



Objektum Orientált Programozás

2. Öröklődés

Répasné Babucs Hajnalka

Répás Csaba



Öröklődés



- Az osztályokból létrehozhatunk leszármazott osztályokat, amelyek öröklík az ősosztály összes tagját
 - Az örökölt metódusok a leszármazottakban módosíthatók
 - A leszármazottak új tagokkal (mezőkkel, metódusokkal) bővíthetik az ősosztálytól örökölt tagok halmazát
- Minden leszármazott osztálynak csak egy őse lehet
- Minden osztály közös őse a **System.Object** osztály





Származtatás



- Származtatott osztályok deklarációjánál „:” karakterrel elválasztva meg kell adni az őssztály nevét is



- Ez nem kötelező akkor, ha az őssztály a System.Object

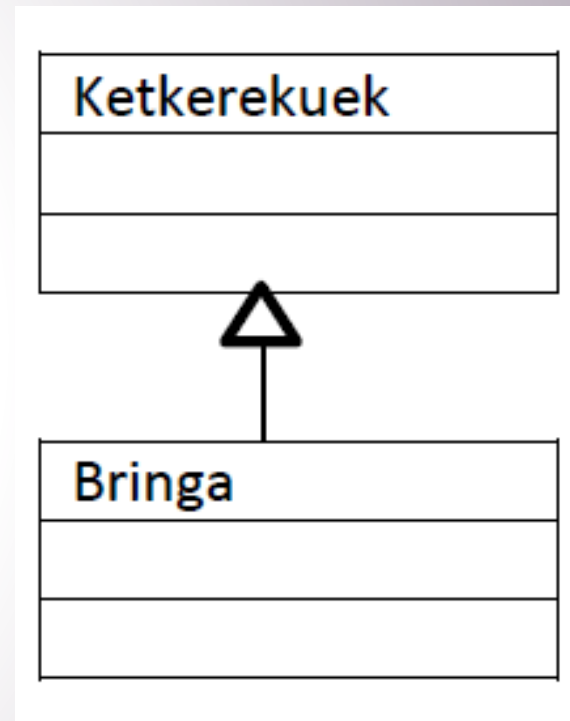
- Csak egyszeres öröklődés van



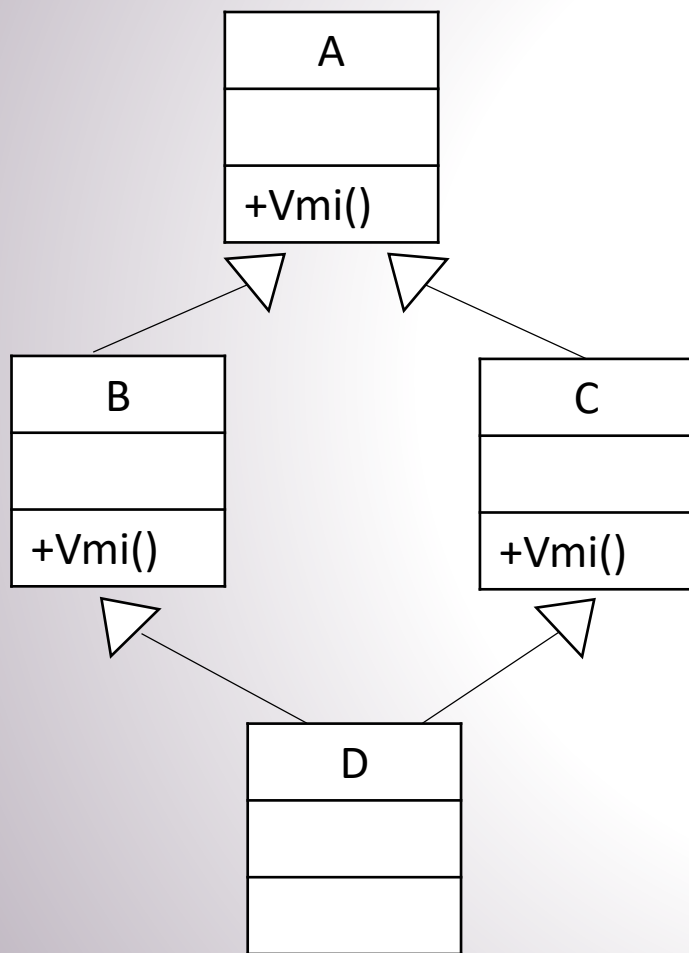
- Ezt követően csak az új mezőket/metódusokat kell felsorolni



```
class Bringa : Ketkerekuek
```



Gyémánt probléma



- A osztályban definiálunk egy Vmi() metódust
- B és C osztály A-ból származik, mindkettőben különböző módon felülírjuk Vmi() metódus működését
- D osztály B-ből és C-ből származik, a Vmi() metódust nem írjuk felül. Melyik verziót örökli D, a B osztályét vagy a C osztályét?



Láthatósági szintek



Szint	Hozzáférési lehetőség
public	Korlátlan
protected	Az adott osztályban és a leszármazottaiban
internal	Az adott projektben
protected internal	Az adott projektben és az osztály leszármazottaiban
private	Csak az adott osztályban
private protected	C# 7.2 óta elérhető A projekten belül protected-ként viselkednek, azon kívül private-ként



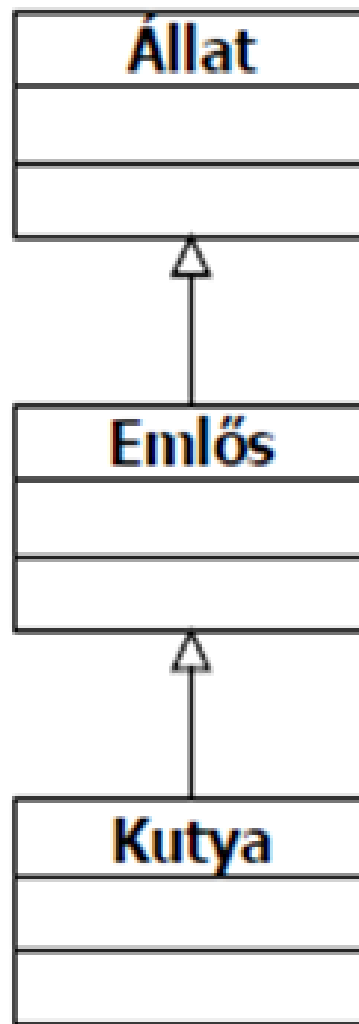
Öröklődés példa



```
class Állat
{
    int lábszám;
    public Állat(){...}
    public void Fut(){...}
}
```

```
class Emlős: Állat
{
    public bool KicsinyétEteti(){...}
}
```

```
class Kutya: Emlős
{
    public void FarkátCsóválja(){...}
    public void Ugat(){...}
}
```



- Az Állat osztályban tároljuk privát mezőként a lábszámot, van egy paraméter nélküli konstruktora, és van egy Fut() metódusa.
- Az Emlős az Állat osztályból származik, saját metódusa a KicsinyétEteti(), és örökli az Állat Fut() metódusát. A lábszám mezőt szintén örökli, de nem elérhető az osztályon belül.
- A Kutya osztály az Emlősből származik, így van KicsinyétEteti() és Fut() metódusa, valamint saját metódusa a FarkátCsóválja() és az Ugat().





sealed kulcsszó



- Ha egy osztályt a **sealed** módosítóval látunk el, akkor az az osztály további osztályok őse nem lehet



```
sealed class Pelda
```

```
{  
}
```

```
class Teszt: Pelda
```

```
{  
}
```

Fordítási hiba, mert a Pelda osztály zárt!
'Teszt': cannot derive from sealed type 'Pelda'

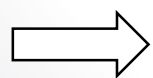


Típuskompatibilitás

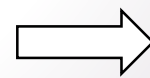
- A gyerekosztály bármely példánya képes helyettesíteni az őszosztály bármely példányát.
- Általánosságban: a gyerekosztály **típuskompatibilis** az őszosztályával!
- A típuskompatibilitás tranzitív, minden osztály kompatibilis nemcsak a közvetlen ősével, hanem annak ősével is, felfelé, egészen a kezdetig. Az osztályok kompatibilisek minden direkt és indirekt ősökkel!
- Egyszerűsítés példa:

```
class Teglalap : Negyzet
```

```
Teglalap t = new Teglalap();  
Negyzet n = new Negyzet();  
n = t;
```



```
Teglalap t = new Teglalap();  
Negyzet n;  
n = t;
```



```
Teglalap t = new Teglalap();  
Negyzet n = t;
```



```
Negyzet n = new Teglalap();
```




Mezők öröklődésének problémái



```
class Elso
{
    private int a = 1;
    protected int b = 1;
    public int c = 1;
}
class Masodik: Elso
{
    private double a = 2.0;
    protected double b = 2.0;
    public double c = 2.0;
    public double d = 2.0;
}
```

⚠ 'Masodik.b' hides inherited member 'Elso.b'.

Use the new keyword if hiding was intended.

⚠ 'Masodik.c' hides inherited member 'Elso.c'.

Use the new keyword if hiding was intended.

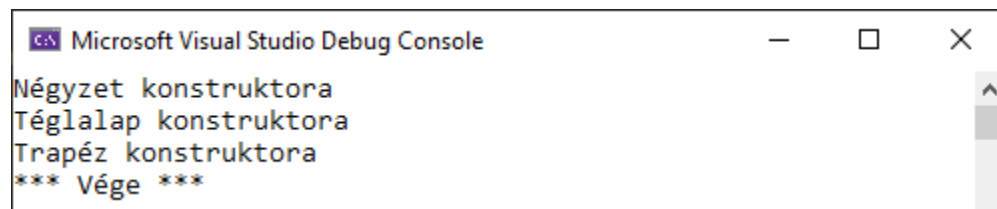
- A Masodik osztály örökli az Elso osztály 3 mezőjét, tehát összesen 7 mezője van, viszont csak 4 különböző néven osztoznak. (Ez kerülendő!)
- A ,d' mezőnév egyedi, nincs jelen az Elso osztályban, ez rendben van.
- Az ,a' mezőre nincs warning, mivel privát, vagyis a hatásköre nem terjed ki a Masodik osztályra, így az ott deklarált másik ,a' mezővel nem fedik át egymás hatáskörét.
- Az Elso osztályban lévő ,b' és ,c' mezők hatásköre kiterjed a Masodik osztályra is, ezért a Masodik osztályban a mezők direkt módon is elérhetőek lennének. => Warning
- Ha ragaszkodunk az elnevezéshez, használjuk a „**new**” kulcsszót.
- Az Elso osztály ,b' és ,c' mezői „**base**” kulcsszóval vagy „**as**” típusmódosító operátorral lesznek elérhetőek.



Konstruktor hívási lánc

```
internal class Negyzet
{
    public Negyzet() {
        Console.WriteLine("Négyzet konstruktora");
    }
}
internal class Teglalap : Negyzet
{
    public Teglalap() {
        Console.WriteLine("Téglalap konstruktora");
    }
}
internal class Trapez : Teglalap
{
    public Trapez() {
        Console.WriteLine("Trapéz konstruktora");
    }
}
```

```
Trapez t = new Trapez();
Console.WriteLine("*** Vége ***");
```



Microsoft Visual Studio Debug Console

```
Négyzet konstruktora
Téglalap konstruktora
Trapéz konstruktora
*** Vége ***
```

- A példány elkészítésében az őszosztályok konstruktorai is részt vesznek.
- Felülről lefelé, a gyökér elem konstruktorától kezdve a levél elemig hívódnak egymás után a konstruktorok.



Konstruktorok és öröklődés



- **A konstruktorok nem öröklődnek!**
- Ha az őssztályban van paraméter nélküli konstruktor, az a leszármazott osztály konstruktorában automatikusan meghívódik
 - Ez egyaránt igaz az automatikusan generált alapértelmezett konstruktorra és a saját, paraméter nélküli konstruktorra
- Ha az őssztályban nincs paraméter nélküli konstruktor, az őssztály konstruktorát meg kell hívni a leszármazott osztály konstruktorából
 - Erre a célra a **base** kulcsszót használjuk



Konstruktorok és öröklődés – van paraméter nélküli konstruktor

```
class A
```

```
{  
}
```

Itt automatikusan létrejön egy paraméter nélküli konstruktor

```
class B: A
```

```
{  
    public B(int x) {...}  
}
```

```
class A
```

```
{  
    public A() {...}  
}
```

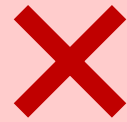
Kézzel megadott paraméter nélküli konstruktor

```
class B: A
```

```
{  
    public B(int x) {...}  
}
```

Konstruktor és öröklődés – nincs paraméter nélküli konstruktor

```
class A
{
    public A(int x) {...}
}
```



Hibás program

```
class B: A
{
    public B(int x) {...}
}
```

```
class A
{
    public A(int x) {...}
}
```



```
class B: A
{
    public B(int x): base(x) {...}
}
```

Hivatkozás az ősz konstruktorára

A metódusok öröklődésének problémái

- Probléma akkor lehet, ha ugyanolyan nevű metódust szeretnénk a gyerekosztályban is készíteni, mint amelyet örököltünk az őssosztályból.
 - Az őssosztályban a metódus private: Nincs probléma, private metódust a gyerekosztály nem látja.
 - Az őssosztályban a metódus nem private, a gyerekosztályban készített metódusban a név megegyezik, de más a paraméterezése: Ha más a paraméterezés, akkor hiába egyforma a 2 metódus neve, nincs átfedés a hatáskörökben.
 - Az őssosztályban a metódus nem private, a 2 osztályban ugyanaz a név és a paraméterezés: Problémás eset.





Nemvirtuális metódusok



- Alapértelmezetten minden metódus nemvirtuális
- A nemvirtuális metódusok változatlanul örökölhetők vagy a leszármazottakban elrejtethetők
- Elrejtés: a leszármazott osztályban azonos néven létrehozunk egy másik metódust
 - A leszármazott osztályban célszerű az újonnan bevezetett metódust a *new* kulcsszóval megjelölni
 - Az ősoosztály azonos nevű metódusa a leszármazottban is elérhető *base* kulcsszó segítségével
- Nem igényelnek semmilyen külön szintaktikai megjelölést





Nemvirtuális metódusok példa 1.



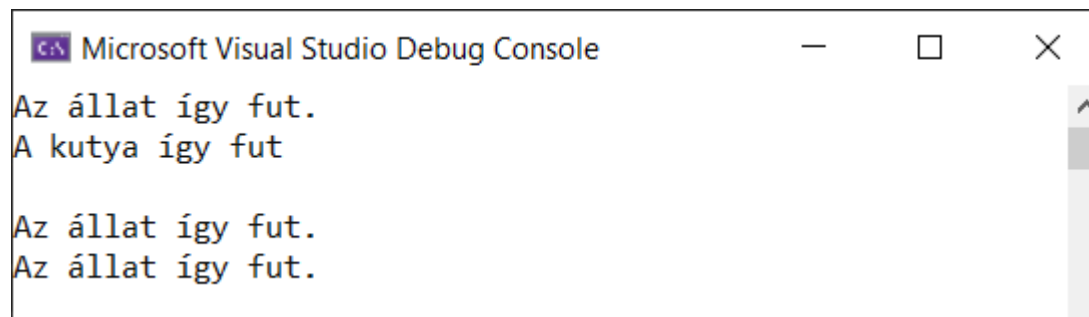
```
internal class Állat
{
    public string Fut()
        => "Az állat így fut.";
}
```

```
internal class Kutya: Állat
{
    public new string Fut()
        => "A kutya így fut";
}
```

```
internal class Macska : Állat
{
    public new string Fut()
        => "A macska így fut";
}
```

```
Állat egyikÁllat = new Állat();
Kutya másikÁllat = new Kutya();
Console.WriteLine(együkÁllat.Fut()); // Az Állat osztály Fut() metódusa hívódik meg
Console.WriteLine(másikÁllat.Fut()); // A Kutya osztály Fut() metódusa hívódik meg
Console.WriteLine();
```

```
Állat háziKedvenc = new Macska();
Console.WriteLine(háziKedvenc.Fut()); // Az Állat osztály Fut() metódusa hívódik meg
háziKedvenc = new Kutya();
Console.WriteLine(háziKedvenc.Fut()); // Az Állat osztály Fut() metódusa hívódik meg
```



```
Microsoft Visual Studio Debug Console
Az állat így fut.
A kutya így fut
Az állat így fut.
Az állat így fut.
```

Az objektum típusa (bal oldal, statikus típus) határozza meg, hogy melyik metódus hívódik meg.





Nemvirtuális metódusok példa 2.

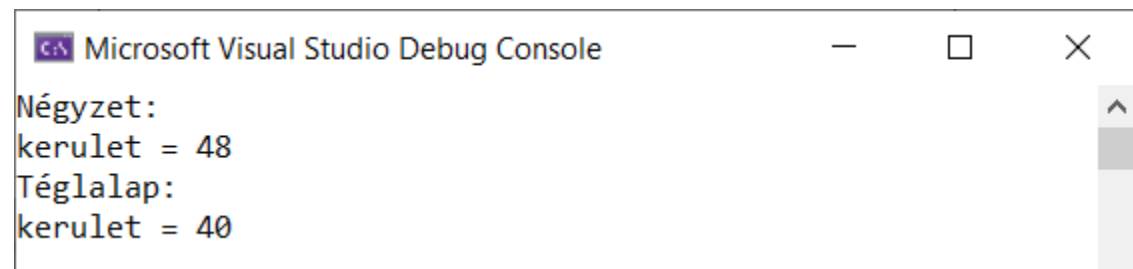


```
internal class Negyzet
{
    public double a_oldal;
    public double Kerulet()
        => 4 * a_oldal;
    public void KiirKerulet()
    {
        Console.WriteLine($"kerulet={Kerulet()}");
    }
}
```

```
internal class Teglalap: Negyzet
{
    public double b_oldal;
    public new double Kerulet()
        => 2 * (a_oldal + b_oldal);
}
```

```
Negyzet n = new();
n.a_oldal = 12;
Console.WriteLine("Négyzet:");
n.KiirKerulet();
```

```
Teglalap t = new();
t.a_oldal = 10;
t.b_oldal = 20;
Console.WriteLine("Téglalap:");
t.KiirKerulet();
```



```
Microsoft Visual Studio Debug Console
Négyzet:
kerulet = 48
Téglalap:
kerulet = 40
```

A kód szintaktikailag hibátlan, mégsem az történik, amit szeretnénk. A KiirKerulet() metódus az ős osztály Kerulet() metódusát hívja.



Korai kötés (early binding)



- 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.
- Ez biztosítja a program maximális futási sebességét.





Késői kötés (late binding)



- Van lehetőség arra, hogy a fordítónak jelezzük, ne a statikus típus alapján hozzon döntést.
- A rugalmasság következményeként a program futási sebessége lassul, hiszen a fordító elhalasztja a döntést későbbi időpontra, a futás idejére.
- A futás során, amikor már a dinamikus típus is ismert, akkor fog a megfelelő metódus kiválasztásra kerülni.



Virtuális metódusok

- A virtuális metódusok a leszármazottakban módosíthatók („felülbírálnak”) vagy elrejtethetők
 - Futási időben, az adott változó típusától dől el, hogy a programkód melyik osztályhoz tartozó virtuális metódust hívja
 - Egy adott virtuális metódusból mindig a változó által ténylegesen hivatkozott osztályhoz legközelebb álló változatot hívja meg a program; a legtöbb esetben ez az osztály saját metódusváltozata
- A virtuális metódusokat az őssztályban a **virtual** kulcsszóval kell megjelölni
- A leszármazottakban felülbírált metódusokat az **override** kulcsszóval kell megjelölni
- Elrejtés: mint a nemvirtuális metódusok esetén





Virtuális metódusok példa

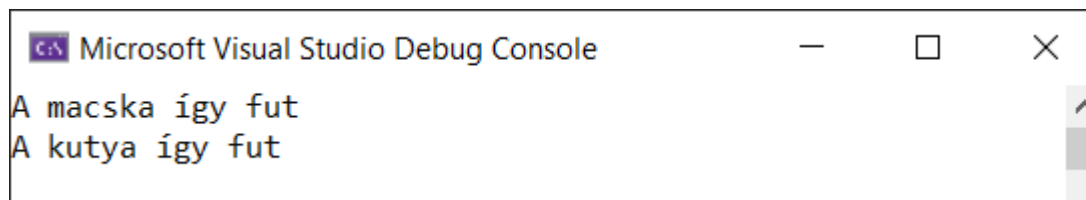


```
internal class Állat
{
    public virtual string Fut()
        => "Az állat így fut.";
}
```

```
internal class Kutya: Állat
{
    public override string Fut()
        => "A kutya így fut";
}
```

```
internal class Macska: Állat
{
    public override string Fut()
        => "A macska így fut";
}
```

```
Állat háziKedvenc = new Macska();
Console.WriteLine(háziKedvenc.Fut());
háziKedvenc = new Kutya();
Console.WriteLine(háziKedvenc.Fut());
```



Futásidőben dől el, hogy melyik osztály metódusa hívódik meg.
A konstruktor hívása (jobb oldal, dinamikus típus) dönti el, hogy melyik osztály metódusát kell meghívni.





Az override egyéb szabályai



- Private védelmi szintű metódusra, property-re nem alkalmazható a virtual jelzés. („*virtual or abstract members cannot be private*”)
- Nem minősül felüldefiniálásnak, ha a gyerekosztályban a metódus felüldefiniálásakor más paraméterezést használunk. („*no suitable method found to override*”)
- Mezőkre nem működik a virtual + override. („*the modifier 'virtual' is not valid for this item*”)
- Osztálysztű metódusok és property-k esetén nem működik a virtual és az override. A késői kötéshez dinamikus típus szükséges, ami példányt feltételez. („*static member cannot be marked as override, virtual or abstract*”)





Típuskényszerítés



- Típuskényszerítésnél (**casting**) egy adott típusú objektumot úgy kezelünk, mintha egy másik típusba tartozna
- Implicit típuskényszerítés: a típusok átalakítása automatikus
 - Példa: egész számok átalakítása valós számmá
- Explicit típuskényszerítés: átalakítás a programozó kérésére
 - Módja: az átalakítandó típus elé, „**()**” karakterek közé kiírjuk a kívánt céltípust





Típuskényszerítés példa

```
Állat Cirmos = new Macska();  
Állat amőba = new Állat();  
Macska Lukrécia;  
Kutya Bodri;
```

Implicit típusátalakítás

(„Állat” helyén mindig szerepelhet „Macska”, mert az „Állat” az őssztálya)

Explicit típusátalakítás a programozó kérésére

(helyes, mert erről az „Állat”-ról biztosan tudjuk, hogy „Macska”, így átalakítható)

```
Lukrécia = (Macska) Cirmos;
```

```
Bodri = (Kutya) Lukrécia;
```

Fordítási hiba: a „Macska” típus nem alakítható át a „Kutya” típusra

```
Lukrécia = (Macska) amőba;
```

Futási idejű hiba: „Macska” helyén nem szerepelhet „Állat”

Az is és as operátorok

- Az **is** operátor segítségével ellenőrizhető, hogy egy objektum egy adott osztályhoz vagy leszármazottjához tartozik-e (típuskompatibilitás)
 - Ez az ellenőrző kifejezés logikai típusú értéket ad vissza
 - Dinamikus típus ellenőrzés
- Az **as** operátor segítségével explicit típus átalakítást hajthatunk végre futási idejű hiba veszélye nélkül
 - Ha az átalakítás nem sikerül, a kifejezés értéke **null** lesz

```
if (p is Teglalap) k = (Teglalap) (p.Kerulet() );
```

```
if (p is Teglalap) k = ((Teglalap) p).Kerulet();
```

```
if (p is Teglalap) k = (p as Teglalap).Kerulet();
```

Szintaktikai hiba: az operátorok prioritása alapján a legerősebb a pont operátor, majd a zárójelpár, végül a típuskényszerítés. Mivel a Kerulet() függvény double értékkel tér vissza, azt nem lehet Teglalap típusra kényszeríteni.



Az is és as operátorok használata

```
class Állatfarm
```

```
{
```

```
    Állat Cirmos = new Macska();
```

```
    Állat amőba = new Állat();
```

```
    Macska Lukrécia;
```

```
    Kutya Bodri;
```

```
    Lukrécia = Cirmos as Macska;
```

```
    if (amőba is Kutya)
```

```
        Bodri = amőba as Kutya;
```

```
    Lukrécia = amőba as Macska;
```

```
}
```

A típusátalakítás sikerülni fog
(Cirmos értéke „Macska” típusú)

A típusátalakítás nem kerül sor, mert amőba értéke nem
„Kutya” típusú, így már a feltétel sem teljesül

A típusátalakítás nem fog sikerülni
(„Macska” helyén nem szerepelhet „Állat”)





Absztrakt osztályok és metódusok



- Az absztrakt metódusok nem tartalmaznak megvalósítást
- Egy osztály akkor absztrakt, ha tartalmaz legalább egy absztrakt metódust
- Az absztrakt osztályok nem példányosíthatók
- Absztrakt metódusaikat leszármazottaik kötelesek felülbírálni, azaz megvalósítást készíteni hozzájuk
- Az absztrakt metódusok mindig virtuálisak (ezt nem kell külön jelölnünk)
- Az absztrakt osztályok garantálják, hogy leszármazottaik tartalmazni fognak bizonyos funkciókat
- Az absztrakt metódusokat és osztályok az **abstract** kulcsszóval kell megjelölnünk





Absztrakt osztályok példa

```
abstract class Alakzat
{
    public abstract void Kirajzol();
}
```

```
class Ellipszis: Alakzat
```

```
{
    public override void Kirajzol()
    {
        //Kirajzol() metódus az Ellipszis osztály megvalósításában
    }
}
```

```
class Kör: Ellipszis
```

```
{
    public override void Kirajzol()
    {
        //Kirajzol() metódus a Kör osztály megvalósításában
    }
}
```

Ebből az osztályból példányt nem hozhatunk létre, leszármazottai viszont biztosan tartalmazznak egy megvalósított Kirajzol() nevű metódust.





Többalakúság (polimorfizmus)



➤ Metóduspolimorfizmus:

- Egy leszármazott osztály egy örökölt metódust újra megvalósíthat

➤ Objektumpolimorfizmus:

- Minden egyes objektum szerepelhet minden olyan szituációban, ahol az őosztály objektuma szerepelhet, (nem csak a saját osztálya példányaként használható)
- A csak utódokban létező publikus jellemzőket (pl. metódusokat) típuskényszerítéssel (cast-olással) érhetjük el.





Interfész



➤ Meghatározza egy osztály felületét

➤ Egy „szerződés”, ami kikényszeríti, hogy az interfészt megvalósító osztályok rendelkezzenek bizonyos műveletekkel

➤ Amennyiben egy osztály őseként megad egy interfészt, azzal azt a "kötelezettséget" vállalja magára, hogy az interfészben megjelölt metódusok és property-k mindegyikét kidolgozza. Amennyiben egyről is „elfeledkezne", a fordítóprogram figyelmezteti az osztályt (szintaktikai hiba), és megtagadja annak lefordítását!

➤ Amennyiben egy osztály őseként választ egy interfészt, és megvalósítja az abban deklarált összes metódust, property-t, indexelőt, akkor azt mondjuk, hogy az osztály implementálta az adott interfészt.

➤ Interfész létrehozása az **interface** kulcsszóval lehetséges

➤ Interfész típus nem példányosítható





Interfész tartalma



➤ Az interfész tartalma lehet:



- Metódus fejrész: visszatérési érték típusa, metódusnév, formális paraméterlista
- Property: a property típusa, neve, és a get és set részek jelölése
- Indexer: hasonlóan a közönséges property-hez



➤ Az interfész nem tartalmazhat:



- Metódus törzset: csak a metódusok fejrészét kell leírni (C# 8.0 már engedi)
- A metódusok nem jelölhetők meg védelmi szintekkel (public, protected, private)
- A metódusok nem jelölhetőek meg abstract sem virtual sem override jelzésekkel
- A property-k hasonlóan nem jelölhetőek védelmi szintekkel, sem egyéb módosítókkal
- Nem tartalmazhat mező deklarációt
- Nem tartalmazhat sem konstruktort, sem destruktort



Interfészek öröklődése



- Az interfészeknek is lehet ősük, de egy interfésznek azonban csak másik interfész lehet az őse, osztály nem.
 - Az interfész típusoknak nem őse az Object, emiatt nem kompatibilisek vele.
- Az interfészben deklarált metódusok, indexelők, property-k az implementáló osztályban kötelezően public védelmi szinttel kell rendelkezzenek
- Az interfész típusnak minősül, és ugyanazon típuskompatibilitási szabály vonatkozik rá, mint a teljes értékű osztályok esetén:
 - Az interfészt implementáló osztályok (az interfész gyerekosztályai) típuskompatibilisek a szóban forgó interfésszel.
 - Interfészek esetére is működnek az **is** és **as** operátorok a megfelelő módokon.





Interfész példa



```
interface IEldönthető
```

```
{
```

```
    bool Eldönt();
```

```
}
```

```
class SajátOsztály : ŐsOsztály, IEldönthető
```

```
{
```

```
    public bool Eldönt()
```

```
    {
```

```
        return false;
```

```
    }
```

```
}
```

Az osztályban kötelezően kell lennie egy *publikus* Eldönt() eljárásnak (ha nincs: a fordító reklamál)





Interfész megvalósítása absztrakt osztályokban



➤ A metódust / tulajdonságot nem szükséges implementálni, azonban kötelezően absztraktként kell megjelölni



➤ Az absztrakt osztály leszármazottainak implementálniuk kell a metódusokat és tulajdonságokat



```
interface IEladható
```

```
{  
    bool Ár { get; set; }  
    void Elad();  
}
```

```
abstract class Termék: IEladható
```

```
{  
    public abstract bool Ár { get; set; }  
    public abstract void Elad();  
}
```



Interfész és absztrakt osztályok összehasonlítása

Interfész

- Követelményrendszernek tekinthető. Csak azt írják elő, hogy milyen metódusokkal kell a megvalósítónak rendelkeznie, de nincs implementáció
- Az interfészek az osztályoktól független hierarchiát alkotnak
- Egy interfész több interfész leszármazottja is lehet, és egy osztály megvalósíthat több interfészt is



Absztrakt osztály

- Egy részben elkészült osztálynak tekinthetők, ahol néhány metódust még nem tudtunk megvalósítani. De lehetnek mezőik, egyéb metódusaik, stb.
- A hagyományos osztály öröklési hierarchia részei
- Nincs többszörös öröklésre lehetőség