# C# keletkezése, története

A C# nyelvet a Microsoft fejlesztette ki, és először 2000-ben mutatták be. A nyelv kidolgozása Anders Hejlsberg nevéhez köthető, aki korábban a Turbo Pascal és a Delphi nyelvek fejlesztésében is részt vett. A C# a .NET keretrendszer része, és eredetileg a Java nyelvre adott válaszként született, bár mára önálló, erősen fejlett nyelvvé vált.

# C# és Visual FoxPro

A Visual FoxPro (VFP) adatbázis-központú programozási nyelv és fejlesztői környezet, amelyet főleg üzleti alkalmazások fejlesztésére használtak. A Visual FoxPro és a C# nagyon eltérő paradigmákon alapul.

| **Tulajdonság** | **C#** | **Visual FoxPro** |
| --- | --- | --- |
| **Paradigma** | Objektumorientált (OOP) és típusbiztos nyelv | Objektumorientált, de dinamikus és laza típuskezelés |
| **Használat** | Modern alkalmazások fejlesztése (*web, mobil, asztali, játékok, cloud, AI, stb.*) | Adatbázis-központú alkalmazások fejlesztése |
| **Típusosság** | Erősen típusos (*statikus*) | Gyengén típusos (*dinamikus adattípusokkal*) |

## Objektum-orientált paradigmára való áttérés

| **Tulajdonság** | **C#** | **Visual FoxPro** |
| --- | --- | --- |
| **OOP támogatás** | Teljes OOP (*osztályok, öröklődés, polimorfizmus, interfészek, absztrakt osztályok*) | OOP-t támogat, de korlátozott |
| **Metódusok túlterhelése** | Igen | Nem |
| **Interfészek és absztrakt osztályok** | Igen | Nem |

## Eltérések adatbázis-kezelésben

A C#-ban nincs beépített adatbázis-kezelés, de a .NET keretrendszer széleskörű támogatást nyújt különböző adatbázisokhoz (*pl. SQL Server, MySQL, SQLite*) az **ADO.NET**, **Entity Framework** vagy **Dapper** segítségével.

| **Tulajdonság** | **C#** | **Visual FoxPro** |
| --- | --- | --- |
| **Adatbázis-motor** | Külső adatbázisokkal (*SQL Server, MySQL, PostgreSQL, SQLite, stb.*) működik | Beépített adatbázis (DBF fájlok), SQL támogatás korlátozott |
| **Kapcsolat adatbázishoz** | ADO.NET, Entity Framework, Dapper | Beépített SQL Pass-Through, Cursor, DBF táblák |
| **Többfelhasználós támogatás** | SQL szerverekkel skálázható | Többfelhasználós, de fájlalapú DBF szerkezet miatt sérülékeny |

## ORM (*Object-Relational Mapping*) használata

Az ORM automatikusan leképezi az adatbázis tábláit **C# osztályokra**, így nem kell SQL lekérdezéseket írnod minden egyes adatkezelési művelethez.

**Két fő ORM lehetőség C#-ban:**

| **ORM** | **Leírás** |
| --- | --- |
| **Entity Framework (EF) Core** | A Microsoft hivatalos ORM-je, erősen integrált a .NET ökoszisztémába. |
| **Dapper** | “Micro ORM”, amely kevesebb absztrakciót tartalmaz és gyorsabb, mint az EF Core. |

**ORM előnyei**

* **Kevesebb kód** → Automatikusan kezeli az adatbázis műveleteket (DbContext, DbSet).
* **Egyszerűbb karbantartás** → Nem kell kézzel írni minden SQL lekérdezést.
* **Típusbiztos** → Az adatok automatikusan leképeződnek C# osztályokra.
* **Automatikus migrációk** → Az adatbázis szerkezete programból kezelhető.
* **Kapcsolatok támogatása** → Pl. One-to-Many, Many-to-Many kapcsolatok kezelése könnyebb.

**ORM hátrányai**

* **Lassabb lehet nagy lekérdezések esetén**, mert túl sok absztrakciót tartalmaz.
* **Nem mindig hatékony SQL-t generál**, ami teljesítményveszteséget okozhat.
* **Tanulási görbe** → Az ORM-ek működését és az optimalizálásukat meg kell tanulni.

**Sebesség-összehasonlítás ORM és ADO.NET között**

| **Művelet** | **Entity Framework Core** | **Dapper** | **ADO.NET** |
| --- | --- | --- | --- |
| **Egyszerű SELECT (1 rekord)** | 🔴 Lassúbb | 🟡 Közepesen gyors | 🟢 Leggyorsabb |
| **Több rekord lekérdezése (10,000+ rekord)** | 🔴 Lassabb | 🟢 Gyors | 🟢 Gyors |
| **Adat beszúrása (INSERT)** | 🟡 Közepes | 🟢 Gyors | 🟢 Gyors |
| **Összetett SQL lekérdezés (JOIN, GROUP BY, stb.)** | 🔴 Lassabb | 🟡 Közepes | 🟢 Gyors |
| **Tanulási görbe** | 🟢 Könnyű | 🟡 Közepes | 🔴 Nehéz |

**Mikor érdemes ORM-et használni?**

* Ha **gyors fejlesztést** szeretnél, és nem akarsz SQL-lekérdezésekkel foglalkozni.
* Ha **kis vagy közepes méretű adatbázist** kezelsz.
* Ha **több fejlesztő dolgozik a projekten**, mert az ORM strukturáltabb kódot biztosít.

## Felhasználói felület (UI)

| **Tulajdonság** | **C#** | **Visual FoxPro** |
| --- | --- | --- |
| **UI keretrendszer** | WinForms, WPF, Blazor, MAUI, ASP.NET, Unity | Beépített UI komponensek |
| **Webes támogatás** | Igen (ASP.NET, Blazor) | Nem (csak asztali alkalmazások) |
| **Korszerű UI támogatás** | Modern UI/UX lehetőségek | Elavult felület, korlátozott lehetőségek |

# Elemi adattípusok

| **Típus** | **Típusnév másképpen (egyenértékű)** | **Helyigény (bit)** | **Példa** |
| --- | --- | --- | --- |
| **Logikai változó** | **bool** | Boolean | 8 (!) |
| **Egész szám (előjel nélküli)** | **sbyte, byte** | SByte, Byte | 8 |
|  | **short, ushort** | Int16, UInt16 | 16 |
|  | **int, uint** | Int32, UInt32 | 32 |
|  | **long, ulong** | Int64, UInt64 | 64 |
| **Lebegőpontos szám** | **float** | Single | 32 |
|  | **double** | Double | 64 |
|  | **decimal** | Decimal | 128 |
| **Karakter** | **char** | Char | 8 |
| **Szöveg** | **string** | String | változó |

# Operátorok

C#-ban az **operátorok** olyan speciális szimbólumok, amelyek műveleteket hajtanak végre az operándusokon. Az operátorokat különböző kategóriákba sorolhatjuk.

## Aritmetikai operátorok

Ezek az alapvető matematikai műveletek végrehajtására szolgálnak.

| Operátor | Leírás | Példa | Eredmény |
| --- | --- | --- | --- |
| + | Összeadás | 5 + 3 | 8 |
| - | Kivonás | 5 - 3 | 2 |
| \* | Szorzás | 5 \* 3 | 15 |
| / | Osztás | 5 / 3 | 1 (egész osztás esetén) |
| % | Maradékos osztás | 5 % 3 | 2 |

[!IMPORTANT] Ha egész számokat (int) osztasz, az eredmény **egész szám** lesz. Ha lebegőpontos eredményt szeretnél, használj double vagy float típusokat:

double result = 5.0 / 3; // 1.6667

## Összehasonlító (relációs) operátorok

Logikai értéket (true vagy false) adnak vissza két érték összehasonlításakor.

| Operátor | Leírás | Példa | Eredmény |
| --- | --- | --- | --- |
| == | Egyenlőség | 5 == 3 | false |
| != | Nem egyenlő | 5 != 3 | true |
| > | Nagyobb mint | 5 > 3 | true |
| < | Kisebb mint | 5 < 3 | false |
| >= | Nagyobb vagy egyenlő | 5 >= 3 | true |
| <= | Kisebb vagy egyenlő | 5 <= 3 | false |

## Logikai operátorok

Logikai műveleteket végeznek, és true vagy false értéket adnak vissza.

| Operátor | Leírás | Példa | Eredmény |
| --- | --- | --- | --- |
| && | Logikai ÉS | (true && false) | false |
| || | Logikai VAGY | (true || false) | true |
| ! | Logikai NEM | !true | false |

bool a = true;  
bool b = false;  
Console.WriteLine(a && b); // false  
Console.WriteLine(a || b); // true  
Console.WriteLine(!a); // false

## Értékadás operátorok

Értékadásra és módosításra szolgálnak.

| Operátor | Leírás | Példa | Ugyanaz, mint… |
| --- | --- | --- | --- |
| = | Egyszerű értékadás | x = 5 | – |
| += | Összeadás és értékadás | x += 5 | x = x + 5 |
| -= | Kivonás és értékadás | x -= 5 | x = x - 5 |
| \*= | Szorzás és értékadás | x \*= 5 | x = x \* 5 |
| /= | Osztás és értékadás | x /= 5 | x = x / 5 |
| %= | Maradékos osztás és értékadás | x %= 5 | x = x % 5 |

int x = 10;  
x += 5; // x most 15  
x \*= 2; // x most 30

## Bitműveleti operátorok

Ezek bináris szinten végeznek műveleteket az egész számokon.

| Operátor | Leírás | Példa (bináris) | Eredmény |
| --- | --- | --- | --- |
| & | Bitenkénti ÉS | 5 & 3 (101 & 011) | 1 (001) |
| | | Bitenkénti VAGY | 5 | 3 (101 | 011) | 7 (111) |
| ^ | Bitenkénti kizáró VAGY (XOR) | 5 ^ 3 (101 ^ 011) | 6 (110) |
| ~ | Bitenkénti negálás | ~5 (~00000101) | -6 |
| << | Bitenkénti balra tolás | 5 << 1 (101 << 1) | 10 (1010) |
| >> | Bitenkénti jobbra tolás | 5 >> 1 (101 >> 1) | 2 (010) |

int a = 5; // 101 binárisan  
int b = 3; // 011 binárisan  
Console.WriteLine(a & b); // 1 (001)  
Console.WriteLine(a | b); // 7 (111)  
Console.WriteLine(a ^ b); // 6 (110)

## 6️⃣ Növelés és csökkentés operátorok

Ezek az operátorok az adott változó értékét módosítják **1-gyel**.

| Operátor | Leírás | Példa | Eredmény |
| --- | --- | --- | --- |
| ++ | Növelés (előtte) | ++x | x egyel nő, az új értéket adja vissza |
| ++ | Növelés (utána) | x++ | x egyel nő, de az eredeti értéket adja vissza |
| -- | Csökkentés (előtte) | --x | x egyel csökken, az új értéket adja vissza |
| -- | Csökkentés (utána) | x-- | x egyel csökken, de az eredeti értéket adja vissza |

int x = 5;  
Console.WriteLine(x++); // 5 (először kiírja az eredeti értéket, majd növeli)  
Console.WriteLine(++x); // 7 (először növeli, majd kiírja)

## Feltételes (ternary) operátor

Egyszerű feltételes értékadást tesz lehetővé egy sorban.

| Operátor | Leírás | Példa | Eredmény |
| --- | --- | --- | --- |
| ? : | Ha igaz, akkor egyik érték, ha hamis, akkor másik | (5 > 3) ? "Igaz" : "Hamis" | "Igaz" |

int a = 10, b = 20;  
string eredmeny = (a > b) ? "A nagyobb" : "B nagyobb";  
Console.WriteLine(eredmeny); // "B nagyobb"

## Null-kezelő operátorok

Speciális operátorok a null értékek kezelésére.

| Operátor | Leírás | Példa | Eredmény |
| --- | --- | --- | --- |
| ?? | Ha null, akkor másik értéket ad | string s = null; s ?? "Alapérték" | "Alapérték" |
| ?. | Null-előrehaladó operátor | obj?.Property | Ha obj null, nem dob hibát |

string? nev = null;  
Console.WriteLine(nev ?? "Nincs név megadva"); // "Nincs név megadva"

# Összetett adatszerkezetek

Nagyon jó irány! A C# összetett adatszerkezetei lehetővé teszik, hogy **több értéket** tároljunk, **strukturáltan** dolgozzunk adatokkal, és **rugalmasan** szervezzük a programunk logikáját.

Nézzük sorban a leggyakoribb összetett adatszerkezeteket, alap példákkal:

## ✅ 1. Tömbök (Arrays)

Statikus méretű, azonos típusú elemeket tartalmaz.

int[] szamok = new int[3];  
szamok[0] = 10;  
szamok[1] = 20;  
szamok[2] = 30;  
  
Console.WriteLine(szamok[1]); // 20

* Mérete fix.
* Az indexelés 0-tól kezdődik.
* Használható foreach vagy for ciklussal.

## ✅ 2. Listák (List<T>)

Dinamikus, típusos lista. Sokkal rugalmasabb, mint a tömb.

List<string> nevek = new List<string>();  
nevek.Add("Anna");  
nevek.Add("Béla");  
  
Console.WriteLine(nevek[0]); // Anna

Jellemzők: - Elemet hozzáadhatsz Add, törölhetsz Remove, kereshetsz Contains stb. - Mérete automatikusan nő.

## ✅ 3. Szótár (Dictionary<K, V>)

Kulcs-érték párokat tárol.

Dictionary<string, int> korok = new Dictionary<string, int>();  
korok["Anna"] = 23;  
korok["Béla"] = 30;  
  
Console.WriteLine(korok["Anna"]); // 23

* Kulcs (K) egyedi, az érték (V) lehet bármi.
* Gyors keresés kulcs alapján.
* Hasznos például adatlekérdezéshez, azonosítókhoz, statisztikákhoz.

## ✅ 4. Halmaz (HashSet<T>)

Egyedi értékeket tároló gyűjtemény.

HashSet<int> szamok = new HashSet<int>();  
szamok.Add(1);  
szamok.Add(2);  
szamok.Add(2); // nem kerül be újra  
  
Console.WriteLine(szamok.Count); // 2

* Nem enged duplikált elemeket.
* Gyors ellenőrzés, hogy egy elem benne van-e (Contains).

## ✅ 5. Saját osztály (class)

Komplex adatmodellekhez, viselkedéssel (metódusok) együtt.

class Ember  
{  
 public string Nev;  
 public int Kor;  
  
 public void Koszon()  
 {  
 Console.WriteLine($"Szia, {Nev} vagyok!");  
 }  
}  
  
Ember e = new Ember();  
e.Nev = "Ádám";  
e.Kor = 25;  
e.Koszon(); // Szia, Ádám vagyok!

* Referenciatípus (memóriában hivatkozással kezelődik).
* Használható OOP (objektumorientált programozás) alapjaként.

## ✅ 6. Struktúra (struct)

Hasonló az osztályhoz, de **értéktípus**. Inkább kisebb, egyszerűbb objektumokra.

struct Pont  
{  
 public int X;  
 public int Y;  
}  
  
Pont p = new Pont();  
p.X = 3;  
p.Y = 4;  
  
Console.WriteLine($"{p.X}, {p.Y}");

* Jobban teljesít, ha sok kis adatot mozgatunk.
* Nincs öröklés, csak egyszerűbb használatra.

## Összehasonlító táblázat (röviden)

| Típus | Dinamikus? | Egyedi elemek? | Kulcs-érték? | OOP? |
| --- | --- | --- | --- | --- |
| Array | ❌ | ❌ | ❌ | ❌ |
| List<T> | ✅ | ❌ | ❌ | ❌ |
| Dictionary<K,V> | ✅ | ✅ (kulcs) | ✅ | ❌ |
| HashSet<T> | ✅ | ✅ | ❌ | ❌ |
| class | ✅ | - | - | ✅ |
| struct | ✅ | - | - | ✅ |

Nagyon jó kérdés! Az enum (felsorolási típus) **nem tipikus értelemben vett összetett adatszerkezet**, mint a tömb vagy lista, de **összetett adattípusnak** számít, mivel több lehetséges **névhez rendelt értéket** tartalmaz egy típus alatt.

# enum?

Az enum (enumeration) egy olyan típus, amely egy előre definiált, **nevesített értékkészletet** tartalmaz. Alapértelmezés szerint ezek mögött egész számok állnak.

Példa:

enum Hetnapja  
{  
 Hetfo,  
 Kedd,  
 Szerda,  
 Csutortok,  
 Pentek,  
 Szombat,  
 Vasarnap  
}

Használat:

Hetnapja nap = Hetnapja.Szerda;  
  
if (nap == Hetnapja.Szerda)  
{  
 Console.WriteLine("Ma szerda van.");  
}

Mögöttes értékek:

Alapértelmezés szerint az első érték 0, és minden következő egyel nő:

Console.WriteLine((int)Hetnapja.Hetfo); // 0  
Console.WriteLine((int)Hetnapja.Szerda); // 2

De manuálisan is megadható:

enum Statusz  
{  
 Folyamatban = 1,  
 Kesz = 2,  
 Hibas = 99  
}

# Vezérlési szerkezetek

A vezérlési szerkezetek határozzák meg, milyen sorrendben, milyen feltételek mellett, hányszor és hogyan hajtódjanak végre az utasítások.

## Feltételes elágazás – if, else if, else

int szam = 10;  
  
if (szam > 0)  
{  
 Console.WriteLine("Pozitív szám");  
}  
else if (szam < 0)  
{  
 Console.WriteLine("Negatív szám");  
}  
else  
{  
 Console.WriteLine("A szám nulla");  
}

[!NOTE]  
A feltétel zárójelben van.

[!NOTE]  
A blokk {} között van.

[!NOTE]  
Ha csak egy sor van, elhagyható a {}, de **ajánlott** mindig használni.

## Többágú elágazás – switch

A switch akkor hasznos, ha egy változót több lehetséges értékkel kell összehasonlítani.

int nap = 3;  
  
switch (nap)  
{  
 case 1:  
 Console.WriteLine("Hétfő");  
 break;  
 case 2:  
 Console.WriteLine("Kedd");  
 break;  
 case 3:  
 Console.WriteLine("Szerda");  
 break;  
 default:  
 Console.WriteLine("Ismeretlen nap");  
 break;  
}

[!NOTE]  
A break utasítás megszakítja a switch-et.

[!NOTE]  
A default akkor fut, ha egyik case sem egyezik.

## Ciklusok – for, while, do-while, foreach

### 🔹 for ciklus

Ismert ismétlésszámnál.

for (int i = 0; i < 5; i++)  
{  
 Console.WriteLine("i értéke: " + i);  
}

### while ciklus

Amíg a feltétel igaz, ismétel.

int szam = 0;  
  
while (szam < 3)  
{  
 Console.WriteLine(szam);  
 szam++;  
}

### do-while ciklus

Legalább egyszer lefut, utána ellenőrzi a feltételt.

int szam = 0;  
  
do  
{  
 Console.WriteLine(szam);  
 szam++;  
}  
while (szam < 3);

### foreach ciklus

Gyűjtemények bejárására – egyszerű és tiszta.

string[] nevek = { "Anna", "Béla", "Cili" };  
  
foreach (string nev in nevek)  
{  
 Console.WriteLine(nev);  
}

## Kivételkezelés – try, catch, finally

A programhibák (*például null érték, osztás 0-val, fájlhiba*) elkapására szolgál.

try  
{  
 int x = 10;  
 int y = 0;  
 int eredmeny = x / y;  
}  
catch (DivideByZeroException ex)  
{  
 Console.WriteLine("Hiba: Osztás nullával.");  
}  
catch (Exception ex)  
{  
 Console.WriteLine("Általános hiba történt: " + ex.Message);  
}  
finally  
{  
 Console.WriteLine("Ez mindig lefut.");  
}

[!NOTE]  
Csak a try blokkban lehet a hibás kód.

[!NOTE]  
A catch hiba esetén hajtódik végre.

[!NOTE]  
A finally **mindig lefut** – akár volt hiba, akár nem.

## Hasznos utasítások ciklusban

| Utasítás | Jelentés |
| --- | --- |
| break | Kilép a ciklusból |
| continue | Átugorja a ciklus hátralévő részét, új iteráció indul |
| return | Kilép a metódusból |

Nagyszerű, a **metódusok és függvények** a C# szíve-lelke – segítségükkel darabolhatod a programodat logikus részekre, újrahasználható kódblokkokat hozhatsz létre, és olvashatóbbá teheted az alkalmazásodat.

# Metódus

A C#-ban a **függvények** hivatalosan **metódusok** (*methods*), és mindig egy osztályhoz vagy struktúrához tartoznak.

visszatérési\_típus MetódusNeve(típus paraméter1, típus paraméter2, ...)  
{  
 // utasítások  
 return érték; // ha a visszatérési típus nem void  
}

Egyszerű példa: Név kiírása

void Koszontes(string nev)  
{  
 Console.WriteLine("Szia, " + nev + "!");  
}

Meghívás:

Koszontes("Anna"); // Kiírja: Szia, Anna!

## Metódus visszatérési értékkel

int Osszead(int a, int b)  
{  
 return a + b;  
}

Meghívás:

int eredmeny = Osszead(5, 3);  
Console.WriteLine(eredmeny); // 8

[!NOTE]  
A void típus azt jelenti, hogy a metódus **nem ad vissza semmit**.

## Paraméterátadás módjai

### 1. Érték szerint (*alapértelmezett*)

Az egyszerű típusok (*int, double, bool, stb.*) másolódnak az eredeti nem változik.

void Novelem(int szam)  
{  
 szam++;  
 Console.WriteLine(szam); // 6   
}  
  
int x = 5;  
Novelem(x);  
Console.WriteLine(x); // 5 – az eredeti nem változott

### 2. Referencia szerint – ref vagy out

#### ref – már inicializált értéket módosít

void Duplaz(ref int szam)  
{  
 szam \*= 2;  
}  
  
int x = 5;  
Duplaz(ref x);  
Console.WriteLine(x); // 10

#### out – a metódusban kap értéket

void Szorzas(int a, int b, out int eredmeny)  
{  
 eredmeny = a \* b;  
}  
  
int r;  
Szorzas(3, 4, out r);  
Console.WriteLine(r); // 12

## Metódus túlterhelés (*overloading*)

Ugyanaz a metódusnév, más paraméterekkel:

int Osszeg(int a, int b)  
{  
 return a + b;  
}  
  
double Osszeg(double a, double b)  
{  
 return a + b;  
}

## Metódus alapértelmezett paraméterekkel

void Udvozlet(string nev = "ismeretlen")  
{  
 Console.WriteLine($"Üdv, {nev}!");  
}  
  
Udvozlet(); // Üdv, ismeretlen!  
Udvozlet("Béla"); // Üdv, Béla!

## Lokális függvények (*függvény egy másikon belül*)

void Kulfuggveny()  
{  
 void Belso()  
 {  
 Console.WriteLine("Ez egy belső metódus.");  
 }  
  
 Belso();  
}

## Összefoglalás

| Fogalom | Jelentés |
| --- | --- |
| void | Nincs visszatérési érték |
| return | Értéket ad vissza |
| ref, out | Hivatkozással történő átadás |
| params | Tetszőleges számú paraméter |
| overload | Több metódus ugyanazzal a névvel, eltérő paraméterekkel |
| default param | Alapértelmezett paraméterérték |

# OOP *(Objektumorientált programozás)*

Az **osztály**, **absztrakt osztály** és **interface (felület)** a C# nyelvben (és más objektumorientált nyelvekben) az **öröklődés** és **polimorfizmus** fontos eszközei, de különböző célt szolgálnak. Itt egy rövid, érthető összehasonlítás:

## Osztály (class)

* Olyan sablon, amely **mezőket**, **tulajdonságokat**, **metódusokat** és **konstruktorokat** tartalmaz.
* Lehet példányosítani (példányt lehet létrehozni belőle).
* Teljesen működőképes egység.

**Példa:**

class Auto {  
 public void Indit() {  
 Console.WriteLine("Az autó elindult.");  
 }  
}

## Absztrakt osztály (abstract class)

Az absztrakt osztály soha nem közvetlen példányosításra szolgál. Ennek az osztálynak tartalmaznia kell legalább egy absztrakt metódust, amelyet az absztrakt kulcsszó vagy módosító jelöl meg az osztálydefinícióban. Az Absztrakt osztályokat általában egy alaposztály meghatározására használják az osztályhierarchiában. Vagy más szóval, az absztrakt osztály egy hiányos osztály vagy speciális osztály, amelyet nem tudunk példányosítani. Az absztrakt osztály célja, hogy mintát biztosítson a származtatott osztályokhoz, és meghatározzon néhány szabályt, amelyeket a származtatott osztályoknak végre kell hajtaniuk.

[!IMPORTANT]  
Nem lehet példányosítani, így nem tartalmazhat konstruktort sem.

[!IMPORTANT]  
Lehet benne **megvalósított** és **absztrakt** (*csak aláírt, de nem definiált*) metódus.

[!IMPORTANT]  
Arra szolgál, hogy **alap viselkedést** biztosítson, amit az alosztályok kiegészítenek vagy felülírnak.

[!IMPORTANT]  
Örökléssel származtatott osztályok **egyetlen** absztrakt osztályból örökölhetnek. Többszörös öröklés nem megengedett.

[!NOTE]  
Konstruktorokat vagy destruktorokat tartalmazhat.

**Példa:**

abstract class Allat {  
 public abstract void HangotAd(); // nincs törzs, kötelező felülírni  
  
 public void Mozog() {  
 Console.WriteLine("Az állat mozog.");  
 }  
}  
  
class Kutya : Allat {  
 public override void HangotAd() {  
 Console.WriteLine("Vau!");  
 }  
}

## Interface (*felület*)

A gyermek osztálynak kötelezően megvalósítandó methódusokat tartalmazza. Az interface nem deklarálhat példányadatokat, például mezőket, automatikusan megvalósított tulajdonságokat vagy tulajdonságszerű eseményeket. Az interfészek használatával például több forrásból származó viselkedést is felvehetünk egy osztályba. Ez a képesség azért is fontos a C# -ban, mert a nyelv nem támogatja az osztályok többszörös öröklését. - Csak **metódus-aláírásokat** és **tulajdonságokat** tartalmazhat (*nincs implementáció* kivéve a static metódust). - Egy osztály **több interfészt** is megvalósíthat (többszörös öröklés). - Használatával **szerződés jellegű elvárásokat** lehet megadni, hogy mit tudjon az adott osztály.

**Példa:**

interface IRepulo {  
 void Felszall();  
}  
  
class RepuloGep : IRepulo {  
 public void Felszall() {  
 Console.WriteLine("A gép felszáll.");  
 }  
}

## Összefoglaló táblázat:

| Tulajdonság | Osztály | Absztrakt osztály | Interface |
| --- | --- | --- | --- |
| Példányosítható? | Igen | Nem | Nem |
| Tartalmazhat megvalósítást? | Igen | Igen (részben) | Nem (C# 8.0-tól korlátozottan igen) |
| Absztrakt tagokat tartalmazhat? | Nem | Igen | Igen (csak aláírásokat) |
| Öröklés típusa | Egy osztályból | Egy absztrakt osztályból | Több interface is megvalósítható |
| Használat célja | Példányosítás, működés | Közös alapviselkedés meghatározása | Képességek definiálása |

1. **Fájlkezelés és adatbázis kapcsolat** – Hogyan olvashatsz és írhatsz fájlokat (StreamReader, StreamWriter), illetve használhatsz adatbázisokat (pl. SQL kapcsolat Entity Framework segítségével).

# Aszinkron programozás

task, thread, process **alapvető fogalmak a párhuzamos programozásban**!

## Process *(folyamat)*

Egy futtatható fájl elindítása hozza létre.

* Egy **önállóan futó program példány**.
* Saját **memóriaterülete van**, saját erőforrásai (fájlok, hálózat, stb.).
* Teljesen **elszigetelve** fut más folyamatoktól.
* A legtöbb alkalmazás egy processzként indul (chrome.exe, notepad.exe, stb.).

## Thread *(szál)*

Egy böngésző *(pl. Chrome)* a processzen belül indíthat külön szálat a letöltésnek, a UI-nak, és a videólejátszásnak.

* Egy processzen **belüli végrehajtási szál**.
* Több szál ugyanazon memóriaterületen osztozik.
* Könnyű és gyors váltani köztük, de **versenyhelyzetek** és **szinkronizációs problémák** előfordulhatnak.

## Task *(feladat)* – (C# specifikus)

* A Task egy **abstrakció a szálak felett**.
* A .NET Task osztály lehetővé teszi a **magas szintű párhuzamosságot**, gyakran async/await-tel együtt.
* Nem mindig új szálat indít! Lehet pl. szálkészletből, vagy egyszerű ütemezésből.

await Task.Run(() => HosszuSzamitas());

Ez párhuzamosan lefuttatja a számítást egy másik szálon, de neked nem kell közvetlenül szálat kezelni.

## Összehasonlító táblázat

| Jellemző | **Process** | **Thread** | **Task** (C#) |
| --- | --- | --- | --- |
| Hol fut? | Önállóan, operációs rendszer alatt | Egy processzen belül | Általában egy szálon belül |
| Memória | Saját, izolált | Osztozik a processz memóriáján | Osztozik, nincs saját memória |
| Létrehozási költség | Nagy | Közepes | Kicsi |
| Egymás közötti kommunikáció | Nehézkes (IPC) | Könnyebb (de kell szinkronizálni) | Könnyű, await megoldja |
| Példa C#-ben | Process.Start("notepad.exe") | new Thread(() => ...) | Task.Run(() => ...) |
| Cél | Külön alkalmazás | Párhuzamos végrehajtás | Aszinkron vagy párhuzamos művelet |

## Összefoglalás – mikor melyiket?

| Feladat típusa | Javasolt megoldás |
| --- | --- |
| Külön program indítása | Process |
| Párhuzamos háttérmunka (pl. letöltés) | Thread vagy Task |
| Aszinkron művelet (pl. HTTP kérés) | Task + async/await |
| Sok könnyű feladat futtatása | Task, szálkészlettel |
| Bonyolult szálkezelés, finom vezérlés | Thread manuálisan |

# Gyakorlati példák

## [Állatos példa abstract class-ra és interface-re](C:\\Users\\Boros\\Downloads\\CSharp_examples\\oop\\)

## [Állatos példa több interfésszel, használattal, Thread-el](C:\\Users\\Boros\\Downloads\\CSharp_examples\\OOP2\\)

## [Hello\_Wpf a felület és kódolás bemutatására](C:\\Users\\Boros\\Downloads\\CSharp_examples\\Hello_Wpf\\)

## [Hello\_WinForm a felület és kódolás bemutatására](C:\\Users\\Boros\\Downloads\\CSharp_examples\\Hello_WinForm\\)