Eseményvezérelt alkalmazások fejlesztése 2.

2. gyakorlat - A C# programozási nyelv alapjai

Szabó Tamás sztrabi@inf.elte.hu

C# nyelv jellemzői

Microsoft több mint 10 éve aktívan fejleszti a .NET Framework keretrendszerrel együtt: jelenleg a .NET Framework 4.6-ös és a C# 6.0-ás verziójánál tartunk.

A nyelv fejlesztését régebben az a **Anders Hejlsberg** vezette, aki a **Turbo Pascal** és a TypeScript nyelv tervezője is egyben.

Tisztán objektum-orientált programozási nyelv. Szintaxisának alapjául a **Java** és a **C++** szolgált. Minden osztály közös őse a System.Object osztály.

Fejlesztőknek lehetősége van eseményvezérelt, sablonvezérelt és funkcionális programozásra is.

Szigorúan típusos, automatikus szemétgyűjtés (*garbage collection*) van benne, de lehetőség van **unsafe** környezetben pointer-aritmetika segítségével irányítani az objektumainkat (nagyon-nagyon kivételes esetben van csak rá szükség).

.NET framework-öt támogató nyelvek

Rengeteg programozási nyelv támogatja magát a .NET Framework-öt, de azok közül is ezeket a nyelveket érdemes megismerni:

- C#
- Visual Basic.NET
- F#
- Managed C++
- IronPython
- IronRuby

Demo - Hello World

A kötelező **Hello World** programunk a következőképpen néz ki:

```
//Definiálunk egy új osztályt
class Program {
  //A statikus Main metódus lesz a belépési pontja a programunknak
  public static void Main(string[] args) {
    //Kiírjuk a console-ra a "Hello World" szöveget
    Console.WriteLine("Hello World!");
  }
}
```

Vezérlési szerkezetek

- Szekvencia

```
Method_1();
// ...
Method_n();
```

- Elágazás

```
if (/* feltétel */) {
//igaz ág
}
else if (/* feltétel */) {
//hamis ág
}
else {
//hamis ág
}
```

- Többszörös elágazás

```
switch (/*felsorolható típusú vagy diszkrét értékű objektum*/) {
case pattern_0:
  method_0();
  break;
...
case pattern_n:
  method_n();
  break;
default:
  method_default();
  break;
}
```

- Előltesztelő ciklus (`while`)

```
while (/* feltétel */) {
/* ... ciklusmag ... */
}
```

- Hátultesztelő ciklus (`do-while`)

```
do {
/* ... ciklusmag ... */
} while (/* feltétel */);
```

- `for` ciklus

```
for (/* inicializációs rész */;
   /* feltétel ellenőrzés */;
   /* terminálló függvény értékének csökkentése */) {
/* ... ciklusmag ... */
}
```

- `foreach` ciklus

```
foreach (int /*ciklusváltozó típusa*/ index /*ciklusváltozó neve*/ in
       new[] { 1, 2, 3 } /*iterálandó kifejezés*/) {
/* ... ciklusmag ... */
}
```

foreach ciklus

Megjelent egy új vezérlési szerkezet is a nyelvben, ami az iterálható objektumok bejárásában és azok feldolgozásában segít: `foreach` ciklus. Ilyet már láthattunk Java-ban, C++-ban vagy akár ECMAScript 6-ban (`for-of` + `symbol` type).

A **for** ciklustól eltéroen, itt nem lehet megváltoztatni a ciklusmagban a bejárt objektum tartalmát (azaz **nem** lehet hozzáadni vagy törölni elemeket a kollekcióból)! Erre a mellékhatások (*side-effect*) elkerülésére van szükség, így biztonságosabb és megbízhatóbb programot tudunk írni.

Csak olyan típusú objektumokon hívhatjuk meg a **foreach**-et, amik megvalósítják az IEnumerable vagy az IEnumerable<T> interfészeket. Használata:

```
string[] words = new string[] { "Hello", " ", "World", "!" };
foreach (string word in words)
{
  //ToUpperCase() metódus az string csupa nagybetűs változatával tér vissza
  //Fontos megjegyezni, hogy a string-ek immutable típusúak
  Console.Write(word.ToUpper());
} //Output: HELLO WORLD!
```

(+) IEnumerable és a foreach kapcsolata

Miért fontos, hogy az iterálható objektumunk megvalósítsa az `IEnumerable` interfészünket?

```
namespace System.Collection {
  public interface IEnumerator {
    //Visszatér az éppen aktuális elemmel
    object Current { get; }
    //Igazzal tér vissza, ha még nem értünk az iteráció végére,
    //illetve beállítja a Current tulajdonságunkat, az aktuális értékre
    bool MoveNext();
    //Visszaállítjuk az iterációt az eredeti állapotára
    void Reset();
  }

  public interface IEnumerable {
    //Visszaadja az objektumunknak az enumerátorát
    IEnumerator GetEnumerator();
  }
}
```

Bejárhatjuk (habár nem igazán kényelmes) az `IEnumerable` objektumainkat, `foreach` segítsége nélkül is (lásd az előző példát).

Típusok

A C# a típusokat két nagyobb csoportra osztja: **értéktípusok** (*value type*) és **referencia típusok** (*reference type*). Az első szembeötlő különbség az, hogy az értéktípusok automatikusan a System.ValueType osztályból származnak.

A másik dolog, hogy az **értéktípusok**-nak nem adhatunk értékül `null` referenciát. Illetve vannak más megszorítások is, de ezekrol majd később ...

Mivel tisztán objektum-orientált nyelvrol van szó, így minden objektumnak minosül. Lehetoségünk van felsorolható

típusokat (`enum`), interfészeket (`interface`), osztályokat (`class`), illetve elemi osztályokat (`struct`) definiálnunk.

Minden típusnak van egy univerzális közös őse: `System.Object` . Ennek megfelelően egy `object` típusú objektumnak, bármilyen értéket adhatunk:

```
int age = 12;
object o = age;     //érvényes értékadás (bár nem túl hatékony: boxing/unboxing)
o = "Hello World!"; //érvényes értékadás
age = o;            //típushiba!
age = (int)o;       //futásidejű hiba: "Hello World" nem egész szám!
o = 12;
age = (int)o;       //ez már jó
```

Alaptípusok

C# típus	CLR neve
bool	System.Boolean
byte	System.Byte
sbyte	System.SByte
char	System.Char
decimal	System.Decimal
double	System.Double
float	System.Single
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
object	System.Object
short	System.Int16
ushort	System.UInt16
string	System.String

Interfészek (interface)

Az interfészekre érdemes úgy gondolni, mint egy halmaz tele absztrakt metódusokkal.

Azok az osztályok, amik öröklik az interfészeket, vállalják azt a kötelezettséget, hogy ezeket az absztrakt metódusokat ők (vagy a leszármazottjai) implementálni fogják. Úgyis gondolhatunk rájuk, mint szerződésekre, melyeket az osztályoknak kötelező betartani.

Öröklődési hiearchiát is definiálhatunk az interfészek között (de csak azok között!). Az interfészek között, megengedett a többszörös öröklődés is.

Szintaxisa nagyon egyszerű:

```
//A konvenció az, hogy minden interfész nevének nagy "I" betűvel kell kezdodnie
interface ILogger {
  //a metódusok szignatúrájánál nem kell megadni a láthatóságot,
  //ugyanis minden metódus, tulajdonság kötelezoen publikus
  void Log(string message);
  void Log(int logType, string message);
}

interface IConsoleLogger : ILogger {
  Color BackgroundColor { get; }
  void SetBackgroundColor(int r, int g, int b);
}
```

Elemi osztályok (struct)

A **C++**-tól eltéroen a **C#** különbséget tesz a `struct`-ok és a `class`-ok között. Az olyan típusokat, amiket `struct`-okként definiálunk, értéktípusoknak fog tekinteni (azaz közvetlen leszármazottja lesz a `System.ValueType` osztálynak).

Több megkötés is van az értéktípusokra vonatkozóan: példányosításnál a legtöbb esetben a **stack**-en tárolódnak, mindig értékként kezelodik. Továbbá nem szerepelhet ősosztályként az öröklődésben, azonban ugyanúgy mint az osztályoknál, bármennyi interfészt implementálhat (viszont veszélyes!).

A megkötések nem állnak meg itt: nem lehet alapértelmezett konstruktora (**default constructor**) - ami egyszerre igaz is, illetve nem ... lásd: **C# 6.0** - nem lehet az elemeit se inicializálni.

Mikor használjuk őket? Leginkább akkor, amikor kis objektumokat akarunk készíteni, amik adattárolás céljára lettek megtervezve, illetve nem akarunk leörököltetni belolük komoly hiearchiákat és gyors adatmozgatásra alkalmasak.

```
struct Point {
  //a pont X koordinátája
  public int X { get; private set; }
  //a pont Y koordinátája
  public int Y { get; private set; }

  //túlterhelt konstruktor
  public Point(int x, int y)
    : this() /* meghívjuk a default konstruktort meghívás elott */ {
      X = x;  //beállítjuk az átadott adatokat
      Y = y;
  }

  //a könnyebb használhatóság miatt felüldefiniáltuk a ToString() metódust
  public override string ToString() {
      return string.Format("({0}, {1})", X, Y);
  }

  //statikus konstans Point, ami a koordinátarendszerünk közepét reprezentálja
  public static readonly Point Origo = new Point(0, 0);
}

//példányosítás ugyanúgy történik, mint a referenciatípusok esetén
Point p = new Point(1, 2);
```

Osztályok (class)

Legfontosabb fogalom a nyelvben maga az osztály, mivel minden érték objektum és minden típus osztály.

Az osztály tartalmazhat **mezoket**, **tulajdonságokat**, **metódusokat**, **eseményeket**, illetve belso osztályok, interfészeket vagy felsorolható típusokat.

Csak egyszeres öröklodés van megengedve az osztályok között (azaz csak egyetlen ososztály lehet), de korlátlan mennyiségben implementálhat interfészeket. Ha nem adjuk meg explicite az ososztály, akkor automatikusan a **System.Object**-ból fog öröklodni.

Osztályok között is megkülönböztetünk több osztályt, amikrol a késobbiekben lesz még szó.

```
public class Person {
  public string Name;
  public int Age;

  public void SayHello() {
    Console.WriteLine("Hello " + Name + " (" + Age + ")");
  }
}

Person p = new Person();
p.Name = "Gipsz Jakab"; p.Age = 34;
p.SayHello(); //Output: Hello Gipsz Jakab (34)
```

Láthatóság az osztályok esetén

Objektum-orientált programozásban (és szoftvertechnológiai szempontból is) fontos, hogy az adataink megfelelo elrejtése. Ezt a nyelv tervezoi is tudták, ezért több lehetoséget bevezettek a fejlesztok számára: `private`, `protected`, `public`, `internal`, `protected internal`.

Mind az osztályok adattagjainak (mezok, metódusok stb.), mind pedig maguknak az osztályoknak (illetve interfészek, felsorolható típusok és elemi osztályok esetében is) be lehet állítani, hogy milyen láthatósággal is rendelkezzen. Ha nem adunk meg semmit se, akkor a fordító automatikusan *privátként* fogja értelmezni.

C++-tól eltér kicsit a szintaxis a láthatóságoknál. Itt kötelezo megjelölni minden adattagnál, hogy az milyen láthatósággal van definiálva. Ez a könnyebb olvashatóságot szolgálja.

Még egy eltérés az is, hogy ebben a nyelvben nincs lehetoség rá, hogy öröklodésnél meghatározzuk, hogy az örökölt adattagok milyen láthatósággal legyenek definiálva az új osztályban:

```
public class Employee : public Person //szintaktikai hiba!
{
  /* ... */
}
```

Metódusok

Mivel a nyelv tisztán objektum-orientált, ezért a névterekben nem definiálhatunk magukban eljárásokat/függvényeket (metódusokat), kötelezően az osztályokon belül kell oket megadni:

```
class Person {
  //a láthatóságot (ha csak nem private), mindig meg kell adni!
  public void Foo(/* paraméterlista */) {
    /* a metódus törzse ... */
  }

  public int Foo2() {
    /* a függvény törzse ... */
    //függvényeknél minden esetben meg kell hívni a return-t,
    //ha ezt nem tesszük meg, akkor fordítási hibát kapunk: a cél,
    //hogy minél biztonságosabb kódot írjunk
    return 0;
  }
}
```

Ugyanúgy, mint **C++** esetében is, itt sem különböztetjük meg szintaktikailag az eljárások a függvényektol. Ha eljárást akarunk írni, akkor egyszerűen a System.Void (`void`) típussal kell visszatérnünk.

Paraméterátadási lehetőségek

Alapértelmezett paraméterátadás:

```
void PrintData(string name, int age) {
  Console.WriteLine("Name: " + name + "(" + age + ")");
  //Console.WriteLine("Name: {0} ({1})", name, age);
}

PrintData("Gipsz Jakab", 10);
//lehetoség van név szerinti paraméterátadásra is
//(amit az ADA már elég régóta támogat, míg a C++ nem ...)
PrintData(age: 10, name: "Gipsz Jakab");
```

Kimenő érték szerinti (`out`):

```
bool TryGetCategoryByAge(int age, out string category) {
  //a category paraméter csak az értékadás bal oldalán jelenhet meg!
  //ha megsértjük ezt a megkötést, akkor a fordító hibát fog dobni
  //(ez azt is jelenti, hogy a bemeno érték nem számít: azaz csak változót adhatunk át neki,
  //kifejezést NEM!)
  if (age < 18) {
    category = "teenager";
    return true;
  }
  else if (age >= 18) {
    category = "adult";
    return true;
  }
  return false;
}

string category = string.Empty; //lehet használni a category = ""; is
//a nyelv szintaxisa megköveteli, hogy explicite kiírjuk metódushíváskor az out kulcsszót
if (TryGetCategoryByAge(12, out category))
  Console.WriteLine("Category: " + category);
```

Cím szerinti paraméterátadás (`ref`)

```
//klasszikus példa a cím szerinti paraméterátadásra
 void Swap(ref string obj1, ref string obj2) {
   string temp = obj1;
   obj1 = obj2;
   obj2 = temp;
 }

 string a = "Hello";
 string b = "World";
 Swap(ref a, ref b);  //ugyanúgy mint az out-nál, itt is kötelezo kitenni a ref kulcsszót
 Console.WriteLine(a, b); //Output: World Hello
```

Változó számú paraméter átadása (`params`)

```
//a fordító a paramétereket egy tömbbe gyűjti, majd ezt a tömböt adja át
//a metódusnak: ezzel a technikával egyszerűen és elegánsan lehet feldolgozni
//a bemeno adatokat
public void LogMessage(params string[] messages) {
  foreach (string message in messages) {
    Console.WriteLine("Logged: " + message);
  }
}

//metódus meghívás szempontjából nem különbözik semmitol se
//ugyanúgy vesszovel elválasztva, több paramétert adunk át neki
LogMessage("Http error!", "Logging error ...");
LogMessage("Http error!"); //ugyanazt a metódust hívtuk meg és nem egy túlterhelt változatát
```

Mezők használata

Ugyanúgy, mint a **C++**-ban (vagy akár **Java**-ban), itt is lehetoség van az osztályokon belül **mezőket** (**field**) definiálni.

Szintaktikai különbség, hogy a láthatóságot minden mezonél explicite meg kell adni!

```
  public class Person {
    public string Name; //A személyünk neve
    public int Age; //A személyünk életkora
  }
```

A mezok definiálásakor a kezdoértéket is adhatunk nekik:

```
public class Person {
  public string Name = "Gipsz Jakab"; //A személyünk neve
  public int Age = 30; //A személyünk életkora
}
```

Illetve lehetnek statikus mezői is az osztályoknak:

```
public class Person {
  public static int Count = 0; //Példányosított személyek száma
  public string Name = "Gipsz Jakab"; //A személyünk neve
  public int Age = 30; //A személyünk életkora
}
```

Mezők hiányosságai

Van egy kis gond a mezőkkel: nem szeretjük őket közvetlenül használni, ugyanis a felhasználó (vagyis a mi esetünkben az osztályunkat használó programozó), úgy megváltoztathatja az objektumunkat, hogy azzal a példányunk inkonzisztens, érvénytelen állapotba kerülhet, anélkül, hogy értesülnénk a hibáról (elmarad a validáció). Mi a helyzet, ebben az esetben?

```
Person person = new Person();
//Érvénytelen állapotba kerül az objektumunk,
//mert szemantikailag mi nem szeretnénk olyan személyt, aki -10 éves!
person.Age = -10;
```

Erre megoldást getter/setter (érték lekérdező/beállító) függvények jelentik. Ilyenekkel leginkább **Java**-ban lehet találkozni:

```
public class Person {
  // Elrejtjük a felhasználótól a mezoket, így kívülrol nem tudják használni
  private string _name; // Névkonvenció (lehet), hogy _ prefixxel kezdődjenek a mezők nevei
  private int _age;

  public void SetName(string value) { _name = value; }
  public string GetName() { return _name; }

  public void SetAge(int value) {
    // Ebben az esetben validálni is tudunk, ezzel próbáljuk megőrizni a mezők érvényességét
    if (value >= 0) _age = value;
  }
  public int GetAge() { return _age; }
}
```

Tulajdonságok használata

Az előző megoldásunk egy nagyon jó irány, hiszen elegánsan tudunk validálni, illetve ha kell akár transzformálni is adatokat, olyan formába, amire nekünk szükségünk van. Két baj van vele: sokat kell gépelni és ronda

```
Person person = new Person();
person.SetAge(-10);
bool isInvalid = person.GetAge() == -10; //false lesz a kifejezés értéke
```

Erre a megoldásra a nyelv készítői egy szintaktikai elemet emeltek a nyelvbe: a **tulajdonságokat** (**property**):

```
public class Person {
  private string _name; //úgynevezett backfield-eknek nevezzük ezeket a mezoket
  private int _age;

  //Névhez tartozó tulajdonság
  public string Name {
    set { _name = value; }  //setter metódus
    get { return _Name; }   //getter metódus
  }

  //Életkorhoz tartozó tulajdonság
  public int Age {
    set {
      if (value >= 0) _age = value; //ugyanígy megtörténik a validáció
    }
    get { return _age; }
  }
}
```

Habár úgy viselkednek, mint a metódusok, használni mégis úgy kell oket, mint egyszerű mezoket. De fontos megjegyezni, hogy a nyelv **property**-ként tartja számon, szóval se nem metódusok, se nem mezok!

```
  Person person = new Person();
  person.Age = -10; // ilyenkor meghívódik a Person.Age tulajdonság set ága

  // false lesz megint az értéke, mert nem közvetlenül a mezőt állítjuk be
  // hanem egy setter eljáráson keresztül állítjuk be a személy életkorát,
  // ennek köszönhetően lefut a saját validációnk
  bool isInvalid = person.Age == -10;
```

A tulajdonságok használatának segítségével, kódunk olvashatóbb marad, illetve kevesebb kódot is kell írnunk (boilerplate code). **WPF**-nél még fontosabb jelentőségük lesz.

Lehetőségünk van arra is, hogy a **getter/setter** metódusok láthatóságát külön-külön megadhassuk az egyes tulajdonságoknak, illetve nem is kötelezo mindkettő metódust definiálni: lehet csak read-only (csak `get` ág van), write-only (csak `set` ág van), illetve mindkettő (megvan a `get` és `set` ág is).

Szóval a láthatóságra még visszatérve: alakítsuk át a **Person** osztályunkat **immutable**-re:

```
public class Person {
  private string _name;
  private int _age;

  //Person osztály konstruktora
  public Person(string name, int age) {
    Name = name;  //viszont az osztályon belül még látjuk a tulajdonságok setter metódusát!
    Age = age;
  }

  public string Name {
    private set { _name = value; } //mostmár csak lekérdezni lehet, beállítani nem
    get { return _name; }
  }

  public int Age {
    private set { _age = value; }
    get { return _age; }
  }
}

Person p = new Person("Gipsz Jakab", 24);
p.Age = -10; //fordítási hiba, az Age setter metódusa private láthatóságú!
```

Automatikus tulajdonságok

Mégis van itt egy kisebb probléma: még mindig sokat kell gépelnünk ... Ezt a nyelv készítői is észrevehették, ugyanis bevezették az automatikus tulajdonságokat (**automatic properties**): célja, hogy a privát **backfield**-eket a fordító majd automatikusan generálja nekünk.

```
public class Person {

  public Person(string name, int age) {
    Name = name;  //viszont az osztályon belül még látjuk a tulajdonságok setter metódusát!
    Age = age;
  }

  //a fordító automatikusan kigenerálja a tulajdonságokhoz a megfelelo mezoket
  public string Name { private set; get; }

  public int Age { private set; get; }
}

Person p = new Person("Gipsz Jakab", 24);
p.Age = -10; //Használatában semmi változást nem lehet megfigyelni kívülrol!
```

Ezzel a megoldással már tényleg sokkal olvashatóbb és áttekinthetőbb lesz a kód.

Getter-only tulajdonságok (C# 6)

```
public class Person {
  public Person(string name, int age) {
    Name = name;  //viszont az osztályon belül még látjuk a tulajdonságok setter metódusát!
    Age = age;
  }

  // nyugodtan elhagyhatjuk a setter-eket
  public string Name { get; }

  // FONTOS: csak a konstruktorban adhatunk nekik értéket az osztályon belül, máshol nem
  public int Age { get; }
}

Person p = new Person("Gipsz Jakab", 24);
p.Age = -10; //Használatában semmi változást nem lehet megfigyelni kívülrol!
```

A háttérben a fordító a következőt generálja:

```
public class Person {
  private readonly string _name;
  private readonly string _age;

  public Person(string name, int age) {
    _name = name;  //viszont az osztályon belül még látjuk a tulajdonságok setter metódusát!
    _age = age;
  }

  public string Name { get { return _name; } }
  public int Age { get { return _age; } }
}
```

Expression-bodied properties and methods (C# 6)

A C# 6.0-tól kezdve bevezették a nyelvbe az úgynevezett **kifejezés-törzsű tulajdonságokat** és **metódusokat**. Nem több egyszerű nyelvi egyszerűsítéstől (*syntactic sugar*), amivel rövidebb kódot lehet írni.

```
public class Person {
  public Person(string name, int age) {
    Name = name;  //viszont az osztályon belül még látjuk a tulajdonságok setter metódusát!
    Age = age;
  }

  public string Name { get; }
  public int Age { get; }

  // expression-bodied property
  public string CapitalName => Name.ToUpper();

  // expression-bodied method
  public string ToString() => Name + "(" + Age + ")"; // megjegyzés: nagyon gagyi implementáció
  // public string ToString() => $"{Name}({Age})";
}
```

Köszönöm a figyelmet!