Haladó Fejlesztési Technikák

ÓBUDAI EGYETEM
NEUMANN JÁNOS INFORMATIKAI KAR

Modul 3

DLL értelme

Fordítási folyamat

Natív DLL-ek

Felügyelt DLL-ek

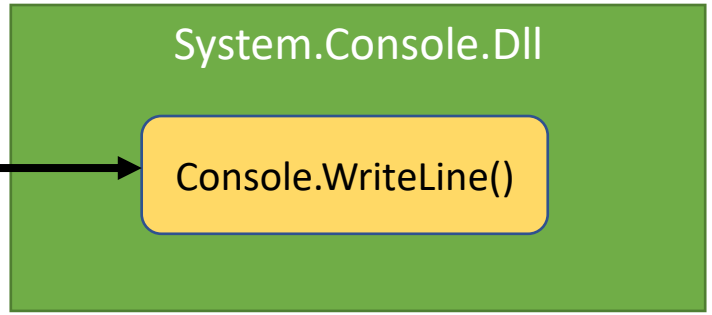
Reflexió

Dynamic típus

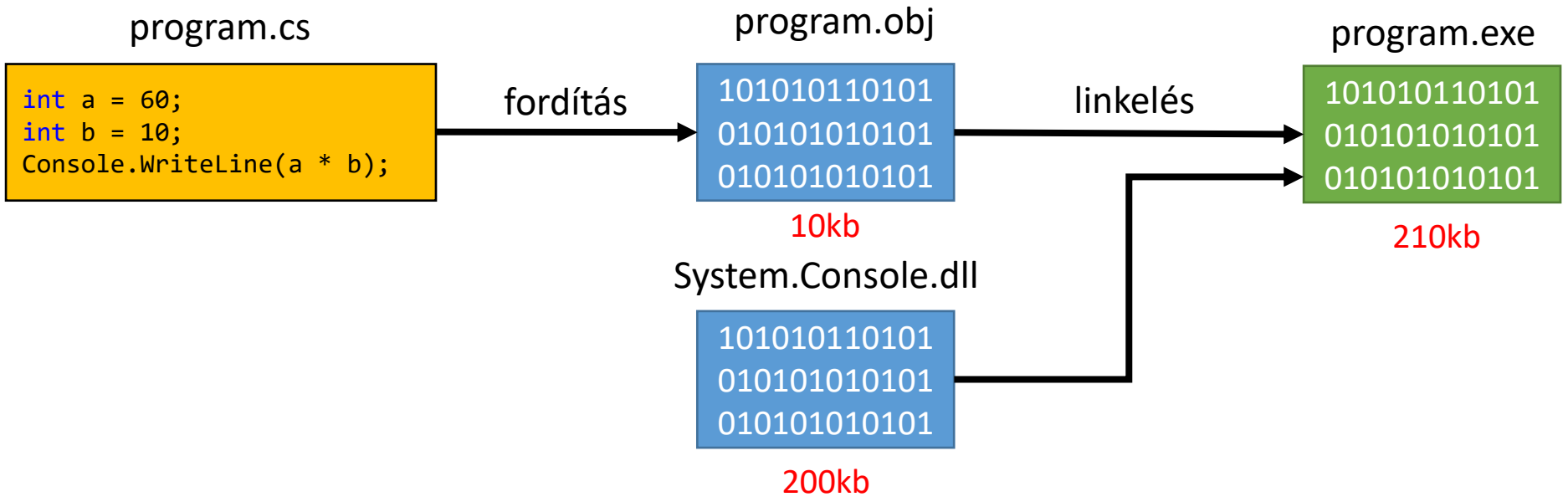
Mi lenne a DLL célja?

- **DLL röviden**: Bizonyos osztályok és metódusok kiszervezése egy olyan fájlba, ami nem forráskód, hanem már lefordított bináris kód
- **Értelme:**
  - Fejlesztünk egy titkos algoritmust, amit használhat más is, de nem a forrást akarjuk átadni
  - Ugyanazt a függvényt használhassa több program is
  - Operációs rendszer különböző szolgáltatásokat így biztosít a fejlesztők részére
  - .NET összes beépített típusát is így érjük el (megfelelő DLL-be belehívunk)

```
int a = 60;
int b = 10;
Console.WriteLine(a * b);
```

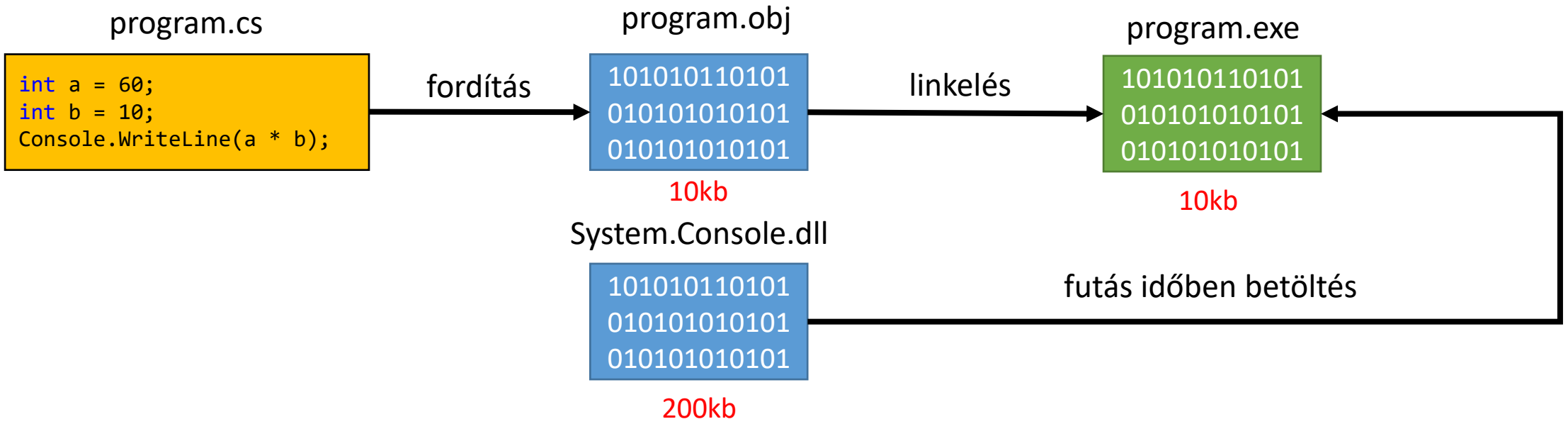


Fordítás és linkelés



- **Statikus linkelés**
  - Függőségek is belekerülnek a végső futtatható állományba
  - Régen készültek így a programok (hatalmas fájlméret)
  - Ugyanazt a függvényt több program is külön-külön tartalmazza → pazarlás

Dinamikus linkelés



- **Dinamikus linkelés**
  - Futásidőben gyakorlatilag belehívunk a dll-ben egy metódusba
  - A betöltést az operációs rendszer végzi
  - Ugyanazt a DLL-t több program is hívhatja (Shared Object) → .NET-ben AppDomain miatt nem mindig
  - Legtöbb mai modern program így működik
  - Update-ek egyszerűen megvalósíthatók: lecseréljük a DLL fájlt egy újabbra
  - **.NET DLL-ek**: C:\Windows\assembly

Futtatható állomány típusok

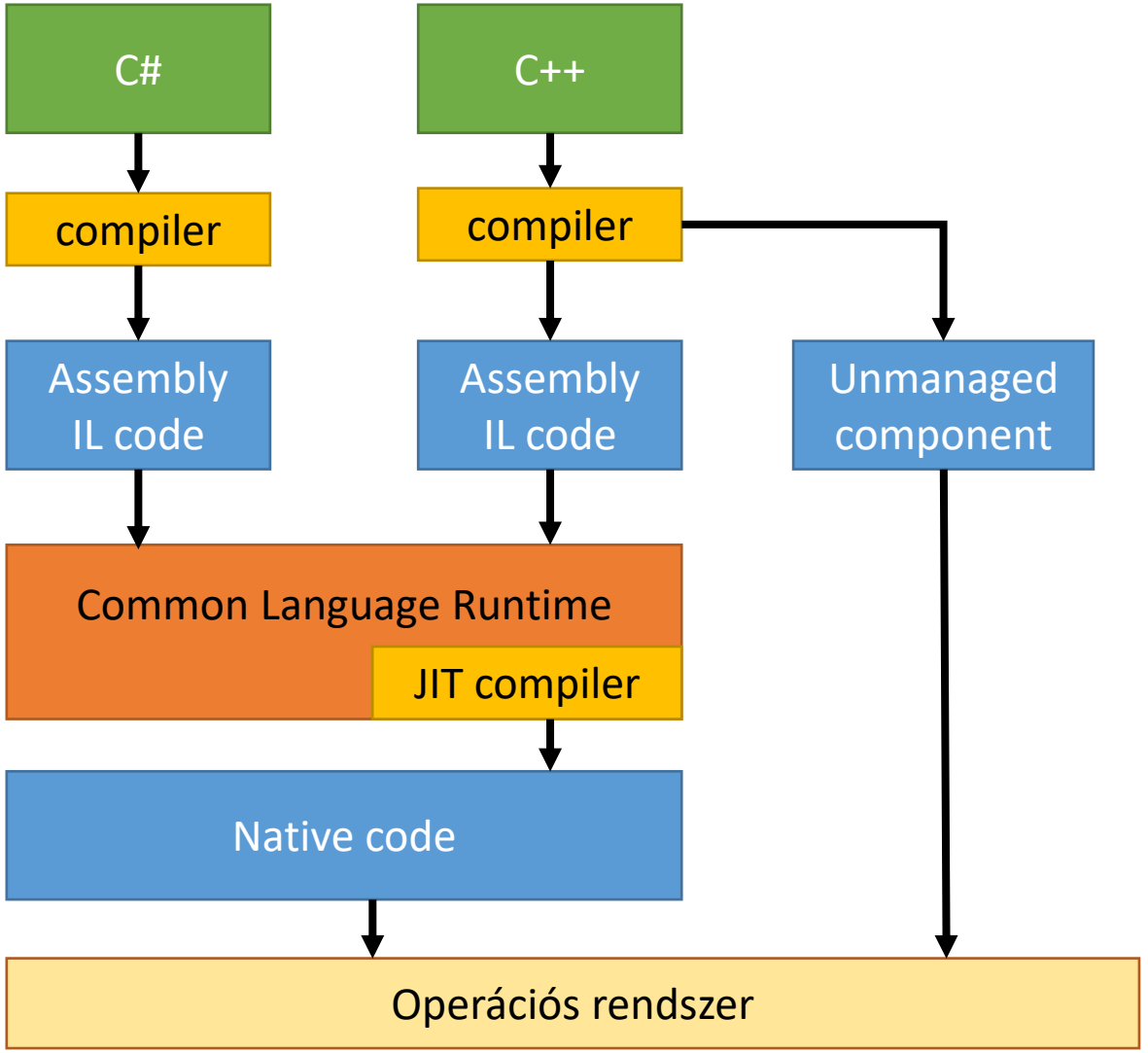
- **Futtatható állomány:** futtatható kód és minden a futtatáshoz szükséges adat

- Linux: **ELF**
  - **E**xecutable and **L**inkable **F**ormat
  - Kiterjesztés nélküli bináris futtatható állomány vagy SO file
  - Extra funkció: fatELF = több platformfüggő állomány egy nagy platformfüggetlen fájlban

- Windows: **PE**
  - **P**ortable **E**xecutable
  - Minden **exe** és **dll** fájl ide tartozik
  - DLL: annyiban különbözik az exe-től, hogy nincs belépési pontja
  - Extra funkció: ikon elhelyezése a fájlban

Unmanaged/managed állományok

- **Unmanaged/natív állomány**
  - OS/CPU számára értelmezhető bytekód
  - Közvetlen hardver elérés
  - Bármilyen nyelven írható
  - Platformfüggő
- **Managed/felügyelt állomány**
  - Egy vagy több osztály van benne
  - .NET értelmező tud vele dolgozni csak
  - Nyelvfüggetlen
  - Platformfüggetlen (de .NET értelmező kell hozzá)
  - .NET értelmező: .NET runtime



Natív DLL-ek

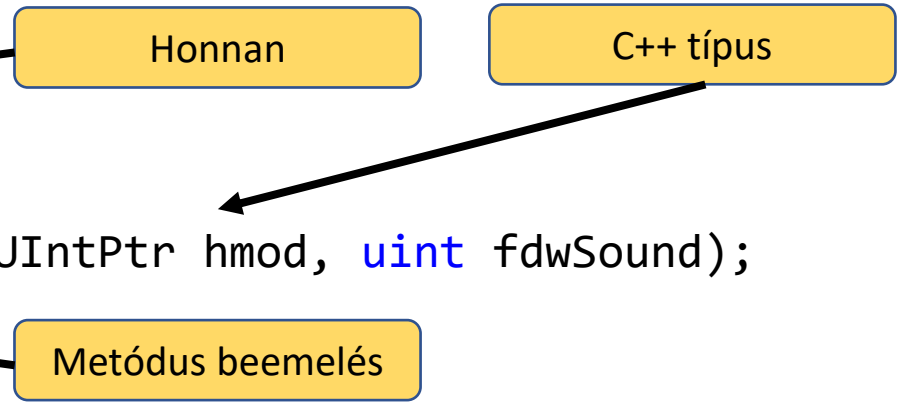
- Aktuális könyvtárból vagy a **%PATH%-**ból töltődnek be
  - **%PATH%:** Windows, System, System32 könyvtárak
- Lassú betöltődés
- DLL HELL: Verziózás nem megoldott
  - Különféle alkalmazások, különféle verziót igényelnek
  - DLL stomping a megoldás vagy minden DLL az exe mellett
  - Linux megoldás: file szintű csomagkezelő
  - .NET megoldás: Global Assembly Cache (GAC)
- Windows API
  - Natív DLL gyűjtemény
  - Operációs rendszer minden publikus funkcionalitása elérhető
  - A fontosabb funkciókhoz tartozik .NET osztály → háttérben WINAPI hívás

Natív DLL-ek

- Natív DLL meghívása

```
public class Program
{
        [DllImport("winmm.dll", SetLastError = true)]
        static extern bool PlaySound(string pszSound, UIntPtr hmod, uint fdwSound);


        static void Main(string[] args)
        {
                string fname = @"c:\Windows\Media\tada.wav";
                PlaySound(fname, UIntPtr.Zero, 1);
                Console.ReadLine();
        }
}
```

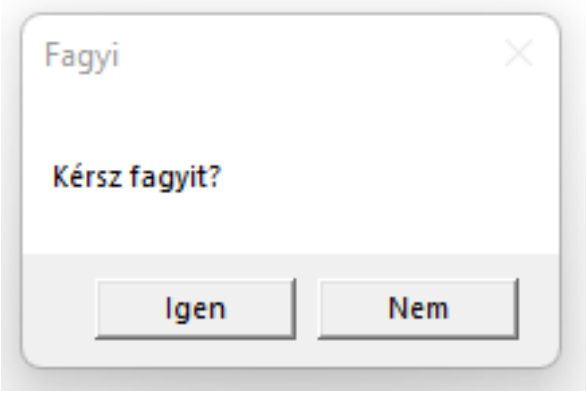


Natív DLL-ek

- Problémák:
  - Platform és OS függő kód
  - Paraméterek és visszatérési értékek típusai
  - Ki mit szabadít fel a memóriában
  - Nem tudjuk mi érthető el a DLL-ben (dumpbin)
  - Nem tudjuk mik a DLL függőségei (dependency walker)
  - Csak futásidőben derül ki, hogy létezik-e a dll és a metódusok
- WINAPI szignatúrák: **pinvoke.net**

Feladat

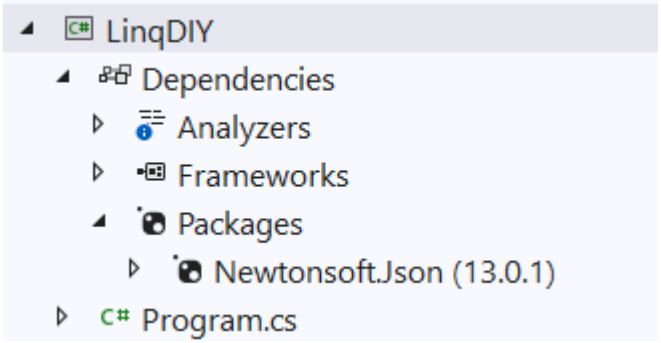
- Dobjunk fel a felhasználó számára egy ilyen ablakot!
- A választása függvényében írjunk ki neki valamit!
- Fedjük el a natív WINAPI hívást C# osztállyal!



- **Segítség**: MessageBox a neve ennek az ablaknak!
- **URL**: https://www.pinvoke.net/default.aspx/user32/MessageBox.html

Felügyelt DLL-ek

- **Minden .NET osztály és metódus hívása egy DLL hívás volt**
- Dependencies szekcióban tároljuk el → csproj fájlban van leírva konkrétan
- Add → Project Reference menüvel adhatunk hozzá mi magunk is DLL-t a projekthez
- Fordító is ellenőrzi a meglétét!
- Gyors betöltődés, ugyanolyan gyors mint a saját kód



- DLL készítése: Add → New Project → Class Library
  - Console Apphoz képest különbség: nincs Main
    - Az exe fájl PE része csak a CLR értelmezőt tölti be Console App esetén
  - Minden osztály és metódus, amit használatra szánunk, legyen publikus
  - Build után elkészül a DLL
- Felügyelt EXE/DLL gyűjtőneve: **.NET Assembly**

Feladat

- **Készítsünk egy saját DLL-t!**
- Valósítsunk meg benne egy olyan Lista adatszerkezetet, ami eseményekkel jelez, hogyha elem lett hozzáadva, vagy törölve belőle!
- A névtér legyen: **OENIK.NotifyCollection**
- Az osztály legyen: **EventList<T>**
- Gyakorlatilag terjessze az osztály ki a beépített List<T> típust!

- Utána használjuk fel a DLL-t egy Console App-ban!

Futtatási folyamat

1. EXE fájlra duplakatt
2. PE formátum validálása, 32/64 bites Process készítése a header alapján
3. Annak eldöntése, hogy az APP felügyelt/natív
4. Ha felügyelt, akkor a megfelelő verziójú CLR betöltése (.NET FW 4.6 vagy .NET 5)
5. App Domain létrehozása (sandboxolt közeg) → exe és dll-ek egységbezárva
6. Függőségek (assembly-k betöltése a Fusion komponenssel)
7. CLR meghívja a Main() metódust

DLL eszközök

- **gacutil.exe**
  - DLL regisztrálása/törlése a Global Assembly Cache-ből (.NET DLL-ek is itt vannak)
  - Verziókezelésre és függőségkezelésre is képes
  - MSDN-en megtalálható, hogy melyik osztály/névtér melyik DLL-ben található
- **NuGet**
  - Központi .NET csomagkezelő, felügyelt DLL-ekhez
  - GUI és CLI verziója is van
  - Szinte az összes C# library/tool letölthető vele
  - Függőségek/frissítések/ellentmondó verziók kezelve vannak
- **Dotpeek/ILDasm/Reflector**
  - EXE/DLL visszafejtő
  - Akkor működik, ha nem használtak Code Obfuscatort

Reflexió célja

- **Az a képesség, amellyel a program önmaga struktúráját és viselkedését futásidőben analizálni és alakítani tudja**
- Jellemzői
  - Magasszintű nyelv kell hozzá (Java, PHP, C#) → különböző támogatás
  - C#-ban két használati mód
    - **Futásidejű típusanalízis**: Milyen tulajdonságai vannak ennek az objektumnak? Milyen értékei vannak?
    - **Futásidejű típusgyártás**: System.Reflection.Emit → nem tárgyaljuk
- Több technológia is használja
  - **Intellisense**: metódusok, tulajdonságok listázása
  - **Szerializáció**: tulajdonságok és hozzájuk tartozó értékek kinyerése/beállítása
  - **Tesztek**: teszt metódusok kigyűjtése egy vezérlőpultra

Típusinformációk kinyerése

- Egy adott osztály jellemzőit szeretnénk
  - Mi a neve, milyen metódusai, tulajdonságai, adattagjai vannak, stb.
- **Type** típusba kapjuk vissza ezeket a jellemzőket
- Egy konkrét osztály típusinformációt kinyerhetjük

```
Type t = typeof(DateTime);
```

- Egy generikus paraméterből is megkaphatjuk a futásidőben behelyettesített típus infóit

```
class SuperList<T>
{
        public SuperList()
        {
                Type t = typeof(T);
                Console.WriteLine($"Ez egy {t.Name} típusú lista");
        }
}
```

Típusinformációk kinyerése

- Egy objektumból is megkaphatjuk a típusinfókat → gyakoribb használati eset

```
static void GetInfo(object obj)
{
        Type t = obj.GetType();
        Console.WriteLine($"Type name: {t.Name}");
}
```

- Stringként is beírható a típus neve, de akkor teljes névtér+típusnév kell

```
Type t = Type.GetType("System.DateTime");
```

- A jelenleg futtatott Assembly-ben lévő típust is elérhetjük

```
Assembly a = Assembly.GetExecutingAssembly();
Type t = a.GetType("Person");
```

Típusinformációk (Type) adatai

- Típus neve

```
string result = t.Name;
```

- Típus teljes neve (névtérrel együtt)

```
string result = t.FullName;
```

- Típus teljes neve + assembly infók (ahogyan a GAC is visszaadná)

```
string result = t.AssemblyQualifiedName;
```

- Típus ősosztálya (szintén Type formában)

```
Type result = t.BaseType;
```

- Ős-e / leszármazottja-e / megvalósítja-e? (Manager: Worker)

```
bool result = typeof(Worker).IsAssignableFrom(typeof(Manager));
bool result = typeof(Manager).IsSubclassOf(typeof(Worker));
bool result = typeof(Manager).IsAssignableTo(typeof(IComparable));
```

Típusinformációk (Type) adatai

- Tulajdonság/tulajdonságok kinyerése

```
PropertyInfo info = typeof(Worker).GetProperty("Name");
PropertyInfo[] infos = typeof(Worker).GetProperties();
```

- Adattag/adattagok kinyerése

```
FieldInfo info = typeof(Worker).GetField("name");
FieldInfo[] infos = typeof(Worker).GetFields();
FieldInfo info = typeof(Worker).GetField("name", BindingFlags.NonPublic); //privátok is
```

- Metódus/metódusok kinyerése

```
MethodInfo info = typeof(Worker).GetMethod("Parse");
MethodInfo[] infos = typeof(Worker).GetMethods();
```

PropertyInfo / FieldInfo

- Miután rendelkezésünkre áll egy **PropertyInfo**, rajta keresztül lekérhető **adott** objektum, **ezen** PropertyInfo-jú tulajdonságához tartozó **konkrét** érték

```
Type t = typeof(DateTime);

PropertyInfo yearInfo = t.GetProperty("Year");

DateTime now = DateTime.Now;

int value = (int)yearInfo.GetValue(now);
```

Milyen típussal dolgozunk

Milyen tulajdonságára vagyunk kíváncsiak

Hol a KONKRÉT objektum?

object-ként kapjuk vissza

Adott tulajdonságát kérjük

Ennek az objektumnak

- Értéket beállítani a **SetValue()**-val lehet
  - (1. param: objektum, 2. param: érték)

Feladat

- Készítsünk olyan programot, amely bármilyen objektum összes tulajdonságát kiírja a konzolra és mellé írja a konkrét értékét is!
- Egészítsük ki a programot arra, hogy valamilyen típusú gyűjtemény összes elemét képes egy szépen formázott táblázatban megjeleníteni!

MethodInfo

- Miután rendelkezésünkre áll egy **MethodInfo**, meghívhatjuk **ezen** metódusát **valamilyen** objektumnak

```
Type t = typeof(Person);

MethodInfo setupinfo = t.GetMethod("Setup");

Person p = new Person();

setupinfo.Invoke(p, new object[] { "Béla", 6 });
```

Milyen típussal dolgozunk

Milyen metódusával szeretnénk dolgozni?

Hol a KONKRÉT objektum?

Adott metódusát meghívjuk

Ennek az objektumnak

A paramétereket object [] –ként adjuk át

```
class Person
   {
        public string Name { get; set; }
        public int Age { get; set; }
        public void SetUp(string name, int age)
        {
            this.Age = age;
            this.Name = name;
        }
   }
```

- Statikus osztályoknál: Invoke első paramétere null

Egyéb hasznos metódusok

- Példányosítunk egyet adott típusból

```
var obj = Activator.CreateInstance(t);
var obj = Activator.CreateInstance(t, new object[] { "Béla", 6 });
```

- Kinyert tulajdonság/adattag típusának lekérése

```
PropertyInfo info = t.GetProperty("Name");
Type propType = info.PropertyType;
```

Feladat

- Készítsünk olyan programot, amely egy általunk megadott típushoz képes egy automatikus párbeszédalapú adatbekérést csinálni!

Feladat

- A párbeszédalapú bekérőt egészítsük ki automatikus XML mentéssel/betöltéssel is!
  - Ezt írjuk meg reflexióval a gyakorlás miatt!

Assembly elérés

- Lehetőség van arra is, hogy típusokat DLL-ből gyűjtsünk be futásidőben

```
Assembly a = Assembly.LoadFrom("data.dll");
Type[] types = a.GetTypes();
```

- Vagy a jelenleg futó Assembly-ből gyűjtjük be

```
Assembly a = Assembly.GetExecutingAssembly();
Type[] types = a.GetTypes();
```

- **DLL-ből begyűjtés hasznossága**:
  - Játékok addon-jai is általában „Addon” mappába elhelyezett DLL-ek
  - Futásidőben begyűjtjük belőlük az Enemy osztály leszármazottjait → új fajta ellenségek is lesznek

Feladat

- A párbeszédalapú bekérő legyen képes arra, hogy az adott típust futásidőben adjuk hozzá!
- Tehát új típus bekéréséhez ne kelljen lefordítani az alkalmazást újra!
- A **„classes”** mappából gyűjtse össze futásidőben a DLL-eket és azokból nyerje ki a típusokat!
- Hogy melyik típushoz akarunk bekérést csinálni, az legyen kiválasztható indításkor!
- Az adatbekéréshez lehessen extension-t írni, ezeket a DLL-eket az **„extensions”** mappából gyűjtse össze futásidőben a program!

Attribútumok

- Amolyan intelligens, programozható „cetlik”, amelyeket osztályokra, metódusokra, tagokra, stb. tehetünk
- **Szakmaian**: saját metaadattal kiegészítjük a típust

```
[Serializable]
class Person
{
        [Required]
        public string Name { get; set; }

        [Range(1,99)]
        [JsonIgnore]
        public int Age { get; set; }
}
```

Attribútum

Paraméteres attribútum

- Más nyelvekben: **annotáció**

Attribútumok

- Célja
  - Információk, kikötések, intelligens kommentek
  - Adott elem működését alapvetően nem befolyásolja
  - Kinyerhetjük minden tulajdonságra/mezőre/osztályra, hogy van-e attribútuma (és van-e benne adat)
- Rengeteg beépített technika is használja
  - NewtonSoft.JSON → JsonIgnore (tulajdonság kihagyása a JSON-ből)
  - XmlSerializer → NonSerialized (mező kihagyása az XML-ből)
  - Adatbáziskezelés → Key, Required, MaxLength, Range, stb.
  - Tesztelés → TestFixture, TestCase, stb.
  - Intelligens komment → Obsolete, DisplayName, Description, stb.

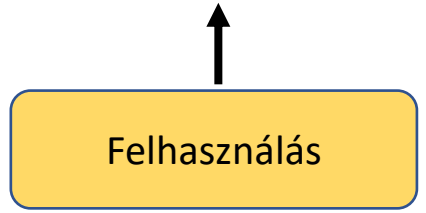
Saját attribútumok írása

- **Attribute** őstől származtatjuk
- Amúgy egy hétköznapi osztály
  - **AttributeUsage**: hol lehessen használni? → csak tulajdonságon!
  - **AllowMultiple**: többször is szerepelhessen

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = true)]
class TranslationAttribute : Attribute
{
        public string Language { get; set; }
        public string Text { get; set; }
        public TranslationAttribute(string language, string text)
        {
                Language = language;
                Text = text;
        }
}
```

```
class Person
{
   [Translation("HU", "név")]
   [Translation("GER", "name")]
   public string Name { get; set; }

   [Translation("HU", "életkor")]
   [Translation("GER", "jahre")]
   public int Age { get; set; }
}
```



Attribútumok elérése reflexióval

- Példakód

```
Person p = new Person();
p.Name = "Péter";
p.Age = 42;

PropertyInfo info = p.GetType().GetProperty("Name");
```

- Minden attribútum lekérése

```
IEnumerable<Attribute> attributes = info.GetCustomAttributes();
```

- Konkrét attribútum/attribútumok lekérése

```
IEnumerable<TranslationAttribute> attributes = info.GetCustomAttributes<TranslationAttribute>();
IEnumerable<Attribute> attributes = info.GetCustomAttributes(typeof(TranslationAttribute));
```

Feladat

- Írjunk egy olyan kiegészítést az adatbekérőhöz, ami adott típusnál nem a tulajdonság nevét írja ki, hanem a [DisplayName] attribútumban szereplő szöveget!

Reflexió hátrányai

- Nagyon lassú futásidőt eredményez!
  - Nyilván több száz objektum feldolgozása esetén jön elő ez a lassúság
  - Ott használjuk, ahol nem lehet másképp megoldani
- Nem arra való, hogy a láthatóságokat megkerüljük vele!
- Hasonlóan flexibilis, de sokkal gyorsabb megoldás: **dynamic** típus

```
static bool ValidateAll(List<dynamic> items)
{
        foreach (dynamic item in items)
        {
                if (!item.IsValid)
                {
                        return false;
                }
        }
        return true;
}
```

Feltételezzük, hogy van IsValid jellemzője

Nincs compiler/intellisense segítség

Köszönöm a figyelmet!

Kérdés esetén e-mailben szívesen állok rendelkezésre.