Eseményvezérelt alkalmazások fejlesztése

3. gyakorlat - A C# programozási nyelv alapjai

[Szabó Tamás sztrabi@inf.elte.hu]

Mai óra tematikája - Miről is lesz szó?

- Ezen az órán elkezdjük körbejárni a nyelvi elemeket.
- Felsorolható típusok - enumerációk (**enumeration**)
- Objektumorientált programozást támogató újabb nyelvi elemek: absztrakt osztályok (**abstract class**), lezárt osztályok (**sealed class**), parciális osztályok (**partial class**), illetve statikus osztályok (**static class**).
- Szót ejtünk még a generikus (**generics**) típusokról, metódusokról és arról, hogy miért érdemes őket használni
- Tömbök használata (a gyűjtemények egy másik órára)
- Delegáltak (**delegates**), események (**events**) és λ-függvények
- Ha valaki nagyon unatkozik, az megpróbálhatja elérni a 6900 pontot: http://helloworldquiz.com/

Tömbök

- Munkánk során (az esetek nagy részében adatokat) fogunk manipulálni, transzformálni valamilyen módon. Ezeket az adatokat, tárolnunk kell a memóriában úgy, hogy azokat egyszerűen el tudjuk érni és használni.

```
  //amikor példányosítjuk, akkor már inicializálhatjuk is, hogy milyen elemek kerüljenek bele
  int[] a = new int[] { 1, 2, 3, 4 };
  int[] b = new int[10]; //egy 10-elemű tömböt szeretnénkű
  //ha a fordító ki tudja következtetni, hogy milyen típusú
  //a tömbünk, akkor még a típust sem kell megadnunk a példányosításnál
  int[] c = new[] { 1, 2, 3, 4 };
  Console.WriteLine(a[0]);
  foreach (int number in a) {
    Console.WriteLine("number: {0}", number);
  }
  int[][] identicalMatrix = new int[][]  //ezt úgy nevezzük, hogy jagged array
  {
      new[] {1, 0, 0},
      new[] {0, 1, 0},
      new[] {0, 0, 1},
  };

  Console.WriteLine(identicalMatrix[0][1]); //Output: 0

  int[,] matrix = new int[3,3]; //multi-dimensional matrix
  Console.WriteLine(matrix[0, 1]); //Output: 0
```

Felsorolható típusok - Enumerációk

- A háttérben ugyanúgy egy osztályt készít nekünk a fordító, amit közvetlenül a System.Enum osztályból származik. Mivel a **System.Enum** pedig értéktípus, ezért minden általunk definiált felsorolható típus is értéktípusnak fog számítani.

```
enum EmployeeType { Intern, Graduated, Manager, VicePresident }

//Használata nagyon egyszerű:
EmployeeType employee = EmployeeType.Manager;
//NEM: employee = Manager; - fordítási hibát kapunk!
```

- Megadhatunk alapértelmezetten konstans értékeket **csak** egész számokat) a felsorolható típus elemeinek, amik a későbbiekben reprezentálni fogják, majd azokat. Illetve beállíthatjuk, hogy milyen típusú értékként (*tipikusan a memóriafoglalás szempontjából tud hasznos lenni*) (csak a következők jöhetnek szóba: **byte**, **sbyte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**) reprezentálja őket.

```
enum EmployeeType : byte /* byte típusú elemeink lesznek */ {
  Intern = 1,   Graduated = 2,
  Manager = 4,  VicePresident = 8
}
```

Absztrakt osztályok (1/2)

- Vannak olyan helyzetek (kódújrafelhasználás szempontjából), hogy szeretnénk egy olyan típust készíteni, aminek egyes részei már meg vannak valósítva, viszont még nem áll készen a tényleges használatra. Az interfészekkel az a baj, hogy csak egy sablont adnak arra vonatkozóan, hogy egy osztálynak "hogyan kellene kinéznie", míg maguk a konkrét osztályokat már használatra készre írjuk meg. Ilyen esetekben az **absztrakt osztályok** jelenthetik a megoldást a programozónak, amelyek valahol az interfészek és a konkrét típusok között foglalnak helyet.
- Az **absztrakt osztályokat** nem lehet példányosítani, egyetlen ősosztálya lehet csak (ha explicite nem adjuk meg, akkor automatikusan a **System.Object**-ből származik), de bármennyi interfészt megvalósíthat.
- Ha egy osztály ősosztálya absztrakt, akkor vagy kötelezően deklarálni kell az absztrakt metódusokat vagy magát az osztályt is absztraktként kell megjelölni.

Absztrakt osztályok (2/2)

```
public interface IPerson {
  string Name { get; set; }
  void Greeting();
}
//az absztrakt osztályokat kötelezően meg kell jelölnünk az abstract kulcsszóval
public abstract class PersonBase : IPerson {
  public string Name { get; set; }
  //de ez nem elég a fordítónak, az absztrakt metódusokat is ugyanúgy meg kell jelölni
  //kódolvashatóság szempontjából is előnyös ez a fajta szintaxis
  //fontos, hogy a metódus törzsét, már nem kell megadnunk, elég csak egyszerűen lezárni egy pontosvesszővel
  public abstract void Greeting();

  //ez nem azt jelenti, hogy nem lehet konstruktora!
  //protected PersonBase() { /* ... ide jön valami kód ... */ }
}

public class Woman : PersonBase {
  //override kulcsszóval jelöljük meg, hogy az absztrakt metódust felüldefiniáltuk
  public override void Greeting() {
    Console.WriteLine("Hi, Miss/Mrs. {0}", Name);
  }
}

public class Man : PersonBase {
  public override void Greeting() {
    Console.WriteLine("Hi, Mr. {0}", Name);
  }
}

//ez a sor fordítási hibát dob
//PersonBase person = new PersonBase();
PersonBase person = new Woman();
```

Lezárt osztályok (sealed) (1/2)

- Mi történik akkor, amikor azt az igényünket szeretnénk megvalósítani, hogy ne lehessen tovább örököltetni egy adott osztályt?
- Egyik megoldásunk lehet az, hogy az összes konstruktorát elrejtjük: probléma, hogy akkor már kívülről nem is tudjuk példányosítani. Ezt könnyen ki lehet kerülni azzal, hogy csinálunk egy statikus metódust, ami visszatér az osztályunk egy példányával (lényegében egy factory pattern-ről van szó).
- Ez szép megoldás, de nem minden esetben egyértelmű: kell valamennyi dokumentáció (vagy teszteset), hogy a programozó tudja, hogyan kell használni.
- Másik megoldásunk, hogy nyelvi szinten támogassuk! Erre vezették be a **sealed** kulcsszót a nyelvben: ugyanúgy példányosíthatjuk mint bármelyik más osztályunkat, viszont az öröklésben nem jelenhet ősosztályként.
- Nem csak osztályokat, hanem metódusokat is meg lehet jelölni a sealed kulcsszóval.

Lezárt osztályok (sealed) (2/2)

```
public interface IPerson {
  string Name { get; set; }
  void Greeting();
}

//az absztrakt osztályokat kötelezően meg kell jelölnünk az abstract kulcsszóval
public abstract class PersonBase : IPerson {
  public string Name { get; set; }
  public abstract void Greeting();
}

//lezártuk az osztályt, nem lehet ősosztályként megadni
public sealed class SuperiorPerson : PersonBase {
  public override void Greeting() {
      Console.WriteLine("Behold the Superior Person!");
  }
}

//Nincs szuperebb a legszuperebb embernél ... ezért (is) kapunk fordítási hibát
public class MostSuperiorPerson : SuperiorPerson {
  //ha lezárjuk a metódusokat, akkor az alosztálya ennek az osztálynak, nem tudja
  //majd felüldefiniálni a Greeting() metódust. Ezzel a megoldással nagyon szépen
  //lehet finomhangolni a metódusainkat
  public sealed override void Greeting() {
      Console.WriteLine("Behold The Most Superior Person!");
  }
}
```

Parciális osztályok (partial)

- Egy-egy osztályunk (tipikusan a generált kódoknál) elérheti a több ezer sort... vagy olyan sok interfészt implementálunk az osztályunkba, hogy egyszerűbb lenne az interfészek alapján külön-külön fájlban tárolni az implementációkat: ezekre a problémákra kínál nekünk megoldást a **C#** a parciális osztályok (partial) képében.
- Egyszerűen arra jó, hogy az osztályunkat több fájlban tudjuk tárolni és ennek segítségével a kódunk sokkal áttekinthetőbb tud maradni. Előnyös logikailag különválasztani (partícionálni!) a nagyobb osztályunkat.

```
//SuperiorPerson.cs
sealed partial class SuperiorPerson : PersonBase {
    public override void Greeting() {
        DoSomethingAwesome();
        Console.WriteLine("Behold the Superior Person!");
    }
}

//SuperiorPerson.ISuperior.cs
interface ISuperior { void DoSomethingAwesome(); }

//Minden implementációnál kötelező megjelölni, hogy az adott osztály parciális!
//Miért is jó ez megint? Hát persze, hogy a kódolvashatóság szempontjából ...
partial class SuperiorPerson : ISuperior {
    public void DoSomethingAwesome() {
        Console.WriteLine("Okay dockey karaoke ;)");
    }
}
```

Statikus osztályok (static)

- Lehetőségünk van a nyelvben tisztán statikus osztályt **static class**) is készíteni. Ez azt jelenti, hogy minden adattagjának kötelezően statikusnak kell lennie.

- A statikus osztályokat nem lehet példányosítani, de ettől függően megadhatunk neki egy privát paraméterek nélküli konstruktort.
- Örököltetni sem lehet ezeket az osztályokat, illetve ők sem örökölhetnek (azaz nem lehet interfészt vagy ősosztályt megadni neki).
- Az öröklés miatt **protected** láthatóságot sem lehet használni (ami teljesen jogos, mivel nem lenne semmi értelme).

```
//készítünk egy statikus Ensure osztályt
static class Ensure
{
    //És egy statikus NotNull metódust
```

```
    public static void NotNull(object obj, string message)
    {
        if (obj == null)
            throw new ArgumentNullException(message);
    }
}

//Az adattagokat az osztály nevén keresztül tudjuk elérni:
void DummyFunction(string text) {
  Ensure.NotNull(text, "'text' cannot be null!");
  /* ... */
}
```

Generikus típusok (generic)

- Nem szeretjük a kód duplikációkat ... minél több a duplikált kód, annál nehezebb karbantartani a kódunkat, illetve fejleszteni azt. Klasszikus állatorvosi ló esete: válasszuk ki két elem közül a maximumot.
- Ilyen esetekben szeretnénk minél általánosabb megvalósítást találni: adjunk az osztályainkhoz (vagy metódusainkhoz) típusparamétert!
- Miért nem lehetne egyszerűen a paraméterátadásoknál az **object**-et használni? Lehet, viszont csúnya megoldás, nem beszélve arról, hogy elveszítjük fordítási időben a típusinformációt. Ennél elegánsabb megoldás a **generikus típusok** használata.

```
//Sokkal egyszerűbb a szintaxis, mint a C++-ban. Egyszerűen csak <..> kell írni
//az osztály neve után
public class MyCollection<TType> : ICollection<TType>
{
  //és valamit implementálunk
}
```

- Ha nem adunk meg semmilyen megszorítás a típusparaméterre vonatkozóan, akkor túl sok lehetőségünk nincs a használatánál: csak annyit tudhatunk biztosan, hogy az **object**-ből származik. A **MyCollection** esetében még azt sem tudjuk, hogy a **TType** értéktípusú vagy referencia típusú-e.

Megszorítások a típusparaméterekre (1/2)

- Adhatunk plusz információkat a fordítónak a típusparaméterre nézve: értéktípus-e vagy sem, milyen interfészeket valósít meg, milyen őosztálya van, van-e default konstruktora vagy sem ... Rengeteg lehetőségünk van, de azért vannak hátrányai is ennek a megoldásnak ...
- Egy-egy típusparaméterre vonatkozó megszorításokat a **where** kulcszó után kell felsorolnunk:

```
public TType Max<TType>(TType a, TType b)
  where TType: IComparable
  //where TType_2: ... stb.
{
  return a.CompareTo(b) > 0 ? a : b;
}
```

Megszorítások a típusparaméterekre (2/2)

| Kulcsszó | Leírás | Használata |

-
|
`class`	Referencia típusú osztályokat vár paraméterül ... Miért fontos? Azért, mert nélküle egy egyszerű null referencia ellenőrzést sem tudunk elvégezni.	**where** TType: **class**
`struct`	Ha van referencia típusú, akkor van értéktípusú osztály is. Ugyanaz a megfontolás, mint a class-nál	**where** TType: **struct**
Osztálynév	Explicite megadhatjuk, hogy mely osztályból öröklődjön a típusunk	**where** TType: **PersonBase**
Interfészek (itt többet is megadhatunk)	Explicite megadhatjuk, hogy mely interfészeinkből öröklődjön a típusunk	**where** TType: **IComparable, ISuperior**
`new()`	Feltesszük, hogy az osztályunknak van default konstruktora. Ezután már példányosíthatjuk is.	**where** TType: **new()**

Delegáltak (delegate)

- Nemcsak generikus típusokkal általánosíthatunk, hanem úgyis, hogy az alprogramokat (eljárások és függvények) értékekként (**first-**

class citizen) kezeljük.

- Definiálhatunk delegáltakat **delegate**) azaz függvénytípusokat. Segítségükkel, a metódusainkat általánosíthatjuk és paraméterekként alprogramokat adhatunk át értékként.

```
//Névterekben, illetve osztályokon belül is definiálhatjuk őket
delegate void Action();

//Lehetőségünk "túlterhelni" is őket, azaz több változatot készíteni
delegate void Action<TParameter>(TParameter parameter);

public void ForEach<T>(IEnumerable<T> enumerable, Action<T> method) {
  foreach (var item in enumerable) {
    method(item);
  }
}

ForEach(new String[] { "Hello", "World" },
  //Használata egyszerű, egyszerűen csak a delegate-t kell használnunk
  //utána pedig megadni, hogy milyen paramétert akarunk átadni
  delegate(string parameter) {
    Console.WriteLine(parameter);
  }
);

//Ha megegyezik a metódusunk típusa a delegálttal, akkor
//egyszerűen paraméterként is átadhatjuk
ForEach(new String[] { "Hello", "World" }, Console.WriteLine);
```

Események (event) (1/2)

- Elérkeztünk az eseményvezérelt programozáshoz. Mit is kell ezalatt érteni? Mi történik akkor, amikor szeretnénk valamilyen eseményről (valami példányosult, a felhasználó kattintott egyet a gombunkon stb.) értesülni?
- Ennek örömére a **C#** bevezette az eseményeket **event**).

```
event Action DoSomethingEvent;

//Event invocator -  ezt a metódust kell meghívnunk, ha azt akarjuk,
//hogy tüzeljen az eseményünk
void OnDoSomethingEvent()
{
    Action handler = DoSomethingEvent;
    if (handler != null) handler();
}

//+= operátorral lehet feliratkozni és
//-= operátorral lehet leiratkozni az eseményről
DoSomethingEvent += delegate() { Console.WriteLine("Event is fired"); };
OnDoSomethingEvent(); //tüzelni fog az eseményünk
```

- Az esemény típusának kötelezően valamilyen delegáltnak kell lennie. A tulajdonságokhoz hasonlóan, itt is megadhatjuk, hogy mi történjen ha valaki felirakozott (vagy esetleg leiratkozott) az eseményünkről.

Események (event) (2/2)

```
  private static event Action _DoSomethingEvent;

  //egyszerűen csak az add és remove kulcsszavakat kell használnunk
  //szintaxisa nagyon hasonlít a tulajdonságokra
  public static event Action DoSomethingEvent
  {
      //az add ág fut le akkor, amikor valaki feliratkozik az eseményünkre
      add
      {
          _DoSomethingEvent += value;
          Console.WriteLine("Adding ...");
      }
      //és a remove ág fut le, amikor leiratkoztunk róla
```

```
    remove
    {
        _DoSomethingEvent -= value;
        Console.WriteLine("Removing ...");
    }
}

protected static void OnDoSomethingEvent()
{
    //viszont használni már csak a privát eseményünket tudjuk használni
    Action handler = _DoSomethingEvent;
    if (handler != null) handler();
}
```

- A **Windows Forms**, de a **WPF** is erősen támaszkodik az eseményvezérelt programozásra. De ha ügyesek vagyunk, akkor a felhasználói felületen történt események nagyrészét ki tudjuk majd kerülni egy kicsit más módszerrel ...