Eötvös Loránd Tudományegyetem
Informatikai Kar

Eseményvezérelt alkalmazások fejlesztése II

2. előadás

A C# nyelvi elemei és nyelvi könyvtára

Giachetta Roberto

A jegyzet az ELTE Informatikai Karának
2014. évi Jegyzetpályázatának támogatásával készült

A C# nyelvi elemei és nyelvi könyvtára

Előfordítási direktívák

- A nyelv tartalmaz előfordítási direktívákat, amelyek előzetesen kerülnek feldolgozásra, így lehetőséget adnak bizonyos kódsorok feltételes fordítására, hibajelzésre, környezetfüggő beállítások lekérdezésére, pl. **`#if`**, **`#define`**, **`#error`**, **`#line`**
- Mivel nem választható szét a deklaráció a definíciótól, a kód tagolását a *régió*k segítik elő, amelyek tetszőleges kódblokkokat foghatnak közre:

```
#region <név>
…
#endregion
```

  - nem befolyásolják a kódot, csupán a fejlesztőkörnyezetben érhetőek el

A C# nyelvi elemei és nyelvi könyvtára

Megjegyzések

- Az egyszerű *megjegyzések* a fordításkor törlődnek
  - sor végéig tartó: `// megjegyzés`
  - tetszőleges határok között: `/* megjegyzés */`
- A *dokumentációs megjegyzés*ek fordításra kerülnek, és utólag előhívhatóak a lefordított tartalomból
  - osztályok és tagjaik deklarációjánál használhatjuk
  - célja az automatikus dokumentálás elősegítése és a fejlesztőkörnyezetben azonnal segítség megjelenítése
  - a `///` jeltől a sor végéig tart, belül XML blokkok adhatóak meg, amelyek meghatározzák az információ jelentését

A C# nyelvi elemei és nyelvi könyvtára

Megjegyzések

- pl.:

```
/// <summary>
/// Racionális szám típusa.
/// </summary>
/// <remarks>Két egész szám hányadosa.</remarks>
struct Rational {
    …
    /// <summary>
    /// Racionális szám példányosítása.
    /// </summary>
    /// <param name="n">Számláló.</param>
    /// <param name="d">Nevező.</param>
    public Rational(Int32 n, Int32 d) { … }
    …
```

A C# nyelvi elemei és nyelvi könyvtára

Kivételkezelés

- A .NET keretrendszerben minden hiba kivételként jelenik meg
  - a kivétel általános osztálya az **`Exception`**, csak ennek vagy leszármazottjának példánya váltható ki
  - kivételt kiváltani a **`throw`** utasítással tudunk:
    `throw new <kivétel típusa>(<paraméterek>);`
  - kivételt kezelni egy kivételkezelő (**`try-catch-finally`**) szakasszal tudunk:

```
try { <kivételkezelt utasítások> }
catch (<elfogott kivétel típusa>){
   <kivételkezelő utasítások>
}
finally { <mindenképp lefuttatandó utasítások> }
```

A C# nyelvi elemei és nyelvi könyvtára

Kivételkezelés

- Pl.:

```
class WorkingClass {
   public void DoSomething(Int32 number)
   {
      if (number < 1)
         throw new ArgumentOutOfRangeException();
         // kivétel kiváltása (a paraméter hibás
         // tartományban van)
      …
      throw new Exception("Too lazy…");
      // kivétel kiváltása (üzenettel)
   }
   public void Finish() { … }
}
```

A C# nyelvi elemei és nyelvi könyvtára

Kivételkezelés

- Pl.:

```
WorkingClass wc = new WorkingClass();
try // kivételkezelő blokk
{
   wc.DoSomething();
}
// a kivételt típustól függően kezelhetjük
catch (ArgumentOutOfRangeException ex)
{ … }
// az Exception típusú kivételt nem kezeljük le
finally {
   wc.Finish(); // de ez mindenképpen lefut
}
```

A C# nyelvi elemei és nyelvi könyvtára

Generikus típusok

- Generikus programozásra futási időben feldolgozott sablon típusok (*generic*-ek) segítségével van lehetőség
  - osztály, metódus és delegált lehet sablonos, a sablon csak osztály lehet
  - a sablon fordításra kerül, és csak a futásidejű fordításkor helyettesítődik be a konkrét értékre
  - pl.:

```
struct Rational<T> {
    private T nom; // használható a T típusként
    …
    public Rational(T n, T d) { … }
    …
}
```

```
…
Rational<SByte> r1 = new Rational<SByte>(10,5);
Rational<Int64> r2 = new Rational<Int64>(10,5);
// különböző értékkészletű racionálisok
```

- A szigorú típuskezelés miatt a sablonra csak a **`Object`**-ben értelmezett műveletek használhatóak, ezt a műveletkört növelhetjük megszorításokkal (**`where`**)
  - pl.:

```
class Rational<T> where T : struct, IComparable,
          IFormattable, IConvertible { …
   // T elemi osztály, amire használható a fenti
   // interfészek összes művelete
}
```

A C# nyelvi elemei és nyelvi könyvtára

Tömbök

- A tömbök osztályként vannak megvalósítva (**`System.Array`**), de egyszerűsített szintaxissal kezelhetőek, pl.:

```
Int32[] myArray = new Int32[10]; // létrehozás
myArray[0] = 1; // első elem beállítása
```

  - referencia szerint kezeltek, méretnek változó is megadható, az értékek inicializálhatóak, pl.:

```
Int32[] myArray = new Int32[] { 1, 2, 3, 4 };
   // a tömb 4 hosszú lesz
```

  - akár több dimenziósak is lehetnek, pl.:

```
Int32[,] myMatrix = new Int32[10, 5]; // mátrix
myMatrix[0, 0] = 1; // első sor első eleme
Double[,,] myMatrix3D = new Double[10, 5, 10];
  // 3 dimenziós mátrix
```

A C# nyelvi elemei és nyelvi könyvtára

Tömbök

- Fontosabb műveletei:
  - hossz lekérdezés (**`Length`**, **`LongLength`**, **`GetLength`**)
  - dimenziószám lekérdezése (**`Rank`**)
- Statikus műveletként számtalan lehetőségünk van, pl.:
  - másolás (**`Copy`**), átméretezés (**`Resize`**)
  - rendezés (**`Sort`**), fordítás (**`Reverse`**)
  - lineáris keresés (**`Find`**, **`IndexOf`**, **`LastIndexOf`**), bináris keresés (**`Binary Search`**)
- A tömböknél (és más gyűjteményeknél) alkalmazott indexelő művelet megvalósítható saját típusokra is (paraméteres tulajdonságként)

A C# nyelvi elemei és nyelvi könyvtára

Gyűjtemények

- A gyűjtemények a **`System.Collections`** névtérben találhatóak, a legtöbb gyűjteménynek van általános és sablonos változata is, pl.:
  - dinamikus tömbök: **`ArrayList`**, **`List<T>`**, **`SortedList`**, **`SortedList<Key, Value>`**
  - láncolt listák: **`LinkedList<T>`**
  - verem: **`Stack`**, **`Stack<T>`**
  - sor: **`Queue`**, **`Queue<T>`**
  - asszociatív tömb: **`Hashtable`**, **`Dictionary<Key, Value>`**, **`SortedDictionary<Key, Value>`**
  - halmaz: **`HashSet<T>`**, **`SortedSet<T>`**

A C# nyelvi elemei és nyelvi könyvtára

Gyűjtemények

- A nem sablonos gyűjteményekbe bármilyen elemeket helyezhetünk
- A dinamikus tömbök indexelhetőek, és változtatható a méretük (bárhova beszúrhatunk, bárhonnan törölhetünk), pl.:

```
List<Int32> intList = new List<Int32>();
   // üres tömb létrehozása
intList.Add(1); … // elemek hozzáadása
intList.Insert(0, 100); // beszúrás az elejére
…
intList.Remove(100); // elem törlése
for (Int32 i = 0; i < intList.Count; i++)
   Console.WriteLine(intList[i]);
   // lekérdezés
intList.Clear(); // kiürítés
```

A C# nyelvi elemei és nyelvi könyvtára

Fájlrendszer-kezelés

- A fájlokkal és fájlrendszerrel kapcsolatos műveletek a **`System.IO`** névtérben helyezkednek el
  - fájlműveleteket a **`File`**, könyvtárműveleteket a **`Directory`** osztály statikus műveleteivel hajthatunk végre, pl.:

```
Directory.CreateDirectory(@"c:\Data");
   // könyvtár létrehozása
String[] paths = Directory.GetFiles(@"c:\Data");
   // könyvtár listázása
File.Copy(@"c:\data.txt", @"c:\Data\data.txt");
   // fájl másolása
```

  - az elérési útvonallal kapcsolatos műveletek a **`Path`** osztályban találhatóak, pl.:

```
Path.GetParent(@"c:\Data"); // szülő lekérdezése
```

A C# nyelvi elemei és nyelvi könyvtára

Fájlkezelés

- Az adatfolyamok kezelése egységes formátumban adott, így azonos módon kezelhetőek fájlok, hálózati adatforrások, memória, stb.
  - az adatfolyamok ősosztálya a **`Stream`**, amely binárisan írható/olvasható
- Szöveges adatfolyamok írását, olvasását a **`StreamReader`** és **`StreamWriter`** típusok biztosítják
  - létrehozáskor megadható az adatfolyam, vagy közvetlenül a fájlnév
  - csak karakterenként (**`Read`**), vagy soronként (**`ReadLine`**) tudunk olvasni, így konvertálnunk kell

- amennyiben a műveletek során hiba keletkezik, **`IOException`**-t kapunk

- Pl.:

```
StreamReader reader = new StreamReader("in.txt");
   // fájl megnyitása
while (!reader.EndOfStream) // amíg nincs vége
{
   Int32 value = Int32.Parse(reader.ReadLine());
      // sorok olvasása, majd konvertálás
   …
}
reader.Close(); // bezárás
```

A C# nyelvi elemei és nyelvi könyvtára

Erőforrások felszabadítása

- A referencia szerinti változók törlését a szemétgyűjtő felügyeli
  - adott algoritmussal adott időközönként pásztázza a memóriát, törli a felszabadult objektumokat
  - sok, erőforrás-igényes objektum példányosítása esetén azonban nem mindig reagál időben, így nő a memóriahasználat
  - a **`GC`** osztály segítségével beavatkozhatunk a működésbe
- A manuális törlésre (destruktor futtatásra) nincs lehetőségünk felügyelt blokkban, de erőforrások felszabadítására igen, amennyiben az osztály megvalósítja az **`IDisposable`** interfészt, és benne a **`Dispose()`** metódust

A C# nyelvi elemei és nyelvi könyvtára

Erőforrások felszabadítása

- Emellett a C# nyelv tartalmaz egy olyan blokk-kezelési technikát, amely garantálja a **`Dispose()`** automatikus futtatását:

```
using (<objektum példányosítása>){
   <objektum használata>
} // itt automatikusan meghívódik a Dispose()
```

- Pl.:

```
using (StreamReader reader = new StreamReader(…)){
   // a StreamReader is IDisposable
   …
}
// itt biztosan bezáródik a fájl, és
// felszabadulnak az erőforrások
```

A C# nyelvi elemei és nyelvi könyvtára

Lamda-kifejezések

- A *lambda-kifejezések (lambda-expressions)* funkcionális programozásból átvett elemek, amelyek egyszerre függvényként és objektumként is viselkednek
- A λ-kifejezést az **=>** operátorral jelöljük, tőle balra a paraméterek, jobbra a művelet törzse írható le, pl.:

```
a => a * a // négyzetre emelés
x => x.Length < 5 // 5-nél rövidebb szövegek
(x, y) => x + y; // összeadás
() => 5; // konstans 5
```

- A λ-kifejezést elmenthetjük változóként is, típusa a sablonos **Func<...>** lesz, pl.:

```
Func<String, Boolean> lt5 = x => (x.Length < 5);
```

A C# nyelvi elemei és nyelvi könyvtára

Lamda-kifejezések

- Az eltárolt kifejezés bármikor futtathatjuk, mint egy függvényt, pl.: `Boolean l = lt5("Hello!"); // l hamis lesz`
- A λ-kifejezések tetszőlegesen összetett utasítássorozatot is tartalmazhatnak, nem csak egy kifejezés kiértékelését, ekkor a tartalmat blokkba kell helyezni, pl.:

```
Func<Int32, Int32> pow2 = x => {
   x = x * x;
   return x;
};
```

- A λ-kifejezések speciális típusa az akció (**`Action`**), amely egy paraméter és visszatérési érték nélküli tevékenység, pl.:

```
Action hello = () => { Console.Write("Hello!"); };
```

A C# nyelvi elemei és nyelvi könyvtára

Nyelvbe ágyazott lekérdezések

- A *nyelvbe ágyazott lekérdezések (Language Integrated Query)* célja, hogy objektumorientált környezetben valósíthassunk meg lekérdező utasításokat
  - hasonlóan a relációs adatbázisok SQL nyelvéhez
  - pl.:

```
List<Int32> nrList = new List<Int32> { 1, … };
var numQuery = from i in numberList // honnan
               where i < 4 // feltétel
               select i; // mit
```

  - az eredmény egy gyűjtemény (**`IEnumerable`**) lesz, és a kifejezés csak akkor értékelődik ki, amikor azt bejárjuk (*késleltetett végrehajtás*)

A C# nyelvi elemei és nyelvi könyvtára

Nyelvbe ágyazott lekérdezések

- A nyelvbe ágyazott lekérdezések mögött λ-kifejezésekkel dolgozó metódusok találhatóak, amelyek bármilyen gyűjteményre futtathatóak (akár külön-külön is)
  - pl.:

```
var numQuery = numberList // honnan
               .Where(i => i < 4) // feltétel
               .Select(i => i); // mit
```

  - a metódusok úgynevezett *bővítő metódus*ként definiáltak, amelyek elérhetőek a **`System.Linq`** névtérben
  - bonyolultabb lekérdezések is megvalósíthatóak (pl. unió, csoportosítás, összekapcsolás, rendezés, …)

A C# nyelvi elemei és nyelvi könyvtára

Nyelvbe ágyazott lekérdezések

- Pl.:

```
Int32[] s1 = { 1, 2, 3 }, s2 = { 2, 3, 4 };
Int32 sum = s1.Sum(); // számok összege
Int32 evenCount = s1.Sum(x => x % 2 == 0 ? 1 : 0);
   // megadjuk, mit összegezzen, így a páros
   // számok számlálása lesz
var union = s1.Union(s2);
   // két gyűjtemény uniója: { 1, 2, 3, 4 }
var evens = union.Select(x => x % 2 == 0);
   // páros számok kiválogatása
Int32 evenCount =
   s1.Union(s2).Sum(x => x % 2 == 0 ? 1 : 0);
   // unió, majd a páros számok számlálása
```