

Szoftvertchnikák laborgyakorlat

2. mérés

Modern nyelvi eszközök

Modern nyelvi eszközök gyakorlása:
tulajdonság, delegate, esemény, attribútum,
generikus osztályok

Tartalom

A GYAKORLAT CÉLJA	3
BEVEZETŐ	3
FELADAT 1 – TULAJDONSÁG (PROPERTY).....	3
FELADAT 2 – DELEGÁT (DELEGATE, FÜGGVÉNYREFERENCIA, METÓDUSREFERENCIA)	7
FELADAT 3 – ESEMÉNY (EVENT)	9
FELADAT 4 – ATTRIBÚTUMOK	10
FELADAT 5 – GENERIKUS OSZTÁLYOK.....	12
FELADAT 6 – DELEGÁTOROK 2.	13

A gyakorlat célja

A kapcsolódó előadások:

- 2. előadás – Modern programozási eszközök

A gyakorlat célja az alábbi C# nyelvi elemek megismerése:

- Tulajdonságok
- Delegátok
- Event kulcsszó
- Attribútumok
- Generikus típusok

A hallgatók már készítettek egyszerűbb C# „Hello World” alkalmazást, tisztában vannak a Visual Studio használatának alapjaival, képesek egy osztályon belül tagokat létrehozni, módosítani.

Bevezető

A labor során a hallgatók megismerkednek a legfontosabb modern, a .NET környezetben is rendelkezésre álló nyelvi eszközökkel. Feltételezzük, hogy a hallgató a korábbi tanulmányai során elsajátította az objektum-orientált szemléletmódot, és tisztában van az objektum-orientált alapfogalmakkal. Jelen labor során azokra a .NET-es nyelvi elemekre koncentrálunk, amelyek túlmutatnak az általános objektum-orientált szemléleten, ugyanakkor nagyban hozzájárulnak a jól átlátható és könnyen karbantartható kód elkészítéséhez. Ezek a következők:

- Tulajdonságok (properties)
- Delegátok (delegates)
- Események (events)
- Attribútumok (attributes)
- Generikus osztályok (generics)

Feladat 1 – Tulajdonság (property)

A tulajdonságok segítségével tipikusan osztályok tagváltozóihoz férhetünk hozzá szintaktikailag hasonló módon, mintha egy hagyományos tagváltozót érnénk el. A hozzáférés során azonban lehetőségünk van, hogy az egyszerű érték lekérdezés vagy beállítás helyett metódusszerűen implementáljuk a változó elérésének a módját.

A következő példában egy `Person` nevű osztályt fogunk elkészíteni, mely egy embert reprezentál. Két tagváltozója van, `name` és `age`. A tagváltozókhoz közvetlenül nem férhetünk hozzá (mivel privátok), csak a `Name` illetve `Age` publikus tulajdonságokon keresztül kezelhetjük őket. A példa jól szemlélteti, hogy a .NET-es tulajdonságok egyértelműen megfelelnek a C++-ból és Java-ból már jól ismert `SetX(...)` illetve `GetX()` típusú metódusoknak, csak itt ez a megoldás nyelvi szinten támogatott.

1. Hozzunk létre egy új C# konzolos alkalmazást
2. Adjunk hozzá egy új osztályt az alkalmazásunkhoz `Person` néven

Új osztály hozzáadásához a Solution Explorerben kattintsunk jobb egérgombbal a projekt fájlra és válasszuk az „Add / class” menüpontot. Az előugró ablakban a létrehozandó fájl nevét módosítsuk `Person.cs`-re, majd nyomjuk meg az Add gombot.

3. Tegyük az osztályt publikussá

Ehhez az osztály neve elé be kell írni a `public` kulcsszót. Erre a módosításra itt igazából még nem volna szükség, ugyanakkor a 4. feladat már egy publikus osztályt fog igényelni.

4. Hozzunk létre az osztályon belül egy `int` típusú `age` mezőt és egy ezt elérhetővé tevő `Age` tulajdonságot.

```
public class Person
{
    private int age;

    public int Age
    {
        get { return age; }
        set { age = value; }
    }
}
```

5. Egészítsük ki a `Program.cs` fájl `Main` függvényét, hogy kipróbálhassuk az új osztályunkat.

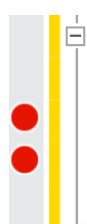
```
static void Main(string[] args)
{
    Person p = new Person();
    p.Age = 17;
    p.Age++;
    Console.WriteLine(p.Age);
}
```

6. Futtassuk a programunkat (F5)

Láthatjuk, hogy a tulajdonság a tagváltozókhoz hasonlóan használható. Az érték lekérdezése esetén a tulajdonságban definiált `get` rész fog lefutni, és a tulajdonság értéke a `return`-nel visszaadott érték lesz. Az érték beállítása esetén a tulajdonságban definiált `set` rész fog lefutni, és a speciális `value` változó értéke ebben a szakaszban megfelel a tulajdonságnak értékül adott kifejezéssel.

Figyeljük meg a fenti megoldásban azt, hogy milyen elegánsan tudjuk egy évvel megemelni az ember életkorát. Java, vagy C++ kódban egy hasonló műveletet a `p.setAge(p.getAge()+1)` formában írhattunk volna le, amely jelentősen körülményesebb és nehezen olvashatóbb szintaktika a fentinél. A tulajdonságok használatának legfőbb hozadéka, hogy kódunk szintaktikailag tisztább lesz, az értékadások illetve lekérdezések pedig az esetek többségében jól elválnak a tényleges függvényhívásoktól.

7. Győződjünk meg róla, hogy a programunk valóban elvégzi a getter és setter függvények hívását. Ehhez helyezzünk töréspontokat (breakpoint) a getter és setter függvények belsejébe a kódszerkesztő bal szélén látható szürke sávra kattintva.



```
public int Age
{
    get { return age; }
    set { age = value; }
}
```

8. Futtassuk a programot lépésről lépésre. Ehhez a programot F5 helyett az F11 billentyűvel indítsuk, majd az F11 további megnyomásaival engedjük sorról sorra a végrehajtást.

Láthatjuk, hogy a programunk valóban minden egyes alkalommal meghívja a gettert, amikor értéklekérdezés, illetve a settert, amikor értékbeállítás történik.

9. A setter függvények egyik fontos funkciója, hogy lehetőséget kínálnak az értékvalidációra. Egészítsük ki ennek szellemében az `Age` tulajdonság setter-ét.

```
public int Age
{
    get { return age; }
    set
    {
        if (value < 0)
            throw new Exception("Érvénytelen életkor!");
        age = value;
    }
}
```

Figyeljük meg, hogy míg az egyszerű getter és setter esetében az értéklekérdezést/beállítást egy sorban tartjuk, addig komplexebb törzs esetén már több sorra tördeljük.

10. Az alkalmazás teszteléséhez rendeljünk hozzá negatív értéket az életkorhoz a `Program` osztály `Main` függvényében.

```
p.Age = -2;
```

11. Futtassuk a programot, győződjünk meg arról, hogy az ellenőrzés helyesen működik, majd a további munka kedvéért hárítsuk el a hibát azzal, hogy pozitívrá cseréljük a beállított életkort.

```
p.Age = 2;
```

A mindennapi munkánk során az interneten találkozhatunk a tulajdonságoknak egy sokkal tömörebb szintaktikájával is. Ez a szintaktika akkor alkalmazható, ha egy olyan tulajdonságot szeretnénk létrehozni, melyben:

- nincs szükségünk a privát tagváltozó közvetlen elérésére
- nem szeretnénk semmilyen kiegészítő logikával ellátni a getter és setter metódusokat

12. Egészítsük ki a `Person` osztályunkat egy ilyen, ún. „autoimplementált” tulajdonsággal (auto-implemented property). Szúrjuk be a következő sort közvetlenül az `Age` tulajdonság záró kapcsos zárójelét követően:

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

Autoimplementált tulajdonság esetén a fordító egy rejtett, kódból nem elérhető változót generál az osztályba, ami a tulajdonság aktuális értékének tárolására szolgál.

13. Majd ellenőrizzük a működését a `Main` függvény kiegészítésével.

```
static void Main(string[] args)
{
    ...
    p.Name = "Luke";
    ...
    Console.WriteLine(p.Name);
}
```

A további feladatok építeni fognak az előző feladatok végeredményeire. Ha programod nem fordul le, vagy nem megfelelően működik, jelezd ezt a gyakorlatvezetődnél a feladatok végén és segít elhárítani a hibát.

Feladat 2 – Delegát (delegate, függvényreferencia, metódusreferencia)

A delegátok típusos függvényreferenciákat jelentenek .NET-ben, a C/C++ függvénypointerek modern megfelelői. Egy delegát definiálásával egy olyan változót definiálunk, amellyel rámutathatunk egy olyan metódusra, amely típusa (paraméterlistája és visszatérési értéke) megfelel a delegát típusának. A delegát meghívásával az értékül adott (beregisztrált) metódus automatikusan meghívódik. A delegátok használatának egyik előnye az, hogy futási időben dönthetjük el, hogy több metódus közül éppen melyiket szeretnénk meghívni.

Néhány példa delegátok használatára: egy univerzális sorrendező függvénynek paraméterként az elemek összehasonlítását végző függvény átadása; egy általános gyűjteményen univerzális szűrési logika megvalósítása, melynek paraméterben egy delegát formájában adjuk át azt a függvényt, amely eldönti, hogy egy elemet bele kell-e venni a szűrt listába. A delegátok egyik leggyakoribb alkalmazása pedig a publish-subscribe minta megvalósítása, amikor bizonyos objektumok más objektumokat értesítenek adott események bekövetkezéséről.

A következő példánkban lehetővé tesszük, hogy a korábban létrehozott `Person` osztály példányai szabadon értesíthessenek külső osztályokat is arról, ha egy ember életkora megváltozott. Ennek érdekében bevezetünk egy delegát típust (`AgeChangingDelegate`), ami paraméterlistájában át tudja adni az emberünk életkorának aktuális, illetve új értékét. Ezt követően létrehozunk egy publikus `AgeChangingDelegate` típusú mezőt a `Person` osztályban, ami lehetővé teszi, hogy egy külső fél megadhassa a függvényt, amelyen keresztül az adott `Person` példány változásairól értesítést kér

1. Hozzunk létre egy új **delegát típust**, mely `void` visszatérési értékű, és két darab `int` paramétert elváró függvényekre tud hivatkozni. Figyeljünk rá, hogy az új típust a `Program` osztály előtt, közvetlenül a névtér scope-jában definiáljuk!

```
namespace PropertyDemo
{
    public delegate void AgeChangingDelegate(int oldAge, int newAge);

    class Program
    { ...
```

Az `AgeChangingDelegate` egy típus, bárhol szerepelhet, ahol típus állhat (pl. lehet létrehozni ez alapján tagváltozót, lokális változót, függvény paramétert, stb.).

2. Tegyük lehetővé, hogy a `Person` példányai rámutathassanak tetszőleges, a fenti szignatúrának megfelelő függvényre. Ehhez hozzunk létre egy `AgeChangingDelegate` típusú mezőt a `Person` osztályban! (Tipp: az új mezőt az osztály elejére szúrjuk be!)

```
class Person
{
    public AgeChangingDelegate AgeChanging;
    ...
}
```

Megjegyzés: a publikus mezőként létrehozott függvénytípus igazából (egyelőre) sérti az OO elveket. Erre később visszatérünk még.

3. Hívjuk meg a függvényt minden alkalommal, amikor az emberünk kora megváltozik. Ehhez egészítsük ki az `Age` tulajdonság setterét a következőkkel.

```
public int Age
{
    get { return age; }
    set
    {
        if (value < 0)
            throw new Exception("Érvénytelen életkor!");
        if (AgeChanging != null)
            AgeChanging(age, value);
        age = value;
    }
}
```

A fenti kódrészlet számos fontos szabályt demonstrál:

- A validációs logika általában megelőzi az értesítési logikát
- Az értesítési logika jellegétől függ, hogy az értékadás előtt, vagy után futtatjuk le (ebben az esetben, mivel a „changing” szó egy folyamatban lévő dologra utal, az értesítés megelőzi az értékadást)
- Fel kell készülnünk rá, hogy a delegate típusú mezőnkhez még senki nem rendelt értéket. Ilyen esetekben a meghívásuk kivételt okozna, ezért meghívás előtt **mindig** ellenőrizni kell, hogy a mező értéke null-e.

4. Kész vagyunk a `Person` osztály kódjával. Térjünk át a fogyasztóra! Ehhez mindenképp először a `Program` osztályt kell kiegészítenünk egy újabb függvénnyel.

```
class Program
{
    ...
    static void PersonAgeChanging(int oldAge, int newAge)
    {
        Console.WriteLine(oldAge + " => " + newAge);
    }
    ...
}
```

Tipp: Fokozottan ügyeljünk rá, hogy az új függvény a megfelelő scope-ba kerüljön!

5. Végezetül iratkozzunk fel a változáskövetésre a Main függvényben!

```
static void Main(string[] args)
{
    Person p = new Person();
    p.AgeChanging = new AgeChangingDelegate(PersonAgeChanging);
    ...
}
```

6. Futtassuk a programot!

Figyeljük meg, hogy az esemény minden egyes setter futáskor, így az első értékadáskor és az inkrementáláskor egyaránt lefut. A megoldásunk azonban még fejleszthető.

7. Egészítsük ki a Main függvényt többszöri feliratkozással, majd futtassuk a programot.

```
p.AgeChanging = new AgeChangingDelegate(PersonAgeChanging);
p.AgeChanging += PersonAgeChanging;
p.AgeChanging += PersonAgeChanging;
```

Láthatóan minden egyes értékváltozáskor mind a három „feliratkozott” függvény lefut. Ez azért lehetséges, mert a delegate típusú mezők valójában nem csupán egy függvény-referenciát, hanem egy függvény-referencia listát tartalmaznak (és tartanak karban).

Figyeljük meg, hogy a függvényreferenciákat az először látottnál tömörebb szintaxissal is leírhatjuk. Ettől függetlenül a második két esetben is egy `AgeChangingDelegate` fogja burkolni a `PersonAgeChanging` függvényeket.

8. Próbáljuk ki a leiratkozást is (szabadon választott ponton), majd futtassuk a programot.

```
p.AgeChanging -= PersonAgeChanging;
```

Feladat 3 – Esemény (event)

Ahogy a tulajdonságok a getter és setter metódusoknak, addig a fent látott delegate mechanizmus a Java-ból ismert Event Listener-eknek kínálnak egy szintaktikailag letisztultabb alternatíváját. A fenti megoldásunk azonban egyelőre még súlyosan sért pár OO elvet (egységbezárás, információrejtés). Ezt az alábbi két példával tudjuk demonstrálni.

1. Az eseményt valójában kívülről is ki tudjuk váltani. Ez szerencsétlen, hisz így az eseményre feliratkozott függvényekkel az osztály nevében hamis adatokat közölhetünk. Ennek demonstrálására szúrjuk be a következő sort a `Main` függvény végére.

```
p.AgeChanging(67, 12);
```

2. Bár a += és a -= tekintettel vannak a listába feliratkozott többi függvényre, valójában az = operátorral bármikor felülírhatjuk mások feliratkozásait. Próbáljuk ki ezt is, a következő sor beszúrásával közvetlenül a fel és leiratkozások utánra.




```
p.AgeChanging = null;
```

3. Az event kulcsszó feladata valójában az, hogy a fenti két jelenséget megtiltva visszakényszerítse programunkat az objektumorientált mederbe. Lássuk el az event kulcsszóval az AgeChanging mezőt Person.cs-ben!

```
class Person
{
    public event AgeChangingDelegate AgeChanging;
    ...
}
```

4. Próbáljuk meg lefordítani a programot. Látni fogjuk, hogy a fordító a korábbi kihágásainkat most már fordítási hibaként kezeli.

```
static void Main(string[] args)
{
    Person p = new Person();
    p.AgeChanging = new AgeChangingDelegate(PersonAgeChanging);
    p.AgeChanging += PersonAgeChanging;
    p.AgeChanging += PersonAgeChanging;
    p.AgeChanging -= PersonAgeChanging;
    p.AgeChanging = null;
    p.Age = 24;
    p.Name = "Luke";
    p.Age++;
    Console.WriteLine(p.Age);
    Console.WriteLine(p.Name);
    p.AgeChanging(67, 12);
}
```

 1 The event 'PropertyDemo.Person.AgeChanging' can only appear on the left hand side of += or -= (except when used from within the type 'PropertyDemo.Person')
 2 The event 'PropertyDemo.Person.AgeChanging' can only appear on the left hand side of += or -= (except when used from within the type 'PropertyDemo.Person')
 3 The event 'PropertyDemo.Person.AgeChanging' can only appear on the left hand side of += or -= (except when used from within the type 'PropertyDemo.Person')

5. Távolítsuk el a három hibás kódsort (figyeljük meg, hogy már az első közvetlen értékadás is hibának minősül), majd fordítsuk le és futtassuk az alkalmazásunkat!

Feladat 4 – Attribútumok

Az attribútumok segítségével deklaratív módon metaadatokkal láthatjuk el forráskódunkat. Az attribútum is tulajdonképpen egy osztály, amit hozzákötünk a program egy megadott eleméhez (típushoz, osztályhoz, interfészhez, metódushoz, ...). Ezeket a metainformációkat a program futása közben bárki (akár mi magunk is) kiolvashatja az úgynevezett reflection mechanizmus segítségével (ezzel részletesen egy későbbi labor foglalkozik).

Az attribútumok funkciója a legkülönbözőbb féle lehet. A következő példában használt attribútumok például az XML sorosítóval közölnek különböző metainformációkat.

6. Szúrjuk be a `Main` függvény végére a következő kódrészletet, majd futtassuk a programunkat!

```
XmlSerializer serializer = new XmlSerializer(typeof(Person));  
FileStream stream = new FileStream("person.txt", FileMode.Create);  
serializer.Serialize(stream, p);  
stream.Close();  
Process.Start("person.txt");
```

A fenti példából az utolsó sor nem a sorosít logika része, csupán egy frappáns megoldás arra, hogy a Windows alapértelmezett szövegfájl nézegetőjével megnyissuk a keletkezett adatállományt.

7. Nézzük meg a keletkezett fájl szerkezetét. Figyeljük meg, hogy minden tulajdonság a nevének megfelelő node-ba lett leképezve.

Attribútumok segítségével olyan metaadatokkal láthatjuk el a `Person` osztályunkat, melyek közvetlenül módosítják a sorosító viselkedését.

8. Az `XmlRoot` attribútum lehetőséget kínál a gyökérelem átnevezésére. Helyezzük el a `Person` osztály fölé!

```
[XmlRoot("Személy")]  
public class Person
```

9. Az `XmlAttribute` attribútum jelzi a sorosító számára, hogy a jelölt tulajdonságot ne xml node-ra, hanem xml attribútumra képezze le. Lássuk el ezzel az `Age` tulajdonságot!

```
[XmlAttribute("Kor")]  
public int Age
```

10. Az `XmlIgnore` attribútum jelzi a sorosítónak, hogy a jelölt tulajdonság teljesen elhagyható az eredményből. Próbáljuk ki a `Name` tulajdonság fölött.

```
[XmlIgnore]  
public string Name { get; set; }
```

11. Futtassuk az alkalmazásunkat! Hasonlítsuk össze az eredményt a korábbiakkal.

Feladat 5 – Generikus osztályok

A .NET generikus osztályai megfelelnek a C++ nyelv template osztályainak. A segítségükkel általános (több típusra is működő), de ugyanakkor típusbiztos osztályokat hozhatunk létre. Generikus osztályok nélkül, ha általánosan szeretnénk kezelni egy problémát, akkor `object` típusú adatokat használunk (mert .NET-ben minden osztály az `object` osztályból származik). Ez a helyzet például az `ArrayList`-tel is, ami egy általános célú gyűjtemény, tetszőleges, `object` típusú elemek tárolására alkalmas. Lássunk egy példát az `ArrayList` használatára:

```
ArrayList list = new ArrayList();
list.Add(1);
list.Add(2);
list.Add(3);
for (int n = 0; n < list.Count; n++)
{
    // Castolni kell
    int i = (int)list[n];
    Console.WriteLine("Value: {0}", i);
}
```

A fenti megoldással a következő problémák adódnak:

- Az `ArrayList` minden egyes elemet `object`-ként tárol
- Amikor hozzá szeretnénk férni a lista egy eleméhez, mindig a megfelelő típusúvá kell cast-olni
- Nem típusbiztos. A fenti példában semmi nem akadályoz meg abban (és semmilyen hibaüzenet sem jelzi), hogy az `int` típusú adatok mellé beszúrjuk a lista egy másik típusú objektumot. Csak a lista bejárása során kapnánk hibát, amikor a nem `int` típust `int` típusúra szeretnénk castolni. Generikus gyűjtemények használatakor az ilyen hibák már a fordítás során kiderülnek.
- Érték típusú adatok tárolásakor a lista lassabban működik, mert az érték típust először be kell dobozolni (boxing), hogy az `object`-ként (azaz referencia típusként) tárolható legyen.

A fenti probléma megoldása egy generikus lista használatával a következőképpen néz ki (a gyakorlat során csak a sárgával jelölt sorokat módosítsuk a korábban begépett példában):

```
List<int> list = new List<int>();
list.Add(1);
list.Add(2);
list.Add(3);
for (int n = 0; n < list.Count; n++)
{
    int i = list[n];
    Console.WriteLine("Value: {0}", i);
}
```

Feladat 6 – Delegátok 2.

Megjegyzés: a vezetett gyakorlat során erre a feladatra lehet, hogy már nem marad idő. Ez esetben a feladatot otthon, önállóan végezhetjük el, hogy referenciaként szolgálhasson a házi feladat 4. feladatához.

A 2. és 3. feladatokban a delegátokkal esemény alapú üzenetküldést valósítottunk meg. A delegátok használatának másik tipikus esetében a függvényreferenciákat arra használjuk, hogy egy algoritmus számára egy előre nem definiált lépés implementációját átadjuk.

A generikus lista osztály FindAll függvénye például képes arra, hogy visszaadjon egy új listában minden olyan elemet, ami egy adott feltételnek eleget tesz. A konkrét feltételt egy függvény formájában adhatjuk meg, mely igazat ad minden olyan elemre, amit az eredménylistában szeretnénk látni. A függvény típusa a következő előre definiált delegate típus:

```
public delegate bool Predicate<T>(T obj)
```

Vagyis bemenetként egy olyan típusú változót vár, mint a listaelemek típusa, kimenetként pedig egy logikai értéket. A fentiek demonstrálására kiegészítjük a korábbi programunkat egy szűréssel, mely a listából csak a páratlan elemeket fogja megtartani.

1. Valósítsunk meg egy olyan szűrőfüggvényt az alkalmazásunkban, amely a páratlan számokat adja vissza:

```
static bool MyFilter(int i)
{
    if (i % 2 == 1)
        return true;
    else
        return false;
}
```

2. Majd egészítsük ki a korábban írt kódunkat a szűrő függvényünk használatával:

```
List<int> list = new List<int>();
list.Add(1);
list.Add(2);
list.Add(3);

list = list.FindAll(MyFilter);

for (int n = 0; n < list.Count; n++)
{
    int i = list[n];
    Console.WriteLine("Value: {0}", i);
}
```

3. Futtassuk az alkalmazásunkat. Figyeljük meg, hogy a konzolon valóban csak a páratlan számok jelennek meg.
4. Érdekességgként elhelyezhetünk egy töréspontot (breakpoint) a MyFilter függvényünk belsejében és megfigyelhetjük, hogy a függvény valóban minden egyes listaelemre külön-külön meghívódik.