helyett **késői kötés** is lehetne használni!

Az ilyen esetekben kell alkalmazni a virtuális metódusokat, amik olyan metódusok, melyeket a gyerekosztályokban valószínűleg felüldefiniálunk.

- Az őszintább metódusai is meg tudják hívni
- Késői kötéssel kezeljük őket
- Az őszintábbban **virtual** kulcsszóval kell őket megjelölni.
- A gyerekosztályban **override** kulcsszóval kell őket felüldefiniálni.

- A metódusok hívása esetén hogyan dől el, melyik metódus kerül végrehajtásra?

Korai kötés esetén:

- A metódushívások által hivatkozott metódusok **fordítási időben** lesznek kiválasztva.

Késői kötés esetén:

- Késői kötésnél csak **futási időben** dől el, hogy a virtuális metódus hívása melyik konkrét metódust futtassa.
- A híváshoz használt példány típusa alapján dől el.
- Az ún. virtuális metódus tábla (VMT) segítségével dönti el ezt a futtatórendszer.
- A VMT fordítási időben készül, minden osztályhoz 1 db.

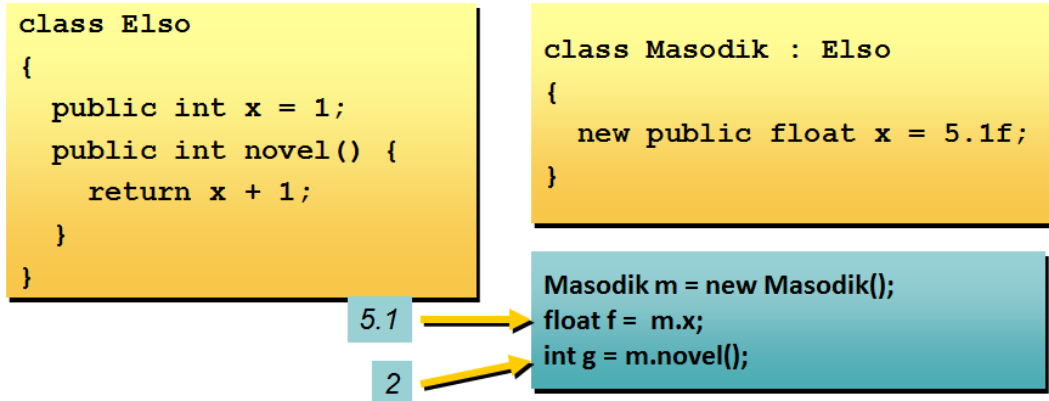
7. Korai és késői kötés.

- A korai kötés lényege, alkalmazása az OOP kódban. Adjon példát, amikor a korai kötés 'hibás' működésűvé válik. Ismertesse ennek körülményeit!

new = korai kötés

Az ősosztály metódusai az ősosztály mezőit használják akkor is, ha gyerekosztályban felüldefiniáltuk a mezőt.

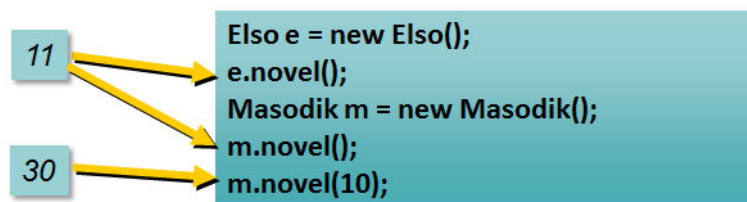
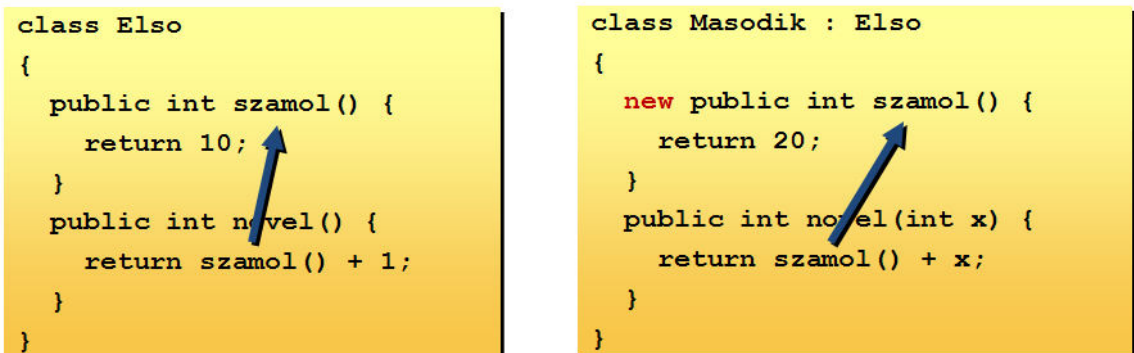
Korai kötés: a fordító „összeköti” a mező hivatkozását a fizikai mezővel.



Ugyanolyan nevű és paraméterezésű metódus „elfedi” az ősosztály metódusát.

Megoldás: **new** kulcsszó – Korai kötés: a fordító „összeköti” a metódus hivatkozását a fizikai metódussal.

A metódushívások által hivatkozott metódusok **fordítási időben** lesznek kiválasztva.



Az alábbi példában ugyanazok a metódusok futnak le!

```
class Repulo {
    public void Felszall() {...}
    public void Leszall() {...}
    public void Repul() {...}
    public void Gyakorlokor() {
        Felszall();
        Repul();
        Leszall();
    }
}
```

```
class Helikopter : Repulo {
    new public void Felszall() {...}
    new public void Leszall() {...}
    new public void Gyakorlokor() {
        Felszall();
        Repul();
        Leszall();
    }
}
```

Ezért a copy-paste SOHASEM jó megoldás!

- A késői kötés lényege, alkalmazása az OOP kódban.

virtual + override = késői kötés

Olyan (virtuális) metódusokat használunk, melyet a gyerekosztályban felül tudunk definiálni, de az őszosztály metódusai is megtudják hívni.

- Felüldefiniáláskor kötelező használni az override-ot!
- Nem szabad meg változtathatni a felüldefiniált metódus egyik adatát sem (név, paraméterezés, visszatérési típus)!
- Csak a metódus törzsét szabad átírni!
- Virtuális metódus nem lehet private! (Nincs értelme.)

- Ismertesse, hogy dől el, hogy melyik kötés lesz korai, melyik késői!

A kötés során metódushívást rendelünk metódushoz.

A **korai kötés**-ről akkor beszélhetünk, ha ez az összerendelés **fordítási időben** történik, a fordítóprogram által. A korai kötés legfontosabb jellemzője, hogy **oldhatatlan**, megváltoztathatatlan kötés.

A másik kezelési technika a **késői kötés**, de ennek alkalmazásához a metódust meg kell jelölni. Az alapértelmezett kezelés a korai kötés, ezért ha késői kötést szeretnénk, akkor jelzést kell adnunk a fordítóprogramnak, hogy valamely metódust ilyen típusúnak kezelje.

- Ismertesse a késői kötés működésének támogatását VMT és DMT táblák esetén! Hasonlítsa össze a két technikát (előnyök, hátrányok)!

Virtuális metódus tábla (VMT):

- Késői kötésnél csak **futási időben** dől el, hogy a virtuális metódus hívása melyik konkrét metódust futtassa.
- A híváshoz használt példány típusa alapján dől el.
- Az ún. virtuális metódus tábla (VMT) segítségével dönti el ezt a futtatórendszer.
- A VMT fordítási időben készül, minden osztályhoz 1 db.

```
class Elso
{
    public virtual int A() {}
    public virtual int B() {}
    public virtual void D() {}
    public void E() {}
}
```

Elso VMT

A		TElso.A
B		TElso.B
D		TElso.D

```
class Masodik : Elso
{
    public override int B() {}
    public virtual void C() {}
}
```

Masodik VMT

A		TElso.A
B		TMasodik.B
D		TElso.D
C		TMasodik.C

Új osztály VMT-jének készítéséhez:

1. Másold le az őosztály VMT-jét
 - ha nincs őőd (Object), akkor üres a VMT
2. Van itt override-olt metódus?
 - ha igen, írd át a megfelelő sort a VMT-ben!
3. Van itt virtual metódus?
 - ha igen, add hozzá új sorként a VMT-hez!

Hátrány: nagy a memóriaigénye

- a gyerekosztály VMT-je legalább olyan hosszú, mint az őosztályé
- ha a gyerekosztályban nincs override-olás, akkor is tartalmazza az adott metódus adatait

Dinamikus metódus tábla (DMT)

- Tárolunk az őosztályra egy „visszamutatót”

```
class Elso
{
    public virtual int A() {}
    public virtual int B() {}
    public virtual void D() {}
    public void E() {}
}
```

Elso DMT

ŐS = null		
A		TElso.A
B		TElso.B
D		TElso.D

```
class Masodik : Elso
{
    public override int B() {}
    public virtual void C() {}
}
```

Masodik DMT

ŐS = TElso.DMT		
B		TMasodik.B
C		TMasodik.C

Új osztály DMT-jének készítéséhez:

1. A DMT üres
2. Állítsd be az ŐS-t az őosztály DMT-jére!
3. Van itt override-olt metódus?
 - ha igen, add hozzá új sorként a DMT-hez!
4. Van itt virtual metódus?
 - ha igen, add hozzá új sorként a DMT-hez!

DMT használata:

- Az osztály DMT-jében keressük a metódust
- Ha nincs ott, akkor folytatjuk az őosztály DMT-jében a keresést.

DMT előnye: kisebb memóriefelhasználás

DMT hátránya: lassabb

8. Konstruktorkok.

- A konstruktor fogalma, feladatai.

Az osztály speciális metódusa, ami az új példány (mezői) számára memóriát foglal.

Feladata: példányosításkor az új objektumot konzisztens belső állapotba helyezi

- A mezők inicializálásakor ellenőrzéseket végez.
- Az objektum későbbi konzisztens állapotban tartását a property-kkel és metódusokkal kell biztosítani.

Kötelező példányosításkor konstruktort hívni!

- Hogyan lehet konstruktor írása nélkül is alaphelyzetbe állítani az objektum-mezőket, illetve milyen esetekben nem működik ez az alternatív módszer?

Amennyiben mi nem készítünk valamely osztályhoz konstruktort, akkor megteszi helyettünk azt a fordító. Elkészít egy olyan konstruktort, amelynek üres a paraméterlistája, és az utasításblokkja is. Ennek a konstruktornak csakis az a szerepe, hogy a példányosításkor meg lehessen hívni. Azonban ez a konstruktor a mezőknek nem ad kezdőértéket, tehát alkalmazása komolyabb, professzionálisabb esetekben nem ajánlott.

- A konstruktor írás szintaktikai szabályai.

- Neve kötött: ugyanaz, mint az osztály neve.
- Nincs visszatérési típusa (még void sem!)
- Lehet paramétere.
- Egy osztálynak több konstruktora is lehet (overloading):
 - Formális paraméterlistájuk különbözzön, mivel a nevük azonos.

- Alapértelmezett konstruktor.

Mi történik, ha egy osztályhoz nem adunk meg konstruktort? Tudjuk, hogy példányosításhoz muszáj legalább 1 db. konstruktornak lennie! Ilyenkor a fordító automatikusan létrehoz egy **paraméter nélküli** konstruktort.

```
class Haromszog
{
    int a, b, c;
}
```

```
Haromszog h = new Haromszog();
```

Ha írtunk konstruktort, akkor nem jön létre alapértelmezett konstruktor.

```
class Haromszog
{
    int a, b, c;

    public Haromszog(int a, int b, int c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }
}
```

~~Haromszog h = new Haromszog();~~

- Hogyan hívhat meg egy konstruktorból másik saját osztálybeli, illetve ősz osztálybeli konstruktort?

- Mivel nem sima metódus, nem hívható egyszerűen a konstruktor törzséből.
- Speciális szintaxisra van szükség.
- A konstruktor fejléce mögé **kettőspontot** kell írni, majd a **this** vagy **base** kulcsszó segítségével lehet másik konstruktorra hivatkozni.

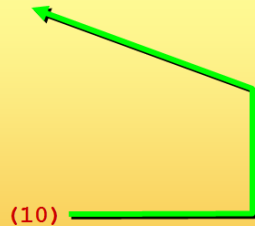
Konstruktorból egy másik osztálybeli konstruktor hívása (':' és this() szó)

```
class TKor
{
    private int X,Y;
    public TKor():this(0,0)
    {
        // itt már más dolgunk nincs is
    }
    public TKor(int ujX, int ujY)
    {
        X = ujX;
        Y = ujY;
    }
}
```

```
TKor k1 = new TKor(30,40);
TKor k2 = new TKor();
```

Konstruktorból az ősz osztály egy konstruktorának hívása (‘:’ és base() szó)

```
class TÖs
{
    public TÖs(int akarmi)
    {
    }
}
class TGyerek:TÖs
{
    public TGyerek():base (10)
    {
        ...
    }
}
```



- Hogyan és miként hívjuk meg a konstruktort példányosításkor, és írja le mely mechanizmus szerint hajtódnak végre a konstruktorok az adott osztályból, illetve annak őssosztályaiban!

Általában nem csak 1 db. konstruktor fut le példányosításkor.

A kulcs: **öröklés**.

- Legyen az O1 osztály az őssosztály, O2 pedig az ő gyerekosztálya. Azaz O2 öröklí O1 mezőit.
- O1 konstruktora inicializálja az O1-ben deklarált mezőket, O2 konstruktora az O2-ben deklaráltakat.
- Mi történik, ha O2-t példányosítjuk? Nem csak O2 konstruktorának, hanem O1-ének is le kell futnia!
- Különösen fontos, hogy O1 privát mezőit csak O1 konstruktora tudja elérni!

Konstruktor hívási lánc:

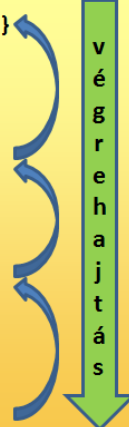
- Konstruktórhíváskor először az **őssosztály** konstruktora fut le, csak utána a gyerekosztályé.
- Ez így fut végig az **összes** őssosztályon, az Object-ig.
- Tehát az őssosztályok konstruktorai **fentről lefelé** futnak le.

```
class Object {
    public Object() { ... }
}

class O1 {
    public O1() { ... }
}

class O2 : O1 {
    public O2() { ... }
}

class O3 : O2 {
    public O3() { ... }
}
```



```
O3 x = new O3();
```

Problémák:

- Ha az ősnek van paraméter nélküli konstruktora, a fordító automatikusan azt futtatja.
- És ha nincs az ősnek ilyen konstruktora?
- És ha mi azt szeretnénk, hogy másikat futtasson le?
- Hogyan legyen az paraméterezve?

```
class Negyzet {  
    int a;  
    public Negyzet(int aOldal) {  
        a = aOldal;  
    }  
}  
  
class Teglalap : Negyzet {  
    int b;  
    public Teglalap(int aOldal, int bOldal) {  
        b = bOldal;  
    }  
}
```

Teglalap t = new Teglalap(10, 20);

HIBA! A fordító nem tudja kiválasztani és felparaméterezni az ősosztály futtatandó konstruktorát!

A **base** kulcsszó: az ősosztály konstruktorának explicit kiválasztására és felparaméterezésére.

```
class Negyzet {  
    int a;  
    public Negyzet(int aOldal) {  
        a = aOldal;  
    }  
}  
  
class Teglalap : Negyzet {  
    int b;  
    public Teglalap(int aOldal, int bOldal) : base(aOldal) {  
        b = bOldal;  
    }  
}
```

- A **base** és **this** ugyanolyan szintaxissal használhatóak.
- Mindketten a konstruktorhívási láncot befolyásolják.
- **this(...)**: az aktuális osztály konstruktora
- **base(...)**: az ősosztály konstruktora

- Az osztályszintű konstruktor fogalma, írása, használata.

Az osztály osztályszintű (static) mezőinek inicializálására is szükség van. Ezt végzi el az osztályszintű konstruktor.

- Neve kötött: ugyanaz, mint az osztály neve
- Nincs visszatérési típusa (még void sem!)
- Nincs paramétere
- static
- Kötelezően private, ezért nem is kell a láthatóságát megadni
- Csak 1 db. lehet belőle egy osztályban (következik a fentiekből)

```
class Student {  
    static uint numberOfStudents = 0;  
    static uint maxNumberOfStudents;  
  
    static Student() {  
        Console.WriteLine("Add meg a hallgatók max. számát!");  
        maxNumberOfStudents = uint.Parse(Console.ReadLine());  
    }  
}
```

- Egyszerűbb static mezők kezdőértékét a deklarációval egy sorban hozzájuk tudjuk rendelni.
- Egyes static mezőknél viszont bonyolultabb kód alapján megy a kezdőértékkadás.

Hogyan lehet lefuttatni?

- Kódból (explicit módon) nem. Nem is tudnánk, hiszen private.
- A futtatórendszer hívja meg automatikusan.

Mikor fut le?

- Nem tudjuk előre megmondani, a futtatórendszer dönti el.
- A rendszer garantálja, hogy azelőtt, mielőtt az osztályra az első hivatkozás történne.

- Ismertesse az object factory fogalmát!

Ha az osztályunknak csak **privát konstruktora** van, akkor a külvilág nem tudja meghívni, nem tudja példányosítani!

Mi az értelme a privát konstruktornak?

- Nem tesszük lehetővé, hogy bárki ellenőrizetlen módon példányokat készítsen az osztályból.
- Készítünk egy **public static metódust**, mellyel **ellenőrzött módon** készíthetünk példányt.
- Az ilyen megoldás neve: **object factory**

Az object factory ellenőrzéseket végezhet el, mielőtt új példányt hozunk létre. Meg is tagadhatja a példány létrehozását.


```

class User {
    string name, passwd;

    private User(string name, string passwd) {
        this.name = name;
        this.passwd = passwd;
    }

    public static User CreateUser(string name, string passwd) {
        if (IsPasswdOK(passwd))
            return new User(name, passwd);
        else return null;
    }
}

```

User u = User.CreateUser("pisti", "jelszo123");
 if (u == null) { ... nem jó a jelszavam ... }
 else { ... dolgozom a példánnyal ... }

Példa az Object factoryra:

```

private static dbSzam = 0;

public static SajatOsztaly Letrehoz()
{
    if (dbSzam >= 10) return null;
    else {
        dbSzam++;
        return new SajatOsztaly();
    }
}

```

```

SajatOsztaly x = SajatOsztaly.Letrehoz();
if (x == null) { ... túl sok létrehozott példány ... }
else { ... dolgozom a példánnyal ... }

```

9. Típuskompatibilitás.

- Ismertesse az OOP területén értelmezett típuskompatibilitás fogalmát!

Az OOP-ben egy 'Y' objektumosztály kompatibilis egy 'X' objektumosztállyal, ha a 'Y'-nak 'X' az őse.

- Y az X minden mezőjét, property-jét, metódusát örökli.
- Ezért mindent, amit el tudunk végezni X-szel, el tudjuk végezni Y-nal is.
- Ezért Y egy példánya képes helyettesíteni X bármely példányát.

```
class X {
    public int szam = 10;
    ...
}
class Y : X {
    ...
}
```

```
static void Kiir(X x) {
    Console.WriteLine(x.szam);
}
static void Main() {
    X x = new X();
    Y y = new Y();
    Kiir(x);
    Kiir(y);
}
```

Tranzitív tulajdonság: ha Z kompatibilis Y-nal és Y X-szel, akkor Z kompatibilis X-szel.

```
class X { ... }
class Y : X { ... }
class Z : Y { ... }

static void Main() {
    X x;
    Y y;
    Z z = new Z();
    y = z;
    x = y;
}
```

x = z;

X x = new Z();

Jelentősége:

```
class Negyzet { int a; ... }
class Teglalap : Negyzet { int b; ... }

static void Kiiras(Negyzet x) {
    Console.WriteLine(x.Kerulet());
}
static void Main() {
    Negyzet n = new Negyzet(12);
    Kiiras(n);
    Teglalap t = new Teglalap(10, 20);
    Kiiras(t);
}
```

Fontos: Általános felhasználású metódusokat lehet készíteni.

- Az Object típus kapcsolata a típuskompatibilitással, ennek használata kollekciók (listák, verem, sor, stb.) fejlesztésekor.

C#-ban ha egy osztálynak nincs megjelölt őse, akkor a nyelv automatikusan az Object osztályt rendeli hozzá ősnek! Mivel minden osztálynak az Object az őse ezért minden osztály kompatibilis az Object-tel.

```
class Negyzet {
    int a;
    public override bool Equals(Object o) {
        return this.a == o.a;
    }
}

static void Main() {
    Negyzet n1 = new Negyzet(12);
    Negyzet n2 = new Negyzet(16);
    if (n1.Equals(n2)) ...;
}
```

Használata kollekciók esetén:

```
class ArrayList
{
    public void Add(Object item) { ... }
```

```
class Stack
{
    public void Push(Object item) { ... }
    public Object Pop() { ... }
```

```
class Console
{
    public static void WriteLine(Object item) { ... }
```

```
Stack st = new Stack();
...
TSajat t = new TSajat();
st.Push( t ); // működik
...
```

```
TSajat x = st.Pop(); // nem működik !
```

```
TSajat x = st.Pop() as TSajat; // nem biztonságos
```

```
TObject o = st.Pop();
if (o is TElso) t = o as TElso; // biztonságos
```

- Ismertesse a korai kötés során fellépő problémás eseteket!

```
class Negyzet {
    int a;
    public int Kerulet {
        get { return 4 * a; }
    }
}
class Teglalap : Negyzet {
    int b;
    public new int Kerulet {
        get { return 2 * (a + b); }
    }
}
```

Ugyanaz az érték kerül kiírásra (48)!

```
Negyzet n = new Negyzet(12);
Kiiras(n);
Teglalap t = new Teglalap(10, 20);
Kiiras(t);
```

```
public void Kiiras(Negyzet x) {
    Console.WriteLine(x.Kerulet);
}
```

- Dinamikus típus azonosítási lehetőség ('is' operátor).

Formája: **<példány> is <osztály>**

Logikai értéket ad vissza:

- **true**, ha a példány típusa kompatibilis az adott osztállyal

```
static void Kiiras(Negyzet x) {
    if (x is Negyzet) ...;
    else if (x is Teglalap) ...;
}
```

```
if (x is Teglalap) ...;
else ...;
```

Mindig felesleges: **x is Object, x as Object**

- Dinamikus típuskényszerítés lehetőségei ('as' operátor), ennek lehetséges veszélyei.

Formája: **<példány> as <osztály>**

A példány úgy viselkedik, mintha az adott osztály példánya lenne. Késői kötés működéséhez ez a típuskényszerítés nem szükséges.

```
class Negyzet {
    int a;
    public override bool Equals(Object o) {
        Negyzet n = o as Negyzet;
        return this.a == n.a;
    }
}
```

`this.a == (o as Negyzet).a`

Formája: **<példány> as <osztály>**

`this.a == (o as Negyzet).a`

Helyette ez is lehet: **(<osztály>)<példány>**

`this.a == ((Negyzet)o).a`

Ha a típuskényszerítés nem végrehajtható (azaz nem kompatibilisek a típusok), akkor **InvalidCastException** kivétel dobódik. Ezért típuskényszerítés ('as') esetén mindig ellenőrizzük le, hogy a kompatibilitás fennáll-e az 'is' operátorral!

10. Sealed, static és absztrakt osztályok.

- A sealed kulcsszó jelentősége, használatának szintaktikai, szemantikai szabályai.

Sealed = lepecsételt.

Sealed osztály:

Egy objektumosztályt megjelölhetek 'sealed' kulcsszóval is. Ez azt jelenti, hogy:

- Az osztályt „véglegesnek” tekintjük, **zároljuk**.
- Nem lehet továbbfejleszteni (nem lehet más osztályok őse).
- A fordító bizonyos optimalizálásokat végezhet el, pl. virtuális metódushívásokat feloldhat korai kötéssel.
- A sealed ('lepecsételt') kulcsszót nem lehet az 'abstract' jelzővel együtt szerepeltetni!

```
sealed class TBefejezett
{
    ...
}
```

Sealed metódus:

- Nem csak osztályok, hanem metódusok is lehetnek sealed-ek.
- Csak virtuális metódusokra alkalmazható.
- **sealed** csak **override**-dal együtt szerepelhet.

```
class RoundShape {  
    public virtual double Area() { ... }  
}  
  
class Circle : RoundShape {  
    ...  
    public override sealed double Area() {  
        return 4 * radius * radius * Math.PI;  
    }  
}
```

Sealed property:

A metódusok zárolásához hasonló működést eredményez, ha a 'sealed' kulcsszót property-kre alkalmazzuk (a property is lehet virtual + override!).

```
class RoundShape {  
    public virtual double Area {  
        get { ... }  
    }  
}  
  
class Circle : RoundShape {  
    ...  
    public override sealed double Area {  
        get {  
            return radius * radius * Math.PI;  
        }  
    }  
}
```

- A static osztályok használatának szintaktikai és szemantikai szabályai.
 - Csak **osztályszintű** (static) mezőket, property-ket, metódusokat tartalmazhat.
 - Nem lehet benne példányszintű elem, így konstruktor sem.
 - Úgynevezett **singleton** osztály = csak 1 db. létezik a memóriában.
 - A fordító sem generál alapértelmezett konstruktort.
 - Ezzel a belőle öröklést is letiltjuk, a konstruktorhívási lánc miatt.
 - Erősebb, mint a sealed.

- Az absztrakt osztályok fogalma, jellemzői.

Olyan esetben használunk absztrakt osztályokat, mikor az osztályban egy metódust még nem tudunk megírni, mert a metódus implementációja csak a gyerekosztályokban nyer értelmet.

- **abstract** kulcsszóval kell megjelölni a nem kidolgozott metódusokat.
 - Ne adjunk törzset nekik!
- Az absztrakt metódusokat tartalmazó osztály elé is **abstract** kulcsszót kell írni.
- A gyerekosztályokban kötelező override-olni az abstract metódusokat!
 - kivéve, ha a gyerekosztály is absztrakt

- Az absztrakt osztályok és a virtuális metódusok előnyei, hátrányai, hasonlósága, különbsége egymással szemben.

Absztrakt osztály:

- Az 'abstract' kulcsszó használatával jelezzük, hogy nem tudjuk megírni a metódusokat, de deklaráljuk őket. Megadjuk a nevüket, és a paraméterezésüket.
- Egyáltalán nem írunk hozzájuk utasításblokkot, helyette pontosvesszőt teszünk.
- Az ilyen jellegű metódusok esetén a késői kötés kötelező.
- Az absztrakt osztályokból nem lehetséges a példányosítás, mivel a fordítóprogram ezt megtiltja.

Előnyei:

- Ha elfelejtkeznénk valamely örökölt absztrakt metódus felüldefiniálásáról, úgy a fordítóprogram erre figyelmeztet.

Virtuális metódus:

- Az ilyen jellegű metódusok esetén a késői kötés kötelező.
- Használjuk a 'virtual' és 'override' kulcsszavakat.
- Ezzel a megoldással valójában csak egy a baj: a gyermekosztály fejlesztője nem köteles a virtuális metódusokat felüldefiniálni.

Hátrányai:

- Ha a programozó "elfelejti" kidolgozni a metódust akkor a fordítóprogram, sem senki más nem ad figyelmeztető jelzést, hogy elfelejtkezett a metódus megírásáról.
- A fordítóprogram a fenti, 'virtual' metódusokat teljes értékű metódusoknak tekinti, és lefordítja azok törzsét (még ha az amúgy üres is)
- Ha ezek a metódusok nem 'void' visszatérési típusúak, akkor az üres törzsű változatban is definiálni kell visszatérési értéket.
- A fordítóprogram megengedi a példányosítást belőle.

- Az absztrakt metódusok használatának szintaktikai és szemantikai szabályai.
 - Minden absztrakt metódus egyben virtuális is, ezért nem kell kiírni a `virtual` kulcsszót.
 - Osztályszintű metódusok nem lehetnek absztraktak.
 - Property is lehet absztrakt.

```
public abstract void Kirajzol();
```

```
public abstract void Letorol();
```

```
public override void Kirajzol() {
```

```
    ...
```

```
}
```

```
public override void Letorol() {
```

```
    ...
```

```
}
```

- Ismertesse az absztrakt osztályokra és gyermekosztályaikra vonatkozó további szabályokat!

A gyerekosztályban továbbra sem kötelező az összes absztrakt metódust és property-t kifejleszteni, de ez esetben a gyerekosztályt is meg kell jelölni az 'abstract' jelzővel, és a gyerekosztályból sem lehet példányosítani.

Az 'abstract' jelzőt akkor is rátehetem egy osztályra, ha az egyáltalán nem tartalmaz egyébként absztrakt metódusokat. Ekkor a fordítóprogram nem fogja engedni, hogy példányosítsanak ebből az osztályból.

11. A 'this' kulcsszó. Indexelők.

- A `this` kulcsszó fogalma.

Néha szükséges az objektum metódusaiban magára az *aktuális példányra* hivatkozni. Ezt a 'this' foglalt szóval tehetjük meg.

- A `this` kulcsszó használatának lehetőségei metódus, property, konstruktor belsejében.

Metódusban:


```

class VadAllat
{
    public void Megeszik(Novenyevo aldozat)
    {
        class Novenyvevo
        {
            public void Megtamad(Vadallat gyilkos)
            {
                if (tulFaradt) gyilkos.Megeszik( this );
                ...
            }
        }
    }
}

```

Property-ben:

- Ha a formális paraméter neve megegyezik a mező nevével.
- **this** nélkül: a paramétert jelöli
- **this**-szel: hangsúlyozza, hogy ez példánymező

Konstruktorban:

- Ha a formális paraméter neve megegyezik a mező nevével.
- **this** nélkül: a paramétert jelöli
- **this**-szel: hangsúlyozza, hogy ez példánymező

```

class TSajat
{
    private int x=0;
    public TSajat(int x)
    {
        this.x += x;
    }
}

```

A this szócskát osztály szintű metódusban (static) nem használható fel, mert a 'this' a példányt azonosítja, de az osztályszintű metódus hívásához nem kell (és nincs is) példány!

- Ismertesse az indexelő fogalmát!

Más néven indexelő property. Akkor írunk ilyen, ha az osztályt olyan alakban akarjuk kezelni, mintha ő egy tömb lenne.

- Az indexelő használatának szintaktikai és szemantikai szabályai.

- Írunk egy 'this' nevű property-t.
- Megadjuk, hogy milyen típusú értékkel akarjuk indexelni (előző példában int)

```
LancoltLista lista = new LancoltLista();  
lista[0] = "hello";
```

this.set hívása
i=0
value="hello"

```
string s = lista[0];
```

this.get hívása
i=0

Példa létező indexerekre:

```
class String {  
    public char this[int i] {  
        get {  
            // a sztring i. karakterének visszaadása  
        }  
    }  
}
```

```
class ArrayList {  
    public object this[int i] {  
        get { // a lista i. elemének visszaadása }  
        set { // a lista i. elemének beállítása }  
    }  
}
```

Mátrixszerű indexelő:

```
class HaromszogMatrix {  
    private object[] elemek; // sorfolytonos tárolás  
  
    public object this[int sor, int oszlop] {  
        get {  
            return elemek[...];  
        }  
        set {  
            elemek[...] = value;  
        }  
    }  
}
```

- Indexelő jelentősége.

Az indexelés nem feltétlenül számmal történhet csak, hanem tetszőleges típusa lehet (pl. string).

```

class EmberLista {
    private List<Szemely> emberek; // Szemely példányok listája

    public Szemely this[string szemIgSzam] {
        get {
            // adott azonosítójú személy visszaadása
            // most: lambda kifejezés felhasználásával
            return emberek.First(p => p.Id == szemIgSzam);
        }
    }
}

```

12. Interface.

- Az interface fogalma.

Új nyelvi elem, amit arra használunk, hogy egymással nem rokon osztályok funkcionalitását írja le.

- Nem számít osztálynak (még absztraktnak sem, de arra hasonlít.)
- Metódusok és property-k szignatúráját tartalmazza (nincsenek implementálva)
- Mindegyikük publikus.
- Nem tartalmazhat mezőket.

- Milyen elemekből épülhet fel egy interface, mely elemeknek mik a szintaktikai szabályai?

Az interface-ek-ben lévő metódusok és property deklarációk mindegyike kötelezően public, ezért az interface belsejében nem kell ezt külön jelölni, az osztályokban viszont ezt kötelezően public-ként kell megvalósítani! Az nincs előírva, hogy a metódust egyszerűen, vagy virtuális módon kell megvalósítani.

```

interface ISzemely {
    string GyerekNeve(int sorszam);
    double Suly {
        get; set;
    }
    int Fizetes {
        get;
    }
    int this[int honap] {
        get;
    }
}

```

The diagram illustrates the components of the `ISzemely` interface. Green arrows point from labels to specific parts of the code:

- Metódus** points to the `string GyerekNeve(int sorszam);` line.
- Property** points to the `double Suly { get; set; }` and `int Fizetes { get; }` blocks.
- Indexer** points to the `int this[int honap] { get; }` block.

- Ismertesse az interface felhasználását, jelentőségét az OOP stílusú programokban!

Az interface-eket arra használjuk, hogy olyan közös tulajdonságokat írjanak le, amelyek egyébként egymással nem rokon objektumosztályok között fedezhető fel. Egyetlen objektumosztálynak csak egyetlen másik objektumosztály lehet az őse, de több interface-t is implementálhat egy időben.

- Ismertesse a 'interface implementálás' fogalmát, szabályait!

Egy osztály implementál egy interface-t, ha

- szerepel az ősei között felsorolva
- az interface-ben leírt összes metódust és property-t a megfelelő szignatúrával megvalósít (tartalmazza a kidolgozását).

- Az interface és az absztrakt osztály között különbségek.

- „abstract class” helyett **interface** kulcsszó
- csak metódus/property szignatúrák
- metódusok/property-k előtt nem kell **public**
- metódusok/property-k előtt nem kell **abstract**
- interfészből öröklés esetén nem kell **override**
- mező tilos
- konstruktor/destruktor tilos
- osztályszintű (static) elem tilos

- Milyen fontos interface-eket ismer a .NET Framework-ben?

- **IComparable**
 - int CompareTo(Object x) metódus
 - példa felhasználás: List.Sort()
- **IComparer**
 - int Compare(Object x, Object y) metódus
 - példa felhasználás: List.Sort(IComparer c)
- **IEnumerator**
 - object Current property
 - bool MoveNext() metódus
- **IEnumerable**
 - IEnumerator GetEnumerator metódus
 - példa felhasználás: foreach
- **IDisposable**
 - void Dispose() metódus
 - Memóriából való takarításhoz.

- **ICollection : IEnumerable**
 - int Count property
 - IEnumerator GetEnumerator metódus
- **ICollection**
 - Add, Insert, Remove, RemoveAt, IndexOf, Contains stb. metódusok

13. Kivételkezelés.

- Milyen egyéb (hagyományos) hibajelzési technikákat ismer, melyiknek mik a korlátai, hátrányai?

A metódus hogyan jelezze a hibát?

- Írjon ki hibaüzenetet.
- Írjon a log-fájlba.
- A függvény térjen vissza speciális értékkel.

Problémák:

- Ha eljárás => át kell írni függvényre
- Ha konstruktor => nem lehet visszatérési értéke!!!

```
StreamReader r = new StreamReader("c:\hello.txt");
// és ha nem létezik a fájl?
```

Példa:

```
double Sqrt(double x) {
    if (x < 0)
        return 0;
    ...
}
```

Hiba esetén a visszatérési érték olyan legyen, ami normális esetben nem fordulhat elő!

```
double Sqrt(double x) {
    if (x < 0)
        return -1; return Double.NaN; // NaN = Not a Number
    ...
}
```

A metódus hívásakor hogyan észleljük a metódus által jelzett hibát?

- if utasításokkal

```
Console.Write("Kérek egy számot:");  
double x = double.Parse(Console.ReadLine());  
double b = Sqrt(x);  
if (Double.IsNaN(b)) Console.WriteLine("nincs gyöke");  
else ...
```

Ha a hívó kód bonyolult, akkor a hiba kezelése túlbonyolítja a kódot.

```
double b = Math.PI * ( Sqrt(x) + Sqrt(y) ) / 2;  
  
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;  
}
```

Nagy projekt esetén nem emlékeznek a programozók a speciális visszatérési értékek jelentésére.

- Ismertesse a kivétel, mint hibajelzési technika jellemzőit!
 - A programunk alapértelmezetten „normál” („optimista”) módban fut.
 - Nincs a kódban if-es hibakezelő kód
 - Ha valamilyen hiba merül fel => a program „hiba” módba lép, melyben:
 - A normál mód utasításai nem kerülnek végrehajtásra.
 - A return-höz hasonlóan visszaugrálunk a metódusok hívási láncán.
 - Ha közben a hibát nem kezeljük le, a program véget ér.
 - A kivétel programozott módon történő előidézésének (kivétel feldobása) technikája, annak szintaktikai szabálya, szemantikai következményei.

„Hiba” állapotba egy **kivétel** (=hibajelző objektum) dobásával tudunk lépni.

- A **throw** utasítást kell használni.
- Az **Exception** osztály egy példányát kell dobni.

```
double Sqrt(double x) {
    if (x < 0)
        throw new Exception();
    ...
}
```

Az Exception osztálynak kétfajta konstruktorát szokás használni:

- Paraméter nélkül.
- String paraméterrel: a paraméter hibaüzenet.


```
double Sqrt(double x) {
    if (x < 0)
        throw new Exception(„Negatív számnak nincs
            négyzetgyöke”);
    ...
}
```

- A kivétel kezelésének módszere, annak szintaktikai és szemantikai szabályai.

A „hiba” mód hívási láncon való visszalépegetését egy **try-catch** utasításpár segítségével állíthatjuk meg.

- **try:** Utasításblokkjában kivétel dobódhat fel és így a program „hiba” módba kerülhet.
- **catch:** Elkapja a try-ban feldobott kivételt és visszaállítja a programot „normál” módba.

```
try {
    ...
    ...
}
catch {
    ...
    ...
}
...
...
```




Az a programszakasz, amelyik kivételt dobhat.

„Hiba” módban lefutó utasítások. A kivétel feldolgozása.

Ezek az utasítások akkor futnak le, ha a catch megszünteti a „hiba” állapotot.

```
double b;
try {
    Console.Write("kérek egy számot:");
    double x = double.Parse(Console.ReadLine());
    b = Math.PI * Sqrt(x) / 2;
}
catch {
    Console.Write("Hiba történt a feldolgozás során");
    b = 0;
}
double c = b*2;
```



B terv: ha kivétel keletkezett, akkor is kell mennitovább.

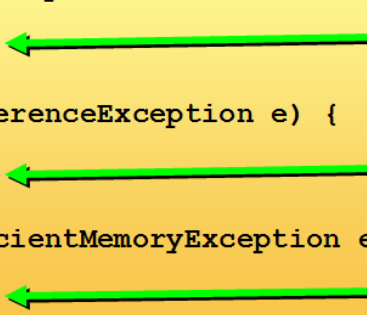
A catch mögött egy paraméteren keresztül elérhetjük a kivétel objektumot.

```
try {
    ...
}
catch (Exception e) {
    Console.WriteLine("A hibaüzenet: {0}", e.Message);
    Console.WriteLine(e.StackTrace);
}
```

StackTrace property: A hívási verem állapotát adja vissza stringként.

Egy try-hoz akár több catch is tartozhat, más-más típusú paraméterrel.

```
try {
    ...
}
catch (FormatException e) {
    ...
}
catch (NullReferenceException e) {
    ...
}
catch (InsufficientMemoryException e) {
    ...
}
```



Ha a feldobott kivétel kompatibilis a FormatException-nel.

Ha a feldobott kivétel kompatibilis a NullReferenceException-nel.

Ha a feldobott kivétel kompatibilis a InsufficientMemoryException-nel.

A catch ágak kiértékelése hasonló a switch kiértékeléséhez:

- Maximum 1 db. futhat le.
- Sorrendben értékelődnek ki.
- Az 1. illeszkedő fog lefutni.

Ezért a catch-ek a speciális kivételtől haladjanak az általános felé!


```

try {
    ...
}
catch (DriveNotFoundException e) {
    Console.WriteLine("Nem találom a meghajtót!");
}
catch (IOException e) {
    Console.WriteLine("Valamilyen I/O hiba!");
}
catch (Exception e) {
    Console.WriteLine("Hiba keletkezett!");
    Console.WriteLine(e.Message);
}

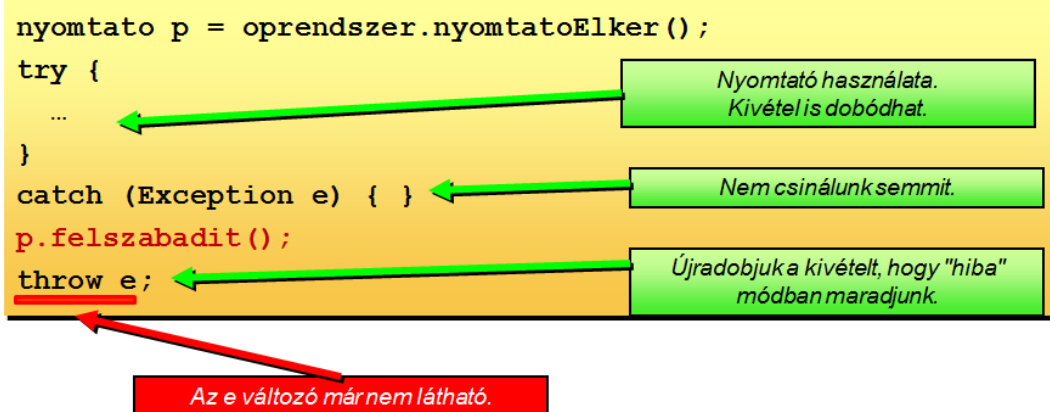
```

Ha a catch paraméterét nem használjuk (csak a típusát), akkor a paraméterváltozó elhagyható.

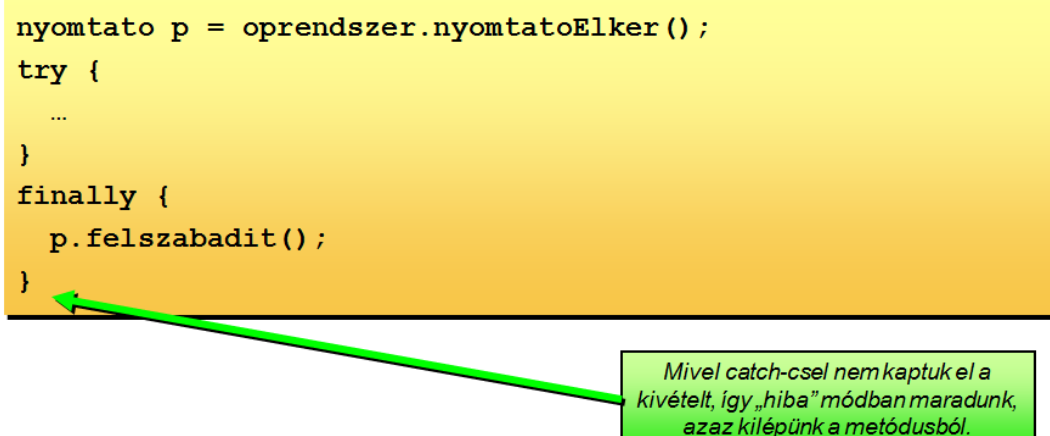
Figyelem: Az alábbi rész elvileg már nem tartozik bele a kérdésbe, de a témakörbe igen, így lehet kérdezni fogja a vizsgán!!!

Saját kivételosztály könnyedén készíthető valamelyik kivételosztályból való örökléssel. Általában az Exception-ből szokás örököltetni, de lehet az Exception bármely gyerekosztályából is.

Kivétel keletkezése esetén is illik felszabadítani a lefoglalt erőforrásokat. Például:



A finally blokk mindig, azaz „normál” módban és „hiba” módban is lefut.



try-catch-finally:

Hármas kombinációt alkotnak ezek az utasítások:

- 1 db. try
- akárhány db. catch
- 0 vagy 1 db. finally

```
try { ... }  
catch (FileNotFoundException) { ... }  
catch (IOException) { ... }  
catch (Exception) { ... }  
finally { ... }
```

A try-catch-finally blokkok egymásba is ágyazhatóak.

Értelme: a hibakezelés során is keletkezhet hiba.

```
try {  
    try { ... }  
    catch { ... }  
    finally { ... }  
}  
catch {  
    try { ... }  
    catch { ... }  
}  
finally { ... }
```

14. Boxing és unboxing. Generikus típusok.

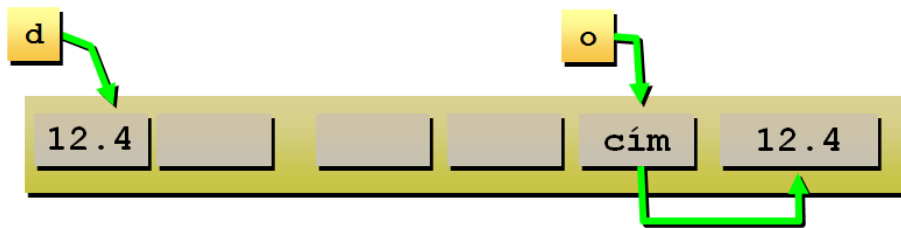
- Ismertesse a boxing művelet értelmét, hol és hogyan használja ezt a rendszer!

Boxing = „bedobozolás”

Az a művelet, mikor egy **értéktípusú** értéket egy **referenciatípusú** változóba helyezünk.

```
double d = 12.4;  
object o = d;
```

- Az értékről másolat készül a memóriában.
- A másolat memóriacíme bekerül a változóba.



Gyakorlati alkalmazása pl. ArrayList-eknél. Az 'ArrayList' osztály '.Add(...)' metódusának paramétere ugyanis 'Object' típusú:

```
class ArrayList
{
    public void Add( Object o )
    {
        ....
    }
}
// használata
ArrayList l = new ArrayList();
int x = 12;
l.Add( x );
```

- Ismertesse az unboxing művelet értelmét, hol és hogyan használja ezt a rendszer!

Unboxing = „kidobozolás”

Az a művelet, mikor egy kinyerjük a **referenciatípusú** változóba helyezett **értéktípusú** értéket.

```
double d = 13.2;
object o = d;
...
double x = (double)o;
```

A gyakorlatban amikor az 'ArrayList'-ből kiveszünk egy benne korábban elhelyezett értéket, akkor pedig ennek fordítottja zajlik le. Az 'ArrayList' indexelője ugyanis 'Object' típusú értéket ad vissza:

```
class ArrayList
{
    public Object this[int index]
    {
        get
        {
            ....
        }
    }
}
```

```
// használata
ArrayList l = new ArrayList();
int x = 12;
l.Add( x );
...
int y = (int)l[0];
```

A boxing és unboxing műveletek automatikusan (implicit módon) hajtódnak végre.

- A generikus típus fogalma, készítésének szabályai.
 - C# 2.0-ban vezették be.
 - Generikus = **Típussal** paraméterezhető osztály/interfész/metódus/stb.

```
class Node<T> {
    T data;
    public T Data {
        get { return data; }
        set { data = value; }
    }

    Node<T> nextNode;
    ...
}
```

```
Node<int> n1 = new Node<int>();
Node<double> n2 = new Node<double>();
```

Akár több (**akárhány**) típusparamétert is fogadhat a generikus.

```
class KeyNode<K, T> {
    K key;
    T data;
    ...
    public T Find(K key) { ... }
}
```

```
KeyNode<int, bool> n1 = new KeyNode<int, bool>();
KeyNode<string, double> n2 = new KeyNode<string, double>();
double x = n2.Find(„dafuq”);
```

Akár **metódus** is paraméterezhető típusokkal.

```
static class Helper {
    public static void Swap<T>(ref T x, ref T y) {
        T temp = x;
        x = y;
        y = temp;
    }
}
```

```
int num1 = 8;
int num2 = 33;
Helper.Swap<int>(ref num1, ref num2);
```

Ha a típusparaméterünknek valamilyen tulajdonságúnak kell lennie:

```
public T Maximum<T>(T a, T b, T c) {
    T x = a;
    if (b > x) x = b;
    if (c > x) x = c;
    return x;
}
```

Operator '>' cannot be applied to operands of type 'T' and 'T'

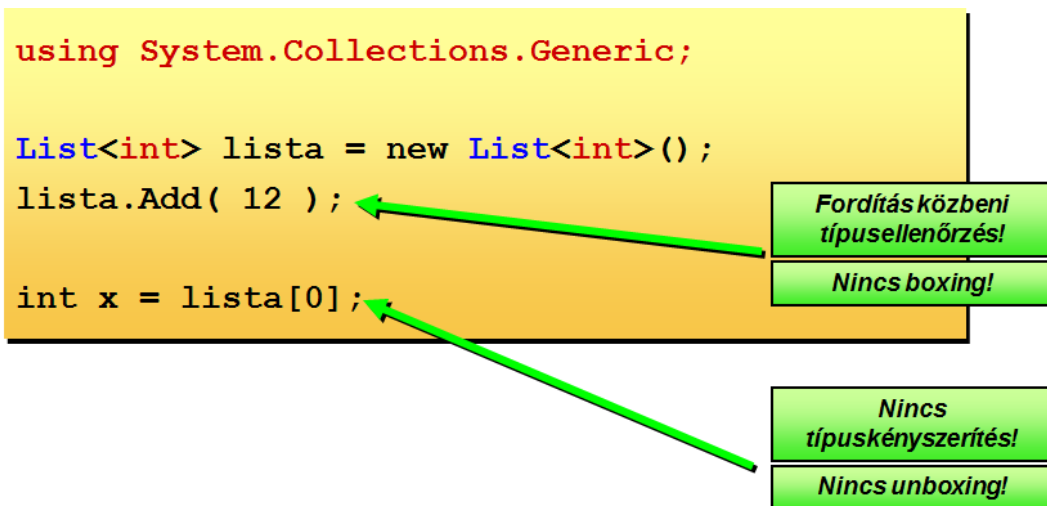
Akkor olyan T típusra van szükségünk, melyre implementált az összehasonlítás. A **where** kulcsszó után lehet a típusparaméterekhez megkötéseket felsorolni.

```
public T Maximum<T>(T a, T b, T c)
    where T: IComparable {
    T x = a;
    if (b.CompareTo(x) > 0) x = b;
    if (c.CompareTo(x) > 0) x = c;
    return x;
}
```

```
class EmployeeList<T>
    where T : Employee, IComparable<T>, ICloneable
{ ... }
```

- A generikus típus felhasználása a programozás során.

A leggyakrabban történő felhasználásra példa a lista (System.Collections.Generic.List) osztály.



- Milyen generikus típusokat ismer a .NET Framework-ből? Hasonlítsa össze ezek használhatóságát a régebbi megoldásokkal!

A 'System.Collections.Generic' névtében a BCL 2.0 változata több, hasonlóan általános, típusbiztos osztályt tartalmaz:

- 'List<T>': lista
- 'Stack<T>': verem
- 'Queue<T>': sor
- 'Dictionary<T,P>': szótár (név-érték páros)

using System.Collections;	using System.Collections.Generic;
ArrayList	List<T>
Stack	Stack<T>
Queue	Queue<T>
Hashtable	Dictionary<K, V>

15. Delegate.

- A delegate fogalma, ennek jelentősége, felhasználhatósága OOP környezetben.

Memóriában kétfajta tartalom:

- adat => erre mutatnak referencia típusú változóink
- kód => erre mutatnak a delegate-ek

Delegate = metódus referencia

- Az eljárás/függvény belépési pontjára mutat.
- Memóriacímen kívül tárolja: a visszatérési típusát és a paraméterezését.

Használata:

1. Delegate típus bevezetése

```
delegate int muvelet(int a, int b);
```

2. Megfelelő szignatúrájú metódus készítése

```
static void int osszead(int x, int y) { return x + y; }
```

3. Delegate típusú változó használata

```
muvelet op = osszead;
```

Metódus neve zárójel nélkül:
REFERENCIA

4. Delegate hívása

```
op(89, -5);
```

Metódus neve zárójellel: **HÍVÁS**

- Callback függvények.

A delegate legfontosabb célja, hogy a 3-rétegű alkalmazásfejlesztési elvet támogassa.



Az alkalmazáslogika **callback** metódusokon keresztül jeleníti meg az adatokat.

Callback függvény: egy megadott típusú, szignatúrájú függvényt, mint paramétert adunk át. Ezen függvényt vissza lehet hívni. A függvény szignatúráját rögzíteni kell, mint típust:

```
delegate void PercentCallback( int );
```

Kulcsszó

Visszatérési típus

A típus neve

A fv paraméterezése

```

delegate void SzazalekCallback(int x);

class ZIP_Osztaly {
    public static void Becsom(string inpFile,
                               SzazalekCallback Kiir) {
        ... // fájl megnyitása
        int szazalek_kesz = 0;
        while (!nincs_file_vege()) {
            ... // egy szakasz becsomagolása
            szazalek_kesz++;
            Kiir(szazalek_kesz);
        }
        ... // fájl bezárása
    }
}

```

- Ismertesse a delegate-ek kezelésével kapcsolatos ismerteket!

A függvény menet közben hajlandó egy 'void'-ot visszaadó, paramétereket nem váró függvényt rendszeresen aktivizálni.

- 'void' típussal tér vissza
- nem vár semmilyen paramétert

A 'delegate' kulcsszóval új **típust** hozhatunk létre. Ezen típusnévvel deklarálhatunk változókat, mezőket, paramétereket. Egy ilyen változó egy, a fenti szignatúrának megfelelő típusú függvény memóriacímét képes tárolni, de tárolhat cím hiányát jelző 'null' értéket is. Egy ilyen változó memóriaigénye **4 byte**.

- Hogy lehet delegate-et kezelni, mint paraméter, mint ilyen típus mező?

Természetesen van arra mód, hogy nem csak 'void'-ot visszaadó, és paraméterket nem váró függvénnel dolgozzunk, hanem ennél bonyolultabb esetekkel is. Ahol paraméterként van jelen a delegate:

delegate void fv_letoltesMutato(int letoltott, int max, int szmperc);

A fenti típusú függvényt értelemszerűen megfelelően felparaméterezve kell majd meghívni:

```

public void File_Letoltese(string url, fv_letoltesMutato userFv)
{
    ...
    DateTime start = DateTime.Now;
    while (...)
    {
        ...
        DateTime akt = DateTime.Now;
        TimeSpan kulonb = akt-start;
        int szmperc = (int)kulonb.TotalMilliseconds;
        if (userFv!=null)
            userFv(letoltott, meret, szmperc);
    }
    ...
}

```


Nem csak delegate típusú változót és paramétert lehet készíteni, hanem ilyen típusú mezőt is. Egyszerűen megoldható, hogy a callback függvényt nem paraméterként kapja meg, hanem előre elhelyezve egy megfelelő mezőben:

```
class FileLetolto
{
    // a mező
    public fv_letoltesMutato userFv;

    public void File_Letoltese(string url )
    {
        ...
        while (...)
        {
            ...
            if (userFv!=null)
                userFv(letoltott, meret, szmperc);
        }
        ...
    }
}
```

A mező lehet értelemszerűen példányszintű, vagy osztályszintű. Érdekes a kódban felkészülni, hogy a mező értéke hátha 'null', ezt a callback aktiválása előtt érdemes egy egyszerű 'if'-el ellenőrizni.

- Hogyan tud több delegate-et kezelni egy időben lista, illetve esemény (event) segítségével?

Delegate lista:

Néha delegate-ek listáját kell kezelni.

Példa probléma: Több helyre is akarunk egyszerre logolni + SMS-ben küldjünk értesítést!

```
class ZIP_Osztaly {
    public List<SzazalekCallback> Kiir;
    ...
    foreach (SzazalekCallback k in Kiir)
        k("hiba: lemez megtelt");
}

...

ZIP_Osztaly zip = new ZIP_Osztaly();
zip.Kiir.Add(logSQL);
zip.Kiir.Add(sendSMS);
```

Event:

Több delegate hívása olyan gyakori feladat, hogy a C# külön konstrukciót biztosít ehhez: event = esemény(kezelő).

- Az event-hez akármennyi metódust hozzá tudunk adni.
- Az event hívásakor az összes metódus meghívódik.

Használata:

1. Delegate típus bevezetése

```
delegate void LogProc(string message);
```

2. Event mező bevezetése az osztályban

```
public event LogProc Log;
```

3. Feliratkozás az eseményre

```
Log += logSQL;  
Log += sendSMS;  
// eseményről való leiratkozás  
Log -= logSQL;
```

4. Esemény kiváltása

```
Log("hiba: lemez megtelt");
```

16. Lambda kifejezés.

- A lambda kifejezés fogalma, jelentősége, felhasználhatósága OOP környezetben.

A lambda kifejezés egy anonim delegate intuitív szintaxissal.

Anonim delegate:

```
delegate(string x) { return x.StartsWith("S"); }
```

Lambda kifejezésként:

```
x => x.StartsWith("S")
```

- x típusát kikövetkezteti a környezetéből!

A => operátor a lambda kifejezések operátora.

- A lambda kifejezések használatának szintaktikai és szemantikai szabályai.

- 1 db. paraméterrel:

```
x => x >= 0 ? x : -x
```

- Több paraméterrel:

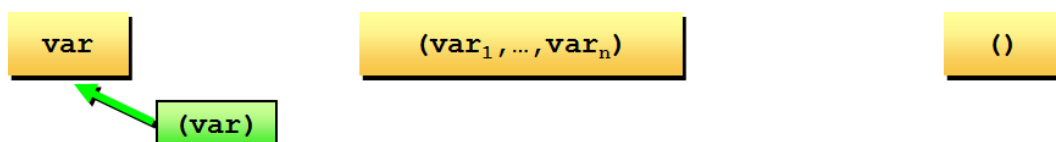
```
(dx, dy) => Math.Sqrt(dx * dx + dy * dy)
```

- Paraméter nélkül:

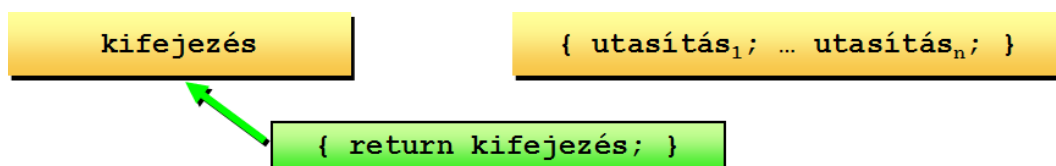
```
() => 42
```

LHS => RHS

- LHS (left-hand side) lehet:



- RHS (right-hand side) lehet:



Figyelem: Az alábbi rész elvileg már nem kell, de megkérdezheti a vizsgán!!!

Lambda kifejezés más nyelvekben:

- Java 8:
 - Ugyanaz, mint C#-ban, csak -> operátorral.

```
p -> p.getGender() == Person.Sex.MALE && p.getAge() >= 18
```

- Python:
 - Más szintaxis, **lambda** kulcsszó használata.

```
lambda x, y: x + y
```

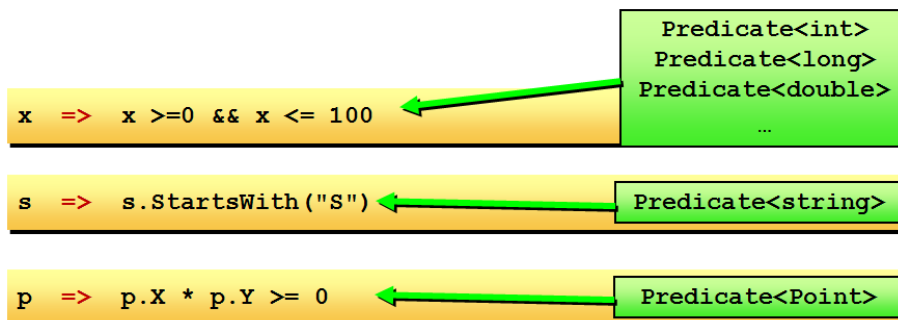
- C++: A C++11 óta.
- PHP, Javascript: Anonim függvények.

- Generikus delegate-ek a .NET Framework-ben, a lambda kifejezések ezekkel való kapcsolata.

Predicate<T>

Paraméter: T típusú.

Visszatérési érték: **bool** típusú.



Action<T>

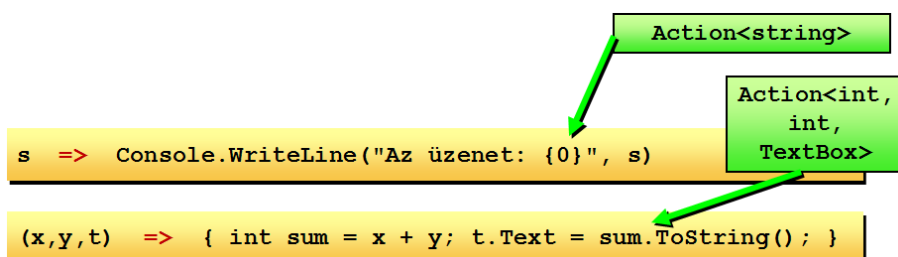
Action<T1,T2>

...

Action<T1,...,T16>

Paraméterek: T1,...,T16 típusúak.

Visszatérési érték: nincs (**void**).



Func<TR>

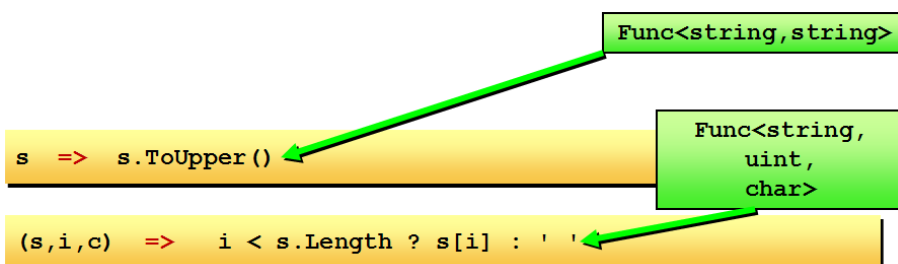
Func<T,TR>

...

Func<T1,...,T16,TR>

Paraméterek: T1,...,T16 típusúak.

Visszatérési érték: TR típusú.



- LINQ bővítő metódusok fogalma, jelentősége, a lambda kifejezések ezekkel való kapcsolata.

"Language Integrated Queries" = nyelvbe integrált lekérdezések.

A System.Linq névtér "kibővíti" a IEnumerable<T> interfészt ezt az interfészt örökli a tömb, a List<T>, stb.

"Bővítő metódusok" (extension methods) jelennek meg.

Szűrés	Where
Projekció	Select, SelectMany
Rendezés	OrderBy, ThenBy
Csoportosítás	<i>GroupBy</i>
Joinok	<i>Join, GroupJoin</i>
Quantifiers	Any, All
Particionálás	Take, Skip, TakeWhile, SkipWhile
Halmazműveletek	<i>Distinct, Union, Intersect, Except</i>
Elemek	First, Last, Single, ElementAt
Aggregáció	Count, Sum, Min, Max, Average
Konverzió	ToArray, ToList, ToDictionary
Kasztolás	OfType<T>, Cast<T>

- Össze lehet fűzni a hívásaikat

```
list.Where (...).OrderBy (...).Take (...).Select (...)
```

- Tipikusan késői kiértékeléssel dolgoznak:

- csak akkor adják vissza a következő elemet, amikor kérem
- erre utal az IEnumerable interfész
- ha azonnali kiértékelés kell: tömbbé/listává/stb. szükséges az eredményt konvertálni

```
list.Where (...).OrderBy (...).Take (...).Select (...).ToList ()
```

Sok sikert a vizsgához! ☺

*Készítette: Köntös Balázs
Programtervező informatikus Bsc*