A delegált típus egy metódusreferencia. Az a célja, hogy egy metódust/függvényt hivatkozni tudjunk. Ez a gyakorlatban úgy néz ki, mintha egy változóba beletöltenénk egy függvényt, majd a változót meghívva tulajdonképpen a függvényt futtatjuk le. Elsőre egy felesleges nyelvi elemnek tűnhet, de segítségével egy függvény átadható egy másik függvénynek paraméterként, szerepelhet visszatérési értékként, továbbá adott függvényeket befogadni képes tömböt, listát hozhatunk létre. Számtalan gyakorlati előnye van mindegyik felsorolt lehetőségnek, ezeket gyakorlati feladatokon keresztül meg fogjuk ismerni.

Fontos kikötés, hogy magának a delegáltnak van egy típusa, ami megköti a visszatérési érték típusát és a paraméterek típusát és sorrendjét. És pont ugyanilyen típusú függvényekre tudunk a delegálttal mutatni. Nézzünk erre egy példát.

Először is létrehozzuk a delegált típusát:

```
delegate void Greeter(string name);
```

Majd a következő lépésben létrehozunk egy ugyanilyen visszatérési értékkel és paraméterlistával rendelkező függvényt:

```
void HungarianGreeter(string name)
{
    Console.WriteLine($"Jó reggelt, {name}!");
}
```

A következő lépésben példányosítjuk a delegálttal és ekkor kötelező jelleggel át is adjuk neki azt a függvényt, amit referálni szeretnénk vele:

```
Greeter greeterFunction = new Greeter(HungarianGreeter);
```

A függvényt meghívhatjuk közvetlenül vagy a delegált közreműködésével:

```
//függvény meghívása

HungarianGreeter("Péter");

//delegált meghívása

greeterFunction("Pál");
```

Ezután kipróbálhatjuk a delegált ún. multicast delegate jellegét, vagyis azt, hogy több függvény is elhelyezhető benne. Ehhez létrehozunk még egy függvényt.

```
void EnglishGreeter(string name)
{
    Console.WriteLine($"Good morning, {name}!");
}
```

Ezt a második függvényt a += operátorral helyezhetjük rá a meglévő delegált példányunkra.

```
greeterFunction += EnglishGreeter;
```

Hogyha ezután a delegáltat futtatjuk, akkor mindkét ráhelyezett függvényt meg fogja hívni és mindkét függvénynek átadja neki átadott paramétert.

```
greeterFunction("Pál");
```

A példányosítás történhet egyszerűbb formában is, a new operátor elhagyható és átadható értékadás operátorral közvetlenül egy függvény.

```
Greeter greeterFunction = HungarianGreeter;
```

Hogyha bármely függvényt szeretnénk eltávolítani a delegáltból, akkor a -= operátorral tehetjük ezt meg.

```
greeterFunction -= EnglishGreeter;
```

Egy delegált létrehozásakor az előzőek alapján kötelező 1 függvényt átadni neki, ezért azt hihetjük, hogy a delegált sosem lehet null értékű. Erre oda kell figyelni, mert semmi sem akadályozza meg, hogy bárhol a kódban nullal tegyük egyenlővé vagy a -= operátorral minden függvényt eltávolítsunk belőle. Tehát minden meghívás előtt futtassunk null ellenőrzést!

```
if (greeterFunction != null)
{
    greeterFunction  ("Pál");
}
```

Ennek viszont van egy szép rövidebb módja is, ami a fenti elágazással ekvivalens.

```
greeterFunction ?.Invoke("Pál");
```

Felmerülhet a kérdés, hogy mi a helyzet a visszatérési értékkel rendelkező függvényekkel? Az alábbi kód szintaktikailag teljesen hibátlan, de a futtatása után azt tapasztaljuk, hogy a legutoljára a delegáltba betöltött függvény visszatérési értéke kerül a változóba.

```
double Add(double a, double b)
{
    return a + b;
}

double Mul(double a, double b)
{
    return a * b;
}

MathDelegate del = Add;

del += Mul;

double result = del(10, 20);

delegate double MathDelegate(double a, double b);
```

Valójában a futás közben minden függvény visszatérési értéke bele került, csak éppen felülírták egymást. A megoldáshoz végig tudunk iterálni a feliratkozott függvényeken és a visszatérési értékeket például egy listába tudjuk gyűjteni.

```
List<double> results = new List<double>();

foreach (var item in del.GetInvocationList())
{
    double? result = (item as MathDelegate)?.Invoke(10, 20);
```

```
    if (result != null)
    {
        results.Add((double)result);
    }

}
```

Ennél a megoldásnál feltűnhet, hogy a visszatérési érték kinyerésére double? típust használtunk. Ez azért fontos, mert a double érték típus, ami nem lehet null. Ellenben, hogyha nullal egyenlő a delegált, akkor a visszatérési értéke is null lesz. A double? típust nullable típusnak nevezzük, vagyis egy olyan kiterjesztett érték típus, ami null-t is felvehet. Megfelelő ellenőrzések után természetesen sima double típusra alakítható.

Utolsó módosítás: 2024. szeptember 7., szombat, 21:09

Az eddigiekben 3 lépést követtünk:

1. Elkészítettük a delegált típusát a delegate kulcsszóval
2. Elkészítettünk egy delegált példányt
3. Erre feliratkoztattunk függvényeket

Az első lépés elhagyható, mert a .NET keretrendszerben rengeteg beépített delegált található, különféle generikus paraméterezésekkel. Ezáltal nagy valószínűséggel találunk olyan beépített delegáltat, amelyre illeszkednek a referálni kívánt függvényeink.

T�pus	N�v
void()	Action | MethodInvoker
void(a)	Action<a>
void(a,b)	Action<a,b>
void(a,b,...,p) max. 16 paraméter	Action<a,b,...,p>
r()	Func<r>
r(a)	Func<a,r>
r(a,b)	Func<a,b,r>
r(a,b,...,p) max. 16 paraméter	Func<a,b,...,p,r>
void(object, EventArgs)	EventHandler
void(object, T: EventArgs)	EventHandler<T>
bool(T)	Predicate<T>
int(T,T)	Comparison<T>

Nézzünk erre 1-2 példát:

- Szeretnénk double visszatérési értékű, két int paraméterrel rendelkező függvényeket referálni

o Bal oldali oszlop, voidosak nem jók nekünk, r visszatérési érték helyére kerül a double és két int paraméterünk lesz a,b helyén --> **Func<int,int,double>**

- Szeretnénk bool visszatérési értékű, int bemenetű függvényeket referálni, amelyek pl. eldöntik, hogy adott életkorú ember beléphet-e valahova

o Bal oldali oszlop, r visszatérési érték helyére kerül a bool, és egy int paraméterünk lesz a helyén --> **Func<int,bool>**

o Bal oldali oszlop alul, bool lesz a visszatérési érték, T paraméter helyére kerül az int --> **Predicate<int>**

- Szeretnénk olyan függvényeket referálni, amelyek kapnak egy string paramétert és elmentik naplófájlba, tehát visszatérési érték nincs

o Bal oldali oszlop, void visszatérési értékűek közül lesz valamelyik, a paraméter helyére kerül a string --> **Action<string>**

Nézzük meg ezt programkódban, az előzőekben készített Add és Mul függvényeket mentsük el beépített delegáltb

```
double Add(double a, double b)
{
    return a + b;
}
```

```
double Mul(double a, double b)
{
    return a * b;
}
```

```
Func<double, double, double> mathFunctions = Add;
mathFunctions += Mul;
```

Van számtalan olyan beépített függvény, amelyek valamilyen utólag megírt logikát várnak tőlünk úgy, hogy egy beépített delegált paraméterük van. Nézzünk egy példát, egy listába vegyünk fel néhány személyt

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Job { get; set; }
     public Person(string name, int age, string job)
    {
        Name = name;
        Age = age;
        Job = job;
    }
}

List<Person> people = new List<Person>()
{
   new Person("John", 44, "ceo"),
   new Person("Jack", 33, "lead developer"),
   new Person("Sarah", 35, "UX designer"),
   new Person("Kate", 28, "frontend developer"),
   new Person("Michael", 20, "trainee")
};
```

A List<T> típus rendelkezik például egy **FindAll()** függvénnyel, amely visszaadja nekünk az összes adott feltételnek eleget tevő elemet. Delegáltak használata nélkül ilyen függvény nem is nagyon létezhetne, hiszen hogyan is adhatnánk meg neki, hogy mi pontosan a logikai feltétel. Tehát a **FindAll()** egy másik függvényt vár paraméterként, ami egy darab **Person** bemenetet vár és egy **bool**-t ad eredményül. Ezt egészen pontosan **FindAll(Predicate<T> match)** formában várja. Hogyha megnézzük a fenti táblázatban, akkor látható, hogy ez egy **bool** visszatérési értékű, **T** bemenetű függvényre illeszkedik. A **T** generikus paraméter pedig **Person**-re változik, azért mert **List<Person>-**on futtatjuk ezt a függvényt. Írjunk tehát egy ilyen vizsgáló függvényt arra, hogy a 30-évesnél fiatalabb munkavállalókat szűrje ki a **FindAll()** függvény.

```
bool YoungerThan30(Person p)
{
    return p.Age < 30;
}
```

Ezután a **FindAll()** meghívható a listán úgy, hogy a **YoungerThan30()** függvényt kapja paraméterül.

```
List<Person> youngWorkers = people.FindAll(YoungerThan30);
```

A korábbi félévekben tanult egyszerűbb programozási tételek rendelkezésre állnak a lista vagy tömb típusokon.

Listán

- **T Find(Predicate<T>** match**)** à Első T tulajdonságú elem (lineáris keresés)
- **List<T> FindAll(Predicate<T>** match**)** à Összes T tulajdonságú elem (kiválogatás)
- **bool Exists(Predicate<T>** match**)** àVan-e ilyen elem? (eldöntés)

Tömbön

- **Array.Sort(T[]** array, **Comparison<T>** comparison**)** à Tömb rendezése

Próbáljuk ki a tömb rendezését! Az interfészek témakörénél korábban tanultuk, hogy az **Array.Sort()** képes primitív típusokból álló tömböt rendezni (pl. int tömb, string tömb, stb.). Viszont hogyha egy saját típusokból álló tömböt adunk át neki paraméterül, akkor nem lesz képes, mert honnan is tudná, hogy két **Person** közül ki a kisebb. Aki fiatalabb, aki kevesebbet keres, aki alacsonyabb? Az **Array.Sort()** nem szeretné ezt kitalálni magától, tőlünk várja, hogy megmondjuk azt a logikát neki, ami alapján 2 db elem közül meg tudja mondani, hogy ki a kisebb. Emlékeztető gyanánt nézzük meg, hogy volt ez interfészekkel.

```
class Person : IComparable<Person>
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Job { get; set; }

    public int CompareTo(Person? other)
    {
        return this.Age.CompareTo(other?.Age);
    }
}
```

Ebben az esetben a célosztályon meg kellett valósítanunk az IComparable vagy IComparable<T> interfészt. Utóbbinál nagy előny, hogy az other paraméter T típusként és nem objectként érkezik, tehát nem kell kasztolni. Itt a CompareTo() függvényben vagy visszaadtuk valamelyik primitív jellemzőn meghívva a CompareTo()-t vagy ha valamilyen komplex kifejezésként szeretnénk eldönteni, hogy ki a kisebb, akkor az alábbi módon kell eredményt szolgáltatni:

- this < other -> 1
- this > other -> 1
- this == other -> 0

Nézzük meg, hogyan kellene implementálni ezt, hogyha az életkor és a név hosszának szorzata lenne az összehasonlítás alapja.

```
public int CompareTo(Person? other)
{
    if (this.Age * this.Name.Length < other?.Age * other?.Name.Length)
    {
        return -1;
    }
    else if (this.Age * this.Name.Length > other?.Age * other?.Name.Length)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

Ez csak egy kis ismétlés volt, az interfészeknél már előkerült ez korábban. A módszer jól használható, de roppant nagy hátránya az, hogy csak akkor tudunk vele élni, hogyha az osztályt képesek vagyunk utólag módosítani. Mi van akkor, hogyha egy keretrendszeri vagy egyéb osztálykönyvtárból betöltött típusok tömbjét akarjuk rendezni? Például egy listában vannak dátumok és az alapján akarjuk őket rendezni, hogy hol kisebb a másodperc értéke. (Nyilván ennek a példának semmi értelme a gyakorlatban)

```
DateTime[] dates = {
    DateTime.Parse("2022.10.23 12:34:23"),
    DateTime.Parse("2021.02.11 08:10:53"),
    DateTime.Parse("2023.05.27 22:31:37"),
    DateTime.Parse("2020.01.02 10:00:01"),
    DateTime.Parse("2021.12.24 18:20:30")
};
Array.Sort(dates);
```

A rendezés után a dátumok növekvő sorrendbe állnak. Ám ha más rendező logikát szeretnénk, akkor sajnos a **DateTime** típust nem tudjuk módosítani és nem tudjuk benne a **CompareTo()** függvényt átírni. De hogyha megvizsgáljuk az **Array.Sort()** függvényt, akkor van egy ilyen túlterhelése: **Array.Sort(T[]** array, **Comparison<T>** comparison**).** Vagyis átadhatunk neki egy **T tömböt** és egy **Comparison<T>** delegáltat. A fenti táblázatból láthatjuk, hogy így a paraméter helyén átadható egy olyan függvény, amely **int** visszatérési értékű és két **T** paramétere van. Vagyis **DateTime** tömb esetében két **DateTime** paramétere.

```
Array.Sort(dates, DateComparer);
int DateComparer(DateTime a, DateTime b)
{
    return a.Second.CompareTo(b.Second);
}
```

Átadva a függvényt, a dátumok a másodperc értéke szerint lesznek növekvő sorrendben.

Utolsó módosítás: 2024. szeptember 8., vasárnap, 15:27

Ebben a részben azt nézzük meg, hogyan lehet a delegáltak használatát a lehető legjobban lerövidíteni. Az előzőekben megnéztük, hogy nem szükséges mindig saját delegált típust létrehoznunk, ugyanis a keretrendszerben biztosan találunk valami olyan beépített delegált típust, amelynek a generikus paraméterei helyére a konkrét típusokat behelyettesítve kapunk egy kész megoldást. Megnéztük továbbá, hogy listákon és tömbökön vannak olyan beépített algoritmusok, amelyek beépített delegálton keresztül, kívülről várnak tőlünk valamilyen logikát. Erre egy példa.

Adott egy lista, feltöltve személyekkel:

```
List<Person> people = new List<Person>()
{
    new Person("John", 44, "ceo"),
    new Person("Jack", 33, "lead developer"),
    new Person("Sarah", 35, "UX designer"),
    new Person("Kate", 28, "frontend developer"),
    new Person("Michael", 20, "trainee")
};
```

Hogyha ki akarjuk szűrni a 30 évnél fiatalabb munkavállalókat, akkor írunk erre egy eldöntő logikát, amelyet átadunk delegálton keresztül a **FindAll()** függvénynek.

```
bool YoungerThan30(Person p)
{
    return p.Age < 30;
}
List<Person> youngWorkers = people.FindAll(YoungerThan30);
```

A megoldás jól működik, ám a **YoungerThan30()** tipikusan egy egyszer használatos függvény, amely elszennyezi a kódot. Korábban megtanultuk, hogy azt érdemes függvénybe kitenni, amit gyakran felhasználunk és minél inkább generikus megoldás. Ez a függvény tipikusan nem az. Lehetőségünk van a függvényt helybe kifejteni a **delegate** kulcsszóval. Ez nevezzük **névtelen függvénynek**. Azért névtelen függvény, mert nem tudunk rá máshol hivatkozni.

```
List<Person> youngWorkers = people.FindAll(delegate(Person p)
{
    return p.Age < 30;
});
```

Látható, hogy így a **FindAll()**-nak kvázi helyben beírtuk a szelekciós logikát. Ám ezt még jobban le tudjuk rövidíteni **Lambda-kifejezések** használatával.

```
List<Person> youngWorkers = people.FindAll(p => p.Age < 30);
```

Ez gyakorlatilag a névtelen függvény rövidítése. A **delegate** kulcsszót a **=>** operátor váltja fel. A **Person** típust teljesen felesleges kiírni, hiszen úgyis csak olyan függvény adható át, ami egy **Person** paramétert várt. A **return** kulcsszó is felesleges, a **=>** operátortól jobbra egy **bool** kifejezést kell írnunk. Így a **=>** bal oldala a bemenet, jobb oldala a kimenet.

Nézzünk egy példát az Array.Sort() hívására is névtelen függvénnyel:

```
Person[] people =
{
    new Person("John", 44, "ceo"),
    new Person("Jack", 33, "lead developer"),
    new Person("Sarah", 35, "UX designer"),
    new Person("Kate", 28, "frontend developer"),
    new Person("Michael", 20, "trainee")
};
```

```
Array.Sort(people, delegate (Person a, Person b)
{
    return a.Age.CompareTo(b.Age);
});
```

Itt is írhatunk komplexebb logikát

```
Array.Sort(people, delegate (Person a, Person b)
{
    if (a.Age * a.Name.Length < b.Age * b.Name.Length)
    {
        return -1;
    }
    else if (a.Age * a.Name.Length > b.Age * b.Name.Length)
    {
        return 1;
    }
    else
    {
        return 0;
    }
});
```

Ugyanez Lambda kifejezéssel. A különbség itt az, hogy több paraméteres delegált esetén a paramétereket **zárójelbe kell írni**!

```
Array.Sort(people, (a, b) => a.Age.CompareTo(b.Age));
```

És a komplexebb logika is felírható ugyanígy:

```
Array.Sort(people, (a, b) =>
{
    if (a.Age * a.Name.Length < b.Age * b.Name.Length)
    {
        return -1;
    }
    else if (a.Age * a.Name.Length > b.Age * b.Name.Length)
    {
        return 1;
    }
    else
    {
        return 0;
    }
});
```

Lambda esetén megkülönböztetünk elméletben **kifejezés lambdát** (Expression Lambda) és **kijelentés lambdát** (Statement Lambda). Előbbinek van visszatérési értéke, utóbbinak nincs.

Kifejezés lambda

```
List<Person> youngWorkers = people.FindAll(p => p.Age < 30);
```

Kijelentés lambda

```
people.ForEach(t => Console.WriteLine(t.Name));
```

Amikor delegáltakkal dolgozunk, akkor belefuthatunk az **Outer Variable Trap** hibába. Értsük meg a jelenséget az alábbi kód futtatásával:

```
Action numWriter = null;
for (int i = 0; i < 10; i++)
{
    numWriter += () => { Console.WriteLine(i); };
}
numWriter();
```

Amit ebben a kódban szeretnénk az az, hogy készítünk egy **numWriter** nevű delegáltat. Egy ciklussal belehelyezünk 10 darab függvényt és a ciklusváltozó függvényében mondjuk meg, hogy a konkrét függvények mit írjanak ki. Vagyis elhelyezünk egy 0-t kiíró függvényt, egy 1-et kiíró függvényt, stb. Az elvárt kimenetünk 0,1,2,3,4,5,6,7,8,9 lenne, de a futtatás után 10,10,10,10,10,10,10,10,10,10 kimenetet látunk a konzolon. Ez azért van, mert a névtelen függvényben/lambdában felhaszn külső változó cím szerint adódik át. Így a ciklus lefutása végén az i értéke 10 lesz és minden függvény ezt az értéket fogja kiírni. A megoldás az lehet, hogy a cikluson belül minden iterációban létrehozunk egy változót még a delegálton kívül. Ebbe érték szerint átmásoljuk a ciklusváltozót. És a delegáltak a saját iterációjukban létrehozott változóra fognak cím szerint hivatkozni.

```
Action szamkiiro = null;
for (int i = 0; i < 10; i++)
{
    int k = i;
    szamkiiro += () => { Console.WriteLine(k); };
}
szamkiiro();
```

Utolsó módosítás: 2024. szeptember 8., vasárnap, 15:34

Az előző leckében megismertük a delegáltak használatát, konkrét gyakorlati feladatot néztünk arra, hogy egy generikus tároló osztály kívülről kapja meg azt a függvényt, amit meg kell hívnia. Viszont felmerül egy probléma a delegáltakkal kapcsolatban. Akkor tudunk rá kívülről feliratkozni, hogyha

- publikus a delegált
- rendelkezésre áll egy feliratkoztató metódus

Az előző leckében megismert Storage<T> osztályt elevenítsük fel egy kicsit leegyszerűsített formában.

```
public class Storage<T>
{
    T[] array;
    int pointer;
     public Storage(int size)
    {
        array = new T[size];
    }
    public void Add(T item)
    {
        if (pointer < array.Length)
        {
            array[pointer++] = item;
        }
        else
        {
            throw new Exception("Storage is full");
        }
    }
}
```

Itt most szeretnénk kivétel dobása helyett egy delegálton keresztül jelezni a külvilág számára, ha a tároló betelt. Ezt egy publikus delegálttal megoldhatjuk könnyen.

```
public class Storage<T>
{
    T[] array;
    int pointer;
    public delegate void StorageSign();
    public StorageSign storageFull;
    public Storage(int size)
    {
        array = new T[size];
    }
    public void Add(T item)
    {
        if (pointer < array.Length)
        {
            array[pointer++] = item;
        }
        else
        {
            storageFull?.Invoke();
        }
    }
}
```

A Storage-t felhasználó osztály kívülről a storageFull delegáltra fel tud iratkozni. Természetesen ez lehetett volna Action típusú, de most maradjunk saját delegált készítésénél. A feliratkozás kívülről így történne:

```
Storage<string> items = new Storage<string>(5);
items.storageFull += () => Console.WriteLine("A tároló megtelt!");
```

Láthatjuk, hogy itt helyesen a += operátor van használva, de ezt megtehettük volna az értékadás (=) operátorral is.

```
items.storageFull = () => Console.WriteLine("A tároló megtelt!");
```

Illetve sok más dolgot is megtehetünk a publikus delegálttal. Például letörölhetjük a más osztályok által feliratkozott függvényeket belőle vagy akár le is futtathatjuk.

```
items.storageFull = null;
items.storageFull();
```

Mindkét műveletre van lehetőségünk, ami felvet egy **nagyon komoly biztonsági problémát** . Képzeljünk el egy hatalmas szoftvert, amiben van egy központi adattároló rész, hasonló, mint ez a Storage<T> osztály. Például egy vállalatirányítási rendszerben egy raktárkezelő osztály tartja nyilván a nyersanyag (pl: szén) mennyiségét. Amikor betelik a raktár, akkor egy delegálttal jelzi. A nyersanyagok tárolását végző osztályt rengeteg alrendszer használja (pl. beszerzés, vezetői kimutatás, eladás, stb.). Ez kódszinten valahogy így nézne ki.

```
class Sourcing
{
    Storage<Carbon> carbonStorage;
    public Sourcing(Storage<Carbon> carbonStorage)
    {
        this.carbonStorage = carbonStorage;
        carbonStorage.storageFull += () => Console.WriteLine("Carbon storage is empty!");
    }
}
 class Management
{
    Storage<Carbon> carbonStorage;
    public Management(Storage<Carbon> carbonStorage)
    {
        this.carbonStorage = carbonStorage;
        carbonStorage.storageFull = () => Console.WriteLine("Carbon storage is empty!");
    }
}
```

Vegyük észre, hogy a **Sourcing** osztályt fejlesztő csapat/személy megfelelően a **+=** operátorral iratkozott fel, míg a **Management** osztály fejlesztő csapat/személy teljesen véletlenül a **+** jelet kihagyta. **Hatalmas kárt okozhatnak ezzel**, mert a raktár telítettségéről a **Sourcing** nem kap értesítést és szerzi be folyamatosan az újabb nyersanyagot, a vezetőség kimutatásában pedig nem érti, hogy ha tele a raktár, akkor a **Sourcing** miért indít újabb és újabb beszerzéseket.

Természetesen mondhatjuk, hogy hibás kódot írtak, miért nem nézték át rendesen? Gondoljunk bele, hogy a **Management** osztály fejlesztői hibáztak, de náluk tökéletes működött a jelzés. A **Sourcing** osztály fejlesztői pedig nem nyúltak a szoftverhez 1 hónapja és nem jön jelzés 4 napja. Ők biztosan nem írták el. Nagyon kellemetlen helyzet.

A helyzet megoldható, hogyha minden delegálthoz készítünk feliratkoztató és leiratkoztató metódusokat. Az első leckében a gyakorló feladatnál erre használtuk az AddTransformer() függvényt. És a delegált nem volt publikus. Leiratkoztató metódus abban a példában nem volt.

A korrekt megoldás az **események** (**event**) használata. Az event egy speciális delegált, amelyre:

- Kívülről csak fel és leiratkozni tudunk, nem tudjuk nullal egyenlővé tenni
- Csak akkor tudunk leiratkoztatni egy függvényt, ha van rá referenciánk (saját függvényünk)
- A létrehozó osztályon belülről lehet csak elsütni

Eventek használatával az alábbira módosul a kód:

```
public event StorageSign storageFull;
```

A delegált példánynál bekerült az event kulcsszó. Az alábbi kódokat nem tudjuk futtatni kívülről többé. Le sem fordul egyik sor sem.

```
items.storageFull = () => Console.WriteLine("A tároló megtelt!");
items.storageFull = null;
items.storageFull();
```

Kizárólag a += és -= operátor működik:

```
items.storageFull += () => Console.WriteLine("A tároló megtelt!");
```

Utolsó módosítás: 2024. szeptember 8., vasárnap, 15:39

Amikor a beépített delegáltakat tanultuk, akkor a jól ismert táblázatunkban volt két ilyen sor:

Típus	Név
void(object, EventArgs)	EventHandler
void(object, T: EventArgs)	EventHandler<T>

Nagyon nem tértünk ki rájuk, de itt az eseménykezelésnél nyernek értelmet. A .NET frameworkben van egy egységes konvenció események kezelésére. Természetesen a programunk működik akkor is, hogyha nem ezt a konvenciót követjük. Pusztán jobb lesz ezáltal a kód olvashatósága és kicsit egyértelműbbé válnak az eseményparaméterek.

Nézzünk egy példát, ahol hagyományos módon futtatunk egy eseményt és még az esemény bekövetkeztekor információkat is küldünk a hívó környezet számára.

```
public class Storage<T>
{
    List<T> list = new List<T>();
    public delegate void InsertSign(T item, bool start);
    public event InsertSign Inserted;

    public void Add(T item)
    {
        Random r = new Random();
        if (r.Next(0,2) == 1)
        {
            list.Add(item);
            Inserted?.Invoke(item, false);
        }
        else
        {
            list.Insert(0, item);
            Inserted?.Invoke(item, true);
        }
    }
}
```

A példában az látható, hogy a Storage itt egy lista, aminek véletlenszerűen vagy az elejére, vagy a végére szúrjuk be az új elemet. A beszúrást követően az eseményen keresztül elküldjük, hogy melyik elemről van szó és azt is, hogy előre lett-e beszúrva (itt most bool-ként).

Ez a példa önmagában rá tud mutatni arra, hogy miért fontos nekünk az eseménykezelés. Ezeket az adatokat, hogy mit és hova szúrtunk be alapvetően visszakaphatnánk visszatérési értékként az Add metódustól. Gond nélkül összecsomagolhatnánk a **T** item és **bool** start változókat egy wrapper osztályba. De az eseménykezelés azért is fontos nekünk, mert lehet, hogy az Add()-ot egy bizonyos alrendszer hívja, de a beszúrás tényéről több másik alrendszer is szeretne információt kapni, ezért feliratkoztak az eseményre. Hogyha visszatérési értékként csak a hívó kapná vissza az adatokat, akkor a többi információt kérő osztály nem tudna hogyan értesülni.

Ezt a példát átírhatjuk a táblázat 2. sora szerinti generikus **EventHandler<T>** delegáltra. Ekkor el kell készítenünk egy **EventArgs** leszármazottat ami egy **wrapper** osztály lesz és az esemény részleteit tartalmazza.

```
public class InsertEventArgs<T> : EventArgs
{
    public enum Direction
    {
        start, end
    }
    public Direction InsertDirection { get; }
    public T Item { get; }
     public InsertEventArgs(InsertEventArgs<T>.Direction insertDirection, T item)
    {
        InsertDirection = insertDirection;
        Item = item;
    }
}
```

Ez tehát egy olyan wrapper osztály, ami tartalmazza az esemény részleteit csak olvasható formában. Módosítsuk az event típusát **EventHandler<T>-**re.

```
public class Storage<T>
{
    List<T> list = new List<T>();
    public event EventHandler<InsertEventArgs<T>> Inserted;
    public void Add(T item)
    {
        Random r = new Random();
        if (r.Next(0,2) == 1)
        {
            list.Add(item);
            Inserted?.Invoke(this, new InsertEventArgs<T>(InsertEventArgs<T>.Direction.end, item));
        }
        else
        {
            list.Insert(0, item);
            Inserted?.Invoke(this, new InsertEventArgs<T>(InsertEventArgs<T>.Direction.start,
item));
        }
    }
}
```

Az eseményre való feliratkozás pedig így módosul:

```
items.Inserted += (obj, args) => Console.WriteLine(args.Item + " inserted at: " +
args.InsertDirection);
```

Utolsó módosítás: 2024. szeptember 8., vasárnap, 15:42

Info

Moodle használati útmutató

GY.I.K.

Kapcsolat

1034 Budapest, Doberdó út 6.

Telefon : +36 1 666-5741

E-mail : moodlesupport@uni-obuda.hu

A fájlok és könyvtárak kezelése elég fontos ismeret. Szinte nincs olyan alkalmazás, ahol ne kellene fájlból olvasni, írni, esetleg mappákat bejárni és kinyerni a bennük található fájlok listáját. Ebben a leckében most megnézzük elsősorban azt, hogy fájlokat hogyan tudunk kezelni.

A legegyszerűbb megoldás, hogyha a projektünk /bin/Debug/net7.0 mappájában az exe fájl mellett található egy tetszőleges szöveges fájl. Ennek sorait be tudjuk könnyedén olvasni egy string tömbbe.

```
string[] lines = File.ReadAllLines("data.txt");
```

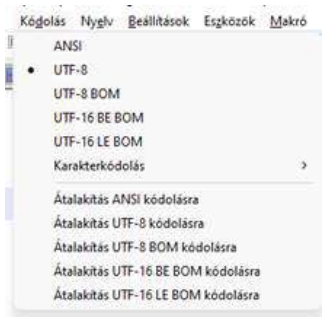
Alapesetben .NET 8-ben a **File** osztály minden függvénye **UTF-8 kódolású** fájlokat feltételez. Korábbi .NET verziókban, például .NET Framework 4.6-ban még az ANSI kódolás volt az alapértelmezett. Ha nem írjuk ki a 2. paraméterben, akkor az ekvivalens lesz ezzel a hívással:

```
string[] lines = File.ReadAllLines("data.txt", Encoding.Default);
```

Hogyha más kódolással készült a fájl, akkor szükséges megadni a 2. paraméterben:

```
string[] lines = File.ReadAllLines("data.txt", Encoding.Unicode);
```

Hogyha nem vagyunk biztosak egy fájl karakterkódolásában, akkor pl. **Notepad++** programmal megnyitva a Kódolás menüben látszik, illetve itt át is alakítható UTF-8-ra.



Hogyha nem szeretnénk soronként feldolgozni a szöveget, akkor lehetőség van a teljes tartalmát egy darab stringbe betölteni. Ez akkor lehet hasznos, hogyha például keresni szeretnénk a fájlban, vagy adott részeket cserélni/törölni (pl. Replace()).

```
string content = File.ReadAllText("data.txt");
```

A fájlok elmentése hasonlóan történhet, vagy egy string tömböt írunk ki egy fájlba, ahol a tömb elemei külön sorokba kerülnek, vagy egy darab stringet írunk ki egy fájlba.

```
string[] lines =
{
    "John",
    "Jack",
    "Kate",
    "Sawyer"
};

File.WriteAllLines("data.txt", lines);
```

```
string content = "This is a secret information";
File.WriteAllText("data.txt", content);
```

Fontos megjegyezni, hogy ezek a kiíró műveletek a teljes fájlt felülírják. Lehetőségünk van arra is, hogy egy meglévő fájl sorait bővítsük a kívánt tartalommal, vagyis hozzáfűzzünk a fájlhoz. Ez persze megoldható a fájl beolvasásával, a tömb bővítésével és újbóli kiírásával, de erre nincs szükség, hogyha az Append.. kezdetű függvényeket használjuk.

```
string[] lines =
{
    "John",
    "Jack",
    "Kate",
    "Sawyer"
};

File.AppendAllLines("data.txt", lines);
```

```
string content = "This is a secret information";
File.AppendAllText("data.txt", content);
```

Ezek a beolvasási módszerek egyszerűek, viszont vagy egy nagy hátrányuk. A teljes fájl beolvasásra kerül a memóriába. Szélsőséges példa, de hogyha van egy 10 gigabájt méretű naplófájl, amiben keresni szeretnénk C#-al és a memóriánk 4 gigabájt, akkor bizony az előző megoldás nem fog működni, mert **MemoryOutOfRange** exceptiont kapunk a beolvasáskor. Illetve ha csak keresnénk valamit egy fájlban, akkor felesleges kivárni azt az időt, amíg a teljes fájl betöltődik a memóriába. Van egy módszer, amivel soronként tudjuk a fájlt végigiterálni és csupán annyi memóriát használunk fel, amennyi az aktuális sor tárolásához szükséges.

```
StreamReader sr = new StreamReader("data.txt");
while (!sr.EndOfStream)
{
    Console.WriteLine(sr.ReadLine());
}
sr.Close();
```

A módszerben az **sr** objektum lesz az, ami végiglépked a fájl sorain. Addig megyünk egy **while** ciklussal, amíg a fájl végére nem érünk **(!sr.EndOfStream)**. Minden lépésben kiolvassuk az adott sort az **sr.ReadLine()**-al és ez a **ReadLine()** fogja a kurzort a következő sorba léptetni.

Az **sr.Close()** ami bontja a kapcsolatot a fájllal. Amíg ezt nem tesszük meg, addig a fájlt megnyitva tartja a C# program. Megoldhatjuk ezt másképp is. Egy **using** blokkba zárva a **StreamReader**-t a blokk végén automatikusan törli az objektumot a keretrendszer és a kapcsolat megszakad.

```
using (StreamReader sr = new StreamReader("data.txt"))
{
    while (!sr.EndOfStream)
    {
        Console.WriteLine(sr.ReadLine()); ;
    }
}
```

A fájlba való kiírásra is van egy másik módszer: **StreamWriter** használata. Hogyha folyamatosan jönnek adatok, amelyeket egy fájlba kell menteni, akkor ezzel a módszerrel úgy tudunk a fájlba írni, mintha a konzolra írnánk.

```
Random r = new Random();
using (StreamWriter sw = new StreamWriter("numbers.txt"))
{
    for (int i = 0; i < 100; i++)
    {
        sw.WriteLine(r.Next(1, 101));
    }
}
```

A **StreamWriter** is megnyitható úgy, hogy nem felülírja a fájlt, hanem hozzáfűzi az adott tartalmat. A második paraméter egy **bool** append paraméter. Ha igazra állítjuk, akkor hozzáfűzés történik.

```
StreamWriter sw = new StreamWriter("numbers.txt", true);
```

Utolsó módosítás: 2024. szeptember 13., péntek, 13:31

Elektronikus és Digitális Tananyagok Iroda

Info

Moodle használati útmutató

GY.I.K.

Kapcsolat

1034 Budapest, Doberdó út 6.

Telefon : +36 1 666-5741

E-mail : moodlesupport@uni-obuda.hu

Most megnézzük azt, hogy könyvtárakat és különböző elérési utakat hogyan tudunk kezelni. Az előző leckében megnéztük, hogyha a **\bin\Debug\net7.0** mappában van a fájlunk az exe fájl mellett, akkor a beolvasása egyszerű. Hogyha nem itt van a fájl, akkor a művelet körülményesebb.

Hogyha ezen belül egy alkönyvtárban található a fájl, akkor megadhatjuk, hogy a jelenlegi könyvtártól relatívan kiindulva még egy adott mappát lefele kell mennünk. Mind a négy megoldás képes a fájlt beolvasni. A .NET 7 verzióban már mindkét irányba dőlő perjel elfogadott. Korábban ez nem volt így, ugyanis windowson \ ezt a per jelet, linuxon és macen / ezt a perjelet használjuk. De .NET 7 alatt már rájön a környezet, hogyan értelmezze a beírtakat.

```
string[] lines = File.ReadAllLines("files/data.txt");
string[] lines = File.ReadAllLines("./files/data.txt");
string[] lines = File.ReadAllLines(@"files\data.txt");
string[] lines = File.ReadAllLines(@".\files\data.txt");
```

A legbiztosabb megoldás azonban, hogyha a **Path.Combine()** függvény segítségével építjük össze az utat. Ez a függvény majd beszúrja az adott operációs rendszernek megfelelő jelölés az elérési útba.

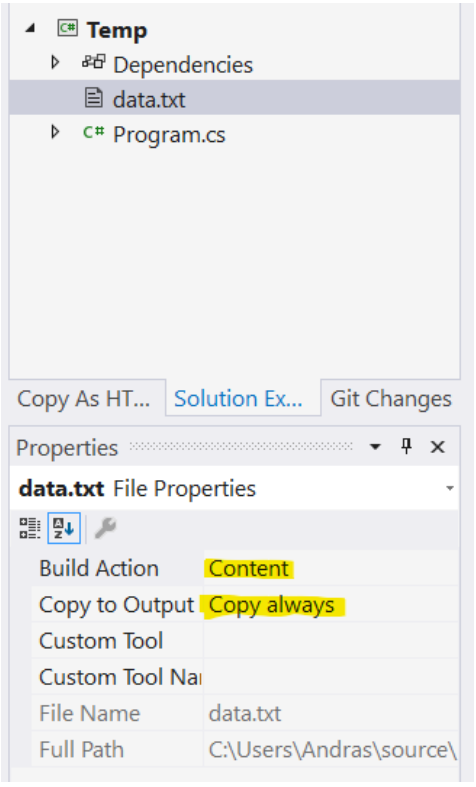
```
string[] lines = File.ReadAllLines(Path.Combine(Directory.GetCurrentDirectory(), "files", "data.txt
```

Hogyha a **\bin\Debug\net7.0** mappától feljebb van a keresett fájl, például a projekt gyökerében, akkor a .. jelöléssel tudunk feljebb navigálni az elérési útban.

```
string[] lines = File.ReadAllLines(@"..\..\..\data.txt");
```

Viszont vegyük figyelembe azt, hogy miután az alkalmazás lefordításra került és a **Release** mappából az exe fájlt átadjuk valakinek futtatásra a forráskód nélkül és ő elhelyezi a C:\ meghajtó gyökerében, ott már nem fog tudni a programja 3 könyvtárral feljebb keresgélni. Tehát sose hivatkozzunk a project könyvtárban lévő fájlra.

Hogyha viszont egy olyan programot írunk, amely pl. egy config fájlból olvas be adatokat, akkor ezt ne a \bin\Debug\net7.0 könyvtárban hozzuk létre, mert egyrészt a Release-be nem kerül bele, másrészt egy Clean Solution + Rebuild Solution törli ezeket a könyvtárakat és újraépíti. A korrekt megoldás elhelyezni a project könytárban, kijelölni a Visual Studioban és a Properties ablakban beállítani az alábbiakat:



Ez lényegében azt jelenti, hogy a project gyökerében lévő data.txt fájl minden egyes buildeléskor bemásolódik a **\bin\Debug\net7.0** mappába és onnan könnyen elérhető lesz.

Fájlokat beolvashatunk **abszolút elérési úttal** is:

```
string[] lines = File.ReadAllLines(@"c:\work\data.txt");
```

Ez viszont nem követendő gyakorlat így egy-az-egyben! A forráskódba sose égessünk be abszolút elérési utakat. Ugyanis a saját gépünkön lévő C:\users\Sanyi\.. mappa biztosan nem létező út bárki másnak a gépén. Viszont akkor lehet létjogosultsága, hogyha egy olyan programot írunk, amely egy felhasználó által megadott könyvtárból olvas fájlokat vagy oda ment fájlokat. Ekkor viszont kívülről kérjük be az abszolút elérési utat.

A példakódban bekérünk egy mappanevet (pl: C:\suli\oktatovideok), majd ide egy video.txt-be beleírjuk a megadott tartalmat.

```
Console.Write("Enter download location: ");
string path = Console.ReadLine();
File.WriteAllText(Path.Combine(path, "video.txt"), "This is a secret video");
```

Felmerülhet olyan igény, hogy egy könyvtárban lévő fájlok nevét szeretnénk listaként megkapni. Például készítünk egy alkalmazást, amely fényképeket tölt fel felhőbe. A felhasználó megadja a mappát, ahol a fényképek vannak és egyenként szeretnénk a fájlokon végiglépkedni és valamit csinálni velük

```
Console.Write("Enter photo album location: ");
string path = Console.ReadLine();
string[] fileNames = Directory.GetFiles(path);
```

Itt ráadásul még egy keresési mintát is megadhatunk, nézzünk erre néhány példát

```
//Csak a jpg kiterjesztésű fájlok
string[] fileNames = Directory.GetFiles(path, "*.jpg");

//Csak a PIC kezdetű fájlok
string[] fileNames = Directory.GetFiles(path, "PIC*");

//Csak a PIc kezdetű és jpg kiterjesztésű fájlok
string[] fileNames = Directory.GetFiles(path, "PIC*.jpg");
```

Ugyanígy az összes almappa nevét is megkaphatjuk

```
Console.Write("Enter photo album location: ");
string path = Console.ReadLine();
string[] directoryNames = Directory.GetDirectories(path);
```

Lehetőségünk van ezt úgy is meghívni, hogy nem csak a közvetlen almappákat, hanem azok almappáit és így tovább tetszőleges mélységben megkapjuk

```
Directory.GetDirectories("files", "*", SearchOption.AllDirectories);
```

Utolsó módosítás: 2024. szeptember 13., péntek, 13:39

Info

Moodle használati útmutató

GY.I.K.

Kapcsolat

1034 Budapest, Doberdó út 6.

Telefon : +36 1 666-5741

E-mail : moodlesupport@uni-obuda.hu

Miután megtanultuk azt, hogy hogyan tudjuk a fájlokat elérni és kikeresni adott mappákból, nézzük meg, hogy milyen műveleteket tudunk velük végezni.

Egy fájlt létrehozhatunk üres tartalommal. Ennek nem biztos, hogy sok gyakorlati haszna van, ugyanis a fájlba író műveletek létre is hozzák a fájlt, ha az nem létezik.

```
File.Create("data.txt");
```

Viszont könyvtárak esetében ennek már jóval több értelme van. Ugyanis ha egy olyan mappába akarunk létrehozni fájlt, ami nem létezik, akkor **Exceptiont** kapunk. Például ha a C:\munka\Attila\suli\data.txt-t akarjuk létrehozni és sem a „munka”, sem az „Attila”, sem a „suli” mappa nem létezik, akkor ezeket nem hozza létre a .NET, hanem mind a hármat külön el kell készítenünk.

```
Directory.CreateDirectory("files");
```

Lehetőségünk van ellenőrizni fájlok és mappák meglétét is

```
bool exists = File.Exists("data.txt");
bool exists = Directory.Exists("files");
```

Fájlokat áthelyezhetünk és akár egyben át is nevezhetünk

```
File.Move("data.txt", Path.Combine(Directory.GetCurrentDirectory(), "files", "mydata.txt"));
```

De lehetőségünk van csak egy adott helyre másolni (itt is megadható új név)

```
File.Copy("data.txt", Path.Combine(Directory.GetCurrentDirectory(), "files", "data.txt"));
```

Az áthelyezés műveletet könyvtárakkal is elvégezhetjük, viszont itt másolás nem áll rendelkezésre. Ezt magunknak kell rekurzívan megírni.

```
Directory.Move("files", Path.Combine(Directory.GetCurrentDirectory(), "items", "files"));
```

Fájlok esetén elérhetünk különböző attribútumokat is, például, hogy mikor volt létrehozva a fájl, mikor használták utoljára és mikor írtak bele utoljára.

```
DateTime creation = File.GetCreationTime("data.txt");
DateTime lastAccess = File.GetLastAccessTime("data.txt");
DateTime lastWrite = File.GetLastWriteTime("data.txt");
```

Ugyanez működik könyvtárakkal is

```
DateTime creation = Directory.GetCreationTime("items");
DateTime lastAccess = Directory.GetLastAccessTime("items");
DateTime lastWrite = Directory.GetLastWriteTime("items");
```

Utolsó módosítás: 2024. szeptember 13., péntek, 13:41

Elektronikus és Digitális Tananyagok Iroda

Info

Moodle használati útmutató

GY.I.K.

Kapcsolat

1034 Budapest, Doberdó út 6.

Telefon : +36 1 666-5741

E-mail : moodlesupport@uni-obuda.hu

Ha fájlokkal és mappákkal akarunk dolgozni, akkor a **File** osztály és **Directory** osztály függvényein kívül lehetőségünk van egy objektum-orientált megközelítést is használni. Létrehozhatunk **FileInfo** és **DirectoryInfo** objektumokat, a konstruktorban átadva az elérési útját. Innentől minden egyes fájlt vagy mappát önálló objektumként kezelhetünk és az adott objektumok tagfüggvényein keresztül érhetjük el a kívánt funkciókat.

Kinyerhetjük stringként a kiterjesztését (ezt **FileInfo** nélkül csak string tördeléssel tudtuk volna kiszedni)

```
string ext = fi.Extension;
```

Lekérhetjük a teljes elérési útját:

```
string fullpath = fi.FullName;
```

Vagy csak magát a fájl nevét elérési út és kiterjesztés nélkül:

```
string filename = fi.Name;
```

Egy adott fájlról lekérhetjük az őt tartalmazó könyvtár referenciáját:

```
DirectoryInfo di = fi.Directory;
```

És egy könyvtárról is lekérhetjük az őt tartalmazó könyvtár referenciáját

```
DirectoryInfo parent = di.Parent;
```

Egy könyvtárban lévő fájlokat és alkönyvtárakat is elkérhetjük objektumként

```
FileInfo[] files = di.GetFiles();
DirectoryInfo[] dirs = di.GetDirectories();
```

Ezeken túlmenően az összes **Create**, **Exists**, **Move**, **Copy**, stb. függvény elérhető, illetve a hozzáférési idők is **DateTime** formában.

Utolsó módosítás: 2024. szeptember 13., péntek, 13:47

GY.I.K.

Kapcsolat

1034 Budapest, Doberdó út 6.

Telefon : +36 1 666-5741

E-mail : moodlesupport@uni-obuda.hu

Ebben a leckében majd a nyelvbe ágyazott lekérdezésekkel (Language Integrated Query – LINQ) fogunk megismerkedni. Ebben a fejezetben most még nem beszélünk róla, hogy mi ez és hogyan működik. Előkövetelményként meg kell ismernünk néhány nyelvi elemet, amelyre a LINQ erősen épít. Ezek a LINQ-tól teljesen függetlenül használható nyelvi elemek.

Névtelen típusok

Hogyha C#-ban értékadás operátort használunk, akkor az értékadás bal oldalára ki szoktuk írni a típust.

```
string result = Console.ReadLine();
```

Ez a lépés valójában teljesen felesleges, mert a fordító pontosan tudja, hogy a Console.ReadLine() stringet ad vissza. Nem is lehet egy függvénynek több fajta visszatérési értéke. Ezért megtehetjük azt C#-ban, hogy egyszerűen egy var kulcsszóval jelöljük meg a célváltozót.

```
var result = Console.ReadLine();
```

Hogyha rámutatunk az egérrel, akkor látjuk, hogy a környezet felismeri, hogy ez a result valójában egy string? típus. De megspórolhatjuk a típus kiírását, mivel egyértelmű. Ezt megtehetjük ebben az esetben is:

```
List<string> list = new List<string>();
var list = new List<string>();
```

A list szóra mutatva az egérrel látjuk, hogy ő egy List<string>? típus.

Az alábbi eset is működhet, ilyenkor az értékadás operátor a jobbra lévő típust veszi alapul, nem tesz fel kérdéseket, hogy szeretnénk-e valamely ősével refereálni.

```
class Person { }
class Manager : Person { }

var john = new Manager();
```

A var kulcsszóval amúgy már találkozhattunk, hogyha foreach-et snippetből írjuk be. Itt is gyakorlatilag annyi történt, hogy a string típus egyértelmű, ez var kulcsszóval helyettesíthető.

```
List<string> names = new List<string>();
foreach (var item in names)
{

}
```

Nagyon fontos megérteni, hogy a var nem egy dinamikus változótípus! Ez csupán egy kulcsszó, amivel rövidíthetjük a kódot. De a változó továbbra is erősen típusos és nem adhatunk neki más típusú értéket egy sorral lejjebb.

```
var peter = new Person();
peter = 32; //hiba!!!

class Person
{

}
```

Illetve olyan eset sem fordulhat elő, hogy nem kap értéket:

```
var peter; //hiba!!!
```

És nem állhat függvény visszatérési értékként sem:

```
public var Counter(int number) //hiba!!!
{
    return number * 10;
}
```

Névtelen osztályok

Lehetőségünk van névtelen osztályok létrehozására is. Ezek egyszer használatos osztályok, a típusuk **anonymus**. Arra való, hogy eltároljunk benne valamilyen eredményt és rögtön fel is dolgozzuk. Nem állhat paraméterként, visszatérési értékként, tehát az adott függvényből gyakorlatilag nem tudjuk továbbítani sehova.

```
var peter = new
{
    RealName = "Peter Paker",
    HeroName = "Spider-Man"
};
```

Kiegészítő metódusok (Extension Methods)

Meglévő típusokhoz képesek vagyunk futásidőben extra függvényeket hozzáadni. Pl. a DateTime típus kódjába értelemszerűen nem tudunk beleírni, mert egy keretrendszeri lefordított bináris. De egy technikával bele tudunk tenni új függvényt.

```
DateTime dt = DateTime.Parse("2020.03.17 12:20:30");
Console.WriteLine(dt.ToCustomFormat());

public static class DateTimeExtender
{
    public static string ToCustomFormat(this DateTime source)
    {
        return $"{source.Year}*{source.Month}*{source.Day}";
    }
}
```

Készítünk egy statikus osztályt, amelyet nem kell sehol meghívni, csak szerepeljen a projektünkben. A kívánt függvényt statikusan helyezzük el benne, a visszatérési érték típusát határozzuk meg. Egy darab paramétere legyen **this** kulcsszóval, ahova az objektum helyettesítődik be (számunkra a híváskor a 2. sorban ez nem látszik). És a függvényben megírjuk a logikát amit szeretnénk a **DateTime** példánnyal csinálni.

IEnumerable<T> interfész

Korábbi tanulmányok során már előkerült ez az interfész. A beépített gyűjtemény típusok mind megvalósítják. A **foreach** függvény ennek segítségével tud működni. Hogyha készítünk egy saját gyűjteményt, akkor készíthetünk hozzá egy bejáró osztályt, ami megvalósítja az **IEnumerator<T>** interfészt.

Ez az interfész továbbá arra is tökéletes, hogy saját üzleti logikai függvényekben elég lenne IEnumerable<T> interfészen keresztül várni az adatforrást. És akkor teljesen mindegy, hogy ez egy tömb, lista, sor, verem, gráf, stb valójában.

Először is készítsünk egy nagyon egyszerű adatszerkezetet, ami a háttérben valójában egy tömböt kezel.

```
class Storage<T>
{
    public T[] Array { get; private set; }
    int pointer = 0;
    public Storage(int size)
    {
        this.Array = new T[size];
    }

    public void Add(T item)
    {
        if (pointer < Array.Length)
        {
            Array[pointer++] = item;
        }
    }
```

```
}
```

Ezt az osztályt nem tudjuk **foreach** segítségével bejárni. A **foreach** a **GetEnumerator()** függvény hiányára panaszkodik. Ezt a függvényt az **IEnumerable<T>** interfész írja elő. Ha megvalósítjuk, akkor kapunk két üres függvényt.

```
public IEnumerator<T> GetEnumerator()
{
    throw new NotImplementedException();
}

IEnumerator IEnumerable.GetEnumerator()
{
    throw new NotImplementedException();
}
```

Ezek a függvények egy **IEnumerator<T>** interfészen akarnak visszaadni egy objektumot. Ez az objektum egy segédosztály példánya, ami hogyha kap egy referenciát a **Storage<T>-**re akkor képes bejárni. Készítsünk egy ilyen segédosztályt.

```
class StorageEnumerator<T> : IEnumerator<T>
{
    Storage<T> storage;
    int pointer = -1;

    public StorageEnumerator(Storage<T> storage)
    {
        this.storage = storage;
    }

    public T Current => storage.Array[pointer];

    object IEnumerator.Current => storage.Array[pointer];

    public void Dispose()
    {

    }

    public bool MoveNext()
    {
        return pointer++ < storage.Array.Length-1;
    }

    public void Reset()
    {
        pointer = 0;
    }
}
```

Látható, hogy konstruktoron át kapja meg a Storage<T> referenciát. A **Current** tulajdonság visszaadja az aktuális elemet. A **MoveNext()** a következő elemre lép és visszatérési értékként közli, hogy vannak-e még további érvényes elemek. A **Reset()** visszaáll az elejére, hogy újra bejárható legyen.

Ezután a **Storage<T>** osztály **GetEnumerator()** függvényei kitölthetőek.

```
public IEnumerator<T> GetEnumerator()
{
    return new StorageEnumerator<T>(this);
}

IEnumerator IEnumerable.GetEnumerator()
{
    return new StorageEnumerator<T>(this);
}
```

IEnumerable interfész kiegészítése

A kiegészítő metódusok segítségével, hogyha az **IEnumerable<T>** interfészt kibővítjük valamilyen függvénnyel, akkor azt lehetőség lesz tömbön, listán, soron, vermen, stb. is tökéletes használni. Gyakorlatilag bármi olyan adatszerkezeten, ami foreachhel bejárható. A Példában készítsünk egy mediánt kiszámító függvényt, amivel bővítjük az interfészt.

```
public static class Enumerable
{
    public static T Median<T>(this IEnumerable<T> source) where T:IComparable<T>
    {
        T[] items = source.ToArray();
        Array.Sort(items);
        return items[items.Length / 2];
    }
}
```

Futtassuk le néhány példával:

```
int[] numbers =
{
    3,8,5,78,91,24,56,78,13,24
};

Console.WriteLine(numbers.Median()); //24
```

```
List<string> names = new List<string>
{
    "John", "Jack", "Kate", "Michael", "Jin", "Sayid"
};

Console.WriteLine(names.Median()); //Kate
```

```
Stack<DateTime> dates = new Stack<DateTime>();
dates.Push(DateTime.Parse("2010.10.23"));
dates.Push(DateTime.Parse("2008.02.10"));
dates.Push(DateTime.Parse("2023.11.01"));

Console.WriteLine(dates.Median()); //2010.10.23
```

Látható, hogy egy hihetetlenül hasznos függvényt írtunk, ami bármilyen adattípusú, bármilyen gyűjteményen működik!

Utolsó módosítás: 2024. szeptember 13., péntek, 13:55

Telefon : +36 1 666-5741

E-mail : moodlesupport@uni-obuda.hu

Az előző fejezetben megismerkedtünk minden olyan építőelemmel ami a LINQ használatához szükséges. Igaz, hogy az előző fejezet kissé tömény volt, ám a jó hír az, hogy a LINQ használata ehhez képest elképesztően egyszerű. Az előző fejezet igazából nem is feltétlenül szükséges a használatához, ám az értéséhez annál inkább. És ez főleg akkor lesz hasznos, amikor az alkalmazásunk teljesítmény problémákkal küzd.

A LINQ nem más, mint a C# nyelvbe ágyazott lekérdezőnyelv. A lényege, hogy teljesen forrásfüggetlenül tudunk mindenféle algoritmust futtatni anélkül, hogy tudnánk, hogy milyen gyűjteménnyel dolgozunk. Teljesen mindegy, hogy tömbbel, listával, veremmel vagy akár adatbázissal dolgozunk, a lekérdezés működni fog.

Az egész LINQ úgy működik, hogy a **System.Linq** névtér az **IEnumerable<T>** típushoz kiegészítő metódusokat fűz. Pontosan úgy, ahogy az előző leckében a **Median()**-t elkészítettük mintának.

Nézzük meg az egyszerűbb LINQ metódusokat!

Concat

Két gyűjtemény egymás után fűzése. Teljesen mindegy, hogy milyen gyűjtemények ezek. A **Concat**ot **IEnumerable<T>** referencián tudjuk futtatni, a paramétere **IEnumerable<T>** referencia és az eredménye szintén **IEnumerable<T>** referencia

```
List<string> heroes = new List<string>()
{
    "Superman", "Batman", "Flash"
};

string[] villains =
{
    "Joker", "Lex Luthor", "Deathstroke"
};

var all = heroes.Concat(villains);
//all: Superman, Batman, Flash, Joker, Lex Luthor, Deathstroke
```

És itt álljunk is meg egy pillanatra és értsük meg a LINQ lényegét! Hogyha van egy 100 megabájt méretű A listánk és egy 100 megabájt méretű B listánk, majd egy C változóba konkatenáljuk ezeket LINQ-val, akkor nem fogunk kapni egy 200 megabájtos gyűjteményt, amivel együtt az alkalmazásunk már 400 megabájt memóriát foglal. Továbbra is csak két konkrét gyűjteményünk lesz a memóriában. Az eredmény csupán egy logikát ad vissza, amelyen folyamatosan hívogatva a **MoveNext()**-et szépen végigmegyünk az A gyűjteményen, majd a B gyűjteményen. Ennek bizonyításaképpen a **JetBrains dotPeek** nevű segédprogrammal a **Concat** függvényt visszafejtettük. Nézzük a működését.

```
public static IEnumerable<TSource> Concat<TSource>
(this IEnumerable<TSource> first, IEnumerable<TSource> second)
{
    if (first == null)
        throw Error.ArgumentNull(nameof(first));
    return second != null ? Enumerable.ConcatIterator<TSource>
(first, second) : throw Error.ArgumentNull(nameof(second));
}

private static IEnumerable<TSource> ConcatIterator<TSource>
(IEnumerable<TSource> first, IEnumerable<TSource> second)
{
    foreach (TSource source in first)
        yield return source;
    foreach (TSource source in second)
        yield return source;
}
```

Azt látjuk az alsó függvényben, hogy először az egyik gyűjteményen fut végig egy **foreach**, majd a második gyűjteménye a **yield** kulcsszót használják a fejlesztők, amit nem biztos, hogy ismerünk. A lényege, hogy az adott függvényt meghívva a **yield** miatt a pillanatnyi állapottal térünk vissza, majd újra meghívva a függvényt, eggyel tovább iterál a **foreach**.

Contains

Elem létezésének vizsgálata

```
bool hasSuperman = heroes.Contains("Superman");
```

Ez itt egy kis csúsztatást tartalmaz, ugyanis ez a **Contains()** nem a LINQ része, hanem a **System.Collections.Generic** névtér része, vagyis a lista saját függvénye.

Distinct

Ismétlődések kivágása, vagyis halmazzá alakítás

```
int[] nums = { 2, 3, 3, 4, 4, 4, 5, 5, 5, 5 };
var uniques = nums.Distinct(); //2,3,4,5
```

Intersect

Halmazmetszet, vagyis két gyűjteményből a közös elemek ismétlődés nélkül

```
int[] a = { 1, 3, 5, 7, 9 };
int[] b = { 1, 2, 3, 4, 5, 6 };
var c = a.Intersect(b); //1,3,5
```

Union

Halmaz unió, vagyis két gyűjtemény összes eleme ismétlődés nélkül

```
int[] a = { 1, 3, 5, 7, 9 };
int[] b = { 1, 2, 3, 4, 5, 6 };
var c = a.Union(b); //1,3,5,7,9,2,4,6
```

Except

Halmaz különbség, A gyűjteményből elhagyjuk azokat az elemeket, amelyek B-ben is szerepelnek

```
int[] a = { 1, 3, 5, 7, 9 };
int[] b = { 1, 2, 3, 4, 5, 6 };
var c = a.Except(b); //7,9
```

OrderBy, OrderbyDescending

Rendezés növekvő/csökkenő sorrendbe. Ugyan már tanultunk tömbrendezést az **Array.Sort()** segítségével, de ez teljesen más. Ez nem módosítja az eredeti gyűjteményt, hanem bármilyen gyűjteményen biztosít egy növekvő sorrendben történő bejárást. Itt az első példánál lambdával kell kijelölnünk a rendezés szempontját. Mivel az int önmagában **IComparable<T>** biztos, ezért önmagát jelöljük ki.

```
int[] a = { 1, 5, 3, 9, 7 };
var c = a.OrderBy(t => t); //1,3,5,7,9
```

Más a helyzet, hogyha olyan típusokat szeretnénk rendezni, amelyek nem hasonlíthatóak össze. Ekkor a lambdával kijelöljük valamelyik olyan tulajdonságukat, amely alapján már lehetséges a rendezés.

```
Point[] points =
{
    new Point(10,20),
    new Point(3,2),
    new Point(15,2),
    new Point(6,6)
};

var points2 = points.OrderBy(t => t.X);
```

Persze itt is megadhatunk valami komplexebb rendező logikát

```
var points2 = points.OrderBy(t => Math.Sqrt(Math.Pow(t.X, 2) + Math.Pow(t.Y, 2)));
```

Where

Kiválogatás, teljesen hasonló, mint listánál a FindAll(). Előnye, hogy mint a legtöbb LINQ metódus, csak egy bejárót ad vissza, nem egy konkrét listát.

```
Point[] points =
{
    new Point(10,20),
    new Point(3,2),
    new Point(15,2),
    new Point(6,6)
};

var filtered = points.Where(t => t.X < 10);
```

Count

Megszámlálás. Int típusként visszaadja, hogy hány olyan elem van a gyűjteményben, amelyre igaz az állítás.

```
int c = points.Count(t => t.X < 10);
//2
```

All

Minden elemre igaz-e az állítás?

```
bool all = points.All(t => t.X < 20);
//true
```

Any

Van-e legalább egy olyan elem a gyűjteményben, amelyre igaz az állítás

```
bool any = points.Any(t => t.X < 5);
//true
```

Take

Első n darab elem visszaadása

```
int[] a = { 1, 5, 3, 9, 7 };
var result = a.Take(3);
//1,5,3
```

Skip

Első n darab elem elhagyása

```
int[] a = { 1, 5, 3, 9, 7 };
var result = a.Skip(3);
//9,7
```

First, FirstOrDefault

Első olyan elem visszaadása, amely az adott feltételeknek megfelel

First: Ha nincs ilyen, akkor Exception dobás, **FirstOrDefault**: Ha nincs ilyen, akkor null az eredmény

```
int[] a = { 1, 5, 3, 9, 7 };
var result = a.FirstOrDefault(t => t / 3 == 3);
//9
```

Single, SingleOrDefault

Az egyetlen T tulajdonságú elem visszaadása

Single: ha nincs vagy több ilyen is van, akkor **Exceptiont** dob. **SingleOrDefault**: Ha nincs akkor null az eredmény, ha több ilyen is van, akkor **Exceptiont** dob

```
int[] a = { 1, 5, 3, 9, 7 };
var result = a.SingleOrDefault(t => t / 3 == 3);
//9
```

Select

Tulajdonság kiválasztás, talán az egyik leghasznosabb LINQ függvény!

A bemenete például egy típusos gyűjtemény és a kimenet formátumát fogalmazzuk meg a lambda kifejezésben.

```
List<Person> people = new List<Person>()
{
    new Person("Peter", 30, "developer"),
    new Person("Paul", 32, "ceo"),
    new Person("Kate", 23, "ux designer")
};

IEnumerable<string> result = people.Select(t => t.Name);
```

A példában látható, hogy bemenet a **Select()** függvénybe egy **Person** lista és kijött egy bejárható **string** gyűjtemény, amelyben csak a nevek szerepelnek.

Csinálhatunk akár itt egy teljes transzformációt is.

```
class Guest
{
    public Guest(string name, int birthYear)
    {
        Name = name;
        BirthYear = birthYear;
    }

    public string Name { get; set; }

    public int BirthYear { get; set; }
}

IEnumerable<Guest> guests = people.Select(t => new Guest(t.Name, DateTime.Now.Year - t.Age));
```

És akár visszatérhetünk egy névtelen típussal is

```
var guests = people.Select(t => new
{
    Name = t.Name,
    BirthYear = DateTime.Now.Year - t.Age
});
```

Utolsó módosítás: 2024. szeptember 13., péntek, 13:56

Info

Moodle használati útmutató

GY.I.K.

Kapcsolat

1034 Budapest, Doberdó út 6.

Telefon : +36 1 666-5741

E-mail : moodlesupport@uni-obuda.hu

Az előző fejezetben megismerkedtünk néhány egyszerűbb Linq függvénnyel. Ezek gyakorlatilag egy szűrt eredménylistát, egy darab elemet vagy számot vagy logikai értéket adnak vissza. Felmerülhet a kérdés, hogy miért nevezünk néhány keretrendszeri függvényt lekérdezésnek.

Nos azért, mert megírhatóak lekérdezés-szerűen is. Amelyek **IEnumerable<T>** eredményt adnak vissza, azokon újból meghívható egy Linq függvény és így láncolhatóak ezek.

```
Point[] points =
{
    new Point(10,20),
    new Point(3,2),
    new Point(15,2),
    new Point(6,6)
};

var result = points
    .Where(t => t.X < 15)
    .OrderBy(t => t.Y)
    .Reverse()
    .Select(t => t.Y);
```

Ezt nevezzük **Method syntaxnak**. De a LINQ támogat egy ettől merőbben szokatlanabb megoldást, a **Query syntaxot**. Ez leginkább SQL lekérdezésekre hasonlít. De nem SQL, csak SQL-szerű a szintaxisa.

```
var result = from t in points
             where t.X < 15
             orderby t.Y descending
             select t.Y;
```

Fontos megjegyezni, hogy a **Query Syntax** esetén kötelező a **select** részt használni, még **Method Syntax** esetén nem.

SelectMany

A select egy okosabb megvalósítása, ami lehetővé teszi, hogy egy komplex struktúrát kisimítsunk. Mit értünk ezalatt? Nézzük az alábbi példát:

```
var rooms = new Room[]
{
    new Room("BA.213", new Subject[]
    {
        new Subject("elektro"),
        new Subject("digit")
    }),
    new Room("BA.210", new Subject[]
    {
        new Subject("hft"),
        new Subject("sztgui")
    }),
    new Room("BA.119", new Subject[]
    {
        new Subject("iba")
    }),
};
```

Itt tantermeket látunk és egy tanterem objektum tartalmazza a benne tartott órákat. Hogyha az a feladat, hogy adjuk vissza a tantárgyakat egy listában és minden tantárgy mellett legyen ott, hogy melyik teremben oktatják, azt elég körülményesen tudnánk megoldani. Valahogy így:

```
class SubjectInRoom
{
```

```
    public string Subject { get; set; }
    public string Room { get; set; }

    public SubjectInRoom(string subject, string room)
    {
        Subject = subject;
        Room = room;
    }
}

List<SubjectInRoom> list = new List<SubjectInRoom>();
foreach (var room in rooms)
{
    foreach (var subject in room.Subjects)
    {
        list.Add(new SubjectInRoom(subject.Name, room.Name));
    }
}
```

Egyrészt kellene készítenünk egy teljesen felesleges wrapper osztályt, amit kiválthatnánk névtelen típussal. Másrészt a SelectMany pont ezt a feladatot oldja meg így:

```
var result = rooms.SelectMany(t => t.Subjects, (room, subject) => new SubjectInRoom(room.Name, subj
```

A lambdával ki kell jelölni az al-gyűjteményt. A kétparaméteres következő lambda pedig a főgyűjtemény aktuális elemét és az al-gyűjtemény egy elemét adja át paraméterként. A feladat megoldható wrapper osztály nélkül névtelen típussal is:

```
var result = rooms.SelectMany(t => t.Subjects, (room, subject) => new
{
    RoomName = room.Name,
    SubjectName = subject.Name
});
```

Aggregáló metódusok

Az eddigiekben olyan LINQ függvényeket néztünk, amelyek egy bemeneti gyűjteményt leszűrtek, vagy transzformáltak. Most nézzünk olyan LINQ függvényeket amelyek 1-1 aggregált értéket vagyis egy darab számszerű eredményt adnak vissza egy gyűjteményről.

Sum

Visszaadja a gyűjtemény összegét, ám egy lambda segítségével meg kell neki mondani, hogy melyik jellemző számszerű értékeit összegezze.

```
List<Person> people = new List<Person>()
{
    new Person ("Peter", 30, "developer"),
    new Person ("Paul", 32, "ceo"),
    new Person ("Kate", 23, "ux designer")
};

var sumAge = people.Sum(t => t.Age); //85
```

Average

Visszaadja egy gyűjtemény átlagát, szintén egy lambdával ki kell jelölni, hogy mely jellemzők átlagát.

```
var avgAge = people.Average(t => t.Age); //28.33
```

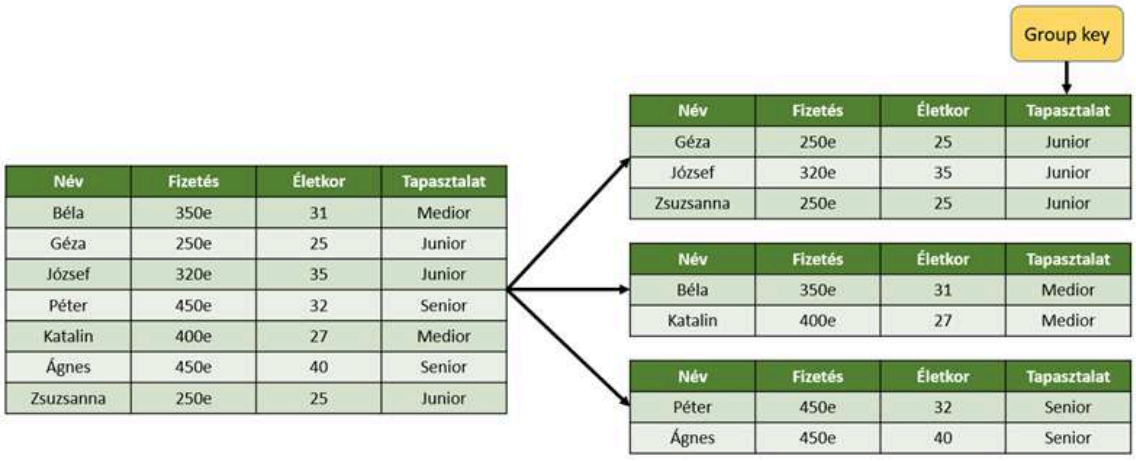
Ezek az aggregáló függvények is nyugodtan lehetnek egy komplexebb lekérdezés részei. Például nézzük meg a develope pozícióban lévő emberek átlagéletkorát:

```
var avgAgeDev = people.Where(t => t.Job == "developer").Average(t => t.Age);
```

Csoportosítás – GROUP BY

A következő nagy LINQ kategória a csoportosításé. Ennek a célja, hogy egy bementi gyűjteményt széttörjön több gyűjteményre kategoriális változók mentén. Ez annyit jelent, hogy kvázi enum-szerű értékek mentén van az egésznek értelme. Ilyen lehet például a munkakör, a részleg, ahol az illető dolgozik vagy céges telephely. De pl. semmi értelme életkor szerint széttördelni egy gyűjteményt, mert az égvilágon semmilyen jelentést nem hordoz magában. Emberek neve szerint sincs sok értelme. Ellenben életkor tartományok vagy fizetési tartományok szerint már annál inkább.

Az egész vizualizálva nagyjából így néz ki:



Tehát van egy bemeneti gyűjteményünk, amely értelmesen felbontható a tapasztalati szint mentén. És így 3 kimenetet kapunk. A **group key**, vagyis a csoportosítás kulcsa az a jellemző, ami mentén a tördelés végezzük.

```
List<Person> people = new List<Person>()
{
    new Person ("Peter", 30, "developer", "senior"),
    new Person ("Paul", 32, "ceo", "senior"),
    new Person ("Kate", 23, "ux designer", "junior"),
    new Person ("Jack", 20, "developer", "junior"),
    new Person ("Michael", 40, "a/b tester", "senior"),
    new Person ("Susan", 27, "developer", "medior")
};

var groups = people.GroupBy(t => t.Level);
```

A **GroupBy()** kimenete egy **IEnumerable<IGrouping<string, Person>>** referencia. Vagyis egy 3 elemű gyűjtemény, amelyet például így járhatunk be:

```
foreach (var item in groups)
{
    Console.WriteLine(item.Key + " level workers:");
    foreach (var worker in item)
    {
        Console.WriteLine("- " + worker.Name);
    }
}
```

Látható, hogy a külső **foreach** halad egy gyűjteményen. Ezek a tapasztalati szintek. Mindegyiknek van egy kulcsa és önmaga is bejárható, mint **Person** gyűjtemény.

Viszont hogyha felbontunk csoportokra valamilyen kulcs mentén, akkor általában nem annak van jelentősége, hogy az adott csoportban kik vannak, hanem valamilyen aggregált mérőszámot szeretnénk látni. Például adott tapasztalati szinten az átlagéletkort:

```
Console.WriteLine("Average Age / level");
foreach (var item in groups)
{
    Console.Write(item.Key + " level: ");
    Console.WriteLine(item.Average(t => t.Age));
}
//senior level: 34
//junior level: 21.5
//medior level: 27
```

Ezeket névtelen típusba ki is tudjuk gyűjteni

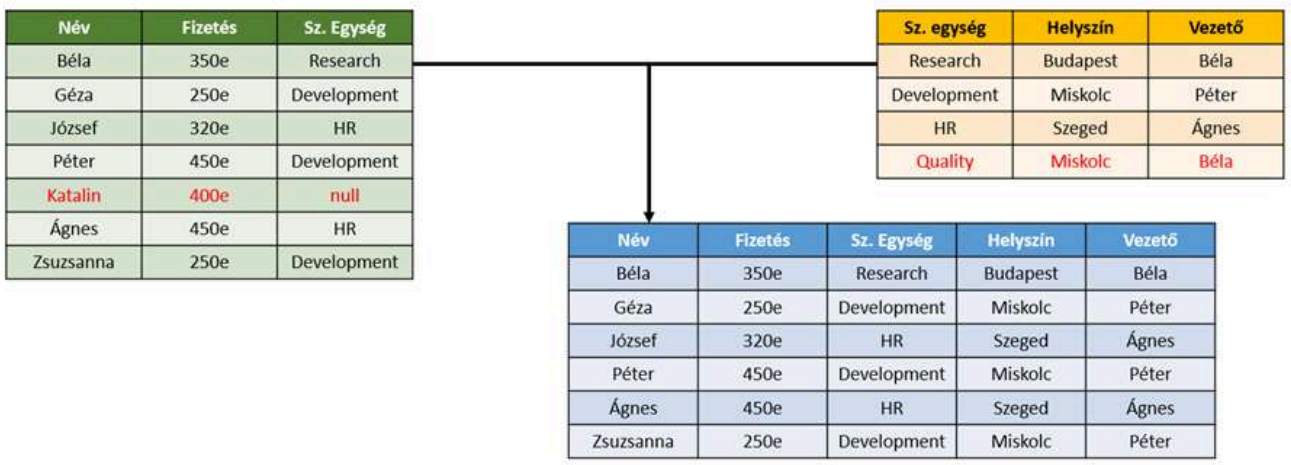
```
var result = people.GroupBy(t => t.Level).Select(t => new
{
    Level = t.Key,
    AvgAge = t.Average(z => z.Age)
});
```

Ugyanez query syntax-szal:

```
var result = from x in people
             group x by x.Level into g
             select new
             {
                 Level = g.Key,
                 AvgAge = g.Average(z => z.Age)
             };
```

JOIN

Az utolsó LINQ kategória a JOIN vagyis összefűzés. A célja, hogy két eltérő bemeneti gyűjteményből egyet csináljon. Nézzünk erre egy ábrát a megértéshez:



Név	Fizetés	Sz. Egység
Béla	350e	Research
Géza	250e	Development
József	320e	HR
Péter	450e	Development
Katalin	400e	null
Ágnes	450e	HR
Zsuzsanna	250e	Development

Sz. egység	Helyszín	Vezető
Research	Budapest	Béla
Development	Miskolc	Péter
HR	Szeged	Ágnes
Quality	Miskolc	Béla

Név	Fizetés	Sz. Egység	Helyszín	Vezető
Béla	350e	Research	Budapest	Béla
Géza	250e	Development	Miskolc	Péter
József	320e	HR	Szeged	Ágnes
Péter	450e	Development	Miskolc	Péter
Ágnes	450e	HR	Szeged	Ágnes
Zsuzsanna	250e	Development	Miskolc	Péter

Van két bemeneti lista és ezekben az a közös, hogy valamelyik jellemző alapján össze lehet őket ragasztani. Képzeletben a jobb oldali sárga táblázatot felvágjuk soronként. És minden szervezeti egységből annyi darabot készítünk, ahányszor a bal oldali táblázatban hivatkozás van rá. Ezeket a darabokat hozzáragasztjuk a zöld táblázathoz és így megkapjuk a kéket. Ez kifejezetten jól fog jönni adatbázisok LINQ-val történő lekérdezése esetén, mert ott a relációs jelleg miatt eleve több táblából lehet bizonyos lekérdezéseket összeépíteni.

Nézzük a megvalósítását:

```
List<Person> people = new List<Person>()
{
    new Person ("Peter", 30, "developer", "senior"),
    new Person ("Paul", 32, "ceo", "senior"),
    new Person ("Kate", 23, "ux designer", "junior"),
    new Person ("Jack", 20, "developer", "junior"),
    new Person ("Michael", 40, "a/b tester", "senior"),
    new Person ("Susan", 27, "developer", "medior")
};

List<Bonus> bonuses = new List<Bonus>()
{
    new Bonus("junior", 1000),
    new Bonus("medior", 5000),
    new Bonus("senior", 10000)
};

var join = from p in people
           join b in bonuses
```

```
            on p.Level equals b.Level
            select new
            {
                PersonName = p.Name,
                Amount = b.Amount
            };
//Peter, 10000
//Paul, 10000
//Kate, 1000
//Jack, 1000
//Michael, 10000
//Susan, 5000
```

Az **on** sorban kell megjelölnünk azt a jellemzőt ami alapján összekapcsolunk.

Left Outer Join

Hogyha valakinél a kapcsoló jellemzőben olyan érték szerepel, ami a kapcsolt táblában nincs benne, akkor az eredményben sem fog szerepelni. Például ha „ceo” levelt adunk a ceo-nak, de ilyen bónusz nincsen, akkor ő nem szerepel az eredmény joinolt adathalmazban.

```
new Person ("Paul", 32, "ceo", "ceo")
```

Ennek megoldása:

```
var join = from p in people
            join b in bonuses
            on p.Level equals b.Level into sub
            from m in sub.DefaultIfEmpty()
            select new
            {
                PersonName = p.Name,
                Amount = m?.Amount
            };
```

Így viszont a bal táblába mappelt értékeknek mindenképp nullable értékeknek kell lenni az eredményhalmazban.

Utolsó módosítás: 2024. szeptember 13., péntek, 14:00

Telefon : +36 1 666-5741

E-mail : moodlesupport@uni-obuda.hu

Ebben a leckében az XML fájlok olvasásával és írásával ismerkedünk meg. Kezdjük rögtön egy XML mentes példával ennek a jelentőségét megérteni. Hogyha vannak objektum hierarchiáink, amelyek több szint mélységben ágyazódnak egymásba, akkor ennek a struktúrának a fájlba írása csak nagyon bonyolult módon tud megtörténni. Nézzünk egy jó komplex, 3 mélységű struktúrát:

```
List<Department> departments = new List<Department>()
{
    new Department("HQ", "Dallas", new List<Person>()
    {
        new Person ("Paul", 32, "ceo", "ceo", new Bonus(5000,12)),
        new Person ("Michael", 40, "a/b tester", "senior", new Bonus(3000,3))
    }),
    new Department("Development", "Chicago", new List<Person>()
    {
        new Person ("Peter", 30, "developer", "senior", new Bonus(4000,4)),
        new Person ("Kate", 23, "ux designer", "junior", new Bonus(1000,1)),
        new Person ("Jack", 20, "developer", "junior", new Bonus(1000,1)),
        new Person ("Susan", 27, "developer", "medior", new Bonus(3000,3))
    })
};
```

Szervezeti egységeket látunk, amelyekben személyek dolgoznak és a személyeknek van egy bónusza, aminek van egy összege és egy éves gyakorisága (hányszor kapja egy évben). Hogyha ezt ki szeretnénk menteni txt fájlba, az valahogy így nézne ki:

```
using (StreamWriter sw = new StreamWriter("company.txt"))
{
    foreach (var dept in departments)
    {
        sw.WriteLine($"dept: {dept.DeptName},{dept.Location}");
        foreach (var worker in dept.Workers)
        {
            sw.WriteLine($"worker: {worker.Name},{worker.Age},{worker.Job},{worker.Level},
{worker.Bonus.Amount},{worker.Bonus.Frequency}");
        }
    }
}
```

És ezt a kimenetet produkálná:

```
dept: HQ,Dallas
worker: Paul,32,ceo,ceo,5000,12
worker: Michael,40,a/b tester,senior,3000,3
dept: Development,Chicago
worker: Peter,30,developer,senior,4000,4
worker: Kate,23,ux designer,junior,1000,1
worker: Jack,20,developer,junior,1000,1
worker: Susan,27,developer,medior,3000,3
```

A kiírás még rendben volt, de a beolvasás már sokkal bonyolultabb. Erre nem is nézünk megoldást. Egy department konstruktorában már meg kell adni a workereket, akiknek a konstruktorában a bónuszt és így tovább. Egy nagyon csúnya kódot eredményezne ami tele van hibalehetőséggel.

Az XML fájlokat arra találták ki, hogy ezt az objektum alá és fölérendeltséget megfelelően tudja reprezentálni. Az első kóc lévő hierarchia így nézne ki XML-ben tárolva:

```
<?xml version="1.0" encoding="utf-8" ?>
<departments>
    <department>
        <name>HQ</name>
```

```
        <location>Dallas</location>
        <workers>
            <worker>
                <name>Paul</name>
                <age>32</age>
                <job>ceo</job>
                <level>ceo</level>
                <bonus>
                    <amount>5000</amount>
                    <frequency>12</frequency>
                </bonus>
            </worker>
            <worker>
                <name>Michael</name>
                <age>40</age>
                <job>a/b tester</job>
                <level>senior</level>
                <bonus>
                    <amount>3000</amount>
                    <frequency>3</frequency>
                </bonus>
            </worker>
        </workers>
    </department>
</departments>
```

Az XML nem is lett teljesen végigírva, csak a HQ departmentet tartalmazza. Az utolsó előtti sorban van vége az adott departmentnek. És ott kezdődhetne egy új <Deparment></Department> blokk. A lényeg, hogy egy XML fájl node-okból áll amelyek teljesen jól illeszkednek az objektum-orientált szemléletmódhoz.

Nézzük meg, hogy hogyan tudnánk kiírni a hierarchiánkat XML-be:

```
XDocument xdoc = new XDocument();
XElement root = new XElement("deparments");
xdoc.Add(root);

root.Add(departments.Select(t =>
{
    return new XElement("department",
        new XElement("name", t.DeptName),
        new XElement("location", t.Location));
}));

xdoc.Save("company.xml");
```

Ez még nem a teljes kód, csupán a departmentek kiírása. Az XML dokumentumban kötelező egy root elem, amelyből csak egy lehet. Ez a departments. És ebbe teszünk bele a Select segítségével sok departmentet. Jelenleg itt tart a kimenetünk.

```
<?xml version="1.0" encoding="utf-8"?>
<deparments>
    <department>
        <name>HQ</name>
        <location>Dallas</location>
    </department>
    <department>
        <name>Development</name>
        <location>Chicago</location>
    </department>
</deparments>
```

Nézzük most a teljes kódot, ami mindent kiír az XML fájlba:

```
XDocument xdoc = new XDocument();
XElement root = new XElement("deparments");
xdoc.Add(root);

root.Add(departments.Select(t =>
{
```

```
    return new XElement("department",
        new XElement("name", t.DeptName),
        new XElement("location", t.Location),
        new XElement("workers", t.Workers.Select(z =>
        {
            return new XElement("worker",
                new XElement("name", z.Name),
                new XElement("age", z.Age),
                new XElement("job", z.Job),
                new XElement("level", z.Level),
                new XElement("bonus",
                    new XElement("amount", z.Bonus.Amount),
                    new XElement("frequency", z.Bonus.Frequency)));
        })));
}));

xdoc.Save("company.xml");
```

Jól kirajzolódik a hierarchia, az **XElement** konstruktora flexibilis, gyakorlatilag beadhatunk neki tartalmat, új **XElement**-et vagy akár több al **XElement**-et.

Nézzük most meg, hogyan lehetne beolvasni XML fájlt

```
List<Department> depts = XDocument.Load("company.xml").Root.Elements("department").Select(t =>
{
    return new Department(t.Element("name")?.Value, t.Element("location")?.Value, t.Element("worker
    {
        return new Person(z.Element("name").Value, int.Parse(z.Element("age").Value), z.Element("jo
            int.Parse(z.Element("bonus").Element("amount").Value),
            int.Parse(z.Element("bonus").Element("frequency").Value)
            ));
    }).ToList());
}).ToList();
```

Talán ez egy kissé kaotikus így, de akár át is szervezhetjük a kódot úgy, hogy minden osztály maga foglalkozik a saját XML node-jának parse-olásával. Így meglepően leegyszerűsödik az egész.

```
public Bonus(XElement element)
    {
        this.Amount = int.Parse(element.Element("amount").Value);
        this.Frequency = int.Parse(element.Element("frequency").Value);
    }
```

```
public Person(XElement element)
{
    this.Name = element.Element("name")?.Value;
    this.Age = int.Parse(element.Element("age")?.Value);
    this.Job = element.Element("job")?.Value;
    this.Level = element.Element("level")?.Value;
    this.Bonus = new Bonus(element.Element("bonus"));
}
```

```
public Department(XElement element)
{
    this.DeptName = element.Element("name").Value;
    this.Location = element.Element("location").Value;
    this.Workers = element.Element("workers").Elements("worker").Select(t => new Person(t)).ToList(
}
```

```
List<Department> depts = XDocument.Load("company.xml").Root.Elements("department").Select(t =>
{
    return new Department(t);
}).ToList();
```

Ezt a fajta kódszervezést felhasználhatjuk az XML mentésénél is. Minden egyes osztály maga gondoskodjon saját maga XML node-ra alakításával.

Bonus osztály

```
public XElement ToXml()
    {
        return new XElement("bonus",
            new XElement("amount", this.Amount),
            new XElement("frequency", this.Frequency));
    }
```

Person osztály

```
public XElement ToXml()
    {
        return new XElement("worker",
            new XElement("name", this.Name),
            new XElement("age", this.Age),
            new XElement("job", this.Job),
            new XElement("level", this.Level),
            this.Bonus.ToXml());
    }
```

Department osztály

```
public XElement ToXml()
{
    return new XElement("department",
        new XElement("name", this.DeptName),
        new XElement("location", this.Location),
        new XElement("workers", Workers.Select(t => t.ToXml())));
}
```

Kimentő függvény

```
XDocument xdoc = new XDocument();
XElement root = new XElement("departments");
xdoc.Add(root);
root.Add(depts.Select(t => t.ToXml()));
xdoc.Save("comp.xml");
```

Utolsó módosítás: 2024. szeptember 13., péntek, 14:15

Telefon : +36 1 666-5741

E-mail : moodlesupport@uni-obuda.hu

Az előző fejezetben megismerkedtünk az XML fájlok készítésével és visszaolvasásával. Viszont ha jobban megnézzük az XML fájlt, akkor nagyon nehezen különül el, hogy mit teszünk gyermekelembe. Mert gyakorlatilag szülő-gyermek kapcsolat van objektumok között (pl. department-person) és ugyanilyen kapcsolat áll fent a person objektum és annak tulajdonságai között.

Éppen ezért, hogyha olyan XML-t szeretnénk, ahol ez a kétfajta kapcsolat szintaxis szinten különválik, akkor az XML támogatja az attribútumok használatát is. Ez a régi XML-ünk:

```
<worker>
    <name>Paul</name>
    <age>32</age>
    <job>ceo</job>
    <level>ceo</level>
    <bonus>
        <amount>5000</amount>
        <frequency>12</frequency>
    </bonus>
</worker>
```

És ezt átírhatjuk erre a formára is:

```
<worker name="Paul" age="32" job="ceo" level="ceo">
    <bonus amount="5000" frequency="12" />
</worker>
```

Vagyis a **worker** tulajdonságai innentől attribútumként szerepelnek, a **bonus** pedig mivel egy objektum kompozíció, ezért gyermekelemként.

Az előző fejezetben ismertetett osztályszintű beolvasás átirata így nézne ki. **element.Element** helyett **element.Attribute**-ot kell lekérni.

```
public Bonus(XElement element)
{
    this.Amount = int.Parse(element.Attribute("amount").Value);
    this.Frequency = int.Parse(element.Attribute("frequency").Value);
}
```

Hogyha pedig attribútumot akarnánk írni, akkor:

```
public XElement ToXml()
{
    return new XElement("bonus",
        new XAttribute("amount", this.Amount),
        new XAttribute("frequency", this.Frequency));
}
```

Ezt az összes osztályban elvégezve, az alábbi XML fájlt kapjuk:

```
<?xml version="1.0" encoding="utf-8"?>
<departments>
    <department name="HQ" location="Dallas">
        <workers>
            <worker name="Paul" age="32" job="ceo" level="ceo">
                <bonus amount="5000" frequency="12" />
            </worker>
            <worker name="Michael" age="40" job="a/b tester" level="senior">
                <bonus amount="3000" frequency="3" />
            </worker>
        </workers>
    </department>
    <department name="Development" location="Chicago">
        <workers>
```

```
            <worker name="Peter" age="30" job="developer" level="senior">
                <bonus amount="4000" frequency="4" />
            </worker>
            <worker name="Kate" age="23" job="ux designer" level="junior">
                <bonus amount="1000" frequency="1" />
            </worker>
            <worker name="Jack" age="20" job="developer" level="junior">
                <bonus amount="1000" frequency="1" />
            </worker>
            <worker name="Susan" age="27" job="developer" level="medior">
                <bonus amount="3000" frequency="3" />
            </worker>
        </workers>
    </department>
</departments>
```

Az, hogy elementeket vagy attribute-okat használunk, tervezési kérdés. De egy jó kiindulási pont lehet az, hogy attribútumba a primitív jellemzők kerülnek, gyermekelembe az összetett jellemzők. Valójában attribútumba nem is kerülhet olyan elem, aminek további gyermekei vannak.

Utolsó módosítás: 2024. szeptember 13., péntek, 14:26

A lecke utolsó fejezetében nézzük meg, hogy milyen automatikus beépített lehetőségeink vannak XML szerializációra és deszerializációra.

XML fájlba kimentés

```
Person p = new Person()
{
    Name = "Peter",
    Job = "Spider-man"
};

XmlSerializer writer = new XmlSerializer(typeof(Person));
FileStream file = File.Create("data.xml");
writer.Serialize(file, p);
file.Close();
```

XML fájlból beolvasás

```
XmlSerializer reader = new XmlSerializer(typeof(Person));
using(StreamReader file = new StreamReader("data.xml"))
{
    Person p = (Person)reader.Deserialize(file);
}
```

Természetesen ez tökéletes egyezést feltételez a tulajdonságok nevei és az XElementek nevei között.

Utolsó módosítás: 2024. szeptember 13., péntek, 14:27

Ebben a leckében az XML-hez nagyon hasonló hierarchikus formátummal ismerkedünk meg: A JSON-nel. A rövidítés a Javascript Object Notation-ből származik. Ahogy a neve is mutatja, a Javascript nyelvvel együtt terjedt el a használata. Nézzünk meg egy JSON fájlt és hasonlítsuk össze az XML-el.

```
[
    {
        "name": "HQ",
        "location": "Dallas",
        "workers": [
            {
                "name": "Paul",
                "age": 32,
                "job": "ceo",
                "level": "ceo",
                "bonus": {
                    "amount": 5000,
                    "frequency": 12
                }
            },
            {
                "name": "Michael",
                "age": 40,
                "job": "a/b tester",
                "level": "senior",
                "bonus": {
                    "amount": 3000,
                    "frequency": 3
                }
            }
        ]
    }
]
```

Ránézésre nem olyan szép, mint az xml és kevésbé tűnik átláthatónak. Észrevehetjük, hogy itt a tömb jelölése támogatott. És mivel a múlt leckében lévő XML-ünk root eleme „departments” névre hallgatott és abban sok department volt, ez itt egyszerűsíthető arra, hogy rögtön egy tömb jelöléssel indítunk és felsoroljuk a departmenteket. Az, hogy department-ről van szó, nem derül ki a jsonból. Annyit látunk, hogy objektumok vannak, amelyeknek vannak jellemzői. Itt most a példarészletben csak 1 db department szerepel, de vesszővel elválasztva tetszőleges mennyiségű jöhet egymás után. A workers is egy gyűjtemény, amelyben objektumok vannak jellemzőkkel. A bonus viszont nem gyűjtemény, hanem 1 db aljellemző.

Az XML-el való összehasonlítás:

Nem kötelező gyökérelem

OOP megközelítéshez jobban illeszkedik

Nincs zárótag

Tömböket is lehet benne használni

Utolsó módosítás: 2024. szeptember 13., péntek, 14:41

Elektronikus és Digitális Tananyagok Iroda

Info

Moodle használati útmutató

GY.I.K.

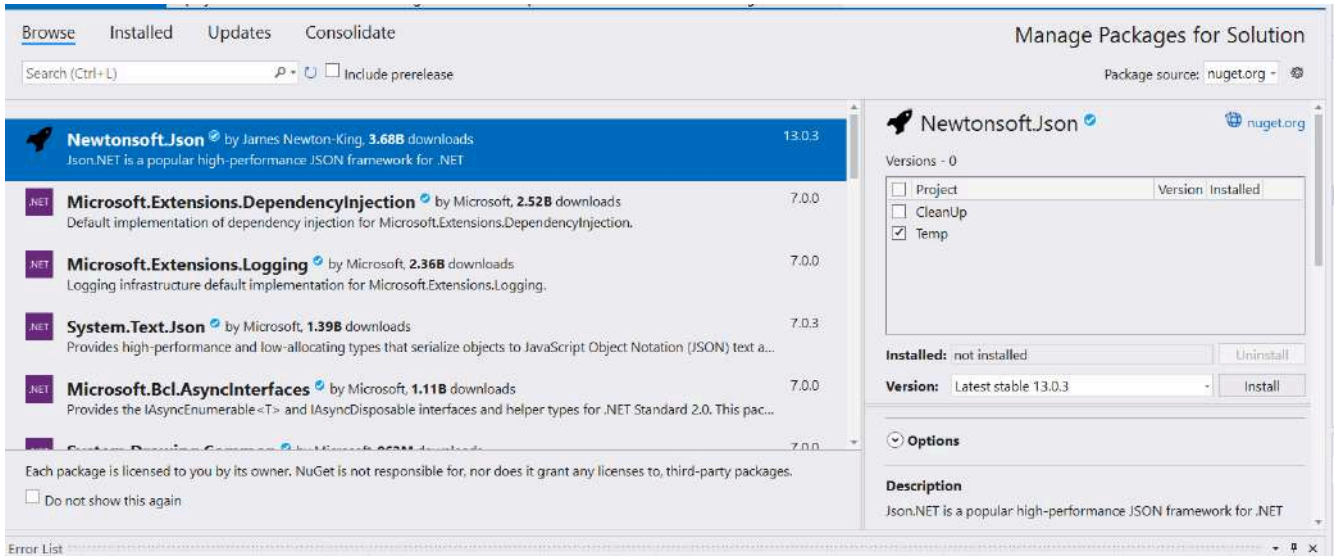
Kapcsolat

1034 Budapest, Doberdó út 6.

Telefon : +36 1 666-5741

E-mail : moodlesupport@uni-obuda.hu

Nézzük meg most azt, hogy JSON fájlokat hogyan tudunk beolvasni és kiírni. Ezek megvalósítása egy nuget csomag (külső library) segítségével történik, amelyet telepítenünk kell. Nyissuk meg a Visual Studioban a nuget package managert: **Tools → NuGet Package Manager → Manage NuGet Packages for Solution**.



Rögtön az első találat lesz a kívánt library: **Newtonsoft.Json**

A jobb oldali panelen jelöljük be a kívánt projectet, amelybe szeretnénk telepíteni, majd kattintsunk az **install** gombra.

Egy JSON fájlt, ha szeretnénk beolvasni, akkor először el kell hozzá készítenünk az adott osztályt.

Egy példa JSON fájl:

```
[
    {
        "name": "John",
        "age": 30,
        "job": "frontend developer"
    },
    {
        "name": "Kate",
        "age": 32,
        "job": "backend developer"
    },
    {
        "name": "Jack",
        "age": 25,
        "job": "tester"
    }
]
```

Látjuk, hogy egy tömbben 3 elem van és mindegyiknek van name, age és job jellemzője. Készítsünk ehhez megfelelő Person osztályt:

```
List<Person> people = JsonConvert.DeserializeObject<List<Person>>(json);

class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Job { get; set; }
}
```

A példában a json változóba stringként belekerült a json fájl tartalma. Ugyanis a JsonConvert osztály stringként várja a bemenetet. Tehát, hogyha nekünk fájlunk van, akkor olvassuk be a ReadAllText() segítségével a teljes tartalmát egy stringbe.

Nézzünk meg egy Visual Studio trükköt! Nem kellett volna bepötyögni feltétlenül a Person osztályt. Hogyha egy JSON fájlt kapunk feldolgozásra, akkor jelöljük ki egy nodeját így:

```
    {
        "name": "Kate",
        "age": 32,
        "job": "backend developer"
    }
```

Majd egy .cs fájlba illesszük be így**: Edit → Paste Special → Paste JSON as Classes**. Az alábbit kapjuk:

```
public class Rootobject
{
    public string name { get; set; }
    public int age { get; set; }
    public string job { get; set; }
}
```

Az osztály nevét értelemszerűen nem tudta kitalálni a VS, merthogy nem része a JSON fájlnak. Ezt nevezzük el tetszés szerint. Illetve a tulajdonságokat is illene nagy kezdőbetűvel írni.

Egy objektum-hierarchia JSON szerializálása is roppant egyszerű módon történik. Értelemszerűen feltöltött gyűjteményt írjunk ki.

```
List<Person> people = new List<Person>();
string json = JsonConvert.SerializeObject(people);
```

Hogyha valamelyik tulajdonságot nem szeretnénk kiírni, mert pl. egy **on-the-fly** érték, akkor megtilthatjuk a **[JsonIgnore]** attribútummal, hogy kimentésre kerüljön a JSON-be. A C# attribútum nem összekeverendő az XML attribútumról! Előbbiről egy későbbi leckében fogunk részletesen tanulni!

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Job { get; set; }

    [JsonIgnore]
    public int Point
    {
        get
        {
            return Age * Name.Length;
        }
    }
}
```

Utolsó módosítás: 2024. szeptember 13., péntek, 14:46

Info

Moodle használati útmutató

GY.I.K.

Kapcsolat

1034 Budapest, Doberdó út 6.

Telefon : +36 1 666-5741

E-mail : moodlesupport@uni-obuda.hu

Ebben a modulban a DLL fájlokkal ismerkedünk meg. Biztosan mindenki találkozott már ilyen fájlokkal, amikor egy telepített játék vagy szoftver telepítési könyvtárában nézelődött. Általában egy szoftver áll egy futtatható **.exe** fájlból, rengeteg **.dll** fájlból és mindenféle egyéb statikus állományból (képek, videók, hangfájlok). Utóbbi tipikusan a játékprogramokra igaz. Nézzük meg, hogy ezek a **.dll** fájlok mire jók és mik a céljuk.

Nagyon röviden a DLL célja: osztályok és/vagy metódusok kiszervezése az exe fájlból egy külön fájlba, amely szintén nem forráskódot tartalmaz, hanem bináris kódot. Vagyis ha egy DLL fájlt megnyitunk egy tetszőleges text editorral, akkor abban sem forráskódot találunk, hanem a processzor számára értelmezhető kódot (.NET esetében köztes kódot – IL kódot). Lényegében ugyanolyan bináris kódot tartalmaz, mint egy exe fájl, csak nincs kitűntetett belépési pontja (ami az exe esetében a Main() metódus .NET 6.0 előtt, utána a Program.cs első sora Top-level statements esetén).

Mi ennek az értelme?

Fejlesztünk egy jól működő algoritmust, ahol nem az a célunk, hogy a forrást, avagy a szellemi terméket átadjuk, hanem az a célunk, hogy mások is használhassák a forráskód megismerése és módosítása nélkül (hozzátesszük, hogy rengeteg eszköz érhető el, amelyekkel egy bináris állomány visszafordítható programkódra)

Célunk lehet, hogy ugyanazt a függvényt/osztályt használhassa több program is és adott hibajavítás esetén csak a dll-t kelljen újrafordítani

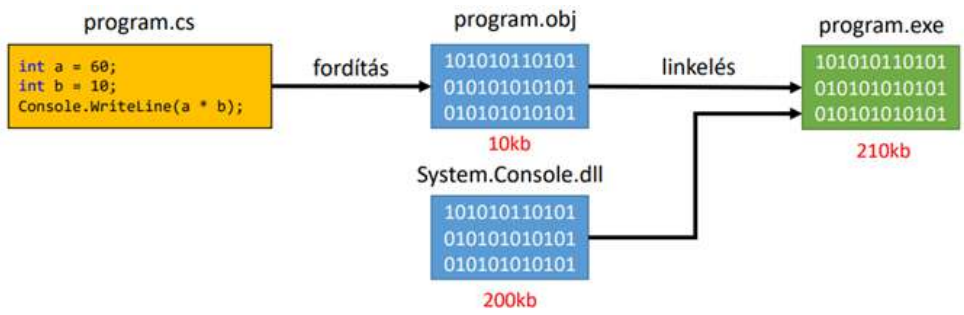
Az operációs rendszer a szolgáltatásait is DLL fájlokon keresztül teszi elérhetővé a fejlesztők részére (pl. vágólap használata, hang lejátszása, memória foglaltság lekérése, stb.)

A .NET keretrendszer összes beépített típusát (pl. Console, List, XDocument) is DLL fájlokon keresztül érjük el.

Utóbbinak a működése úgy néz ki, hogyha a Console osztályt használjuk, akkor a System.Console.Dll névtér betöltődik a C:\Windows\Assembly mappában található megfelelő DLL-ből.

Statikus és dinamikus linkelés

A statikus linkelés egy elavult technika. A lényege, hogy egy forráskódból (pl. program.cs) egy futtatható állományt (program.exe) szeretnénk készíteni úgy, hogy a Console osztályt is felhasználjuk benne. Ekkor a program.cs-ből fordított köztes kód (program.obj) és a System.Console.dll egyesítéséből keletkezik a futtatható állomány. Ami viszont így relatíve nagy méretű lesz. További probléma egy pl. 5-6 GB méretű statikusan linkelt exe fájllal, hogy elindítása akár több perc is lehet, mire a GB nagyságrendű állomány a háttértárból áttöltődik a memóriába. És a program bezárásáig konstans módon foglalni fogja azt a néhány GB-ot a memóriában.



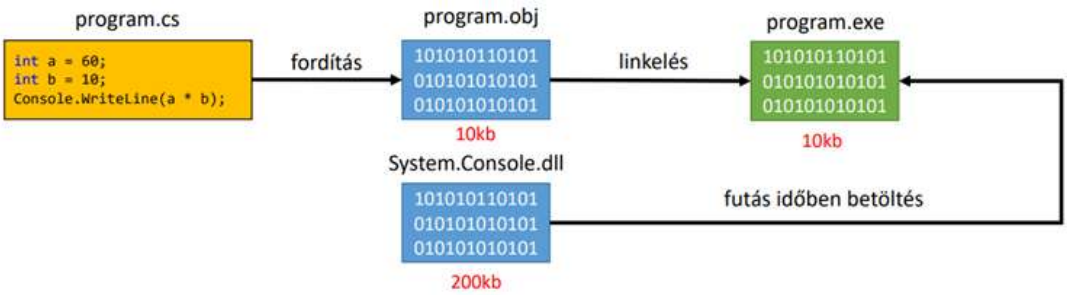
Összegezve

A függőségek is belekerülnek a végső állományba

Ugyanazt a függvényt/osztályt így több program külön-külön is tartalmazza (pazarlás)

Hatalmas lesz a fájlméret

A dinamikus linkelés viszont egy korszerű megoldás. A szükséges dll fájlokat a futtatókörnyezet futásidőben fogja betölteni és csak akkor, amikor éppen szüksége van rá. És csak annyi ideig foglalja a memóriát, amíg használatban van.



Összegezve

Futásidőben gyakorlatilag DLL fájlokba hívunk bele

A betöltést az operációs rendszer végzi

Ugyanazt a DLL-t több program is hívhatja (Shared Object)

Legtöbb mai modern program így működik

Update-ek egyszerűen megvalósíthatóak: lecseréljük a DLL fájlt egy újabbra

Natív és felügyelt állományok

Amikor DLL fájlokról beszélünk, akkor két nagy csoportot különböztetünk meg: natív (unmanaged) és felügyelt (managed állományokat). A különbség hasonló, mint pl. A C++ nyelven és C# nyelven írt programok között. Előbbi képes olyan kódot kibocsátani, amely közvetlenül a processzoron fut, utóbbi pedig a .NET környezet számára értelmezhető köztes kódot bocsát ki. Korábbi tanulmányok során tisztáztuk, hogy utóbbi megoldás kicsit lassabb futásidőt eredményezhet.

Utolsó módosítás: 2024. szeptember 13., péntek, 21:13

E-mail : moodlesupport@uni-obuda.hu

Unmanaged/natív állomány

OS/CPU számára értelmezhető bájtkód

Közvetlen hardver elérés

Bármilyen nyelven írható

Platformfüggő

Lehetőségünk van C# nyelven natív DLL-t felhasználni. Ezeket a PATH környezeti változó által leírt elérési utakról tudjuk betölteni (általában Windows, System, System32 könyvtárak) vagy az aktuális könyvtárból, ahol az exe fájlunk található. Windows operációs rendszeren a DLL-ek használata nehézkes, mert a verziózás nem megoldott. Ezért gyakran a programok az általuk igényelt verziót telepítik az exe fájl mellé, ami pazarlás.

Nézzük most meg, hogyan tudnánk C#-ban egy natív dll-t meghívni. A windows API-t fogjuk használni, amely egy natív DLL gyűjtemény. Ezen keresztül az operációs rendszer minden publikus funkcionalitása elérhető. Ezek nagyrészét a .NET elfedi előlünk támogató osztályokkal/metódusokkal. A windows apiról bővebben a pinvoke.net oldalon olvashatunk.

```
using System.Runtime.InteropServices;

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


string fname = @"c:\Windows\Media\tada.wav";
PlaySound(fname, UIntPtr.Zero, 1);
Console.ReadLine();
```

Az extern kulcsszóval jelölhetjük meg a külső függvényt. A DLL import attribútummal jelezzük, hogy melyik DLL-ben található a függvény. Ez a függvény amúgy C++-ban íródott, tehát olyan típusokat is tartalmazhat, amelyeket C#-ban nem találunk meg. Ilyen az UIntPtr típus is, amely egy unsigned int mutató (~referencia).

A natív dll hívások problémái:

Platform és OS függő kód

Paraméterek és visszatérési értékek típusai nem triviálisak adott nyelven

Nem tudjuk, hogy ki felelős a memória felszabadításáért

Nem tudjuk mi érhető el a DLL-ben, csak ha van dokumentációja

Nem tudjuk mik a DLL további függőségei

Csak futásidőben derül ki, hogy létezik-e egyáltalán a DLL és az általunk leírt metódus

Utolsó módosítás: 2024. szeptember 13., péntek, 21:14

Elektronikus és Digitális Tananyagok Iroda

Info

Moodle használati útmutató

GY.I.K.

Kapcsolat

1034 Budapest, Doberdó út 6.

Telefon : +36 1 666-5741

E-mail : moodlesupport@uni-obuda.hu

Managed/felügyelt állomány

Egy vagy több osztály van benne

.NET értelmező tud vele dolgozni csak (.NET runtime)

Nyelvfüggetlen

Platformfüggetlen (de .NET értelmező kell hozzá a célplatformon)

A korábbi tanulmányok során minden .NET osztály és metódus meghívása egy DLL hívás volt. A project-en belül a Dependencies szekcióban tároljuk el ezeket, amelyek fizikailag a .csproj fájlban vannak leírva. Például JSON kezelés esetén a telepített Newonsoft.Json nuget csomag is látható a Dependencies → Packages szekcióban.

Hogyha készítünk saját DLL fájlt, akkor az Add → Project Reference menüvel adhatjuk hozzá a solution explorerben a projekthez. Fontos megjegyzés, hogy felügyelt dll-ek esetében a fordító is ellenőrzi az állományok meglétét. Gyors betöltődés jellemzi, ugyanolyan gyors, mint a projetkben található saját kód.

DLL készítése: Add → New Project → Class Library

Tudnivalók:

Console Apphoz képest a különbség: nincs program.cs, nincs kitűntetett belépési pont

Minden osztály és metódust, amelyet el szeretnénk érni más projektből, az legyen publikus

Buildelés után a release/debug mappában megtaláljuk az elkészült DLL-t

Az elkészült DLL publikálható bárhol, ezt mások úgy tudják beemelni a projektjükbe, hogy Add → Project Reference → Browse → tallózás

Felügyelt EXE/DLL fájlok gyűjtőneve: .NET Assembly

Nézzük most meg egy exe fájl elméleti futtatási folyamatát:

1. EXE fájlra dupla kattintás

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

Felügyelt DLL kezelő 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

Utolsó módosítás: 2024. szeptember 13., péntek, 21:16

1034 Budapest, Doberdó út 6.

Telefon : +36 1 666-5741

E-mail : moodlesupport@uni-obuda.hu

Ebben a modulban a reflexió fogalmával és eszközkészletével ismerkedünk meg. A reflexió az a képesség, amellyel a program önmaga struktúráját és viselkedését futásidőben analizálni és alakítani tudja. Sok kezdő programozó először nem érti meg ezt a témakört, nem érti, hogy mi lehet a haszna.

Nézzünk egy egyszerű feladatot: szeretnénk készíteni egy függvényt, amely egy **Dog** típus összes jellemzőjén képes végig iterálni és mindegyik jellemző nevét és értékét képes kiírni a konzolra. Ezt csak úgy tudjuk megtenni, hogyha a **Dog** típus jellemzőit kézzel begépeljük.

```
void DisplayDog(Dog d)
{
    Console.WriteLine($"Name: {d.Name}");
    Console.WriteLine($"Age: {d.Age}");
    Console.WriteLine($"OwnerName: {d.OwnerName}");
    Console.WriteLine($"Color: {d.Color}");
}
```

Adott szintig ez működhet, ám 20-30 tulajdonság esetén ez már eléggé repetitív feladatnak számít és ezekre már nagyon rég megtanultuk, hogy ciklust érdemes használni. Csak ezzel egy gond van: jelenlegi ismereteinkkel nem vagyunk arra képesek, hogy egy objektum jellemzőit bejárjuk. Erre való a reflexió. Egy típusról vagy konkrét objektumról futásidőben képes olyan információkat szolgáltatni, hogyan épül fel. Többek között képes a tulajdonságait gyűjteményként visszaadni. A feladatot így lehetne megoldani:

```
void DisplayDog(Dog d)
{
    foreach (PropertyInfo propInfo in d.GetType().GetProperties())
    {
        Console.WriteLine($"{propInfo.Name}: {propInfo.GetValue(d).ToString()}");
    }
}
```

Mindjárt részletesen taglaljuk az összes építőelemet. A függvény lényege a következő: kap egy **Dog** példányt. Lekéri a **GetType()** segítségével a típusinformáció róla. Ez a **Type** objektum tartalmazza a benne lévő metódusokat, tulajdonságokat és adattagokat, továbbá még rengeteg információt (mi az őse, melyik assemblyben található, lezárt osztály-e, absztrakt osztály-e, stb.)

A **Type** típusról a **GetProperties()** visszaadja az összes tulajdonságot **PropertyInfo** gyűjteményként. Egy ilyen **PropertyInfo** elem tartalmazza a tulajdonság nevét (**Name**) és egy függvényt, amely **object** típusként visszaadja az ilyen értékét a paraméterként átadott konkrét objektumnak (**GetValue**).

A fenti reflexiós megoldás legnagyobb erőssége, hogy a paramétere lehetne **Dog** helyett **object** is a kód bármilyen módosítása nélkül és így **tetszőleges típus összes tulajdonságának kiírására képes általános célú függvényt írtunk**.

A reflexió jellemzői

Magasszintű nyelv kell hozzá (Java, PHP, C#) → mindegyikben különböző mértékű támogatás

C#-ban két használati módja van:

Futásidejű típusanalízis: Milyen tulajdonságai vannak egy bizonyos objektumnak? Milyen értékei vannak?

Futásidejű típusgyártás: System.Reflection.Emit → nem tárgyaljuk

Több technológia is használja a reflexiót:

Intellisense: metódusok, tulajdonságok listázása

Szerializáció: tulajdonságok és hozzájuk tartozó értékek kinyerése/beállítása

Tesztek: teszt metódusok kigyűjtése egy vezérlőpultra

Utolsó módosítás: 2024. szeptember 13., péntek, 21:46

Elektronikus és Digitális Tananyagok Iroda

Info

Moodle használati útmutató

GY.I.K.

Kapcsolat

1034 Budapest, Doberdó út 6.

Telefon : +36 1 666-5741

E-mail : moodlesupport@uni-obuda.hu

Nézzük meg most szépen apránként, hogy milyen lehetőségeink vannak a reflexió segítségével. Egy adott objektumról vagy típusról a típusinformációkat egy **Type** típussal tudjuk reprezentálni. Ez kinyerhető egy konkrét objektumból (t1 eset) vagy a **typeof** operátorral egy konkrét típusból is (t2 eset).

```
Dog d = new Dog()
{
    Name = "Baron",
    Age = 3,
    OwnerName = "Jack",
    Color = "white"
};

Type t1 = d.GetType();
Type t2 = typeof(Dog);
```

De egy generikus paraméterből is kinyerhetjük a futásidőben odahelyettesített konkrét típus infóit a **typeof** operátorral:

```
class OrderedList<T>
{
    public OrderedList()
    {
        Type t = typeof(T);
        Console.WriteLine($"{t.Name} type list created!");
    }
}
```

Nézzük meg, hogy milyen jellemzőket kaphatunk meg a típusra vonatkozóan:

```
string name1 = t.Name;
string name2 = t.FullName;
string name3 = t.AssemblyQualifiedName;
```

Három módon is elérhetjük a típus nevét. Az első a rövid név (pl: Dog), a második a névtérrel együtt lévő neve (pl: Models.Dog), a harmadik ahogyan a Global Assembly Cache visszaadná (pl: Dog, Temp, Version=1.0.0.0, Culture=neutral,PublicKeyToken=null).

Nézzük meg, hogy milyen lehetőségeink vannak a származtatási viszonyok felderítésére:

```
Type baseclass = typeof(Dog).BaseType;
bool result1 = typeof(Animal).IsAssignableFrom(typeof(Dog));
bool result2 = typeof(Dog).IsSubclassOf(typeof(Animal));
bool result3 = typeof(Dog).IsAssignableTo(typeof(IComparable));
```

Az első sor megadja **Type**-ként az ősosztályt. Az első kifejezés akkor igaz, hogyha a **Dog** az **Animal** leszármazottja (közvetlen vagy közvetett). Pontosabban azt jelentené, hogy referálhatjuk-e a **Dog** típust **Animal**-ként. A második kifejezés megmutatja, hogy a **Dog** konkrétan az **Animal**-nek leszármazottja-e. Az utolsó pedig megmutatja, hogy **Dog**-ot referálhatjuk-e **IComparable** referenciával.

A tulajdonságok kinyerése az alábbi:

```
PropertyInfo? info = typeof(Dog).GetProperty("Age");
PropertyInfo[]? infos = typeof(Dog).GetProperties();
```

A mezők kinyerése hasonló:

```
FieldInfo? info = typeof(Dog).GetField("name");
FieldInfo[]? infos = typeof(Dog).GetFields();
```

Illetve a metódusoké is:

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

Illetve bármelyik esetében megadhatjuk, hogy a privát tagokat is szeretnénk elérni:

```
MethodInfo[]? infos = typeof(Dog).GetMethods(BindingFlags.NonPublic);
```

Itt viszont felhívnánk arra a figyelmet, hogy a **reflexiónak nem az a célja, hogy a láthatóságokat megkerülje**. Képes rá, de hogyha csak ilyen módon tudunk megoldani egy feladatot, akkor az tervezési hiba.

Könnyen összezavarodhatunk, hogyha egy tulajdonság típusát szeretnénk megállapítani. Például kíváncsiak vagyunk, hogy egy adott tulajdonság string típusú-e. Ekkor az alábbi kód hibás lesz:

```
Type t = typeof(Dog);
PropertyInfo? nameprop = t.GetProperty("Name");
if (nameprop?.GetType() == typeof(string))
{
    Console.WriteLine($"type of string!");
}
```

Itt sosem fogunk az igaz ágba belépni, pedig a **Name** property egyértelműen **string**. Nade amikor a **nameprop** típusát kérjük le, akkor az **PropertyInfo**! A helyes megoldás az, hogy a **PropertyInfo** példánytól elkérjük, hogy ő milyen típusú tulajdonságot referál. Ez pedig a **PropertyType** tulajdonságában található.

```
Type t = typeof(Dog);
PropertyInfo? nameprop = t.GetProperty("Name");
if (nameprop?.PropertyType == typeof(string))
{
    Console.WriteLine($"type of string!");
}
```

Utolsó módosítás: 2024. szeptember 13., péntek, 21:47

Telefon : +36 1 666-5741

E-mail : moodlesupport@uni-obuda.hu

Mezők/tulajdonságok értékének írása/olvasása

Miután van referenciánk egy PropertyInfo-ra vagy FieldInfo-ra, lehetőségünk van a hozzá tartozó értéket írni vagy olvasni. Nézzünk erre egy példát.

```
Dog d = new Dog()
{
    Name = "Baron",
    Age = 3,
    OwnerName = "Jack",
    Color = "white"
};

Type t = d.GetType();
PropertyInfo? ownerInfo = t.GetProperty("OwnerName");
string? name = (string?)ownerInfo?.GetValue(d);
```

A fenti kód alsó három sorát vizsgáljuk meg.

Az első sor megszerzi a **Dog** objektumról a típusinformációt. A második sor megszerzi az **OwnerName** tulajdonság információit (ha van olyan – ezért **nullable**). Az utolsó sor pedig lekéri az ehhez a tulajdonsághoz tartozó értéket a paraméterként átadott konkrét objektumból. Ezt **object**-ként kapjuk vissza, tehát szükséges még a céltípusra alakítani.

Nézzünk egy példát olyan függvényre, amely kap egy tetszőleges típusú gyűjteményt. Végig iterál az elemein és minden **Id** nevű tulajdonságot beállít egy véletlen értékre:

```
List<Dog> dogs = new List<Dog>()
{
    new Dog() { Name = "Baron" },
    new Dog() { Name = "Brother" }
};

IdInit(dogs);

void IdInit<T>(IEnumerable<T> items){
    Type t = typeof(T);
    PropertyInfo? prop = t.GetProperty("Id");
    foreach (var item in items)
    {
        prop?.SetValue(item, Guid.NewGuid().ToString());
    }
}
```

A függvény generikus és egy T típusú tetszőleges gyűjteményt vár. A meghívásakor nem kell **IdInit<Dog>(dogs)** módon hívni, mert a **dogs** beírása miatt egyértelmű a generikus paraméter. A generikus paraméterről lekérjük a típusinfót, arról az **Id** tulajdonságot, ha van. Végig megyünk a konkrét gyűjteményen és minden elemnek az **Id** tulajdonságát a **Guid** segédosztállyal beállítjuk egy random karakterláncra (hexadecimális karakterekkel).

Metódusok végrehajtása

Egy típusról kigyűjtött metódusokkal alapvetően egy dolgot lehet csinálni: meghívni adott időben. A példában egy nagyon egyszerű teszt keretrendszert készítünk.

```
class DogTester
{
    Dog dog;

    public DogTester(Dog dog)
    {
        this.dog = dog;
    }
```

```
    private bool HasName()
    {
        return dog.Name != null;
    }

    private bool HasAge()
    {
        return dog.Age > 0;
    }

    private bool HasOwner()
    {
        return dog.OwnerName != null;
    }

    public void RunTests()
    {
        var tests = typeof(DogTester).GetMethods(BindingFlags.NonPublic | BindingFlags.Instance)?
            .Where(t => t.Name != "RunTests" && t.Name != "Finalize" && t.Name != "MemberwiseClone"
        foreach (var test in tests)
        {
            bool? result = (bool?)test?.Invoke(this, new object[] { });
            Console.WriteLine($"Test: {test.Name} result: {result}");
        }
    }
}
```

A **DogTester** egy olyan osztály, amely a konstruktorában egy **Dog** példányt vár és számtalan tesztet tud lefuttatni abból a célból, hogy az objektum valid-e. A teszteket a példa kedvéért privátként hoztuk létre. A **RunTests()** megkeresi az összes olyan metódust, amely privát és példányszintű. Kizárja az **object** őstől örökölt **Finalize()** és **MemberwiseClone()** függvényeket, mert ezeket nem akarjuk értelemszerűen futtatni. Végigmegy az összes megmaradt függvényen és visszajelez, hogy mi a teszt, sikeres volt-e vagy sem.

A **foreach**-en belül az **Invoke** futtatja le az adott függvényt. Az első paramétere az objektum, amelyen le akarjuk futtatni ezt a függvényt. A második paraméterben a paraméterlistát kell **object** tömbként átadni. Mivel ezek paramétermentes metódusok, ezért egy üres tömböt adunk át.

Megjegyzés: statikus osztály metódusainak hívásakor az első paraméter null legyen.

Objektumgenerálás

Lehetőségünk van tetszőleges típusból példányt is létrehozni a reflexió segítségével. Ez ugyan nem olyan nagy újdonság, hiszen a generikus típusoknál már tudtunk ilyet csinálni.

```
var dogs = Generate<Dog>(100);

List<T> Generate<T>(int count) where T:new()
{
    List<T> items = new List<T>();
    for (int i = 0; i < count; i++)
    {
        items.Add(new T());
    }
    return items;
}
```

Ennek a megoldásnak a hátránya, hogy a T típusra ki kellett kötni, hogy legyen paramétermentes konstruktora. De mi a helyzet akkor, hogyha a konstruktorok paraméteresek?

A reflexió az alábbi két lehetőséget biztosítja:

```
var obj1 = Activator.CreateInstance(typeof(Dog));
var obj2 = Activator.CreateInstance(typeof(Dog), new object[] { "Baron", 4, "Jack" });
```

Látható, hogy egy metódushíváshoz hasonlóan kell paraméterezni a konstruktor hívását.

Assembly analizálás

A reflexió lehetőséget biztosít arra is, hogy a típusokat futásidőben DLL-ekből gyűjtsük be.

```
Assembly assembly = Assembly.LoadFrom("models.dll");
Type[] types = assembly.GetTypes();
```

Tipikusan játékok és szoftverek kiegészítői működnek így. Az addon mappában elhelyezett pluginok/modok futásidőben kerülnek beemelésre.

Lehetőség van DLL-ek mellett a jelenlegi assembly-t is vizsgálni. Így például lehetőség van arra, hogy egy projekthez hozzáadott új osztálynak automatikusan legyen pl. menüpontja.

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

Utolsó módosítás: 2024. szeptember 13., péntek, 21:49

Ebben a modulban megismerjük az attribútumokat és használatuk célját.

Röviden: az attribútum amolyan intelligens, programozható „cetli”, amelyet osztályokra, metódusokra, adattagokra és tulajdonságokra helyezhetünk rá a forráskódban, majd reflexió segítségével lekérhetjük őket futásidőben.

Szakmai nyelven: saját metaadatokkal kiegészítünk típusokat

Nézzük meg a **Dog** osztályunkat most, amelyet elláttunk néhány beépített attribútummal. Ezek közül van olyan, amelynek csak az ottléte hordoz információt (Required – el van helyezve vagy sem valahol) illetve van olyan is, amelynek az attribútuma tartalmaz paramétereket. (Range – minimum és maximum, StringLength – maximum). Más programozási nyelveken ezt szokás annotációnak is nevezni.

```
[Serializable]
class Dog
{
    [Required]
    [StringLength(100)]
    public string Name { get; set; }

    [Required]
    [Range(1,20)]
    public int Age { get; set; }

    [StringLength(100)]
    public string OwnerName { get; set; }

    [StringLength(20)]
    public string Color { get; set; }
}
```

Fontos megjegyezni, hogy ezek az **attribútumok alapvetően nem befolyásolnak semmilyen viselkedést**. Tehát ha létrehozunk egy kutyát és az életkorát 30-ra állítjuk, attól még az objektum gond nélkül létre fog jönni.

Viszont lehetőségünk van olyan ellenőrző függvényeket írni, amelyek végig mennek egy objektum minden tulajdonságán, minden tulajdonságról lekérik az összes attribútumot és megnézik hogy a benne lévő pl. validációs szabályok mindegyike teljesül-e.

Rengeteg beépített technika használ attribútumokat:

NewtonSoft.Json → [JsonIgnore] tulajdonság kihagyása a szerializációból

XmlSerializer → [NonSerialized] mező kihagyása az XML-ből

Adatbáziskezelés → [Key], [Required], [MaxLength], [Range], stb.

Tesztelés → [TestFixture], [TestCase]

Intelligens kommentek → [Obsolete], [DisplayName], [Description]

Utolsó módosítás: 2024. szeptember 13., péntek, 21:54

Elektronikus és Digitális Tananyagok Iroda

Info

Moodle használati útmutató

GY.I.K.

Kapcsolat

1034 Budapest, Doberdó út 6.

Telefon : +36 1 666-5741

E-mail : moodlesupport@uni-obuda.hu

Most megnézzük, hogy reflexióval hogyan érhetjük el az attribútumokat egy tagról.

Maradjunk még ennél a kutyás példánál. A Name tulajdonságról szeretnénk lekérni a StringLength-et.

```
class Dog
{
    [Required]
    [StringLength(100)]
    public string Name { get; set; }
}
```

Az alábbi kóddal tehetjük ezt meg:

```
Type t = typeof(Dog);
PropertyInfo? propInfo = t.GetProperty("Name");
StringLengthAttribute? sla = propInfo?.GetCustomAttribute<StringLengthAttribute>();
int? length = sla?.MaximumLength;
```

Az első sorban megszerezzük a **Dog** osztály típusinfóit. Lekérjük a kívánt tulajdonság típusinfóit. Lekérjük az adott tulajdonságról a **StringLengthAttribute**-ot. Ezt **nullable** típusként hozzuk létre, mert nem biztos, hogy lesz ilyen attribútum a tulajdonságon. A lekért attribútumból kiolvasható a beleírt paraméter.

Léteznek olyan attribútumok, amelyeket többször is el lehet helyezni egy-egy tagon más-más paraméterekkel. Hogyha le akarjuk kérni az összeset, akkor listaként kaphatjuk meg. A példa csak szintaxist mutatja be, a **StringLength**-et nem lehet kétszer elhelyezni egy tagon.

```
var slas = propInfo?.GetCustomAttributes<StringLengthAttribute>();
```

Az alábbi kód az összes attribútumot megadja egy tagról.

```
var attrs = propInfo.GetCustomAttributes();
```

Nézzük meg egy példán keresztül az attribútumok lekérésének hasznát. A **Dog** osztályt kiegészítjük **[DisplayName]** attribútumokkal, amelyben magyarul írjuk le a tulajdonságok neveit, mert pl. elvárt, hogy a szoftver felülete magyar legyen.

```
class Dog
{
    [DisplayName("Név")]
    public string Name { get; set; }

    [DisplayName("Életkor")]
    public int Age { get; set; }

    [DisplayName("Gazdi neve")]
    public string OwnerName { get; set; }

    [DisplayName("Szín")]
    public string Color { get; set; }
}
```

Ezután készítünk egy olyan függvényt, amely képes egy tetszőleges típusokból álló lista táblázatos megjelenítésére úgy, hogy a táblázatfejlécben a **[DisplayName]**-ben leírt értéket használja.

```
void ToTable<T>(IEnumerable<T> items)
{
    var titles = typeof(T)
        .GetProperties()
        .Select(t => t.GetCustomAttribute<DisplayNameAttribute>()?.DisplayName);
    Console.WriteLine(string.Join("\t\t", titles));
```

```
    foreach (var item in items)
    {
        foreach(var prop in typeof(T).GetProperties())
        {
            Console.Write(prop.GetValue(item) + "\t\t");
        }
        Console.WriteLine();
    }
}
```

A függvény első része kigyűjti a tulajdonságokat, illetve ezekről elkéri a **[DisplayName]**-hez tartozó értéket. A **string.Join()** függvény ezeket egy nagy stringgé alakítja dupla tabokkal szeparálva. Ebből lesz a táblázat fejléce.

A függvény alsó része bejárja a tetszőleges lista összes elemét és minden elem összes tulajdonságát és ezeket táblázatosan kiírja.

Utolsó módosítás: 2024. szeptember 13., péntek, 21:57

Saját attribútumok készítése

Célunk lehet a beépített attribútumok használatán felül sajátok készítése. Ehhez le kell származnunk az **Attribute** ősosztályból. Illetve elvárt az attribútum nevébe beleírni, hogy **Attribute**, ahogyan saját **Exception**-ök esetén is beleírjuk, hogy **Exception**.

Elkészítünk egy olyan attribútumot, amely adott nyelven való fordítását tartalmazza az adott tulajdonság nevének.

Az **AttributeUsage** attribútum által leírt jellemzők:

AttributeTargets: hol lehessen használni? Ez esetben csak tulajdonságon

AllowMultiple: többször is rá lehessen-e tenni egy tulajdonságra. Ez esetben igen.

```
[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;
    }
}
```

Amikor elhelyezzük egy osztályon, akkor már az **Attribute** végződést nem kell kiírni.

```
class Dog
{
    [Translation("hu", "Név")]
    [Translation("de", "Name")]
    public string Name { get; set; }

    [Translation("hu", "Életkor")]
    [Translation("de", "Jahre")]
    public int Age { get; set; }

    [Translation("hu", "Tulajdonos")]
    [Translation("de", "Besitzer")]
    public string OwnerName { get; set; }

    [Translation("hu", "Szín")]
    [Translation("de", "Farbe")]
    public string Color { get; set; }
}
```

Utolsó módosítás: 2024. szeptember 13., péntek, 21:58

Info

Moodle használati útmutató

GY.I.K.

Kapcsolat

1034 Budapest, Doberdó út 6.

Telefon : +36 1 666-5741

E-mail : moodlesupport@uni-obuda.hu

Ebben a modulban az adatbázisokkal való kommunikációt tanuljuk meg. Felhívnánk a figyelmet, hogy az adatbáziskezelés megismerése nem része a tananyagnak, elvárt ezen a tárgyon az előismerete vagy legalább a tárggyal párhuzamos hallgatása.

Járjunk körül rögtön a kérdést: miért kell nekünk egyáltalán adatbáziskezeléssel foglalkozni?

Az elmúlt modulokban megtanultuk a saját struktúrával kialakított szöveges fájlok helyett az adataink XML vagy JSON fájlokban való tárolását. Ezeknél eljutottunk körülbelül 2-3 soros megoldásokhoz is, ahol beépített módszerekkel képesek vagyunk szerializálni és deszerializálni a gyűjteményeinket.

A fájlokban való tárolással viszont lehetnek problémák

Nem biztosítanak tranzakciókezelést, vagyis annak a kivédését, hogyha egy írási művelet megszakad

Több szálú környezetben, ahol az alkalmazásunk szerverként viselkedik majd, egyszerre egy szál képes a fájlt írni vagy olvasni, addig a többieknek várakozni kell

Szintén szálak esetén nem biztosított az atomi végrehajtás, hogyha egy szálról érkezik egy adatbeszúrás, rögtön egy másodiktól az adat törlése majd ezután néhány ms-al az előző szál ellenőrzi, hogy bekerült-e az adat, akkor fals eredményt kap

A felsorolt esetek mindegyikét kivédhetjük különböző algoritmusok implementálásával, a szálakat egy várósorba helyezhetjük (Queue), az atomicitást biztosíthatjuk lock blokkokkal. Mindenre lenne megoldásunk C# nyelven. De pont ezeket a megoldásokat implementálja egy adatbáziskezelő rendszer. Leveszi a vállunkról a terhet és megoldja az összes ilyen adatintegritási problémát. Tehát célszerű lenne ilyet használni.

Az adatbázis kezelő rendszer gyakorlatilag egy program, amelyet telepítünk egy számítógépre/kiszolgálóra. Nem feltétlenül tartozik hozzá grafikus felület a táblák kezelésére. Alapvetően egy bizonyos portszámon figyel és a saját adatbázis motorának nyelvén vár tőlünk utasításokat és ezekre válaszol azzal, hogy sikeresen mentette/módosította/törölte amit kértünk vagy visszaadja a konkrét lekérdezésünkre a választ.

Néhány elterjedt adatbázis motor

MySQL (port: 3306)

MSSQL (port: 1434)

PostgreSQL (port: 5432)

Oracle (port: 1521)

MongoDb (port: 27017)

SQLite (nem TCP/UDP)

Az előzőekben felsoroltak akár mindegyike telepíthető a gépre és futtatható egyidőben, hiszen más-más portszámon kommunikálnak. A MondoDb egy kicsit kakukktojás ezek közül, mert nem relációs adatbáziskezelő és nem is SQL nyelvet használ, az SQLite pedig nem a hálózaton kommunikál, csupán tetszőleges programozási nyelvhez egy extension. Ezeken felül még számtalan fajtájú és nyelvű adatbázis motor érhető el.

A tantárgy során az MSSQL-el fogunk ismerkedni, amely a Microsoftnak a saját SQL implementációja és C# nyelvhez a legjobban illeszkedik. De bármelyik másik felsorolt adatbázismotorral tudnánk C#-ból kommunikációt létesíteni.

Windows operációs rendszer esetén három módon tudunk MSSQL-t igénybe venni

Microsoft SQL Server 2022 telepítésével, amely licenszköteles termék és professzionális, skálázható adatbázismotort n
(tanulásra nem ajánljuk)

Microsoft SQL Server 2022 Express telepítésével, amely ingyenes termék, csak 1 CPU magot használ ki, maximum 1GB ramot használ fel és egy adatbázis mérete 10GB-ban van limitálva (tanulásra tökéletes)

LocalDb igénybevételével, ami valódi szerver helyett egy azonos funkcionalitást nyújtó library, amelyet a Visual Studio telepítésekor a „Data Storage and Processing" workload kiválasztásával tudunk telepíteni, akár utólag is (tanulásra szintén tökéletes)

MacOS operációs rendszer esetén (Intel és Apple M1/M2 esetén is) egy lehetőségünk van

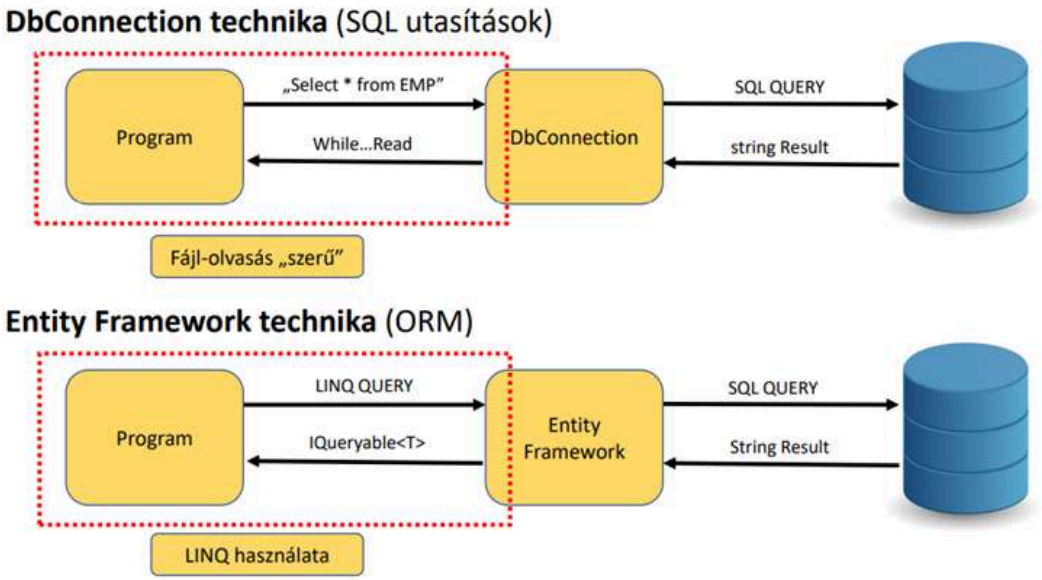
Docker telepítése, valamint SQL server image letöltése és konténerként indítása

Nézzük meg, hogy C# nyelvből hogyan tudnánk kapcsolódni egy adatbázishoz:

ADO.NET DbConnection technikával, ami SQL utasításokat string formájában vár, az eredményeket object tömbként adja vissza. Ezekből a számukra szükséges típusokat kasztolnunk kell

Entity Framework könyvtárral, amely egy úgynevezett ORM réteget (Object Relational Mapping) képez az adatbázis és a C# kód között. A táblákat mint objektumgyűjteményeket érjük el, amelyeken elérhetőek Add(), Remove() függvények is

A két technika közti különbséget szemlélteti az alábbi ábra:



Látható, hogy az Entity Framework által visszaadott gyűjtemény **IQueryable<T>** típusú és ami számukra a legfontosabb: **LINQ**-zható!

Utolsó módosítás: 2024. szeptember 14., szombat, 17:02

Info

Moodle használati útmutató

GY.I.K.

Kapcsolat

1034 Budapest, Doberdó út 6.

Telefon : +36 1 666-5741

E-mail : moodlesupport@uni-obuda.hu

Az előző modulban megtanultuk, hogy két lehetőségünk van C# kódból adatbázishoz csatlakozni:

ADO.NET DbConnection segítségével, string formátumban SQL lekérés elküldésével

Entity Framework ORM segítségével, amely az adatbázistáblánkat egy LINQ-zható gyűjteményként láttatja velünk

Értelemszerűen a második lehetőség csábítóbb, de nagyon röviden nézzük meg az elsőt.

Telepítsük a **System.Data.SqlClient** nuget csomagot, majd hozzuk létre a kapcsolatot!

```
string connStr = @"Data Source=
(LocalDB)\MSSQLLocalDB;Initial Catalog=marveldb;Integrated Security=True";

using(var conn = new SqlConnection(connStr))
{
    SqlCommand cmd = new SqlCommand("select * from MOVIES", conn);
    using (var reader = cmd.ExecuteReader())
    {
        while (reader.Read())
        {
            Console.WriteLine(reader["Title"].ToString());
        }
    }
}
```

Az első sorban egy úgynevezett connetion stringgel kell megadnunk az adatbázis helyét. A példában LocalDb megoldást használtunk. Az SqlCommand látható, hogy stringként küldi ki az SQL utasítást, a választ egy reader objektummal kell bejárnunk, hasonlóan ahhoz, mint amikor egy szöveges fájlt dolgozunk fel soronként.

Nézzünk egy példát beszúrásra is:

```
using(var conn = new SqlConnection(connStr))
{
    SqlCommand cmd = new SqlCommand("insert into MOVIES (MovieId,Title,DirectorId) values(28, 'Doct
    int affected = cmd.ExecuteNonQuery();
    Console.WriteLine("rows modified: " + affected);
}
```

Az egész adatbáziselérés így roppant körülményes és ráadásul kódszinten védekezni kell az SQL injection támadás ellen. Nézzük az alábbi lekérdezést:

```
string uName = "yyy";
string uPass = "xxx";
string sql = $"SELECT * FROM users WHERE username='{uName}' AND userpass=sha2('{uPass}')";
```

A felhasználónév (uName) és jelszó (uPass) változókat képzeljük el, hogy egy webes bejelentkező felületről jönnek. Meg akarjuk nézni az adatbázisban, hogy van-e ilyen felhasználó ezzel a felhasználónévvel és jelszóval. Hogyha igen, akkor visszaküldjük neki a titkos adatokat.

Mi történik, hogyha egy támadó a felületen a jelszó mezőbe ezt gépeli be:

```
x') OR 1=1 OR 1<>sha2('x
```

Az SQL utasítás így fog összeállni:

```
SELECT * FROM users WHERE username='Joe' AND userpass=sha2('x') OR 1=1 OR 1<>sha2('x')
```

És mivel az 1 = 1 mindig igaz, ezáltal a VAGY kapcsolat mindig igaz, úgy fogja érzékelni a rendszerünk, hogy jó jelszót írt be.

A jó hír, hogy az ORM rendszer önmagában megoldja nekünk az SQL injection elleni védelmet is!

Entity Framework

Az Entity Framework azon túl, hogy egy meglévő adatbázisból a lekérdezéseket és módosításokat egyszerűbb tudja tenni, képes C# osztályokból táblákat is generálni. Ezáltal kétfajta megközelítést alkalmazhatunk:

Dbfirst módszer

A lényege, hogy már van egy kész adatbázis, táblákkal, mezőkkel, kapcsolatokkal, netán adatokkal feltöltve. Ez vagy SQL tábla létrehozó utasításokkal készítették el vagy pedig valamilyen adatbázis menedzsment szoftverrel (pl. Microsoft SQL Server Management Studio vagy Azure Data Studio). Ehhez szeretnénk egy olyan C# nyelvű programot írni. Ekkor az Entity Framework képes az ORM réteget generálni az adatbázisból és a különböző adattáblákból képes C# osztályokat generálni.

CodeFirst módszer

Elterjedtebb megközelítés. Szeretnénk egy szoftver írni, amely objektum orientáltan készül, a kezdeti fázisban képes a gyűjteményeket pl. XML/JSON-be szerializálni. És egy ponton szeretnénk áttérni arra, hogy SQL adatbázist használjunk. Ekkor az Entity Framework képes az osztályainkból adatbázis táblákat generálni. Persze nem fogja tudni kitalálni, hogy mi legyen az elsődleges kulcs, mit szeretnénk idegen kulcsként használni, stb. Ezeket a többlet adatokat közölni kell az EF-el.

A jelenlegi modulban a **DbFirst** módszert nézzük meg. Generálunk a gépünkön egy adatbázist SQL scripttel, ami fel is tölti minta adatokkal. Ezt úgy vesszük, mintha egy régóta létező adatbázis lenne és készítünk hozzá egy konzolos C# nyelvű kliensprogramot. A következő modulban lesz szó arról, hogyan készítsünk adatbázist a meglévő programunkhoz.

Adatbázis létrehozása és feltöltése

A Visual Studioban navigáljunk az alábbi menübe

View → SQL Server Object Explorer → SQL Servers → localdb.. → Databases → jobb kattintás → Add New Database

Az adatbázisnak adjunk valamilyen nevet

Ezután az adatbázison jobb klikk → New Query…

És ide beszúrhatjuk azt az SQL scriptet, ami létrehozza a táblákat és feltölti adatokkal

Ezután a táblák tartalma böngészhető és szerkeszthető

C# osztályok generálása

Az alábbi nuget csomagokra lesz szükségünk (mindegyikből olyan főverziót válasszunk, amely az adott .NET verziónkhoz illeszkedik. NET 7.0 esetén: 7.0.11)

Microsoft.EntityFrameworkCore

Microsoft.EntityFrameworkCore.Proxies

Microsoft.EntityFrameworkCore.SqlServer

Microsoft.EntityFrameworkCore.Tools

Nyissuk meg a Tools → Nuget Package Manager → Package Manager Console-t és illesszük be az alábbi utasítást:

```
Scaffold-DbContext "Data Source=(LocalDB)\MSSQLLocalDB;Initial Catalog=marveldb;Integrated Security=True" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

Az Initial Catalog= után a saját adatbázisunk nevét írjuk be!

Ezután létrejön egy **...DbContext** osztály, amelyet példányosíthatunk és elkezdetjük a LINQ lekérdezések írását!

```
MyDbContext ctx = new MyDbContext();
var movies = ctx.Movies.Where(t => t.Title.Contains("Avengers"));
```

Utolsó módosítás: 2024. szeptember 14., szombat, 17:03

Ebben a modulban azt nézzük meg, hogyan tudjuk egy meglévő C# alkalmazás adatkezelési rétegét (perzisztencia rétegét) fájlkezelés helyett adatbázisba kiszervezni.

Első lépésként szükséges telepítenünk az alábbi nuget csomagokat (ugyanazokat, mint DbFirst módszer esetén):

Microsoft.EntityFrameworkCore

Microsoft.EntityFrameworkCore.Proxies

Microsoft.EntityFrameworkCore.SqlServer

Microsoft.EntityFrameworkCore.Tools

Illetve szükségünk van egy connection stringre:

```
string connStr = @"Data Source=
(LocalDB)\MSSQLLocalDB;Initial Catalog=persondb;Integrated Security=True;MultipleActiveResultSets=t
```

CodeFirst esetben nem szükséges előre az SQL Server Object Explorerben elkészíteni az üres adatbázist, az Entity Framework létre fogja hozni a connection stringben megjelölt adatbázist (ebben az esetben a persondb-t).

Nézzük most egy olyan C# osztályt, amelyből adattáblát szeretnénk készíteni. El kell látnunk néhány attribútummal.

```
public class Person
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }

    [StringLength(100)]
    [Required]
    public string Name { get; set; }

    [StringLength(100)]
    [Required]
    public string Job { get; set; }

    [Required]
    public int Age { get; set; }
}
```

Az attribútumok közül ami kötelező az a [Key]. Muszáj egy tulajdonságot elsődleges kulcsként megjelölni. Minden további attribútum csak arra szolgál, hogy finomhangolja a mezők létrehozását. A [StringLength]-et azért célszerű minden stringen megadni, mert az adatbázisok is mindig kérik a hoszt, ezáltal tudnak egy optimális sémát és indexelést kialakítani. Illetve az adatbázisoknál minden mezőre be szokás állítani, hogy kötelező-e vagy sem.

A következő lépés egy **PersonDbContext** osztály létrehozása, amelyet most mi magunk írunk meg kézzel, míg **DbFirst** esetben ez is ki lett generálva.

```
public class PersonDbContext : DbContext
    {
        public DbSet<Person> People { get; set; }

        public PersonDbContext()
        {
            Database.EnsureCreated();
        }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
```

```
        {
            string connStr = @"Data Source=
(LocalDB)\MSSQLLocalDB;Initial Catalog=persondb;Integrated Security=True;MultipleActiveResultSets=t
            optionsBuilder.UseSqlServer(connStr);
            base.OnConfiguring(optionsBuilder);
        }
    }
```

A **DbSet<>** egy gyűjtemény, ami egy táblát fog reprezentálni. Ezen tudunk LINQ lekérdezéseket futtatni.

Az **OnConfiguring()** metódus arra szolgál, hogy beállítsuk az SQL szerver elérését.

A konstruktorban található **EnsureCreated()** pedig megbizonyosodik róla, hogy létezik-e már az adatbázis. Ha nem, akkor létrehozza.

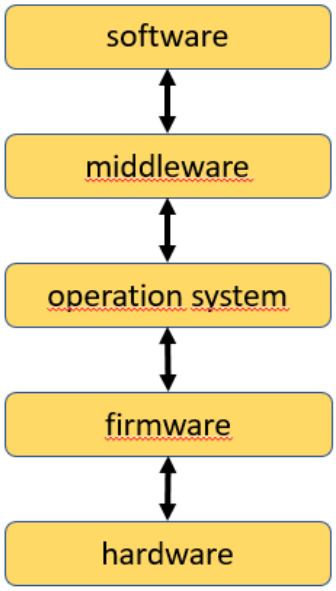
A főprogramban példányosítható a **DbContext** osztály és már tudunk is dolgozni a táblával:

```
PersonDbContext ctx = new PersonDbContext();
ctx.People.Add(new Person()
{
    Name = "John",
    Age = 33,
    Job = "driver"
});
ctx.SaveChanges();
```

Utolsó módosítás: 2024. szeptember 14., szombat, 17:08

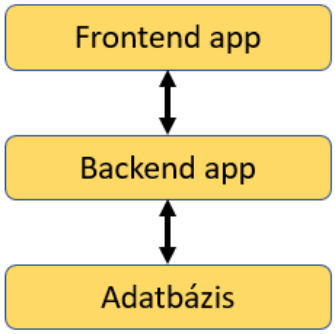
A rétegek célja, hogy elrejtsük az egyes implementációs részleteket, hogy egy felső réteg ne függjön egy nem szomszédos alsó rétegtől.

Az szemlélteti, hogy egy adott szoftvernek, legyen az egy általunk írt C# alkalmazás, neki teljesen mindegy, hogy milyen proccesszor van a gépen és milyen operációs rendszeren futtatunk. Egyedül egy .NET futtatható környezetet vár el.

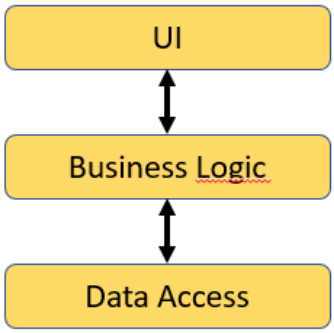


Két féle felbontásról beszélhetünk, ezek a **Tier** és a **Layer**.

A Tier egy fizikai felbontása az alkalmazás komponenseknek. Ezek szoftver vagy hardver komponensek amik a rétegeket futatják. Nem feltétlenül azonos felbontás, mint a rétegek.



A layer egy logikai felbontása az osztályoknak. Az egy rétegbe tartozó osztályok egy közös, magasabb rendű feladotot látnak el. Egyértelműen definiált hogy mi érhető el kivűlről. A rétegek között interfészekkel teremtünk kapcsolatot. Az alkalmazásunk akkor lesz megfelelően rétegezve, ha egy réteg lecserélhető egy másik komponensre amely másképpen van imlepentálva de azonos interfészekkel dolgozik.



Utolsó módosítás: 2024. szeptember 28., szombat, 21:30

Elektronikus és Digitális Tananyagok Iroda

Info

Moodle használati útmutató

GY.I.K.

Kapcsolat

1034 Budapest, Doberdó út 6.

Telefon : +36 1 666-5741

E-mail : moodlesupport@uni-obuda.hu

Ahhoz, hogy megfelelően rétegezhessük az alkalmazásunkat néhány elvet le kell fektetnünk.

A SOLID elvek azért lettek kitaláva, hogy a szoftverek könnyebben megérthetők és karbantarhatók maradjanak.

Az elvek:

Single Responsibility: Egy osztály egy felelősségi kör elve. Lényege, hogy egy osztály vagy metódus csak egyetlen egy dologért feleljen. Ne írjunk olyan osztályt ami egyszerre elvégzi a fáljba írást, adatbázis müveletet és egyéb üzleti logikákat.

Open/Closed Principle: Egy osztály zárt a módosításra de nyitott a bővítésre.Ne írjunk át egy kódot helyette bővítsük leszármazotakkal.

Liskov Substituition: Ősosztály helyett bárhol a kódban legyen lehetőség a leszármazottat használni hiba nélkül

Interface seggragation: Sok kicsi interfész egy nagy helyett.

Dependency Inversion: Egy osztály ne függjön egy másik osztálytól vagy másképpen fogalmazva konkrét implementációtól. Helyette interfészektől függjünk.

Utolsó módosítás: 2024. szeptember 28., szombat, 21:40

A félév során egy egyszerűbb 3 rétegű architektúrával ismerkedik meg minden hallgató.

A 3 réteget a következőképpen nevezhetjük: Infrastrucure, Application, Presentation.

Ezek kiegészíthetjük a Domain réteggel ami a Modelleket fogja tartalmazni és jelen tudásunkkal minden rétegben felhasználható. Illetve egy Test réteg, ami valójában egy önálló projekt ami az alkalmazásunkhoz kapcsolódik nem pedig egy réteg.

Infrastructure:

> Hogyan van az adatunk elmentve? Relációs adatbázis, nem relációs adatbázis stb. Teljesen mindegy, de ennek a rétegnek lesz a feladata a kommunkáció. Szakirodalmakban különböző patterneket találhatunk itt mint a Repository pattern. A tananyag viszont nem tartalamazza a patternek elsajátítását így mi ebben az esetben úgy nevezett "DataProvider"éeket fogunk létrehozni. Ezeknek egyetlen feldata lesz. A szükséges adatok lekérése az adatbázisból. Ehhez mi mi egy Persistence.MsSql projektet fogunk mindig majd létrehozni.
>
> Ő az egyetlen aki entity frameworkot ismeri (lásd adatbázisok), az üzleti logika felé biztosítja az adatokat.

Application:

> A üzleti logika helye. Ha adatra van szüksége azt az Infrastructuren keresztül kéri le, majd ezeket esetleg továbbítja a megjelenítés réteg felé. Ezenfelül biztosíthat, byonolutabb műveleteket is.

Presentation:

> A megjelenítési réteg. Példányosítja az applicationt. Felhasználói interakciókat továbbítja az üzleti réteg felé. Semmiféle tudomása nincs arról, hogy milyen adatbázissal dolgozik az alkalmazás, egyedül az Application rétegről tud. Illetve minden példányosítás itt történik.
>
> Egy Console-s alkalmazás esetén csak ez a réteg rendelkezhet Console.Read/Write metódusokkal.
>
> Az üzleti logika eredményeit jeleníti meg. Tipp: ConsoleMenu-Simple nuget csomag à menügenerálás

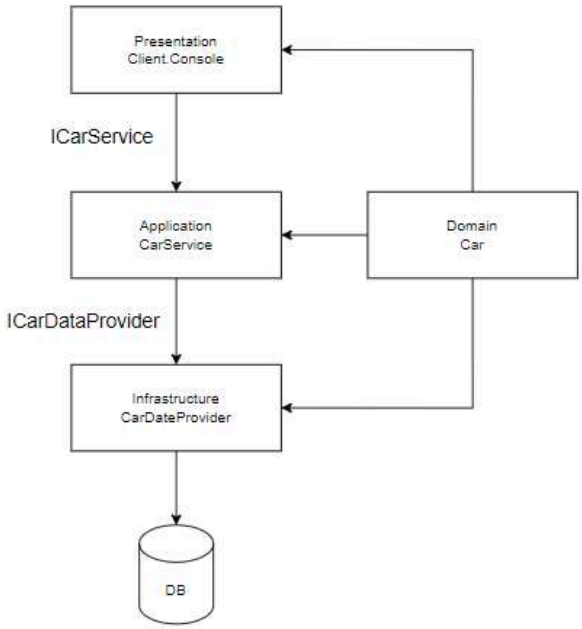
Domain:

> Az alkalmazás által használt entitások helye. Vannak architektúrák ahol hivatalosan ez nem is réteg. Viszont a mi megfogalmazásunkban ezt a réteget mindenkinek ismernie kell. Ezzel a későbbiekben lehet problémam, ugyanis az egyes rétegek logikától függően nem feltétlenül kéne, hogy felhasználják a teljes egyed osztályunkat, hanem ott abban a rétegben egy saját osztállyal kellene dolgozni annyi tulajdonsággal amennyire ott szükségünk van. Ez lehet több de lehet kevesebb is mint mondjuk amivel az adott egyed osztály amúgy rendelkezne

Tests:

> Az alkalmazáshoz kapcsolódó unit tesztek helye

Az interfészek jelentősége: Minden réteg között szükségesesek, nem függhetünk konkrét implementációtól.



Ha interfészeken keresztül kommunikálunk valahol példányosíteni kell. Ennek a helye a Console-os alkalmazás lesz. Mivel ennek van program belépési pontja, ekkor kell nekünk meghatározni, hogy az egyes interfészek hogyan jöjjenek létre, milyen példányokkal stb.
Az alábbi kód részlet a Program.cs első soraiban adjuk meg.

```
1  var host = Host.CreateDefaultBuilder()
2      .ConfigureServices((hostContext, services) =>
3      {
4          //TODO: Add services
5          services.AddTransient<AppDbContext>();
6          services.AddSingleton<ICarServiceDataProvider, CarDataProvider>();
7          services.AddSingleton<ICarService, CarService>();
8      })
9      .Build();
10
11 await host.StartAsync();
12
13 using IServiceScope serviceScope = host.Services.CreateScope();
14
15 IServiceProvider serviceProvider = serviceScope.ServiceProvider;
16
17 var carService = serviceProvider.GetService<ICarService>();
```

A Host.CreateDefaultBuilderrel mondhatjuk meg milyen szervíz konfigurációkat szeretnénk felvenni.

Amire ügyelni kell, hogy azzal kell kezdeni aminek nincs függősége. Az AppDbContext jelöli az adatbázis kontextusoknak. Ennek nincsenek függőségei. Viszont a DataProvider osztálynak van egy függősége még pedig maga az AppDbContext. Ha ezt a két sort felcseréljük a program nem tud elindulni, ugyanis a CarDataProvider példányosításánél hiányolni fogja az AppDbContext példányt.

Mit mond nekünk a **service.AddSingleton<ICarServiceDataProvider, CarDataProvider>**? Ekkor definájuk azt, hogy milyen interfészre milyen implementációt példányosítunk (ragasztunk) milyen élteciklussal.

Az osztályok életciklusa a következő félévekben, specializációs tárgyaknál jobban előjönnek. Viszont amit érdemes tudni, hogy amikor elindítunk egy alkalmazást akkor az egyes osztály példányoknak különböző életciklust adhatunk meg, hogy meddig éljen az adott példány amikor használjuk. Az ismert életciklusaink: Transient, Singleton, Scoped.

Többynire a Singletont fogjuk használni. Az Ő életciklusa addig tart ameddig a program fut és mindig egy darab van az adott osztály példányunkból. Esetünkben a CarDataProvider osztályból egyetlen egy jön létre a program indításakor és akkor fog megszűnni amikor bezárjuk az alkalmazásunkat.

A Tranziens életciklus minden egyes kéréskor új példányt hoz létre a szolgáltatásból. Ez azt jelenti, hogy minden egyes feloldott függőségnek saját egyedi példánya lesz. Az adatbázist ezzel hozzuk létre.

A Scoped lifetime az alkalmazáson belül minden egyes hatókörhöz vagy logikai művelethez egy új szolgáltatáspéldányt hoz létre. A hatókör egy bizonyos kontextust, például egy webes kérést vagy egy munkaegységet képvisel. Az azonos hatókörön belül feloldott függőségek ugyanazt a példányt kapják.

Utolsó módosítás: 2024. október 6., vasárnap, 22:20