

# Haladó fejlesztési technikák

Követelmények

Delegate – ismételés

Delegate-ek modern használata

# Követelmények

- **Az előadás felvezeti majdnem minden héten a labort, ezért az előadásanyag elsajátítása a labor előtt kötelező**
  - Első hét kivételt képez
- **Érdemjegy**
  - 1 db labor ZH 9. héten (40 pont) → programozási feladat
  - 1 db előadás ZH 13. héten (40 pont) → teszt
  - 1 db féléves feladat teljesítése a 13. hét csütörtökig (20 pont)
- **Minden pótolható egy alkalommal a szorgalmi időszakban**
  - Pót labor ZH 14. héten
  - Pót előadás ZH 14. héten
  - Pót féléves leadás a 14. hét csütörtökig (2000 ft különjárási díj)
- **Évközi jegy pótló vizsga (4000 ft vizsgadíj)**
  - A 3 komponensből a sikerteleneket kell csak pótolni

# Féléves Feladat

- **4. oktatási héten kapja meg minden hallgató**

- Felkerül a tárgy weboldalára is ([nikprog.hu](http://nikprog.hu))
- A [@stud.uni-obuda.hu](mailto:@stud.uni-obuda.hu) email címre is elküldjük

- **Feladat**

- Az előadáson lesz pontosan ismertetve minden szükséges elvárás
- Hétről-hétre kell dolgozni rajta
- Mérföldköveket kell teljesíteni

# Delegate

- **Olyan típus, aminek változóiban metódusokat tudunk elhelyezni**
  - A **delegate** típusa meghatározza, hogy milyen szignatúrájú függvényt tehetünk bele

```
delegate double MyDelegate(int param1, string param2);
```

- A konkrét **delegate** **változóban** tárolhatjuk el a függvényeket (a háttérben: osztály=típus és példány+változó+lista)

```
double funct(int első, string második)
{
    return első + második.Length;
}
```

```
MyDelegate del = new MyDelegate(funct); //hosszú megadás
MyDelegate del = funct;                 //rövid megadás
```

- A delegátnak null értéke van, amíg nincs benne függvény

# Delegate használata

- A C#-os delegate multicast tulajdonságú, több függvény is lehet benne – függvény hozzáadása, eltávolítása:

```
del += new MyDelegate(Function1); //hosszú megadás  
del += Function1;                 //rövid megadás
```

```
del -= new MyDelegate(Function1); //hosszú megadás  
del -= Function1;                 //rövid megadás
```

- A delegate-ben lévő függvények hívása:

```
MyDelegate temp = del; //Az ideiglenes változó  
if (temp != null)      //szálbiztonság (thread safety)  
    temp(0, "alma");    //miatt kell.
```

```
del?.Invoke(0, "alma"); //Újabb szintaxis, ATOMI művelet
```

- A hívási sorrend a keretrendszer által nincs garantálva, ne építsünk rá!  
(.NET 4.5-ös állapot: abban a sorrendben hívja, amiben beleraktuk)
- Visszatérési érték használata esetén az utoljára hívódó metódus visszatérési értékét kapjuk meg

# Saját vs. beépített delegate típus

- **Delegate megadása:**

```
delegate double MyDelegate(int param1, string param2);
```

- „Olyan függvényt képes fogadni, aminek double visszatérési értéke van, és egy int és egy string paramétere”

- **Szinte soha nincs rá szükség, a keretrendszerben rengeteg a beépített delegate típus, használjuk inkább ezeket!**
- **A delegate-változó típusa így nem MyDelegate lesz, hanem valami olyan keretrendszeri osztály, ami rögzíti, hogy a változó milyen szignatúrájú metódusokat tud magában foglalni (eredmény + paraméter típusai)**

# Beépített delegate típusok

Predicate<T>	bool(T)	List<T>.Find(), .Exists(), RemoveAll()...
Comparison<T>	int(T1,T2)	List<T>.Sort(), Array.Sort()
MethodInvoker	void()	
EventHandler	void(object,EventArgs)	
EventHandler<T>	void(object,T) (T EventArgs utód)	
Action	void()	
Action<T>	void(T)	
Action<T1,T2>	void(T1,T2)	
Action<T1,T2,...,T16>	void(T1,T2,...,T16)	
Func<TRes>	TRes()	
Func<T, TRes>	TRes(T)	
Func<T1, T2, TRes>	TRes(T1,T2)	
Func<T1, T2, ... T16, TRes>	TRes(T1,T2,...,T16)	



# Delegate használata a gyakorlatban

- Sokszor paraméterként!

```
private bool ParosE(int i)
{
    return i % 2 == 0;
}
private int ParosakatElore(int i1, int i2)
{
    bool i1Paros = ParosE(i1);
    bool i2Paros = ParosE(i2);
    if (i1Paros && !i2Paros) return -1;
    else if (!i1Paros && i2Paros) return 1;
    else return 0;
}
```

```
int[] tomb; List<int> lista;
// ...
int elsoParos = lista.Find(ParosE);
List<int> osszesParos = lista.FindAll(ParosE);
bool vanParos = lista.Exists(ParosE);
Array.Sort(tomb, ParosakatElore);
```



# Az Array.Sort újra-felfedezése

```
delegate bool Osszehasonlito(object bal, object jobb);

class EgyszeruCseresRendezo
{
    public static void Sort(object[] tomb, Func<object, object, bool> nagyobb)
    public static void Rendez(object[] tomb, Osszehasonlito nagyobb)
    {
        for (int i = 0; i < tomb.Length; i++)
            for (int j = i + 1; j < tomb.Length; j++)
                if (nagyobb?.Invoke(tomb[j], tomb[i]))
                {
                    object ideiglenes = tomb[i];
                    tomb[i] = tomb[j];
                    tomb[j] = ideiglenes;
                }
    }
}
```

# Az Array.Sort újra-felfedezése

```
class Diak
```

```
{
```

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

```
    public int Kreditek { get; set; }
```

```
    public Diak(string nev, int kreditek)
```

```
    {
```

```
        this.Nev = nev; this.Kreditek = kreditek;
```

```
    }
```

```
}
```

```
Diak[] csoport = new Diak[] {  
    new Diak("Első Egon", 52),  
    new Diak("Második Miksa", 97),  
    new Diak("Harmadik Huba", 10),  
    new Diak("Negyedik Néró", 89),  
    new Diak("Ötödik Ödön", 69)
```

```
};
```

# Az Array.Sort újra-felfedezése

```
bool KreditSzerint(object első, object második)
```

```
{
```

```
    return ((első as diák).Kreditek <
            (második as diák).Kreditek);
```

```
}
```

```
EgyszeruCseresRendezo.Rendez(csoport, KreditSzerint);
```

# Esemény vs. delegate

- Delegate tagváltó megadása osztályban:

```
DelegateType változoNeve;
```

- Esemény megadása osztályban:

```
event DelegateType esemenyNeve;
```

➔ Az esemény csak egy event kulcsszóval ellátott egyszerű delegate! – az event kulcsszó célja a védelem

- A védelem miatt az event nagyon gyakran publikus

Delegate	Event
Bárhonnan meg lehet hívni	Csak a deklaráló osztályból lehet meghívni, „tüzelni”
Értékadással (=) felülírható	= operátor nem megengedett, += és -= van csak
Standard tulajdonság készíthető hozzá get és set kulcsszavakkal	Korlátozott képességű tulajdonság add (+=) és remove (-=) kulcsszavakkal
Nem szerepelhet interface-ben	Szerepelhet interface-ben

# Eseménykezelés – Névkonvenciók

Feladat	Név	Elhelyezés
<b>Eseményparaméter</b>	<b>...EventArgs</b> (PropertyChangedEventArgs)	<b>A névtérben vagy a kiváltó osztályban</b>
<b>Delegate</b>	<b>...EventHandler</b> (PropertyChangedEventHandler)	<b>A névtérben vagy a kiváltó osztályban</b>
<b>Esemény</b>	<b>...</b> (PropertyChanged)	<b>Kiváltó osztályban</b>
<b>Eseményt közvetlenül meghívó metódus</b>	<b>On...</b> (OnPropertyChanged)	<b>Kiváltó osztályban</b>
<b>Esemény lekezelése</b>	<b>---</b>	<b>Lekezelő osztályban</b>

# Névtelen függvények

- Delegate-ek fő használati módjai:
  - Események
  - Metódussal való paraméterezés
- Probléma: az egyszer használatos függvények „elszennyezik” a kódot, nehezebben érthető lesz
- Megoldás: névtelen függvények
- Az angol dokumentáció szerint: (<http://msdn.microsoft.com/en-us/library/bb882516.aspx>)
- Nem keverendő össze: local/inline functions, ami ROSSZ

Anonymous functions

anonymous methods ☺

lambda expressions



# Anonim függvények

```
int elseParos =  
    lista.Find(delegate(int i) { return i % 2 == 0; });  
List<int> osszesParos =  
    lista.FindAll(delegate(int i) { return i % 2 == 0; });  
bool vanParos =  
    lista.Exists(delegate(int i) { return i % 2 == 0; });  
  
Array.Sort(tomb,  
    delegate(int i1, int i2)  
    {  
        bool i1Paros = i1 % 2 == 0;  
        bool i2Paros = i2 % 2 == 0;  
        if (i1Paros && !i2Paros)  
            return -1;  
        else if (!i1Paros && i2Paros)  
            return 1;  
        else return 0;  
    });
```

- Ma már nem használjuk (-> lambdák)



# Lambda függvények

- **Új operátor:  $\Rightarrow$  (Lambda operátor)**
  - Bemenet és a kimenet összekötésére
  - “Ha a bemenet egy X nevű egész szám, akkor a kimenet...”
- **Szintaxis: paraméter[ek]  $\Rightarrow$  kimenetet meghatározó kifejezés**
- **Használata:**
  - delegate típus (saját v. keretrendszeri), ez tipikusan paraméter típusa

```
delegate double SingleParamMathOp(double x)  
Func<double, double>
```

- delegate változó elkészítése és függvény megadása lambda kifejezés formájában, metódus meghívása

```
SingleParamMathOp muvelet = x => x * x;  
double j = muvelet(5);
```

# Lambda kifejezések

```
delegate double TwoParamMathOp(double x, double y);
```

```
TwoParamMathOp myFunc = (x, y) => x + y;  
double j = myFunc(5, 10);    //j = 15
```

- **Beépített delegált típussal:**

```
Func<int, int> myFunc = (x) => x * x;  
int j = myFunc(5);    //j = 25  
Func<int, int, int> myFunc2 = (x, y) => x + y;  
int j2 = myFunc2(5, 10);    //j = 15
```

- **Több paramétert zárójelezni kell**
- **A paraméterek típusozása nem kötelező, csak speciális esetekben**

# Lambda kifejezések

```
int elsoParos =  
    lista.Find(i => i % 2 == 0);  
List<int> osszesParos =  
    lista.FindAll(i => i % 2 == 0);  
bool vanParos =  
    lista.Exists(i => i % 2 == 0);  
  
Array.Sort(tomb,  
    (i1, i2) =>  
    {  
        bool i1Paros = i1 % 2 == 0;  
        bool i2Paros = i2 % 2 == 0;  
        if (i1Paros && !i2Paros)  
            return -1;  
        else if (!i1Paros && i2Paros)  
            return 1;  
        else return 0;  
    });
```

# Lambda kifejezések

- **Altípusok:**

- **Kifejezéslambda (Expression Lambda)**

- Szigorúan egyetlen kifejezés a kimenetet meghatározó oldalon

$x \Rightarrow x * x$

- **Kijelentéslambda (Statement Lambda)**

- Akár több sorból álló kód a kimenetet meghatározó oldalon, változólétrehozás, .NET függvény hívása, return is megengedett

$x \Rightarrow \{ \text{Console.WriteLine}(x); \}$

- **Különbség:**

- Az kifejezéslambda adott helyeken (pl. adatbázis felé való kommunikáció) nem delegáltra fordul, hanem kifejezésfára (Expression Tree) – pl. adatbázisszerver tudja az SQL dialektusára való fordítás után végrehajtani

# Névtelen függvények és lambdák

- **Előny:**
  - A függvény azonnal olvasható a felhasználás helyén
  - Kevesebb „szemét” tagfüggvény az osztályban
- **Csak az egyszer használatos, és lehetőleg rövid függvényeket írjuk így meg:**
  - A hosszú kód olvashatatlan
  - Mivel „beágyazott kód”, ezért nem újrafelhasználható
- **Lehetőleg ne ágyazzunk egymásba névtelen metódusokat**
- **Óriási hibalehetőség: Outer Variable Trap**

# Outer Variable Trap

- Lambda kifejezések jobb oldalán felhasználhatók a külső változók (closure), ez mindig speciális figyelmet igényel:

```
Action szamkiro = null;
for (int i = 0; i < 10; i++)
{
    szamkiro += () => { Console.WriteLine(i); };
}
szamkiro();
```

- „Elvárt” kimenet: 0, 1, 2, 3, 4, 5, ...
- DE a külső változókat a függvény referencia formájában kapja meg – az érték típusúakat is!
- Valós kimenet: 10, 10, 10, 10, 10 ...
- Megoldás ideiglenes változó bevezetésével (amit sehol nem változtatunk később)

```
// ...
int f = i;
szamkiro += () => { Console.WriteLine(f); };
// ...
```

# Példa

- Naplózó alkalmazást akarunk írni, amelynél a Logger osztály nem tudja, hogy pontosan milyen naplózási módszerek léteznek (email, adatbázis, helyi OS eseménynapló, syslog-ng, ...)

```
class Logger
{
    private Action<string> logMethods;
    public void AddLogMethod(Action<string> logMethod)
    {
        logMethods += logMethod;
    }
    public void Log(string message)
    {
        // if (logMethods != null) ....
        logMethods?.Invoke($"[{DateTime.Now}] {message}");
    }
}
```



# Példa - Főprogram

```
static void ConsoleLog(string msg)
{
    Console.WriteLine(msg);
}

static void Main(string[] args)
{
    Logger log = new Logger();
    log.AddLogMethod(ConsoleLog);
    log.AddLogMethod(delegate (string msg) { Console.WriteLine(msg); });
    log.AddLogMethod(x => Console.WriteLine(x));
    log.AddLogMethod(x =>
    {
        using (StreamWriter writer
        {
            writer.WriteLine(x);
        }
    ));

    log.Log("Starting Apache");
    System.Threading.Thread.Sleep(1000);
    log.Log("Starting MySQL");
    System.Threading.Thread.Sleep(1000);
    log.Log("Starting ProFTPd");
    System.Threading.Thread.Sleep(1000);
    log.Log("Killing ProFTPd");
    log.Log("Stopping Apache");
}
```

# Bővítés: Szűrés


```
List<string> entries;  
public Logger()  
{  
    entries = new List<string>();  
    AddLogMethod(x => entries.Add(x));  
}
```

```
public List<string> Filter1(Func<string, bool> condition)  
{  
    List<string> output = new List<string>();  
    foreach (string akt in entries)  
    {  
        if (condition(akt)) output.Add(akt);  
    }  
    return output;  
}
```

# Reinvent the wheel???

```
public List<string> Filter2(Predicate<string> condition)
{
    return entries.FindAll(condition);
}
```

```
public List<string> Filter3(Func<string, bool> condition)
{
    return entries.Where(condition).ToList();
}
```

 (extension) `IEnumerable<string> IEnumerable<string>.Where<string>(Func<string, bool> predicate)` (+)  
Filters a sequence of values based on a predicate.

Returns:

An `IEnumerable<out T>` that contains elements from the input sequence that satisfy the condition.

Exceptions:

`ArgumentNullException`

# Szűrés - Főprogram

```
Console.WriteLine("Filtering...");
foreach (string akt in
{
    log.Filter(x => x.ToLower().Contains("apache")))
{
    Console.WriteLine(akt);
}
```

# Gyakorlat

- **Visszajelzéseket akarunk kezelni (Opinion, Bugreport, FeatureRequest)**
- **Mindegyik visszajelzés típushoz több fajta feldolgozó rutint akarok hozzárendelni**
- **Periodikusan minden tizedik (a gyakorlaton: harmadik) visszajelzés után hívjuk meg minden visszajelzésre a feldolgozó metódusokat**
- **A feldolgozó metódusokat egy Dictionary<Category, Action<Feedback>> adatszerkezetben akarjuk tárolni**

