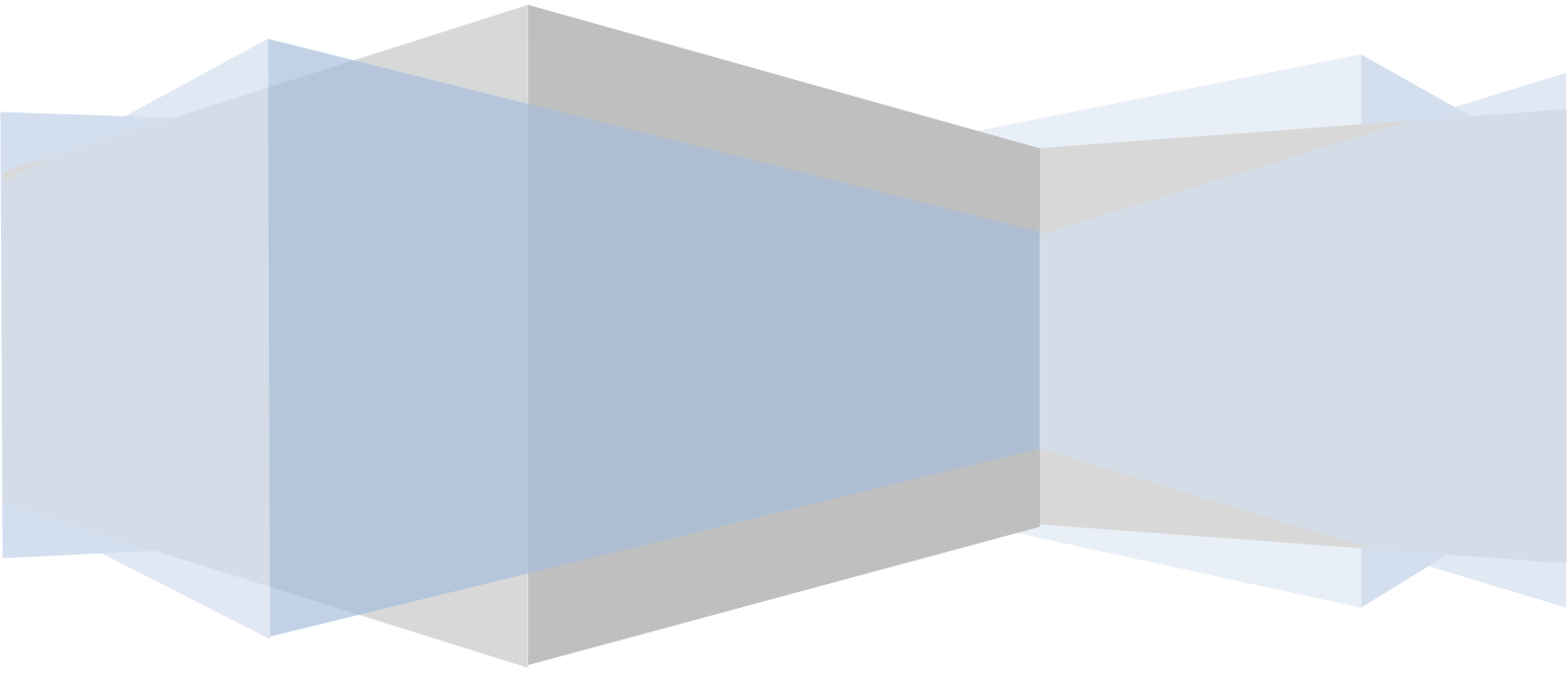
BME AAIT

Nyelvi eszközök 1.

Szoftvertechnikák

Benedek Zoltán

2024.02.25.



Tartalom

Tartalom

Property (tulajdonság)

Az osztályaink tagváltozóit általában nem célszerű publikussá tenni, mert akkor nem garantálható az objektumok konzisztens állapota (pl. a születési év tagváltozónak negatív érték is adható). Erre klasszikus megoldás az, ha a tagváltozókat védetté tesszük, majd lekérdezésükhöz egy getXXX, szabályozott beállításukhoz egy setXXX tagfüggvényt vezetünk be. A C# nyelvben erre egy szintaktikailag egyszerűbb megoldás is van, melynek neve **property** (**tulajdonság**). A főbb jellemzők a következők (a következő példa alapján):

- A definíciója a tagváltozókéhoz hasonló, de megadunk egy get {} és egy set {} ágat is.
- A get {} ág a tulajdonság *lekérdezésekor* fut le. Ha nem írunk get{} ágat, akkor a tulajdonság nem kérdezhető le.
- A set {} ág a tulajdonság állításakor fut le. Ha nem írunk set{} ágat, akkor a tulajdonság nem állítható be.
- A tulajdonság egy közönséges nyelvi elem lett, osztályoknak most már nemcsak tagváltozói és tagfüggvényei lehetnek, hanem tulajdonságai is.
- Egy objektum adott tulajdonságának elérése olyan szintaktikával történik, mintha az objektum tagváltozója lenne.

```
class Person
{
    private string name;
    private int yearOfBirth;

    //  Name nevű property bevezetése
    public string Name
    {
        get { return name; }
        set
        {
           if (value == null)
                  throw new ArgumentNullException(nameof(Name));
           name = value;
        }
    }

    //  YearOfBirth nevű property bevezetése
    public int YearOfBirth
    {
        get { return yearOfBirth; }
        set
        {
            if (value < 1800 || value > 5000)
                  throw new ArgumentException("Invalid yearOfBirth value.");
           yearOfBirth = value;
        }
    }

    // Számított érték, csak olvasható property.
    public int Age
    {
        get { return DateTime.Now.Year- yearOfBirth; }
    }

    // Ez kell ahhoz, hogy konzisztens legyen!
    // A property-n keresztül állítjuk, hogy meglegyen a validáció.
    public Person(string myName, int yearOfBirth)
    {
        Name = myName;
```

```
        YearOfBirth = yearOfBirth;
    }
 }

class Program
{
    static void Main(string[] args)
    {
        Person p1 = new Person("Béla", 1980);
        p1.YearOfBirth = 1995; // set-et hív
        Console.WriteLine("Név: {0}, kor: Age:{1}", p1.Name, p1.Age); // get-et hív

        p1.Age = 20; // Fordítási hiba, nincs set
        p1.YearOfBirth = 1000; // Futás közbeni hiba
        ...
    }
}
```

A property minden .NET nyelvben elérhető, nem csak C#-ból (csak más a szintaktika).

Auto-implementált tulajdonság (Auto-implemented property)

.NET-ben gyakran fordul elő, hogy olyan tulajdonságot készítünk, mely lekérdezéskor pusztán visszaadja egy tagváltozó értékét, beállításkor egyszerűen minden validáció nélkül beállítja azt. Pl.:

```
class Person
{
    public string name;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```

Jelen tudásunk alapján még nem érthető, mi értelme lehet ennek, hiszen az egészet kiválthatnánk egy normál publikus tagváltozó használatával. Bizonyos .NET „modulok” viszont kifejezetten építenek a property-k meglétére (pl. Form-ok estében az adatkötés), így mégiscsak nem is ritkán találkozunk hasonlóval. C# 3.0-tól kezdve lehetőség van auto-implementált property írására, amivel a fenti példa tömörebb formába hozható:

```
class Person
{
    public string Name
    {
        get; set;
    }
}
```

Ha a get és a set kulcsszavak után nem adunk meg implementációt, akkor auto-implementált property keletkezik. Az osztály a property-hez generál egy láthatatlan (kódból nem is elérhető) tagváltozót, és a property lekérdezésekor ennek értékét adja vissza, illetve ezt állítja.

Alapértelmezett érték (default value)

Az autoimplementált tulajdonságok esetében megadható a kezdeti értékük is a deklaráció során.

Adjunk kiinduló értéket a Name tulajdonságnak.

```
    public string Name { get; set; } = "anonymous";
```

Tulajdonságok láthatósága

Lehetőségünk van, hogy a get vagy a set ágra vonatkozóan szigorítsuk a láthatóságot. Csak szigorítani lehet az eredeti láthatóságot, lazítani nem. Pl. az alábbi osztályban a property-t a külvilág csak olvasni tudja, míg a saját műveletei írni is:

```
class Person
{
    public string Name
    {
        get;
        private set;
    }
}
```

Ez a példa autoimplementált tulajdonságokra mutatott példát, de nem autoimplementált esetre ugyanez érvényes.

Csak olvasható tulajdonság (readonly property)

Lehetőség van a get vagy a set ág elhagyására. A gyakorlatban a csak set ággal rendelkező tulajdonságoknak (csak írható tulajdonság) nem sok értelme van, így ezt mostantól nem tárgyaljuk. Amennyiben csak get ágat írunk, egy **olyan tulajdonságot kapunk, mely csak olvasható (readonly).** Autoimplementált tulajdonság esetén ennek is adható kezdőérték: erre csak konstruktorban, vagy alapértelmezett értékkel való ellátással (lásd fent) van lehetőség, ellentétben a privát setterrel rendelkező tulajdonságokkal, melyek settere bármely, az osztályban található tagfüggvényből hívható.

Csak olvasható tulajdonság definiálását a következő kódrészletek illusztrálják:

a) Autoimplementált eset

```
public string Name { get; }
```

b) Nem autoimplementált eset

```
private string name;
...
public string Name { get {return name; } }
```

Megjegyzés: korábban láttunk már példát arra, hogy egy tulajdonságnak csak get ága volt: az Age tulajdonság esetében egy **számított érték** előállítására használtuk ezt a megközelítést.

További property témakörök

További tulajdonságokhoz kapcsolódó témakörök, melyek nem szerepelnek a tárgy keretében, csak érdekességképpen:

1. Tulajdonság virtuális is lehet
2. Csak inicializálható tagok (init only members)

Delegate (delegát, metódusreferencia)

Olyan, mint a C-ben a függvénypointer, csak objektumorientált, illetve a C-vel szemben nemcsak egy, hanem több függvényre (metódusra) is lehet vele mutatni (hivatkozni). **A delegate-ek használatának**

egyik előnye az, hogy futási időben dönthetjük el, hogy több metódus közül éppen melyiket szeretnénk meghívni.

A delegate-ek (hasonlóan a C függvénypointerekhez) **típusosak**, egy delegate objektummal a típusának megfelelő szignatúrájú és visszatérési értékű metódusra lehet hivatkozni. Amikor delegate-ekkel dolgozunk, első lépésben egy delegate típust kell definiálni. Ez annak felel meg, mint amikor C-ben a typedef kulcsszóval egy függvénypointer típust definiáltunk. Ennek megfelelően egy **delegát típus** definiálásával egy olyan típust definiálunk, amelynek változóival rámutathatunk egy vagy több olyan metódusra, amely kompatibilis (paraméterlistája és visszatérési típusa) a delegát típusával. Pl.:

```
delegate bool FirstIsSmallerDelegate(object a, object b);
```

Itt a FirstIsSmallerDelegate egy delegate **típus**. Ebből pont úgy hozhatunk létre változókat (lokális, tagváltozók), vagy szerepeltethetjük függvényparaméterben, mintha egy közönséges osztály lenne, pl.:

```
FirstIsSmallerDelegate fis1;
```

Itt a fis1 delegate változó (objektum) értéke null. Értéket így adhatunk neki:

```
fis1 = new FirstIsSmallerDelegate(FirstIsSmaller_Complex);
```

A delegate típus „konstruktorának” a visszahívandó függvény nevét kell megadni. Itt feltettük, hogy a hívó kód osztályában létezik egy FirstIsSmaller_Complex nevű olyan függvény, amely kompatibilis a FirstIsSmallerDelegate delegate típussal (van két object paramétere és bool-lal tér vissza.)

Amikor értéket adunk egy delegate objektumnak, lehet az **egyszerűsített szintaktikát** is használni, amikor csak a függvény nevét adjuk meg:

```
fis1 = FirstIsSmaller_Complex;
```

A gyakorlatban csak ezt az egyszerűsített szintaktikát használjuk tömörsége miatt. Ez a szintaktika megfelel annak, amikor C nyelven egy függvénypointernek úgy adtunk értéket, hogy egyszerűen megadtuk a függvény nevét.

A delegát meghívásával a delegate objektum által hivatkozott metódus automatikusan meghívódik:

```
bool res = fis1(obj1, obj2); // meghívódik a FirstIsSmaller_Complex
```

A delegátok használatának egyik előnye az, hogy futási időben dönthetjük el, hogy több metódus közül éppen melyiket szeretnénk meghívni.

Példa

Az alábbi példában a **Sorter** osztály **HyperSort** függvénye egy általános sorrendező művelet. Tetszőleges típusú elemeket tud sorrendezni. Ehhez paraméterként megkapja az elemlistát, valamint az elemeket összehasonlítani képes metódusreferenciát.

```
class Complex
{
    public double Re, Im;

    //  Konstruktor
    public Complex(double re, double im)
    {
        this.Re = re;
        this.Im = im;
    }
 }

delegate bool FirstIsSmallerDelegate(object a, object b);
```

```
class Sorter
{
    // A lista rendezése egy paraméterként kapott delegate segítségével.
    // tetszőleges típusra használni szeretnénk. A Complex forráskódja
    // nincs meg, nem tudjuk megoldani, hogy implementálja az
    // IComparable-t. Megoldás: átadjuk az összehasonlító "függvényt" is.
    public static void HyperSort(ArrayList list,
        FirstIsSmallerDelegate firstIsSmaller)
    {
        for (...)
        {
           ...
                if ( firstIsSmaller(list[j], list[j - 1]) )
                  ...
            }
        }
    }

}

class Program
{
    static void Main(string[] args)
    {
        ArrayList list = new ArrayList();
        list.Add(new Complex(1, 2)); // Egy elem, biztosra megyünk :).
        // ...

        // Metódusreferencia statikus tagfüggvényre, csak megadjuk a hívandó függvény nevét.
        // Ez az ún. egyszerűsített szintaktika.
        Sorter.HyperSort(list, FirstIsSmaller_Complex);
        // Ez ekvivalens az előzővel, ez a teljes szintaktika, ma már ritkán használjuk,
        // mert terjengős: példányosítjuk a delegate típust, és konstrukruktor paraméteben
        // adjuk át a hívandó függvény nevét. Nem fogjuk ezt használni, de ismerni kell.
        Sorter.HyperSort(list, new FirstIsSmallerDelegate(FirstIsSmaller_Complex));

        // Metódusreferencia objektum (vagyis nem statikus) tagfüggvényre
        // (kicsit erőltetett a példa, ez is lehetne statikus itt)
        Comparers comps = new Comparers();
        Sorter.HyperSort(list, comps.FirstIsSmaller_Complex);

        // A delegate egy típus, lehet lokális változó, tagváltozó is.
        FirstIsSmallerDelegate fis1 = FirstIsSmaller_Complex;
        bool isFIS = fis1(new Complex(1, 1), new Complex(2, 2));

        // Minden delegate egy MultiCastDelegate leszármazott. Több metódusreferenciát
        // is tud tárolni. A += operátorral vehetők fel újak. Az alábbi példában kétszer
        // is meghívódik a FirstIsSmaller_Complex, nincs sok értelme. Majd az event-eknél
        // látjuk, miért jó ez.
        fis1 += FirstIsSmaller_Complex;
        fis1(new Complex(1, 1), new Complex(2, 2));
    }

    public static bool FirstIsSmaller_Complex(object a, object b)
    {
        Complex ca = (Complex)a;
        Complex cb = (Complex)b;
        return Math.Sqrt(...);
    }
}

class Comparers
{
```

```
    public bool FirstIsSmaller_Complex(object a, object b)
    {
        // mint a Program osztályban
        --||--
    }

    public bool FirstIsSmaller_Person(object a, object b)
    {
        ...
    }
}
```

Érdemes átgondolni, milyen alternatív megoldások lehetnek a sorrendező függvény esetén arra, hogy az elem összehasonlító logika ne legyen a függvénybe beégetve:

1. Delegate alapú megoldás, ez szerepelt a példánkban.
2. A `Sort` műveletnek nem adunk át delegate-et. Helyette az sorrendezendő objektumok típusának kell egy pl. az `IComparable` (Java-ban `Comparable`) interfészt implementálniuk, mely két elem összehasonlítását elvégzi (egy `CompareTo` művelettel).
3. A `Sort` műveletnek nem egy delegate-et, hanem pl. egy, az `IComparer` (Java-ban `Comparator`) interfészt megvalósító objektumot kell átadni, melynek `Compare` művelete két elem összehasonlítására használható.

Az olyan nyelvek esetében, melyek nem támogatják a függvénypointer/metódusreferencia koncepcióját, a 2-3 megközelítést szokás használni (ezek szerepeltek is Java előadás/gyakorlatok során). .NET környezetben mindhárom megoldás használható, az esettől függ, melyiket célszerűbb választani. Az interfész alapú megközelítés kicsit „egységbezártabb” megközelítést jelent, a delegate alapú kicsit rugalmasabb, egyszerűbb:

- Lehet statikus függvényre is metódusreferencia, nem kell hozzá objektum.
- .NET környezetben delegate-ek esetében tudunk lambda kifejezéseket használni. Rövidesen látni fogjuk.

A delegate-ekről leírás példákkal többek között itt található: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/using-delegates

Event (esemény)

Az alkalmazások többsége manapság már eseményvezérelt. Ez azt jelenti, hogy az alkalmazás bizonyos elemei eseményeket váltanak ki bizonyos sorrendben (például egy gomb megnyomása a felhasználói felületen), amelyekre az alkalmazás más részei reagálnak.

Az eseménykezelés .NET-ben, mint ahogy általában minden más programozói nyelvben (feltéve, hogy támogatja azt), **a Publisher/Subscriber tervezési mintára épül. Ennek lényege, hogy tetszőleges osztály publikálhatja eseményeknek egy csoportját, amelyekre tetszőleges más osztályok objektumai előfizethetnek. Amikor a publikáló osztály objektuma elsüti az eseményt, az összes feliratkozott objektum értesítést kap erről.** Megjegyzés: bár sokkal ritkább, lehetőség van osztály szinten is eseményeket definiálni (statikus esemény), illetve eseményekre osztály szinten feliratkozni.

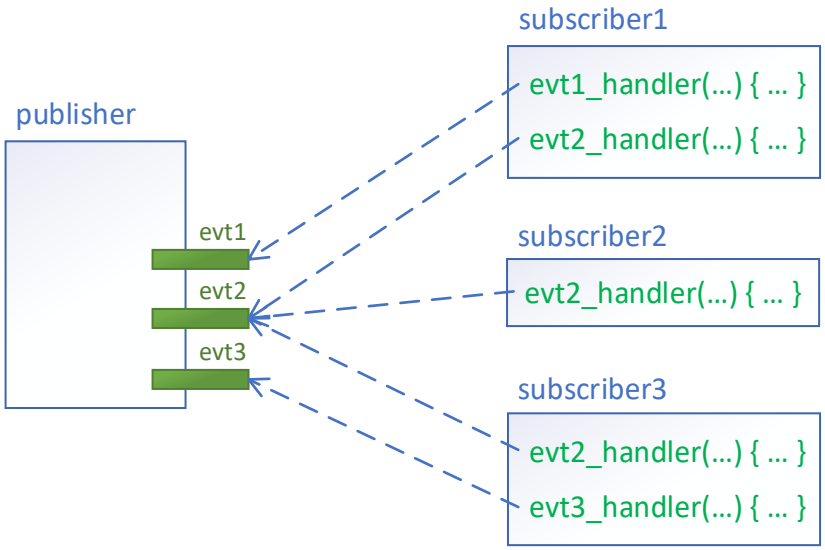
.NET-ben az eseménykezelés hátterében a delegátok állnak. Az eseményt publikáló osztály tulajdonképpen egy multicast delegátból egy speciális **tagváltozót** definiál az **event** kulcsszóval, amely a feliratkozott osztályok egy-egy metódusára mutat. Az event egy közönséges nyelvi elem lett,

osztályoknak most már nemcsak tagváltozói, tagfüggvényei és tulajdonságai lehetnek, hanem eseményei is.

Amikor az esemény elsül, a feliratkozott osztályok megfelelő metódusai a delegáton keresztül meghívódnak. A feliratkozott osztályok azon metódusait, amelyekkel az egyes eseményekre reagálnak, eseménykezelő metódusoknak nevezzük.

Minden eseménynek van egy típusa és van egy neve. A típusa annak a delegátnak a típusa, amelyik tulajdonképpen megfelel magának az eseménynek.

Az alábbi ábra egy publikáló és három előfizető objektumot ábrázol sematikusan:



- A publikáló objektumnak három eseménye van: evt1, evt2 és evt3.
- A subscriber1 előfizető a publikáló evt1 és evt2 eseményére, a subscriber2 előfizető a publikáló evt2 eseményére, a subscriber3 előfizető a publikáló evt2 és evt3 eseményére fizet elő. Ezt mutatják az ábrán a szaggatott kék nyilak.
- Minden esemény a gyakorlatban egy delegate tagváltozó (pontosabban property) a publikálóban, vagyis egy metódusreferencia lista. A példában az evt2 három metódusra hivatkozik az előfizetéseknek megfelelően: a subscriber1 objektum evt2_handler metódusára, a subscriber2 objektum evt2_handler metódusára és a subscriber3 objektum evt2_handler metódusára.
- Amikor a publikáló elsüti az evt2 eseményét, akkor végig iterál az evt2 delegate tag metódusain, és sorban egymás után meghívja ezeket: vagyis meghívódnak az előfizetők beregisztrált metódusai (eseménykezelő függvények), így az előfizetők értesülnek az eseményről. Ennek során a publikáló - amennyiben az eseményhez tartozik paraméter – azt átadja az eseménykezelő függvénynek paraméterben. Lehetőség van több paraméter használatára is. Példa: ha a publikáló egy személy objektum, akinek van neve, és az esemény azt jelzi, hogy a személy neve megváltozott, akkor az esemény elsütésekor a publikáló az előfizető eseménykezelő függvényének át tudja adni két paraméterben a név eredeti és új értékét is.

Példa

Megjegyzés: előadáson nem az alábbi Logger példa, hanem a GitHub-on elérhető előadásdemók (lásd https://github.com/bmeviauab00/eloadas-

demok/tree/main/02%20Nyelvi%20eszk%C3%B6z%C3%B6k%20(lang%20tools) ***közül a „03 Event-DirCopy” példa szerepel!***

Az alábbi példában a **Logger** egy általános célú naplózó osztály. Ez egy event-tet definiál **Log** néven. A Logger osztály esetében naplózni a WriteLine függvénnyel lehet, mely nem csinál mást, mint elsüti a Log eseményt (a null vizsgálattal előbb megvizsgálja, van-e legalább egy előfizető). Az **App** osztály az alkalmazást reprezentálja. A konstruktorában két előfizető műveletet is beregisztrál a **+= operátorral**: a saját **writeConsole** tagfüggvényét (mely a konzolra naplóz), valamint egy **FileLogListener** objektum **WriteToFile** tagfüggvényét (fájlba naplóz). Eseményről leiratkozni a **-= operátorral** lehet (lásd App.Cleanup művelet).

```
public delegate void LogHandler(string msg);

class Logger
{
    // Osztálynak .NET-nem nemcsak tagváltozója és tagfüggvénye
    // lehet, hanem event-je is!
    public event LogHandler Log;
    // Ide jöhetne a többi, de most nincs több

    public void WriteLine(string msg)
    {
        // Esemény elsütése (null vizsgálat: meg kell nézni, van-e előfizető)
        if (Log != null)
            Log(msg);

        // Alternatíva esemény elsütésére:
        // A null vizsgálat és esemény elsütés egyben egyszerűbben is megtehető C#6-tól,
        // a null-conditional operátorral, vagyis a „?.”-tal:
        // csak akkor süti el az eseményt, ha nem null, egyébként semmit nem csinál.
        Log?.Invoke(mgs);
    }
}

class App
{
    Logger log = new Logger();
    FileLogListener fileLogListener = new FileLogListener();

    public App()
    {
        // Feliratkozás a Log eseményre (a writeConsole és fileLogListener.WriteToFile
        // műveleteklet regisztráljuk be)
        log.Log += new LogHandler(writeConsole);
        log.Log += new LogHandler(fileLogListener.WriteToFile);

        // Egyszerűsített formával is feliratkozhatunk, csak a kezelőfüggvény nevét
        // adjuk meg. A fenti két sorral ekvivalens:
        log.Log += writeConsole;
        log.Log += fileLogListener.WriteToFile;

    }

    public void Process()
    {
        log.WriteLine("Process begin...");
        //...
        log.WriteLine("Process end...");
    }
```

```
    public void Cleanup()
    {
        // Leiratkozás eseményről
        log.Log -= new LogHandler(writeConsole);
        log.Log -= new LogHandler(fileLogListener.WriteToFile);
    }

    void writeConsole(string msg)
    {
        Console.WriteLine(msg);
    }
}

class FileLogListener
{
    // FileStream tag.
    public void WriteToFile(string msg)
    {
        // ...
    }
}

class Program
{
    static void Main(string[] args)
    {
        App app = new App();
        app.Process();
        app.Cleanup();
    }
}
```

Miben más az event, mint a delegate?

- Egy delegate objektumból akkor lesz event, ha elé írjuk az **event** kulcsszót.
- Az event osztályok tagváltozója lehet csak (így pl. lokális event objektum nincs, lokális delegate objektum létezik viszont.)
- Plusz védelmet biztosít a nem korrekt használattal szemben!
    - Nem lehet az = operátort használni, csak a += és -= -t, így egy külső objektum nem tudja kitörölni a tőle független feliratkozottakat a listáról.
    - Csak a tartalmazó osztály sütheti el.

Az események minden .NET nyelvben elérhetők, csak más szintaktikával.

Az event-ekről leírás példákkal többek között itt található: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/events

Attribute (attribútum)

Az attribútumok segítségével **deklaratív jelleggel metaadatokat közölhetünk a kód bizonyos részeire** vonatkozóan. Az attribútum is tulajdonképpen egy osztály, amit hozzákötünk a program egy megadott eleméhez (típushoz, osztályhoz, interfészhez, metódushoz, …). Ezeket a metainformációkat a program futása közben mi magunk is kiolvashatjuk az úgynevezett reflection mechanizmus segítségével, de általában az attribútumokkal a .NET „beépített” osztályai vagy pl. a .NET fordító számára szeretnénk információkat közölni. A .NET attribútumoknak a Java nyelvben az annotációk felelnek meg.

Az attribútumok funkciója a legkülönbözőbb féle lehet. A Serializable attribútum segítségével például egy osztályról jelezhetjük, hogy az binárisan sorosítható, azaz tetszőleges adatfolyamba (akár egy file-ba, akár egy hálózati adatfolyamba) az állapota bináris formában elmenthető:

```
[Serializable] // Jelezzük, hogy az osztály sorosítható
class User
{
    string name;

    [NonSerialized] // Jelezzük,hogy ezt a mezőt nem kell sorosítani
    string password;

    // Ezzel a DeleteUser függvényt elavultnak jelüljük meg, ha valaki hívja, akkor
    // fordításkor warningot kap.
    [Obsolete("This method is obsolete. Call DeleteUser2 instead.", false)]
    public static void DeleteUser(int userId) { ... }

    public static void DeleteUser2(int userId) { ... }

    // ...
}

class Program
{
    static void Main(string[] args)
    {
        User user = new User();
        // felparaméterezzük ...

        //  Szerializalas egy file stream-be
        BinaryFormatter formatter = new BinaryFormatter();
        FileStream stream1 = new FileStream("Dump.dat", FileMode.Create);
        formatter.Serialize(stream1, user);
        stream1.Close();

        //  Deszerializalas a file stream-bol
        FileStream stream2 = new FileStream("Dump.dat", FileMode.Open);
        User u = (User)formatter.Deserialize(stream2);
        stream2.Close();

    }
}
```

Lényeges tudni, hogy lehetőségünk van többek között az alábbiakra (kódot nem nézünk rá):

- Olyan kódot írni, mellyel lekérdezhetjük, hogy egy osztálynak milyen attribútumai vannak.
- Olyan kódot írni, mellyel lekérdezhetjük, hogy egy osztálynak milyen tagváltozói/tagfüggvényei vannak, ezek milyen attribútumokkal rendelkeznek (a fenti példában a BinaryFormatter is ezt csinálja!).
- Saját attribútumot is tudunk bevezetni, ehhez mindössze a beépített System.Attribute osztályból kell leszármazni. Ez pont úgy használható, mint a .NET-be beépített attribútumok.

Érték és referencia típusok (!)

.NET környezetben minden típus vagy az érték vagy a referencia típusok csoportjába tartozik, és ennek megfelelően viselkedik.

A lényegi jellemzők a következők:

- **Érték típus (value type):** A változó magát az értéket tartalmazza (nem egy mutató/hivatkozás rá). int, char, decimal, double, float, bool, stb. egyszerű típusok tartoznak ide, valamint amit mi

definiálunk a **`struct`** vagy **`enum`** kulcsszóval. Inline módon foglalnak helyet összetett típusokban. Lokális változónál a vermen foglalódik nekik hely. Függvényparaméter átadáskor pont úgy kezelődnek, mint C++-ban az int, vagy minden más: másolat készül az eredeti adatról. Gyors az allokációjuk. Korlátozások: nem örökölhetnek, nem lehet belőlük származni. Interfészt viszont implementálhatnak (lásd később). **Kisméretű objektumok esetén használjuk**: ekkor a másolatkészítés függvényhíváskor nem lassít még, viszont a GC műveleteket megspóroljuk. Pl. Point, Rect, Coordinate, Complex, DateTime és hasonlók, ezek struct-ok legyenek, ne class-ok.

- **A referencia típus (reference type):** két részből áll: egy mutató/hivatkozás, mely alapértelmezésben null, és maga a mutatott/hivatkozott objektum, ami a felügyelt heapen foglal helyet, és a garbage collector gyűjti be, ha már nincs rá hivatkozás. Ez utóbbinak nekünk kell a **`new`**-val helyet foglalni. A .NET beépített osztályai (pl. string, File, stb.) ilyenek, a tömbök, meg amit mi hozunk létre a **`class`** kulcsszóval. Az interfészek is ide tartoznak, később lesz róluk szó. Nagyobb méretű objektumok esetén használjuk.

Vagyis, ha egy saját típust hozunk létre a **class** kulcsszóval, akkor az alapvetően úgy viselkedik, mint Java-ban az osztályok. A beépített egyszerű típusok (pl. int, char, stb.), valamint az általunk létrehozott **struct** típusok viszont úgy viselkednek, mint a C/C++ egyszerű típusai (illetve az összetett típusok is, amikor nem használunk referenciát/pointert).

Ha Person egy struct, akkor a
```
Person p1;
```
esetében a p1 maga az objektum, létre is jön (nem kell a new-val létrehozni). Vagy egy tömb példa:
```
Person[] persons = new Person[10];
```
, ahogy létre is jön mind a 10 Person objektum magában a tömbben egymás mellett.

Ha a Person class, akkor a
```
Person p1;
```
csak egy referencia, objektum nem jön létre, azt a new-val külön létre kell hozni, és a referenciát rá kell állítani:
```
Person p1;
p1 = new Person();

vagy egy sorban:
Person p1 = new Person();
```
Vagy egy tömb példa:
```
Person[] persons = new Person[10];
```
a tömbben csak referenciák vannak, értékük null. A Person objektumokat egyesével létre kell hozni és a tömb elemeit ráállítani, egyesével vagy ciklusban:
```
persons[0] = new Person();
```
Értelemszerűen a **null** érték csak referencia típusok esetében használható.