

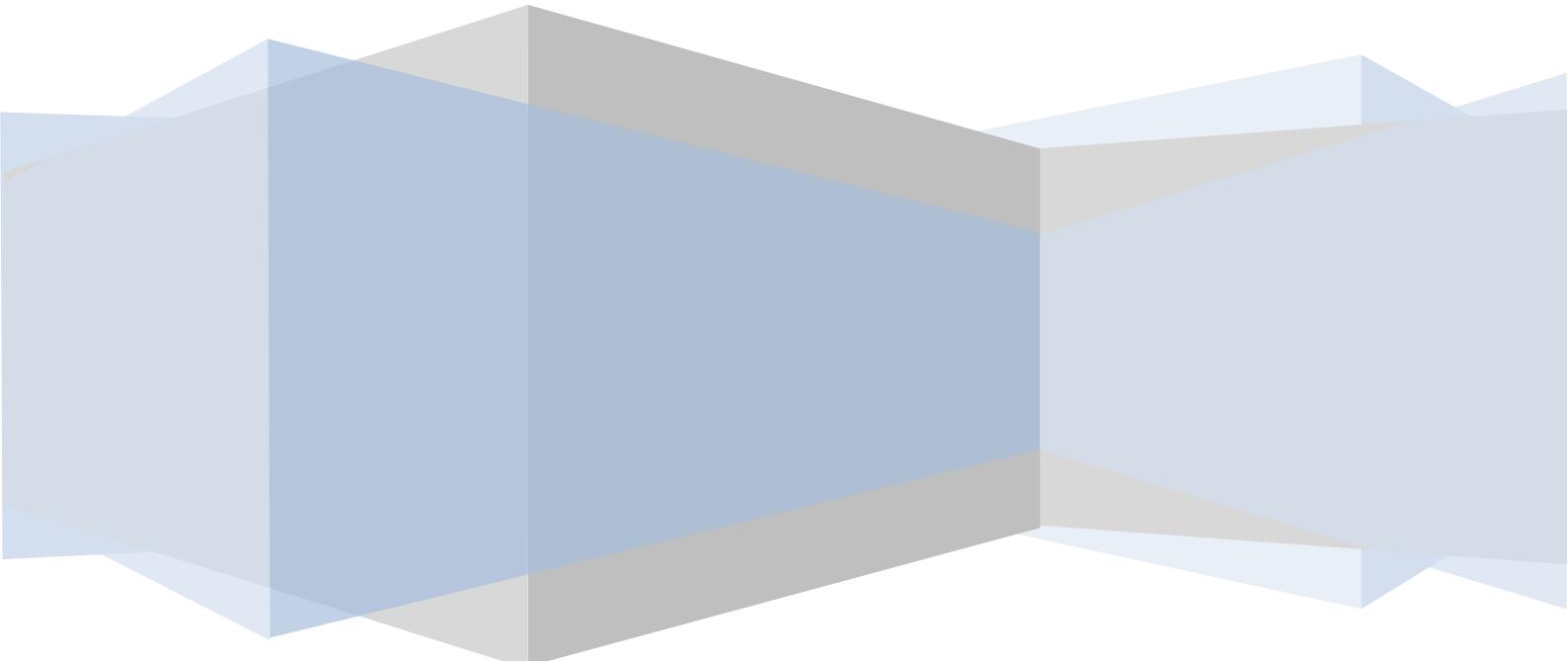
BME AAIT

Nyelvi eszközök 2

Szoftvertechnikák

Benedek Zoltán

2024.04.18.



Tartalom

Tartalom

Tartalom.....	2
var	3
Generics (generikus típusok)	3
Object-ként kezelés problémái	3
Generikus osztály példa	4
Generikus típusok jellemzői *	5
Mi lehet generikus?	5
.NET beépített generikus típusok.....	6
Gyakorlópélda	7
Generikus kényszerek	7
Lambda expression (lambda kifejezés)	9
Bevezető	9
Utasítás (statement) lambda	10
Kifejezés (expression) lambda	12
Egyéb lehetőségek	12
Beépített delegate típusok – Func generikus delegate (fontos).....	12
Beépített delegate típusok – Action és generikus Action delegate (fontos)	13
Lambda kifejezésekben a paraméterek megadásának szintaktikája	14
Irodalom.....	15
LINQ - Language-Integrated Query	15
Egyéb C# nyelvi eszközök.....	16
partial class (részleges osztály) – fontos!.....	16
Expression-bodied members (kifejeéstörzsű tagok)	17
Object initializer (objektuminicializáló)	17

var

Lokális változók esetében a típus megadása helyett használhatjuk a **var** kulcsszót, amennyiben a változót inicializáljuk. Példa:

```
var list = new Complex(10, 20);
```

A példában a `list` változó típusa `Complex`, a "`=`" jobb oldalából a fordító kikövetkezteti a típusát. A fenti kód ezzel ekvivalens:

```
Complex list = new Complex(10, 20);
```

Csak lokális változók esetében használható (tagváltozó, függvényparaméter esetén nem).

Generics (generikus típusok)

Generikus típusokkal már C++ és Java nyelv vonatkozásában is megismerkedtünk (C++ esetében az osztály és függvény sablonok tartoztak ide). A következőkben röviden áttekintjük alkalmazásának előnyeit, illetve a .NET/C# nyelvű specifikumait.

Generikus típusokat akkor használunk, ha nem szeretnénk előre megkötni, hogy milyen típusú adatokkal dolgozik egy osztály/függvény. Egy másik megközelítés az adatok `object`-ként kezelése. Először ez utóbbi megoldást tekintjük át, ezt követően térünk át a generikus típusok alkalmazására.

.NET-ben minden osztály/típus az `object` ősből származik (még az olyan egyszerű típusok is, mint pl. az `int`). Erre építve lehet általános, bármilyen típussal működő osztályokat (pl. `ArrayList` tároló), függvényeket stb. írni. Nézzünk egy minimalisztikus (hibakezelést mellőző) verem tároló megvalósítást:

```
public class Stack
{
    readonly int size;
    int current = 0;
    object[] items;

    public Stack(int size) {
        this.size = size;
        // Helyfoglalás a referenciaknak
        items = new object[size];
    }

    public void Push(object item) {
        ...
        items[current++] = item;
    }

    public object Pop() {
        ...
        return items[current--];
    }
}
```

```
Stack objectStack = new Stack();
objectStack.Push(1);
objectStack.Push("sss"); // lefordul ☺
int x = (int)objectStack.Pop();
```

Object-ként kezelés problémái

Az előző órán már alkalmazott `object` alapú `ArrayList` gyűjteménnyel demonstrálva:

1. Castolni kell (plusz kódot kell írni)

```
ArrayList list = new ArrayList();
```

```
list.Add( new Person() );
...
Person i = (Person)list[0];
```

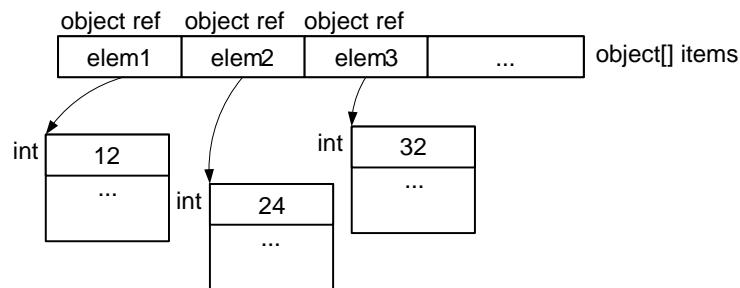
2. Csak futási időben derül ki, ha hiba van

```
ArrayList list = new ArrayList();
list.Add( new Person() );
...
Person i = (int)list[0];
```

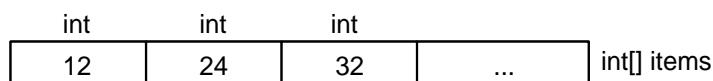
3. Nincs kikénszerítve, hogy ne keveredjenek a különböző típusú objektumok

```
ArrayList list = new ArrayList();
list.Add( new Person() );
list.Add( 12 ); // Ezt bizony elfogadja, de ritkán akarunk ílyet
```

4. Érték típusoknál (pl. egyszerű int) teljesítményromboló tényező a be- és kidobozolás (boxing/unboxing). Pl. a fenti Stack példában az elemek egy object[]-ben vannak tárolva, vagyis a tömb elemei referenciaiak/pointerek: ebbe pl. egy int változó úgy tud bekerülni, ha a .NET runtime egy heap-en allokált objektumba csomagolja (bedobozolás/boxing) az int értéket, és a tömbben erre tesz egy referenciát:



De ez a megoldás nagyságrenddel lassabb és helypazarlóbb, mintha egy egyszerű int[] items tömbbe egymás mellé bekerülnének az int elemek (ahogy pl. C/C++-ban megszoktuk):



A fenti problémákra (1-4) a megoldást a generikus típusok alkalmazása jelenti .NET-ben.

Generikus osztály példa

Alakítsuk át a korábbi, object alapú Stack osztályt generikussá. Vagyis olyan típussá, melyben az elemek típusa nem kötött, hanem **az osztály felhasználása során lehet megadni**. A szintaktika jóval letisztultabb, mint C++-ban, a generikus/sablonparaméterek bevezetése során nem kell a template kulcsszót használni, egyszerűen csak az osztály neve után vesszővel elválasztva fel kell a generikus paramétereket sorolni (esetünkben csak egy sablonparaméter van):

```
// Ahol eddig object volt, T-t írunk az
// osztályban
public class Stack<T>
{
    readonly int size;
    int current = 0;
    [] items;

    public Stack(int size) {
        this.size = size;
```

```
        items = new T[size];
    }
    public void Push(T item) {
        items[current++] = item;
    }
    public T Pop() {
        return items[current--];
    }
}
```

```
// A használat során meg kell adni
// a generikus paraméterek típusát
Stack<int> intStack = new Stack<int>();
intStack.Push(1);
int i = intStack.Pop();
// Az alábbi fordítási hibát okoz 😞
intStack.Push("ss");
```

A bal oldalon a **türkizzel** kiemelt helyen bevezetjük, a **zölddel** kiemelt helyeken felhasználjuk a generikus paramétert. Jelen példában ez triviális, de ha egy összetettebb példát látunk, és „beleveszünk” a paraméterek értelmezésébe, akkor ez a kettéválasztás minden legyen meg a fejünkben (ezt a színezést a későbbiekben is alkalmazzuk majd néhány helyen).

A generikus paramétereknek tetszőleges név adható, a példánkban a `T` helyett használhattuk volna a pl. a beszédesebb, de hosszabb `ItemType` nevet is.

Ez egy „type-safe” (típusbiztos) megoldás, a korábban említett problémák nem jelentkeznek.

Generikus típusok jellemzői *

A **C++ sablonok**, önmagukban le se fordulnak a fordítás/build során. A felhasználás során fejtődnek ki, minden sablonparaméter-kombinációra külön kód generálódik. Ennek három kellemetlenebb következménye is van:

- Ha nem használunk egy sablont, ki sem derülnek bizonyos hibák a fordítás során.
- Kódburjánzs (code bloat) veszély, ha sok különböző paraméter típussal használjuk.
- A sablon forráskódja a felhasználás során a fordításkor rendelkezésre kell álljon! A forráskód védelme nem megoldott! 😟

Ezzel szemben **.NET-ben a generikus típusok lefordulnak a build során IL (köztes) kódra**. És bár az érték típusok esetében a fordító különböző típusok esetén külön kódot generál, a **referencia típusok (osztályok)** esetében csak egy közös kód generálódik, így a kódburjánzástól gyakorlatilag nem kell tartani.

Mi lehet generikus?

.NET-ben generikus lehet: osztály, művelet (tagfügvény), struktúra, interfész, delegate. Osztályra már láttunk példát, nézzünk most a többire is.

Generikus struktúra

```
struct Point<T>
{
    public T X;
    public T Y;
}

Point<float> point;
point.X = 1.2f;
point.Y = 3.4f;
```

Generikus művelet

Swap példa (két érték cseréje):

```

class MyClass {
    public static void Swap<T>(ref T lhs, ref T rhs)
    {
        T temp; temp = lhs; lhs = rhs; rhs = temp;
    }
}

int a = 2, b = 3;
MyClass.Swap<int>(ref a, ref b);
// Ekvivalens az előzővel, nem kell kiírni a típust, kikövetkezteti a fordító a környezetből
MyClass.Swap(ref a, ref b);

```

Megjegyzés: a példában használtuk a C# nyelv `ref` kulcsszavát: ennek segítségével lehet egy paramétert cím szerint (vagyis referenciaiként) átadni, hasonlóan ahhoz, ahogy C/C++-ban a „&” használatával. Ilyenkor nem másolat készül a paraméterről a hívás során, hanem az eredeti változóra egy mutató adódik át, és azon keresztül az eredeti változó is megváltoztatható! A fenti swap példában is erre építünk.

Generikus interfész

Az alábbi példa a .NET beépített `IList<T>` interfészének egy részét mutatja:

```

interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerator<T> {
    public void Add (T item);
    ...
}

```

Generikus delegate

Az alábbi példa a .NET beépített `Predicate<T>` delegate típusdefinícióját mutatja:

```

delegate bool Predicate<T>(T obj);

```

.NET beépített generikus típusok

A .NET számos beépített generikus típust biztosít. Ezek közül a tárgyban csak azokat kell fejből tudni, melyek egyéb példákban is felbukkanak a félév során (félkövérrel kiemeltük alább), de mindenkiéppen érdemes áttekinteni őket, mert hasznosak a minden nap munkában. Pl.:

Néhány generikus gyűjtemény:

- **List<T>:** Dinamikusan nyújtózkodó tömb (mint a Java ArrayList)
- **LinkedList<T>**
- **HashSet<T>**
- **Dictionary<K,V>:** Kulcs-érték párok. K a kulcs típusa, megfelelő GetHashCode impl. kell!
- **SortedList<K,V>**
- **Stack<T>**
- **Queue<T>**
- **ReadOnlyCollection<T>**

Néhány generikus interfész:

- **IEnumerable<T>:** Csak iterálható, egyesével előre (GetEnumerator). Nem módosítható.
- **ICollection<T> :** IEnumerable<T> + Add, Remove, Contains
- **IList<T>:** ICollection<T> + Count tulajdonság, tömbindex operátor
- **IReadOnlyCollection<T>:** IEnumerable<T> + Count

- IReadOnlyList<T>: I IReadOnlyCollection<T> + Count, tömbindex operátor
- Stb.: IDictionary<K,V>, IEnumerator<T>, IComparable<T>, IComparer<T>

Beépített generikus delegate-ekre jelen jegyzetben később, a lambda kifejezések tárgyalásakor nézünk példákat.

Gyakorlópélda

Alakítsuk át az előző előadáson a delegate témakörnél taglalt univerzális, kissé „csúnyácska” object alapú HyperSort példát típusbiztosra (generikusra). Az alábbi kódban kiemeléssel jelöljük a változásokat, a törölt részeket pedig áthúzással (előadáson folyamatában kerül ismertetésre, lásd előadás demók között „01 Generic Types sort example”).

```
delegate bool FirstIsSmallerDelegate<T>(object a, object b);

// Itt választhatunk, hogy az osztályt, vagy csak a művetetét tesszük generikussá:
// az utóbbit választjuk.
class Sorter
{
    public static void HyperSort<T>(List<T> list, FirstIsSmallerDelegate<T> fis)
    {
        for (int i = 1; i < list.Count; i++) {
            for (int j = list.Count - 1; j >= i; j--) {
                if (fis(list[j], list[j - 1])) {
                    object tmp = list[j];
                    list[j] = list[j - 1];
                    list[j - 1] = tmp;
                }
            }
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        ArrayList list = new ArrayList();
        List<Complex> list = new List<Complex>();

        list.Add(new Complex(3, 2));
        list.Add(new Complex(1, 2));

        Sorter.HyperSort(list, FirstIsSmaller_Complex );
    }

    public static bool FirstIsSmaller_Complex(object Complex a, object Complex b)
    {
        Complex ca = (Complex)a;
        Complex cb = (Complex)b;
        return a.Re < b.Re;
    }
}
```

Generikus kényszerek

(óra végén térünk rá idő függvényében)

Írunk egy olyan generikus gyűjtemény osztályt, mely támogatja az általa tárolt elemek sorrendezését (Sort művelet). A korábbi delegate alapú elem összehasonlítással szemben itt az elemeknek kell

implementálni az `IComparable` interfész, melynek egy `CompareTo` művelete van az adott, és egy paraméterben kapott másik elem összehasonlításához:

```
class SortableCollection<T>
{
    private T[] items;

    public void Sort()
    {
        for (int i = 1; i < items.Length; i++) {
            for (int j = items.Length - 1; j >= i; j--) {
                // Ha nem jó a sorrend, csere
                if (items[i].CompareTo(items[i + 1]) > 0) {
                    T tmp = items[j];
                    items[j] = items[j - 1];
                    items[j - 1] = tmp;
                }
            }
        }
    }
    // ...
}
```

A fenti kód azonban nem fordul, a sárgával kiemelt kódra panaszkodik a fordító. Az eddigi generikus feladatainkban, és ebben a példában is `T` **bármilyen** típus lehet a felhasználás során. Vagyis `T` lehet `int`, `Complex`, vagy bármilyen saját típus. Ugyanakkor erre a `T` típusra a fenti példában a kiemelt sorban meghívtuk a `CompareTo` műveletet. Vagyis feltettük, hogy `T`-nek van ilyen művelete, de erre jelen pillanatban semmiféle garancia nincs, amikor a compiler a fenti osztályt lefordítja IL kódra (C++ esetében azért nem volt gond, mert a sablon maga le se fordult, hanem csak az adott típusokkal történő alkalmazása!). A compiler egyet tehet: általános esetben csak azon műveletek alkalmazását engedi, melyek minden típusra értelmezettek valamilyen módon, pl.: operátor`=`, `ToString()` és ami még a minden típus közös ősében, az `object`-ben definiált. Ezek alkalmazásából ugyanis utólag értelemszerűen nem lehet gond. A `CompareTo` nincs benne az `object` ősben, vagyis nem támogatja minden típus, így fordítási hibát kapunk.

Mit lehet tenni? Le kell szűkíteni a `T`-re alkalmazható típusok körét a példánkban, olyan típusokra, melyek rendelkeznek megfelelő `CompareTo` művelettes. Ha `T`-re kikötjük pl., hogy csak olyan típus lehet, mely megvalósítja az `IComparable` interfész, akkor annak biztosan lesz megfelelő `CompareTo` művelete. Célba értünk: **ezt .NET kényszerek (constraints) segítségével tudjuk kikényszeríteni**. A fenti kódot módosítuk egy kényszer felvételével:

```
class SortableCollection<T> where T : IComparable
{
    ...
}
```

A kiemelt rész új, a `where` kulcsszót követően adhatunk meg a generikus paraméterekre kényszereket, vesszővel elválasztva többet is. A legfontosabb lehetőségek:

- **Interfész és/vagy alaposztály** megkötések, mint a példában is volt, vesszővel elválasztva több is megadható.
- **new()**: a paraméternek kell legyen alapértelmezett konstruktora
- **struct**: az adott paraméter csak érték típus lehet
- **class**: az adott paraméter csak referencia típus lehet

További lehetőségekről itt lehet többek között olvasni: <https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/constraints-on-type-parameters>.

Nézzünk egy összetettebb példát (a példa nem valós, a .NET beépített Dictionary<K, V> osztálya másként néz ki):

```
class Dictionary<K,V>: IDictionary<K,V>
    where K: IComparable<K>
    where V: class, IKeyProvider<K>, IPersistable, new()
{
    ...
}
```

Értelmezzük a fentieket:

- A Dictionary osztálynak két generikus paramétere van, K és V (a kulcs és az érték típusa).
- Az osztály implementálja a generikus IDictionary interfészt, K és V típusokkal felparaméterezve.
- K: csak olyan típus lehet, mely megvalósítja az IComparable<K>-t.
- V: csak referencia típus lehet, meg kell valósítsa a IKeyProvider<K> és IPersistable interfészeket, és kell legyen alapértelmezett konstruktora.

Lambda expression (lambda kifejezés)

Bevezető

Bár alapvetően a funkcionális programozás eszköztárából származik, a lambda kifejezések támogatása és alkalmazása pár évvel ezelőtt berobbant a modern objektumorientált nyelvek világába is (úgymint C#, Java, C++ stb.). Lambda kifejezéseket pont olyan gyakorisággal használjuk a minden nap munka során, mint pl. az interfészeket vagy hasonló nyelvi elemeket.

Az alábbiakban „nulláról indulva” ismerkedünk meg a koncepciójával és alkalmazásával, noha az alapok Java nyelvű programozásból már szerepeltek. Az alapkoncepció természetesen ugyanaz, de vannak jelentős eltérések is, például:

- Java nyelvben interfészekre épül (olyan interfész esetén használható, melynek egy absztrakt művelete van), .NET esetében a delegate-ekre,
- Java nyelvben a „->”, C# nyelvben a „=>” az operátora.

Lehetőséget biztosítanak **névtelen (anonymous) függvények** definiálására. Ehhez a lambda (deklaráció) operátort kell használni, melynek jelölése: =>.

Amikor egy lambda kifejezést adunk meg, egy név nélküli függvényt definiálunk, melynek paraméterlistáját a => bal oldalán, a törzsét a => jobb oldalán adjuk meg. Pl.:

```
(int x, int y) => { return x+y; }
```

vagy sokkal egyszerűbb formában:

```
(x, y) => x + y
```

A szintaktika precízebb megközelítésben: a => operátor bal oldalán kell megadni a paramétereket, a jobb oldalán pedig:

- **utasítás/statement lambda** változat (fenti első példa) esetén {} között utasítások (statementek) sorozatát, vagy egyetlen utasítást. A szintaktika:

```
(bemenő paraméterek) => { <utasítások sorozata> }
```

- **kifejezés/expression lambda** változat (fenti második példa) esetén {} nélkül egyetlen kifejezést.

A szintaktika:

(bemenő paraméterek) => kifejezés

Kérdés még, hogy **mi egy lambda kifejezés típusa**, vagyis milyen típusú változóval tudunk egy lambda kifejezésre hivatkozni: egy vele **kompatibilis delegate típussal**. Pl.:

```
delegate bool FirstIsSmallerDelegate(int a, int b); // Delegate típus
// Lambda eltárolása lokális változóban:
FirstIsSmallerDelegate fis = (int x, int y) => { return x+y; }
```

A következő fejezetekben nézzük a részleteket egy átfogóbb példán keresztül.

Utasítás (statement) lambda

Ennél a változatnál a => jobb oldalán {} között akárhány utasítás állhat, ez az általános szintaktika:

(bemenő paraméterek) => { <utasítások sorozata> }

A gyakorlatban 3-10 sornál hosszabbat ritkán szoktunk írni. A kiindulási példánk legyen a jelen jegyzet korábbi, Generikus típusok fejezetének zárópéldája, a már generikussá alakított HyperSort függvény, és annak használata (előadás demók között a „02a Lambda sort example” példát nézzük). Az áttekinthetőség kedvéért itt megismételjük a kódot:

```
delegate bool FirstIsSmallerDelegate<T>(T a, T b);

class Sorter
{
    public static void HyperSort<T>(List<T> list, FirstIsSmallerDelegate<T> fis)
    {
        for (int i = 1; i < list.Count; i++) {
            for (int j = list.Count - 1; j >= i; j--) {
                if (fis(list[j], list[j - 1])) {
                    T tmp = list[j];
                    list[j] = list[j - 1];
                    list[j - 1] = tmp;
                }
            }
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        List<Complex> list = new List<Complex>();

        list.Add(new Complex(3, 2));
        list.Add(new Complex(1, 2));

        Sorter.HyperSort(list, FirstIsSmaller_Complex);
    }

    public static bool FirstIsSmaller_Complex(Complex a, Complex b)
    {
        return a.Re < b.Re;
    }
}
```

Eddig ismételtünk, még nem használtunk lambda kifejezést. De nem minden akarunk „hagyományos”, névvel rendelkező függvényt (példánkban a `FirstIsSmaller_Complex`) definiálni, csak azért, hogy valamilyen „kódot”/logikát paraméterként tudjunk függvénynek paraméterként megadni. Sok esetben van ennek kényelmesebb módja is. A következő példában már egy lambda kifejezést használunk:

```
delegate bool FirstIsSmallerDelegate<T>(T a, T b);

class Sorter
{
    public static void HyperSort<T>(List<T> list, FirstIsSmallerDelegate<T> fis)
    {
        // Változatlan!
    }
}

class Program
{
    ...
    static void Main(string[] args)
    {
        List<Complex> list = new List<Complex>();

        list.Add(new Complex(3, 2));
        list.Add(new Complex(1, 2));

        Sorter.HyperSort(list, FirstIsSmaller_Complex);
        Sorter.HyperSort(list, (Complex a, Complex b) => { return a.Re < b.Re; });
    }

    public static bool FirstIsSmaller_Complex(Complex a, Complex b)
    {
        return a.Re < b.Re;
    }
}
```

A példában minden változatlan és pont úgy működik, mint korábban a klasszikus delegate-es megoldásnál egy kivétellel: itt nem vezettünk be és használtunk egy külön, névvel rendelkező függvényt (`FirstIsSmaller_Complex`), hanem a függvényt név nélkül, helyben definiáltuk. Mégpedig egy („utasítás lambda” formájú) **lambda kifejezés** formájában. A `HyperSort` változatlan, hiszen a lambda kifejezést a már meglévő `FirstIsSmallerDelegate` típusként kapja meg, és a szokásos módon hívja meg. Természetesen itt is figyelni kell, hogy **csak olyan lambda kifejezés adható meg, mely paraméterlistája és visszatérése megfelel a delegate típusának!**

További egyszerűsítési lehetőség, hogy **a compiler a paraméterek típusát ki tudja az esetek túlnyomó többségében következtetni**, így ezeket nem kell, és nem is szoktuk megadni:

```
Sorter.HyperSort(list, (a, b) => { return a.Re < b.Re; });
```

Összegezve: a lambda kifejezés alkalmazásának előnye az, hogy **nem kellett a delegate által hivatkozott logika/kód kedvéért külön klasszikus/nevesített függvényt írni**, hanem helyben definiáltuk egy lambda kifejezés segítségével. Összegészében sokkal tömörebb, rövidebb kódot kaptunk lambda kifejezés használatával.

Kifejezés (expression) lambda

Amennyiben a lambda függvényünk **egyetlen utasításból áll**, lehetőségünk van **kifejezés (expression) lambda** használatára, mely a fenti utasítás (statement) lambdánál tömörebb lambda kifejezés formát tesz lehetővé.

```
Sorter.HyperSort(list, (a, b) => a.Re < b.Re);
```

A szintaktika a következő: (bemenő paraméterek) => kifejezés

A => jobb oldala { }-ek nélkül csak egyetlen kifejezésből áll, ennek értékével tér vissza. "return" sem kell, sőt nem is használható, illetve „;” sem kell a végére, hiszen ez egy kifejezés.

Látható, hogy **milyen egyszerű és tömör formában tudunk „logikát”/”kódot” egy másik függvénynek**, esetünkben a HyperSort-nak megadni.

Egyéb lehetőségek

A fenti példában a lambdát egy függvényparaméternek adtuk át. De természetesen bármilyen (kompatibilis) delegate változónak értékül lehet egy lambda kifejezést adni, pl. tagváltozónak vagy lokális változónak is. Példa lokális változóra:

```
static void Main(string[] args)
{
    FirstIsSmallerDelegate<Complex> fis = (a, b) => a.Re < b.Re;
    bool b = fis(new Complex(10, 20), new Complex(1, 2));
}
```

Beépített delegate típusok – Func generikus delegate (fontos)

A fenti példában az alábbi, általunk definiált delegate típust használtuk:

```
delegate bool FirstIsSmallerDelegate<T>(T a, T b)
```

Valójában ennek bevezetésére nem is lett volna szükség. Ugyanis a .NET rendelkezik számos **Func** nevű beépített generikus delegate-tel. Pl.:

- a **Func<TResult>** olyan delegate típus, melynek nincs paramétere és TResult típussal tér vissza,
- a **Func<T1, TResult>** olyan delegate típus, melynek egy T1 típusú paramétere van és TResult-tal tér vissza
- a **Func<T1, T2, TResult>** olyan delegate típus, melynek egy T1 és egy T2 típusú paramétere van és TResult-tal tér vissza,
- stb. számos paraméter számig, **mindig a visszatérési típus az utolsó paraméter**.

Ennek megfelelően az előző fejezet lokális változós kispéldáját alakítsuk át:

```
static void Main(string[] args)
{
    FirstIsSmallerDelegate<Complex> fis = (a, b) => a.Re < b.Re;
    Func<Complex, Complex, bool> fis = (a, b) => a.Re < b.Re;
    bool b = fis(new Complex(10, 20), new Complex(1, 2));
}
```

Az új megoldásban nem egy saját FirstIsSmallerDelegate típusú változóval hivatkoztunk a függvényükre (ami esetünkben egy lambda függvény, de lehetne normál függvény is), hanem a beépített Func egyik variálásával. Kérdés, miért pont a három paraméteres Func-ot használtuk, és miért pont Complex, Complex, bool> paraméterekkel. A példában egy olyan függvényre hivatkozuk

vele, melynek **paraméterei** és **visszatérési típusa sorrendben**: `Complex`, `Complex`, `bool`. **Pontosan ezeket kell ebben sorrendben a Func-nak generikus paraméterben megadni:** `Func<Complex, Complex, bool>`. Ez a típus lesz kompatibilis a függvényünkkel, ezt is használtuk a példánkban.

Most térjünk át a `HyperSort` példára. Ezt is alakítsuk át úgy, hogy saját delegate típus helyett egy megfelelő `Func` variánst használunk. Haladjunk fokozatosan, első körben még ne a generikus, hanem a korábbi előadáson használt `object` alapú `HyperSort`-ot alakítsuk át.

```
// Töröljük, nincs rá szükség
delegate bool FirstIsSmallerDelegate(object a, object b);

class Sorter
{
    public static void HyperSort(List<object> list, FirstIsSmallerDelegate fis)
    public static void HyperSort(List<object> list, Func<object, object, bool> fis)
    {
        // Változatlan!
    }
}
```

Ebben az `object` alapú megoldásban a hivatkozni kívánt függvénynek két `object` paramétere van, és `bool`-lat térnek vissza, így a `Func<object, object, bool>` variánst kell használni.

Most már legyünk bátrabbak, alakítsuk a `HyperSort`-ot ismét generikussá az `object` alapú helyett, és használunk `Func` típust is:

```
class Sorter
{
    public static void HyperSort(List<object> list, Func<object, object, bool> fis)
    public static void HyperSort<T>(List<T> list, Func<T, T, bool> fis)
    {
        // Változatlan!
    }
}
```

A Generikus típusok fejezetben tanultaknak megfelelően bevezettünk egy `T` paramétert (szokásosan türkizzel jelölve), és a generikus függvényünkben használjuk azt (szokásosan zölddel), így kapjuk a fenti megoldást.

Beépített delegate típusok – Action és generikus Action delegate (fontos)

A `Func` delegate olyan esetben használható, amikor van visszatérési érték is. Amennyiben nincs (a visszatérési típus `void`) beépített **Action** delegate típus különböző változatai használhatók: Pl.:

- az **Action** olyan delegate típus, melynek nincs paramétere (és `void`-dal tér vissza),
- az **Action<T1>** olyan delegate típus, melynek egy `T1` típusú paramétere van (és `void`-dal tér vissza),
- az **Action<T1, T2>** olyan delegate típus, melynek egy `T1` és egy `T2` típusú paramétere van (és `void`-dal tér vissza),
- stb. számos paraméter számig.

Példa 1:

```
Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
```

```
greet("World"); // Kiírja, hogy "Hello World"
```

Példa 2:

```
Action<string, string> labelAndTextWriter =
    (label, text) => Console.WriteLine($"{label}:\t{text}");
labelAndTextWriter("Név", "Ennio Morricone");
labelAndTextWriter("Szül. év", "1928");
```

Lambda kifejezésekben a paraméterek megadásának szintaktikája

Ha nincs paraméter, üres zárójel:

```
Action line = () => Console.WriteLine();
```

Ha egy paraméter van, a zárójel opcionális (nem szoktuk kiírni):

```
Func<double, double> cube = x => x * x * x;
```

Ha több paraméter is van, ()-ek között vesszővel elválasztva soroljuk fel, lásd fenti példák, illetve:

```
Func<int, int, bool> testForEquality = (x, y) => x == y;
```

Emlékezzünk: ha a fordító nem tudja kikövetkeztetni a paraméterek típusát, akkor adjuk meg a típusokat is (ebben a példában egyébként ki tudná):

```
Func<int, string, bool> isTooLong = (int x, string s) => s.Length > x;
```

Variable capturing, closure

Egy normál függvény alapesetben csak a paramétereiben kapott változókon tud dolgozni. A lambda függvényeknek van egy speciális tulajdonsága: a paramétereken túlmenően a kontextusukban levő lokális változókat és tagváltozókat is látják. Ezt variable capturingnek, vagy closure-nek nevezik, és sok esetben nagyon megkönnyíti az életünket, mert nehéz lenne megoldani az adott értékek klasszikus paraméterben való megadását. Egyszerűbb példa:

```
class Incrementer
{
    int delta1 = 10;

    public void DoIt()
    {
        int delta2 = 10;

        Action<int> printIncremented = n => Console.WriteLine(n + delta1 + delta2);

        printIncremented(100);
    }
}
```

A példában az látható, hogy a lambda függvény nemcsak a paraméterben megkapott `n`-t, hanem a lokális `delta1` és a tagváltozó `delta2`-t is fel tudja használni.

Egy másik, kicsit izgalmasabb (bár az egyszerűség érdekében kissé mesterkélt) példával megvilágítjuk, hogy ez nem önmagától adódó helyzetekben is működik:

```
...
static void Main()
{
    IncrementFactory incrementFactory = new IncrementFactory();
    var increment = incrementFactory.GetIncrementFunc();
    res = increment(100);
```

```

        }
        ...

class IncrementFactory
{
    public Func<int, int> GetIncrementFunc()
    {
        int delta = 20;
        return n => n + delta;
    }
}

```

A példában az `IncrementFactory` osztály `GetIncrementFunc` művelete egy olyan függvénnyel tér vissza, mely egy számot tud megnövelni (Pontosabban visszaad egy megnövelt számot). Hogy mennyivel, az a `delta` lokális változóban van benne. A `Main` függvényben az `IncrementFactory`-tól szerünk egy ilyen függvényt, egy lokális változóban eltároljuk, majd meghívjuk a 100-as paraméterrel. A megoldás jól működik, annak ellenére, hogy „érzésre” inkonzisztens. Amikor ugyanis a meghívjuk az `increment` által hivatkozott lambda függvényt, a `delta` lokális változónak már rég nem is kellene léteznie, hiszen a `GetIncrementFunc()` függvény egy sorral feljebb már visszatért, vagyis a lokális változói megszűntek. De a lambda kifejezéseknel a variable capture pont ezt a problémát oldja meg: a lambda függvénnyel nemcsak a függvény kódja tárolódik el, hanem a színpalak mögött egy anonim osztály jön létre, mely tagváltozóiban tartalmazza az ilyen módon felhasznált/behivatkozott (capture-ölt) változókat, esetünkben a `delta` változót.

A fenti példánk kissé mesterkélt. Remek gyakorló feladat egy olyan `GetIncrementFunc` változat elkészítése, melybe nincs lokális változóba beégetve, hogy az általa visszaadott függvény mennyivel növeli meg a paraméterének értékét, hanem a `GetIncrementFunc2` függvénynek ez a `delta` paraméterben átadható (vagyis a „`delta`” legyen a `GetIncrementFunc` paramétere, a lambda függvény számára az is egyfajta lokális változó).

Irodalom

- <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions>

LINQ - Language-Integrated Query

A LINQ a .NET/C# egyik fejlesztők által leginkább kedvelt szolgáltatása. Legegyszerűbb formájában nagyon leegyszerűsíti a gyűjteményekkel való munkát (pl. szűrés, sorrendezés, vetítés, csoportosítás, min/max/szumma stb.), de használható XML feldolgozásra, adatbázis lekérdezések megfogalmazására is. Mi csak gyűjtemények kezelésre nézünk pár alapműveletet ízelítőként, szélesebb körű használata későbbi témaiban szerepel. Kétféle szintaktikával dolgozhatunk: mi csak az ún. **fluent** szintaxist nézzük (az ún. query expression szintaxis a téma keretében nem szerepel).

Nézzünk egy nagyon egyszerű példát:

```

List<string> fruits = new List<string>() { "apple", "grape", "peach", "banana",
"pineapple" };

var fruits2 = fruits.Where(f => f.Length <= 5);

foreach (var f in fruits2)
    Console.WriteLine(f);

```

A példában a gyűjteményen definiált `Where` műveletet használjuk az elemek szűrésére. A `Where` egy olyan gyűjteménnyel tér vissza, melyre teljesül a lambda kifejezésben megadott feltétel (a példánkban az 5-nél nem hosszabb stringekre szűrünk). A `Where` megvalósítása a szílfalak mögött nagyon egyszerű: minden elemre meghívja paraméterben kapott delegate-et/lambda kifejezést, paraméterként átadva az adott elemet (ez a példában `f`), és azon elemeket veszi bele a kimenetbe, melynél a lambda `true`-val tér vissza (a `Where` valójában egy generikus függvény, `Func<T, bool>` delegate-et vár paraméterként, a példánkban ez `Func<string, bool>`).

Nézzünk egy összetettebb példát:

```
var fruits2 = fruits
    .Where(f => f.Length <= 5) // Szűrés
    .OrderBy(f => f.Length) // Sorrendezés
    .Select(f => $"Reverse: {f.Reverse()} Length: {f.Length}"); // Projekció
```

Az egészet írhattuk volna egy sorba, de ez így sokkal olvashatóbb. A példában a `fruit` gyűjteményt a `Where`-rel szűrtük, majd annak kimenetét az `OrderBy`-jal sorrendeztük hossz szerint, majd annak kimenetét a `Select`-tel vetítettük (projekció) /transzformáltuk. A `Select` minden elemhez egy a lambda kifejezés által meghatározott, tetszőleges típusú új objektumot készít (a példánkban egy új stringet), a kimeneti gyűjteményében ezek szerepelnek.

A példa jól szemlélteti, hogy a LINQ műveletek egymás után fűzhetők („.”-tal). Azt is érzékelhető, hogy az egész egy kicsit az SQL-re „rímel”, csak itt nem táblákon, hanem objektumokon dolgozunk.

A `Where/OrderBy/Select/stb.` tetszőleges .NET gyűjteményen használható, mert az `IEnumerable<T>` típuson definiáltak, és .NET-ben ezt az interfész minden gyűjtemény megvalósítja.

Ezek a műveletek `IEnumerable<T>`-vel térnek vissza (`T` az elem típusa): tulajdonképpen egy iterátorral kapunk, mely az iteráció során (amikor egy `foreach` művelettel végig megyünk a kimeneten) állítja elő a kimeneti halmazt. Ha nekünk a kimenti elemekre egy új gyűjtemény objektumban van szükségünk, akkor hívjuk meg a `ToList()` vagy `ToArray()` műveletet. Pl.:

```
var fruits2 = fruits.Where(f => f.Length <= 5).ToList(); // A fruits2 List<string>
```

Egyéb C# nyelvi eszközök

partial class (részleges osztály) – fontos!

Egy osztály több .cs fájlban is lehet definiálva. A fordító fésüli össze a részeket egy osztálytáblájá. A `partial` kulcsszót kell használni, anélkül fordítási hibát kapunk. Példa:

```
// Customer_a.cs
partial class Customer
{
    private int id;
    private string name;
    private List<Order> orders;

    public void SubmitOrder(Order order)
    {
        orders.Add(order);
    }
}
```



```
// Customer_b.cs
partial class Customer
{
    public bool HasOutstandingOrders()
    {
        return orders.Count > 0;
    }
}
```

Compile

```
// A fordítás során ez születik
partial class Customer
{
    private int id;
    private string name;
    private List<Order> orders;

    public void SubmitOrder(Order order)
    {
        orders.Add(order);
    }

    public bool HasOutstandingOrders()
    {
        return orders.Count > 0;
    }
}
```

A fenti példában a `Customer` osztály egyes részei két fájlban szerepelnek: a fordítás összefésüli a részeket, egyetlen `Customer` osztály születik, melyben minden rész benne van.

Az egyik fő alkalmazási területe: a generált és a kézzel írt kód különválasztása.

Expression-bodied members (kifejezéstörzsű tagok)

Időnként olyan rövid függvényeket, illetve tulajdonságok esetén kifejezetten gyakran olyan rövid `get/set/init` (az `init` a tárgy keretében nem szerepel) definíciókat írunk, melyek **egyetlen kifejezésből** állnak. Ez esetben a függvény, illetve tulajdonság esetén a `get/set/init` törzse megadható ún. **kifejezéstörzsű tagok (expression-bodied members)** szintaktikával is, a `=>` alkalmazásával. Ez akkor is megtehető, ha az adott kontextusban akár van visszatérési érték (`return` utasítás), akár nincs.

A példákban látni fogjuk, hogy a kifejezéstestű tagok alkalmazása nem több, mint egy kisebb szintaktikai "csavar" annak érdekében, hogy ilyen egyszerű esetekben minél kevesebb körítő kódot kelljen írni.

Nézzünk először egy függvény példát (feltessük, hogy az osztályban van egy `yearOfBirth` tagváltozó):

```
public int Age() => DateTime.Now.Year - yearOfBirth;
public void DisplayName() => Console.WriteLine(ToString());
```

Mint látható, elhagytuk a {} zárójeleket és a `return` utasítást, így tömörebb a szintaktika.

Fontos: Bár itt is a `=>` tokenet használjuk, ennek semmi köze nincs a korábban tárgyalta kifejezésekhez: egyszerűen csak arról van szó, hogy ugyanazt a `=>` tokenet (szimbólumpárt) két teljesen eltérő dologra használja a C# nyelv.

Példa tulajdonság getter megadására:

```
public int Age { get => DateTime.Now.Year - yearOfBirth; }
```

Sőt, ha csak getterje van a tulajdonságnak, a `get` kulcsszót és a kapcsos zárójeleket is lehagyhatjuk.

```
public int Age => DateTime.Now.Year - yearOfBirth;
```

Ezt az különbözteti meg a korábban látott függvények hasonló szintaktikájától, hogy itt nem írtuk ki a kerek zárójeleket.

```
public int Age() => DateTime.Now.Year - yearOfBirth;
```

Megjegyzés: A Microsoft hivatalos dokumentációjának magyar fordításában az "expression-bodied members" nem "kifejezéstörzsű", hanem "kifejezéstestű" tagként szerepel. Köszönjük szépen, de a függvényeknek sokkal inkább törzse, mint teste van a magyar terminológiában, így ezt nem vesszük át...

Object initializer (objektuminicializáló)

A publikus tulajdonságok/tagváltozók inicializálása és a konstruktorhívás kombinálható egy úgynevezett objektuminicializáló (object initializer) szintaxis segítségével. Ennek alkalmazása során a

konstruktorhívás után kapcsos zárójelekkel blokkot nyitunk, ahol a publikus tulajdonságok/tagváltozók értéke adható meg, az alábbi szintaktikával.

```
var p = new Person()
{
    Age = 17,
    Name = "Luke",
};
```

Az tulajdonságok/tagok inicializálása a konstruktor lefutása után történik (amennyiben tartozik az osztályhoz konstruktor). Ez a szintaktika azért is előnyös, mert egy kifejezésnek számít (azon hárommal szemben, mintha létrehoznánk egy inicializálatlan, Person objektumot, és két további lépéssben adnánk értéket az Age és Name tagoknak). Így akár közvetlenül függvényhívás paramétereként átadható egy inicializált objektum, anélkül, hogy külön változót kellene deklarálni.

```
void Foo(Person p)
{
    // do something with p
}

Foo(new Person() { Age = 17, Name = "Luke" });
```

Ahogy a fenti példákban is látszik, hogy nem számít, hogy az utolsó tulajdonság megadása után van-e vessző, vagy nincs.

Egyelőre az Object initializer használatának előnye kevésbé érezhető: de bizonyos – a tárgyban nem tanult, de később a képzésben szereplő – nyelvi konstrukciók, pl. Linq esetében sokszor elkerülhetetlen, így fontos az ismerete.