Köztes C# szerkezetek

Jánosi-Rancz Katalin Tünde

Sapientia EMTE
tsuto@ms.sapientia.ro

Metódusreferenciák, események és lambdák

Delegates, Events, Lambdas

- delegate: egy típusbiztos objektum, mely **hivatkozik** egy/több metódusra
- event: állapot megváltozás értesítésére használjuk
- lambda kifejezés: egy névtelen metódus / függvény

A .NET Delegate Típus

- hasonló a C/C++ függvénymutatóihoz
- delegate: egy **referenciatípus**, de nem egy objektumra hanem egy/több metódusra hivatkozik. Rajta keresztül hívható meg a metódus.
- egy típusbiztos objektum, amely egy listát tárol a meghívandó metódusokról
- deklarációjánál megadjuk, hogy milyen szignatúrával rendelkező metódusok megfelelőek
- 3 fontos információt kezel:
  - a metódusnak a címét, amelyen hívásokat hajt végre
  - a metódus paramétertípus-listáját (ha vannak)
  - a metódus visszatérítési értékét (ha vannak)

A .NET Delegate Típus -2

- metódusait **szinkron** vagy **aszinkron** módon hívja meg (később)
  - meghívhatunk egy metódust egy másik szálon, anélkül, hogy manuálisan létrehoznánk és kezelnénk egy Thread objektumot
- legnagyobb haszna, hogy nem kell előre megadott metódusokat használnunk, ehelyett később tetszés szerint adhatjuk meg az elvégzendő műveletet
- a következő esetekben használható: eseménykezelésben, Callback (egy fgv. más fgv-t visszahív), LINQ, Design pattern-ek implementációjában

A metódusreferencia definiálása C# -ban

- példa: ez a delegate **típus** olyan metódusra mutathat, amelynek visszatérési értéke `int` típusú és két `int` típusú paramétere van:

```
public delegate int BinaryOp(int x, int y);
```

- egy delegált típussal azok a metódusok kompatibilisek, amelyek paraméterlistája és visszatérési értéke megegyezik a delegált típuséval
- **példányosítás** és **meghívás**:

```
public class SimpleMath {
  public static int Add(int x, int y) { return x + y; } //NB: lehet
      static!
}
...
BinaryOp binop = new BinaryOp(SimpleMath.Add);
Console.WriteLine("1 + 1 is {0}", binop(1, 1));
```

- példányosítás alternatív szintaxisa:

```
BinaryOp binop = SimpleMath.Add;
```

Delegatek mint paraméterek

A delegate-ket azért használjuk, hogy egy metódusnak paraméterként egy másik metódust tudjunk átadni.

```
public delegate int Transformer (int x);

class Util {
  public static void Transform (int[] values, Transformer t) {
    for (int i = 0; i < values.Length; i++)
      values[i] = t(values[i]);
  }
}

class Test {
  static int Square(int x) { return x * x; }
  static void Main() {
    int[] values = { 1, 2, 3 };
    Util.Transform(values, Square);  // Dynamically hook in Square
    foreach(int v in values)
      Console.Write("{0} ", v);      // 1 4 9
  }
}
```

Többes küldés (Multicasting) engedélyezése

- a delegate támogatja a többes küldést - **multicast**-ot:
- a delegate-ekhez egynél több metódust is hozzáadhatunk, a += operátort kell használni
- a delegate hívásakor a listáján lévő összes metódust meghívja a megadott paraméterre

```
SomeDelegate d = SomeMethod1;
d += SomeMethod2; //or d = d + SomeMethod2;
```

- meghívva `d` -t meghívódik a `SomeMethod1` majd a `SomeMethod2`
- metódus eltávolítása a hívási listáról:

```
d -= SomeMethod1;
```

- delegate megváltoztathatatlan **immutable**, i.e. += egy új delegate példányt hoz létre
- **fontos:** ha egy multicast delegate-nek nem void típusú visszatérítési értéke van, akkor a meghívó csak az utolsó metódus visszatérítési értékét kapja meg

Multicast Delegate példa

```
public delegate void ProgressReporter(int percentComplete);
public class Util {
  public static void HardWork(ProgressReporter p) {
    for (int i = 0; i < 10; i++) {
      p(i * 10);                          // Invoke delegate instance
      System.Threading.Thread.Sleep(100); // Simulate hard work
    }
  }
}
class Test {
  static void WriteProgressToConsole(int percentComplete) {
    Console.WriteLine(percentComplete);
  }
  static void WriteProgressToFile(int percentComplete) {
    System.IO.File.WriteAllText("progress.txt",
      percentComplete.ToString());
  }
  static void Main() {
    ProgressReporter p = WriteProgressToConsole;
    p += WriteProgressToFile;
    Util.HardWork(p);
  }
}
```

Delegate példa

```
delegate void MyDelegate(string s);

class MyClass
{
    public static void Hello(string s)
    {   Console.WriteLine(”  Hello, {0}!”, s);}
    public static void Goodbye(string s)
    {   Console.WriteLine(”  Goodbye, {0}!”, s); }
    public static void Main()
    {
        MyDelegate a, b, c, d;

        // a egy delegate objectum, amely
        // hivatkozik a Hello metódusra
        a = new MyDelegate(Hello);
        b = new MyDelegate(Goodbye);
        c = a + b;
        d = c - a;

        a(”A”); // az a delegate meghívása
        b(”B”);
        c(”C”);
        d(”D”);
    }
}
```

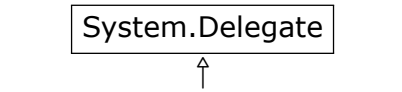
```
Output:

  Hello, A!
  Goodbye, B!
  Hello, C!
  Goodbye, C!
  Goodbye, D!
```

Delegatek, mi történik a háttérben?

- egy delegate definició:

```
public delegate int BinaryOp(int x, int y);
```

- delegate base osztályok:

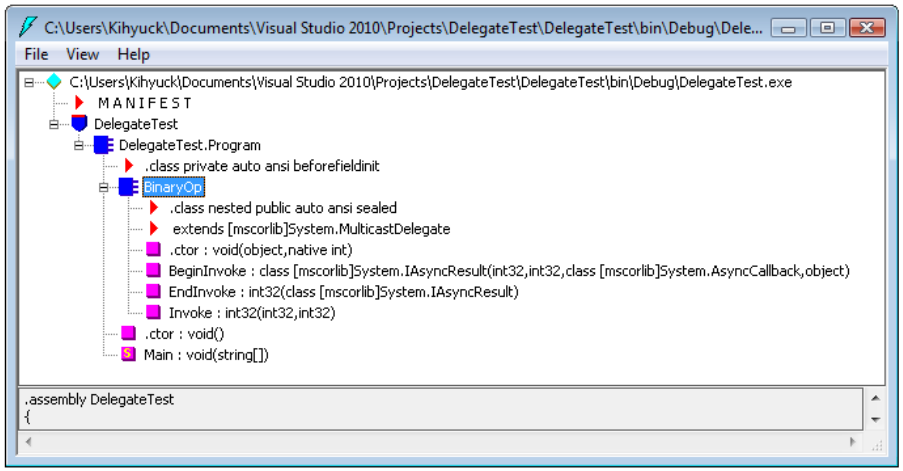


típusok rejtett hívására épül

- a fordító generál egy BinaryOp sealed osztályt (approx):

```
sealed class BinaryOp : System.MulticastDelegate {
 public BinaryOp(object target, uint functionAddress);
 public int Invoke(int x, int y);
 ...
}
```

- ildasm.exe



The System.MulticastDelegate and System.Delegate Base Classes

- System.MulticastDelegate/System.Delegate tagok:

Member	Meaning
Method	Reflection.MethodInfo típust ad vissza. A delegate által kezelt statikus metódusokról tartalmaz információt.
Target	a delegate által kezelt függvényt reprezentálja.
Combine()	A delegate listájához ad hozzá egy metódust. Kiválthatjuk a += túlterhelt operátorral.
GetInvocationList()	A System.Delegate tipusok tömbjét adja vissza, amelyek mindegyike egy adott, meghívható metódust reprezentál.
Remove() RemoveAll()	Eltávolítanak egy metódust a hívási listából. Kiválthatjuk a -= túlterhelt operátorral.

Generikus delegatek készítése

- egy delegate típus tartalmazhat generikus típus paramétereket

```
public delegate void MyGenericDelegate<T> (T arg);
// ez a generikus delegate bármely metódust meghívhatja, amely void értékkel tér
     vissza  és egyetlen bemeno paramétere van
...
static void Main(string[] args)
{
    //példányosításkor meg kell adnunk a típusparaméter értékét és annak a metódusnak
         a nevét, amelyet a delegate hívni fog
    MyGenericDelegate<string> strTarget = new MyGenericDelegate<string>(StringTarget);
    strTarget(”Some string data”);

    MyGenericDelegate<int> intTarget =  new MyGenericDelegate<int>(IntTarget);
    intTarget(9);
}
    static void StringTarget(string arg)
    {
        Console.WriteLine(”arg in uppercase is: {0}”, arg.ToUpper());
    }

    static void IntTarget(int arg)
    {
        Console.WriteLine(”++arg is: {0}”, ++arg);
    }
...
```

Delegatek kompatibilitása

- két delegate szerkezetileg nem egyenlő, még akkor sem ha a szignatúrájuk megegyezik:

```
delegate void D1();
delegate void D2();
...
D1 d1 = Method1;
D2 d2 = d1;                    // Fordításideju hiba
D2 d2 = new D2(d1);            // OK
```

- két delegate egyenlő, ha mindkettő értéke null, illetve ha a híváslistájukon ugyanazon objektumok ugyanazon metódusai szerepelnek

```
public delegate void TestDelegate();
static public void Method1() { }
static public void Method2() { }
static void Main(string[] args)
{
TestDelegate t1 = null;
TestDelegate t2 = null;
Console.WriteLine(t1 == t2); // True
t1 = Program.Method1;
t2 = Program.Method1;
Console.WriteLine(t1 == t2); // True
t1 += Program.Method2;
Console.WriteLine(t1 == t2); // False
...
```

A `Func<T>` és `Action<T>` generikus kifejezések

- új delegate típusok létrehozása helyett használhatjuk a `Func<T>` és `Action<T>` delegateket
- generikus kifejezések
- a `Func` adhat visszatérési értéket és az `Action` nem

```
static void Main(string[] args)
{
Func<int, int> func = (x) => (x * x); // int 1 paraméter, int 2 visszatérési érték
//a generikus kifejezésnek egy lambda kifejezést adtunk, amely nem igényli egy elozoleg
     definiált delegate jelenlétét
Console.WriteLine(func(10)); // 100
Console.ReadKey();
}
```

```
Func<int, int, bool> func = (x, y) => (x > y);  Console.WriteLine(func(10, 5)); // True
```

```
Func<bool> func = () => true; Console.WriteLine(func()); // True
```

- a `Func` minden esetben rendelkezik legalább egy paraméterrel, mégpedig a visszatérési érték típusával
- a visszatérési érték mindig (balról jobbra) utolsó helyen, előtte pedig a bemenő paraméterek (max. 4) kapnak helyet

A `Func<T>` és `Action<T>` generikus kifejezések -2

- A `Func<T>` párja az `Action<T>`, amely szintén maximum négy bemenő paramétert kaphat, de nem lehet visszatérési értéke

```
static void Main(string[] args)
{
    Action<int> act = (x) => Console.WriteLine(x);
    act(10);
}
```

Predicate delegate

- a predicate egy metódus, amely `true`-t vagy `false`-t térít vissza
- a predicate delegate egy referencia egy predicate-re
- hasznos: értéklisták szűrésére

```
public class PredicateDelegate
{
    static void Main()
    {
        List<int> list = new List<int> { 4, 2, 3, 0, 6, 7, 1, 9 };

        Predicate<int> predicate = greaterThanThree;

        List<int> list2 = list.FindAll(predicate);

        foreach ( int i in list2)
        {
            Console.WriteLine(i);
        }
    }

    static bool greaterThanThree(int x)
    {
        return x > 3;
    }
}
```

Névtelen metódusok a C# -ban

- névtelen metódusok **menet közben** adhatóak meg

```
public class Program {
  public static void Main() {
    Func<String, String> beupcase = delegate(string s) {
      char [] c = s.ToCharArray();
      c[0] = char.ToUpper(c[0]);                    //átalakítani az elso
      c[c.Length-1] = char.ToUpper(c[c.Length-1]); //és az utolsó char-t
      return new string(c);                                 // nagybetussé
    };                                                      // ; a } után
    Console.WriteLine(beupcase("sUn")); // SUN
  }
}
```

- ha ugyanarra a funkcióra többször is szükségünk van akkor ne használjunk névtelen metódusokat

Lambda kifejezés - Lambda Expressions

- egy lambda kifejezés gyakorlatilag egy névtelen metódus (amit egy delegate példány helyett írunk)

```
delegate int Transformer(int i);
Transformer sqr = x => x * x;
Console.WriteLine(sqr(3));  // 9
```

- általános formája

`(paraméterek) => kifejezés-vagy-utasítás-blokk`

- az operátor bal oldalán a bemenő változók, jobb oldalán pedig a bemenetre alkalmazott kifejezés áll
- a lambdák több sorra is kinyúlhatnak

```
Func<string, string> lambda = param => {
  param += ” and this was added to the string.”;
  return param;
};
```

Explicit meghatározható a lambda paraméter típusát

- a fordító képes megállapítani a lambda paraméter típusát:

```
Func<int,int> sqr = x => x * x; //x int-ként van kikövetkeztetve
```

- explicit módon is megadhatjuk az x típusát a köv. képpen:

```
Func<int,int> sqr = (int x) => x * x;
```

Lambdák: **Külső/fogoly** változók elérése

- egy lambda kifejezésben hivatkozhatunk annak a metódusnak a paramétereire és lokális változóira, amelyben definiáltuk

```
static void Main() {
  int factor = 2;
  Func<int, int> multiplier = n => n * factor;
  Console.WriteLine(multiplier(3));  // 6
}
```

- a `factor` itt a lambda kifejezés **külső/fogoly változó**-ként ismert
- a külső változók akkor értékelődnek ki, amikor a delegate ténylegesen **meghívódik**, nem pedig a deklaráláskor, vagyis az adott változó legutolsó értékadása számít

```
int factor = 2;
Func<int, int> multiplier = n => n * factor;
factor = 10;
Console.WriteLine(multiplier(3));  // 30
```

- a névtelen metódus nem éri el a definiáló metódus **ref** vagy **out** paramátereit

Lambdák: **Külső/fogoly** változók elérése -2

- lambda kifejezések módosíthatják a fogoly változókat

```
int seed = 0;
Func<int> natural = () => seed++;
Console.WriteLine(natural());   // 0
Console.WriteLine(natural());   // 1
Console.WriteLine(seed);        // 2
```

- a szokásos módon a `seed` lokális változó hatálya eltűnik ha a Natural befejeződik. De mivel a `seed` fogoly lett, az élettartama kibővül.

```
static Func<int> Natural() {
  int seed = 0;
  return () => seed++;
}
static void Main() {
  Func<int> natural = Natural();
  Console.WriteLine(natural()); // 0
  Console.WriteLine(natural()); // 1
}
```

Lambdák: lokális és iterációs változók

- egy lambda kifejezésen belül létrehozott változó egyedi a delegate példány meghívásnál

```
static Func<int> Natural() {
  return() => { int seed = 0; return seed++; };
}
static void Main() {
  Func<int> natural = Natural();
  Console.WriteLine(natural());     //0
  Console.WriteLine(natural());     //0
}
```

- az iteraciós változók a for és a foreach utasításokban: C# úgy kezeli őket mintha a cikluson **kívül** lettek volna deklarálva

```
Action[] actions = new Action[3];
for (int i = 0; i < 3; i++)
  actions[i] = () => Console.Write(i);

foreach (Action a in actions)
  a();  //333
```

A C# események

A C# események

- esemény: a program által kezelhető, lereagálható interakciók eseményeknek
  - valaminek a bekövetkezéséről tájékoztat
  - egy lehetősége az osztálynak arra, hogy értesítse egy objektum felhasználóit a saját állapota megváltozásakor: **kivált egy eseményt**
  - példák:
    - Tick – a Timer komponensnél lejárt az előre beállított időtartam
    - Click – a felhasználó kattintott pl. egy nyomógombon
    - MouseMove – egér mozgatása
    - Changed – az állományrendszer megfigyelt részében változás állt be
    - definiálhatunk saját eseményeket is
  - reagálhatunk az eseményre ún. eseménykezelő metódusok végrehajtásával
  - megvalósítás: metódusreferenciák segítségével
  - használatukkal időt takaríthatunk meg

A C# = Eseményvezérelt programozás

- C# -ban bármelyik objektum közzétehet eseményeket, amelyekre más alkalmazások is feliratkozhatnak. Ha a közzétevő osztályban kiváltódik az esemény, minden feliratkozott alkalmazás értesítést kap
- minden amit csinálunk egy esemény
  - C# = Eseményvezérelt programozás

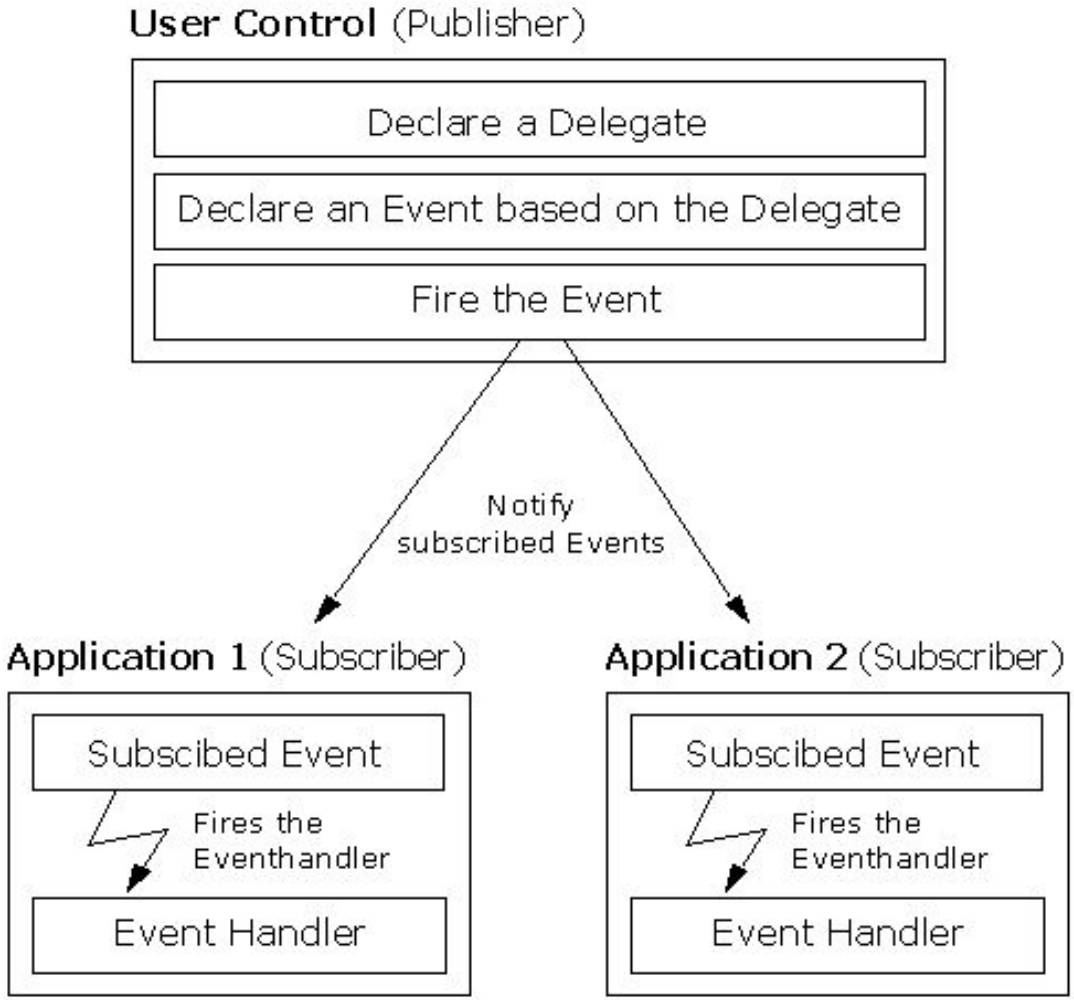
A közzétevő - feliratkozó (Publisher-Subscriber) modell

- az objektum amelyik előidéz egy eseményt: **közzétevő** vagy **küldő**
- más osztályok tudnak ezekre az eseményekre **hallgatni** vagy **feliratkozni** és végrehajtanak vmilyen kódot, amikor az esemény bekövetkezik. Regisztrálja magát a közzétevőnél.
- a közzétevő kontrollálja, hogy **mikor** váltódjon ki az esemény
- feliratkozó kontrollálja **mit** /milyen intézkedéseket kíván hozni válaszul
- az intézkedés egy metódusban van definiálva ún. **eseménykezelő** -vel (**event handler**)
- egy eseményre többen is feliratkozhatnak, de az is előfordulhat, hogy nincs érdeklődő
- C# -ban az események a közzétevő osztály tagjai

```
window.ClickedEvent += MyEventHandler;
```

  - `MyEventHandler`: a feliratkozó
  - `ClickedEvent`: a közzétevő osztályában (window) van definiálva

A közzétevő - feliratkozó (Publisher-Subscriber) modell



Megvalósítás

- 1. Metódusreferencia típus definiálása (az esemény bekövetkeztekor meghívandó metódusok listáját tartalmazza)

```
public delegate void MTípusNév(paraméterlista); // a visszatérési érték
      típusa void kell legyen
```

- 2. A metódusreferencia típuson alapuló esemény definiálása

```
public event MTípusNév EseményNév; // mindig public kell legyen
```

- 3. Feliratkozás az eseményre
  - először létrehozunk egy metódusreferencia objektumot benne megadva a hivatkozott metódust, majd ezt hozzárendeljük az eseményhez

```
EseményNév += new MTípusNév(objnév.esemkeznév)
// vagy:
EseményNév += objnév.esemkeznév
// vagy:
EseményNév += osztnév.esemkeznév
```

  - egy eseményre több metódus is feliratkozhat, valamint a metódus lehet az eseményt előidéző objektum osztályának tagja, egy másik objektum tagja vagy egy másik osztály statikus metódusa

Megvalósítás -2

- 4. Esemény előidézése

```
if(EseményNév != null) //ha van feliratkozó
   EseményNév(this);
```

  - meghívja sorban az egyes eseménykezelőket
  - a sorrend nem biztos
  - ha közben kivétel keletkezik, akkor a még „várakozó” eseménykezelők meghívása elmaradhat

Példa - Diák értesíti szüleit, ha átmegy (:

- Közzétevő:
  - definiálja az eventet
  - kontrolálja mikor váltódjon ki
  - meghívja a feliratkozott metódusokat

```
public delegate void
     NotifyDelegate(int Mark);
class Student
{
 public event NotifyDelegate
      NotifyToParent;
 public String Name { get; set; }
 public int Mark { get; set; }
 public void RecordMark()
{
     if (Mark> 4 &&
          NotifyToParent != null)
        NotifyToParent(Mark);
}
```

```
class Parent //Feliratkozó
{
  public String Name { get; set; }
  public void OnNotifyMe(int pMark)
  { //mit kíván tenni
    Console.WriteLine((”{0} notified
         about Mark {1}”, Name, pMark.
         ToString());
}}
class Program
{
 static void Main(string[] args)
 {
   Student oStudent = new Student();
   oStudent.Name = ”James”;
   oStudent.Mark = 8;

   Parent oParent = new Parent();
   oParent.Name = ”Daddy Cool”;

   oStudent.NotifyToParent += oParent.
        OnNotifyMe;
   oStudent.RecordMark();}}
```

Esemény = speciális delegate

- egy esemény tulajdonképpen egy speciális delegate
- különbségek:
  - az esemény az osztály egy tagja, a delegate típus definiálása történhet az osztályon kívül
  - esemény lehet része interfésznek, míg delegate nem
  - egy eseményt csakis az az osztály „hívhat” meg, amely deklarálta
  - egy esemény rendelkezik add és remove „metódusokkal”, amelyek felülbírálhatóak

Mi a baj a delegate-ekkel, miért events?

- a nyílvános delegate tagok megtörik az egységbe zárást, potenciális biztonsági réseket nyitnak

```
using System;

class Window {
  public delegate void ClickedEventDelegate();
  public ClickedEventDelegate ClickedEvent; //public!
  //...
}

class Program {
  public static void WindowClickHandler() {
    Console.WriteLine(”window click handled by benign click handler”);
  }
  public static void Main() {
    Window window = new Window();
    window.ClickedEvent += Program.WindowClickHandler;
    window.ClickedEvent();  //(public) delegate tagok közvetlenül
        meghívhatóak
    window.ClickedEvent = null; //(public), ezért módosíthatóak is és törölni
         lehet a delegate által meghívandó függvények listáját
  }
}
```

Mi a baj a delegate-ekkel, miért events? -2

```
using System;

class Window {
  public delegate void ClickedEventDelegate();
  public event ClickedEventDelegate ClickedEvent; //NB: event kulcsszó
  //...
}

class Program {
  public static void WindowClickHandler() {
    Console.WriteLine("window click handled by benign click handler");
  }
  public static void Main() {
    Window window = new Window();
    window.ClickedEvent += Program.WindowClickHandler; //ok
    window.ClickedEvent();                              //error
    window.ClickedEvent = null;                         //error
  }
}
```

Saját paraméterek átadása

- Első paraméter konvencionálisan a közzétevő objektum: `object sender`, mert egy eseménykezelő több objektum eseményeire is figyelhet
- A további paramétereket egy objektum adattagjaiként kell átadni, ennek osztálya az `EventArgs` leszármazottja kell legyen
- Létre kell hozni hozzá egy osztályt

- lépés 1: Hozzunk létre egy EventArgs osztályt

```
public class StartEventArgs : System.EventArgs {
    // konstruktor, mezok...
}
```

- lépés 2: Hozzunk létre egy delegatet

```
public delegate void StartEventHandler(object sender, StartEventArgs e);
```

- lépés 3 - Hozzunk létre egy event-et

```
public class Sender {
    public event StartEventHandler BeforeStart;
    public event StartEventHandler AfterStart;
    //...
}
```

- lépés 4 Hozzunk létre OnEvent metódusokat

```
protected virtual void OnBeforeStart(StartEventArgs e) {
  if (BeforeStart != null) BeforeStart(this, e);
}
// meghívás
OnBeforeStart(this, new StartEventArgs());
```

- NB: Nem feltétlenül kell delegate–et deklarálnunk, mivel rendelkezésünkre áll a beépített általános EventHandler delegate, amely két paraméterrel rendelkezik, és void visszatérési típussal bír.

Indexelők, operátorok és mutatók

Az indexelő metódus

- lehetővé teszi olyan egyedi típusok létrehozását, melyek tömbszerű szintaxissal biztosítanak hozzáférést a belső típusokhoz
- az indexelők hasonlóak a tulajdonságokhoz, azzal a különbséggel, hogy nem névvel, hanem egy index-el férünk hozzá az adott információhoz
- pl. a string osztálynak van indexelője:

```
string s = ”hello”;
Console.WriteLine (s[0]); // ’h’
Console.WriteLine (s[3]); // ’l’
```

- indexelő előállítása: `this` kulcsszóval, megadva a paramétert [] zárójelek között

```
public class PeopleCollection {
  private ArrayList people = new ArraList();
  public Person this[int index] {  //indexelo metódus
    get { return people[index]; } // visszaadja a megfelelo obj-ot
    set { people.Insert(index, value); } // a bejövo obj-ot elhelyezi
         az index helyére
  }
}
```

Az indexelő metódus -2

- objektumok létrehozása indexelő szintaxissal

```
PeopleCollection myPeople = new PeopleCollection();
myPeople[0] = new Person(”Homer”, ”Simpson”, 40);
myPeople[1] = new Person(”Marge”, ”Simpson”, 38);
myPeople[2] = new Person(”Lisa”, ”Simpson”, 9);
...
Console.WriteLine(”Age: {0}”, myPeople[2].Age );
```

- az indexelő paramétere bármilyen típusú lehet, így bármilyen object úgy indexelhető, mint egy tömb

```
Dictionary<string,string> hunglish = new Dictionary<string, string>();
hunglish[”alma”] = ”apple”;
Console.WriteLine(hunglish[”alma”]);
```

- Többdimenziós indexelők

```
public class SomeContainer {
  private int[,] my2DintArray = new int[10, 10];
  public int this[int row, int column]   { // get - set 2D tömben
  //...
  }
}
```

Az indexelő metódus -3

- Indexelő definíciók interfésztípusokon

```
public interface IStringContainer<KeyType> {
  string this[KeyType key] { get; set; }   //aki megvalósítja az kell
}                                          //definiálja a get és set-et

class MyStrings : IStringContainer<int> {
  string[] strings = { ”First”, ”Second” };
  public string this[int Key] {
    get { return strings[Key]; }
    set { strings[Key] = value; }
  }
}
```

Az operátor-túlterhelés

```
int a =100; int b = 240; int c = a+b;
string s1 = ”Hello”; string s2 = ”world”; string s3= s1+s2;
// Point p3 = p1 + p2????
```

C# Operátor	Túlterhelhetőség
+, -, !, ∼, ++, --, true, false	Ezek az unáris operátorok túlterhelhetőek
+, -, *, /, %, &, |, ^, <<, >>	Ezek a bináris operátorok túlterhelhetőek
==, !=, <, >, <=, >=	Túlterhelhetőek
[]	Nem terhelhető túl. Az indexelő konstrukció ugyanezt a funkcionalitást biztosítja
()	Nem terhelhető túl. Az egyedi konverziós metódusok ugyanezt a funkcionalitást biztosítják
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	A rövidített értékadó operátorok nem terhelhetők túl, viszont a kapcsolodó operátorok túlterhelésekor ezeket automatikusan megkapjuk.

Túlterhelt bináris operátor, példa

```
class Complex {
  double re, im;

  public Complex(double re, double im) {
    this.re = re;
    this.im = im;
  }
  public static Complex operator + (Complex a, Complex b) {
    return new Complex(a.re+b.re, a.im+b.im);
  }
  public override string ToString() {
    return(String.Format("{0} + {1}i", re, im));
  }
}
class Program {
  public static void Main() {
    Complex a = new Complex(1, 2);
    Complex b = new Complex(3, 5);
    Complex c = a + b;
    c += b;                                  //automatikus
    Console.WriteLine(c);
  }
}
```

A túlterhelés szabályai

- A metódus neve az `operator` kulcsszóval van specifikálva, amit követ az operátor szimbólum
- Az operátor függvény `public` és `static` kell legyen
- Az operátor függvény paraméterei reprezentálják az operandusokat
- Az operátor függvény visszatérítési típusa mutatja a kifejezés eredményét
- A paraméterek közül legalább az egyiknek meg kell egyeznie a definiáló osztály típusával

Egyedi implicit és explicit típuskonverziók

- emlékeztető:
  - numerikus konverzió

```
int a =123;
long b = a; //implicit int-rol long-ra, amikor kisebb típust
     akarunk elhelyezni nagyobb típusban
int c= (int) b; // explicit long-ról int-re, amikor nagyobb
     értéket akarunk kisebb tárolóban tárolni
```

  - konverzió egymásból származó osztálytípusok között

```
class Base{}
class Derived : BAse{}
...
Base myBaseType= new Derived();// imlicit kasztolás, mindig
     konvertálható az alaptípusra

Derived myDerivedType = (Derived)myBaseType; // explicit, ha az
     ososztálytípust tároljuk származtatott változóban
```

Egyedi implicit és explicit típuskonverziók -2

- `implicit` és `explicit` konverziók azok típuskonverziós operátorok
- szabályozzák, hogy a típusok hogyan reagáljanak a konverziós próbálkozásokra
- implicit konverzió: garantált a siker, nincs információ vesztés
- explicit konverzió: a konverzió sikeressége ismeretlen a futásig; információvesztéssel járhat

```
...
// Hertz-é konvertál
//a Note implicit módon konvertálható  double-á
public static implicit operator double (Note x) {
  return 440 * Math.Pow (2, (double) x.value / 12 );
}
// Hertz-bol konvertál
 //a double explicit módon konvertálható Note-á
public static explicit operator Note (double x) {
  return new Note ((int) (0.5 + 12 * (Math.Log (x/440) / Math.Log(2) ) ));
}
...
Note n = (Note)554.37;  // explicit konverzió
double x = n;           // implicit konverzió
```

Pointerek

Mutatótípusok használata - Pointers

- ritkán vagy soha nem lesz szükségünk rá
- lehetőség van, hogy a memóriakezelést a CLR megkerülésével kézben tartsuk
- mutatók használatához: adott metódust, osztályt, adattagot vagy blokkot az `unsafe` kulcsszóval kell jelölni
- értesíteni kell a fordítót a **nem felügyelt kód** használatáról: `csc /unsafe *.cs` // (parancssorból, Visual Studio esetén jobb klikk a projecten, Properties és ott állíthatjuk be)
- mikor használjuk pointereket C# -ban:
  - optimizálni akarjuk az alkalmazásunk egyes részeit
  - C-alapú DLL metódusokat hívunk meg, amelyek paraméterként pointert várnak

```
unsafe public static void UnsafeSwap(int* i, int* j) {
  int temp = *i;
  *i = *j;
  *j = temp;
}
```

- mutatócentrikus operátorok: `*`, `&`, `->`, `[]`, `++`, `--`, `+`, `-`, `==`, `!=`, `<`,

Mutatótípusok használata - Pointers -2

- a közvetlen mutatómanipulálás nem kötelező
- a cserealgoritmus felügyelt változata

```
public static void SafeSwap( ref int i, ref int j) {
  int temp = i;
  i = j;
  j = temp;
}
```

A C# előfordítói direktívái

A C# előfordítói direktívái

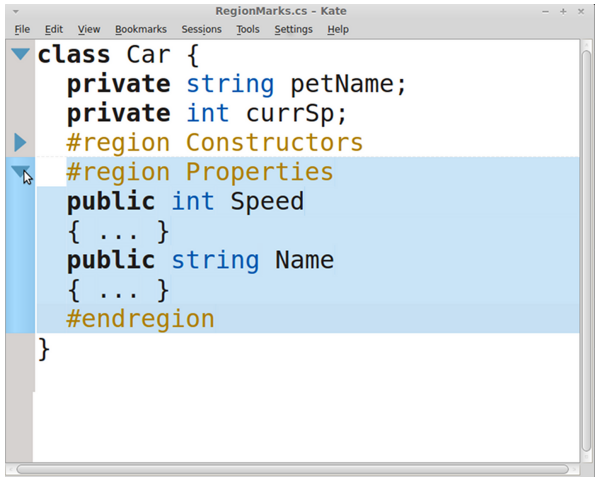
- C# -ban nincs **előfeldolgozási** lépés, az előfeldolgozó direktívák feldolgozása a fordító lexikaianalízis-fázisának része
- gyakori előfeldolgozó direktívák

Direktíva	**Jelentés**
`#region, #endregion`	az összecsukható forráskód jelölésére
`#define, #undef`	feltételes fordítási szimbolumok definiálására és visszavonására
`#if, #elif, #else, #endif`	a forráskód egyes szakaszainak feltételes átugrása
`#line`	alter the filename and line number information that is output by the compiler in warnings and error messages
`#pragma`	elfojtják vagy helyreállítják a warningokat

Kódrégiók megadása

- olyan kódblokkok, amelyet elrejthetünk egy szöveges jelölővel

```
class Car {
  private string petName;
  private int currSp;
  #region Constructors
  public Car()
  { ... }
  public Car (int currSp, string petName)
  { ... }
  #endregion
  #region Properties
  public int Speed
  { ... }
  public string Name
  { ... }
  #endregion
}
```



Feltételes kódfordítás

- hasonló mint C/C++ -ban
- egy kódblokk megjelölése, hogy csak hibakeresési konfiguráció alatt legyen lefordítva

```
#define DEBUG // definiáljuk a DEBUG szimbólumot
class Program {
  static void Main(string[] args) {
    #if DEBUG
    Console.WriteLine("App directory: {0}", Environment.CurrentDirectory);
    Console.WriteLine("Box: {0}", Environment.MachineName);
    Console.WriteLine("OS: {0}", Environment.OSVersion);
    Console.WriteLine(".NET Version: {0}", Environment.Version);
    #endif
  }
}
//...
App directory: /home/.../net/04-c#/code
Box: virtualia
OS: Unix 3.5.0.17
NET Version: 4.0.30319.1
```

- `#define` szimbólumok definícióját mindig a forrásfile elején kell megtennünk, ellenkező esetben a program nem fordul le.

Pragmák használata

- a “nem használt mezők” warning letiltása és visszaállítása
- ha nem adjuk meg a warning számát akkor az összeset letiltja és visszaállítja

```
#pragma warning disable 169
public class MyClass {
  int neverUsedField;
}
#pragma warning restore 169
```

Állítások - Assertions

- assertion: fejlesztés során használt kód, amely lehetővé teszi egy programnak, hogy ellenőrizze magát futás közben
- az állítás igaz => minden az elvártaknak megfelelően működik
- az állítás hamis => egy váratlan bug-ot jelez a kódban
- használat `Debug.Assert (File.Exists (”data.txt”), ”File data.txt does not exist!”);)`

Szerződések - Contracts

- .NET 4.0 -től **code contract** funkció, `System.Diagnostics.Contracts`
- egy erősebb mechanizmus az állítások érvényesítésére
- lehetőség van előfeltételek, utófeltételek és invariánsok megadására

```
public static bool AddIfNotPresent<T> (IList<T> list, T item) {
  Contract.Requires(list != null);         // Elofeltétel
  Contract.Requires(!list.IsReadOnly);     // Elofeltétel
  Contract.Ensures(list.Contains(item));  // Utófeltétel, a metódus
       végrehajtása után kell teljesülnie
  if (list.Contains(item)) return false;
  list.Add (item);
  return true;
}
```

Szerződések - Contracts -2

- az előfeltétel és az utófeltétel a metódus legelején kell szerepeljen
- az előbbi metódus feltételei a **contract** alapján:
  "Egy nem null írható listával kell meghívj."
  "Mikor visszatérek, a lista tartalmazni fogja az általad specifikált elemeket."
- ezeket a tényeket kiírhatjuk az assembly-nak egy XML dokumentum file-ába
- a szerződések lehetőséget biztosítanak a kódunk statikus elemzésére
- egy nagyon erős szempont: **interfészekhez** is lehet megadni contract-ot. Egy külön contract osztályt kell készíteni. (attribútumokkal kapcsoljuk össze- később)
- egy szerződés megszegése jobban testre szabható, mint a hagyományos állítások vagy kivételek
- hátrányok:
  - lassabb fordítási folyamat
  - futási teljesítmény csökkenés (bár szerződés ellenőrzés letiltható)

Attribútumok, Dinamikus kötés

Attribútumok - Attributes

- **attribútum** itt **nem** az objektumok egy-egy tulajdonságára utal
- az attribútum egy bővíthető mechanizmus, hozzákapcsolhatunk egyéni információkat a programhoz
- szögletes zárójelek közt adjuk meg
- példa:
  - tesztelés során általánosan használt attribútum a `Conditional`, amely egy előfordító által definiált szimbólumhoz köti programrészek végrehajtását

```
#define DEBUG // definiáljuk a DEBUG szimbólumot
 using System;
 using System.Diagnostics; // ez is kell

  class DebugClass
   {
     [Conditional(”DEBUG”)] // ha a DEBUG létezik
     static public void DebugMessage(string message)
     {
     Console.WriteLine(”Debugger üzenet: {0}”, message);
      }
   }
...
```

Attribútumok - Attributes -2

- megadható:
  - milyen entitásokhoz legyen kapcsolható (pl. csak osztályhoz, vagy interfészhez)

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.
    Interface)]
public class SimpleAttribute: System.Attribute{}
```

  - az attribútum milyen paramétereket vár
  - mik legyenek a metódusai
- olyan osztályok definiálhatják amelyek a `System.Attribute` absztrakt osztályból származnak
- mi magunk is készíthetünk attribútumokat

```
class TestAttribute : System.Attribute { }

[TestAttribute]  //konvenció alapján írhattunk volna csak Test-et
 class C { }
```

Attributes, example

```
[Serializable] // az osztály szerializálható lesz
public class MyObject {
  public int n1 = 0;
  public int n2 = 0;
  public String str = null;
}

//--- szerializálás ---
MyObject obj = new MyObject();
obj.n1 = 1;
obj.n2 = 24;
obj.str = ”Some String”;
IFormatter formatter = new BinaryFormatter();
Stream stream = new FileStream(”MyFile.bin”, FileMode.Create, FileAccess.Write, FileShare.None);
formatter.Serialize(stream, obj);
stream.Close();

//--- deszerializálás ---
IFormatter formatter = new BinaryFormatter();
Stream stream = new FileStream(”MyFile.bin”, FileMode.Open, FileAccess.Read, FileShare.Read);
MyObject obj = (MyObject) formatter.Deserialize(stream);
stream.Close();

Console.WriteLine(”n1: {0}”, obj.n1);
Console.WriteLine(”n2: {0}”, obj.n2);
Console.WriteLine(”str: {0}”, obj.str);
```

(Szerializálás $\equiv$ a memóriabeli objektumainkat egy olyan szekvenciális formátumba (pl. bináris, vagy XML) konvertáljuk, amelyből vissza tudjuk alakítani az adatainkat. A kiírt adatokat megfeleltetjük egy osztálynak, szerializálva kiírhatjuk, és onnan vissza is olvashatjuk (deszerializálás)

Dinamikus típusok

Dinamikus típusok

- `dynamic` kulcsszóval(C# 4-től) dinamikusan típusossá tehetünk objektumokat
- fordítási időben nem ismert egy változó vagy kifejezés típusa, nem dönthető el, hogy milyen metódusokkal rendelkezik, azok milyen paramétereket fogadhatnak
- `dynamic`-al jelölt objektum bármit megtehet fordítási időben, még olyan dolgokat is, amelyek futásidejű hibát okozhatnának
- valójában object típusúak (lásd IL az kód). Bármilyen típusú értéket megkaphat, ugyanis az object minden osztály őse)
- a dynamic kulcsszó ugyan, azonban nem foglalt szó, így változónévként használható

```
dynamic x = 10;
Console.WriteLine(x); // x most 10

x = ”szalámi”;
 Console.WriteLine(x); // x most szalámi. Futásidoben
     megváltoztathatja a típusát is
```

Dinamikus kötés - Dynamic Binding

- **dinamikus kötés** ≡ elhalasztja a metódusok, operátorok és indexerek, mező- és tulajdonságelérők ténlyleges típusát fordítási időről, futásidőig
- mi tudjuk, hogy egy függvény, tag, operátor létezik, de a compilátor **nem**

```
dynamic d = GetSomeObject();
d.Quack();
```

- ha az adott metódus nem hívható meg az objektumon, futásidőben egy `RuntimeBinderException` típusú kivétel váltódik ki

```
dynamic d = 5;
d.Hello(); // RuntimeBinderException
```

- előnye:
  - jelentősen átláthatóbbá teszi a COM-mal, HTML DOM-mal együttműködő kódunkat,
  - a dinamikus típusalkalmazás akár egy nagyságrenddel is gyorsabb, mint a normál reflexió
- 2 típusa van: nyelvi kötés és egyedi kötés

Egyedi kötés - Custom Binding

- a kötés testreszabható ha implementáljuk a `IDynamicMetaObjectProvider` interfészt

```
using System;
using System.Dynamic;
public class Test
 {
  static void Main() {
    dynamic d = new Kacsa();
    d.Hápogás();
    d.Kacsázás();
  }
}

public class Kacsa : DynamicObject {
  public override bool TryInvokeMember (InvokeMemberBinder binder,
      object[] args, out object result) {
    Console.WriteLine (binder.Name + ” űnev metódus volt meghívva”);
    result = null;
    return true;
  }
}
```

- A Kacsa osztálynak nincs Hápogás, Kacsázás metódusa. Ehelyett testreszabható kötést használ, hogy elkapja és értelmezze azokat
- További megfontolandó metódusok : `TryGetMember, TrySetMember, TryUnaryOperation` etc

Nyelvi kötés - Language Binding

- nyelvi kötés $\equiv$ az objektum nem valósít meg semmilyen speciális interfészt
- előny: megkerülhetünk bizonyos típus korlátozásokat, pl: nem kell megismételni az alábbi kódot minden numerikus típusra

```
static dynamic Mean(dynamic x, dynamic y) {
    return (x+y)/2;
}
...
int x = 3, y = 4;
Console.WriteLine(Mean(x,y));
```

- **Figyelem:** elveszítjük a fordításidejű típusbiztonságot, óvatosan alkalmazzuk!