I. mérés: C# 3.X, Linq

A mérés során a hallgatók megismerkednek a C# nyelv 3.0, 3.5-ös verziójának újdonságaival, valamint a LINQ technológia objektumokra vonatkozó részével.

***Tipp****:Az alábbi anyag rendkívül könyű és intuitív lehet a témakör ismeretében, ugyanakkor előzetes ismeretek nélkül az új nyelvi szerkezetek sokszor furcsák, idegenek és nehezen értelmezhetőek. A magyarázatokat fokozottan érdemes a hallgatók előképzettségéhez igazítani*

A mérés során egy C# konzolos alkalmazást (Console Application) fogunk használni. A project létrehozásakor figyeljünk rá, hogy a project nyelve C#, verziója legalább 3.5-ös legyen.

A különböző példákat a Main függvényben készítjük el, a közben feleslegessé váló részletek kikommentezésével.

# C# 3.X újdonságok

## Implicit típusú lokális változók és tömbök

Az implicit típusú lokális változók jelzésére a *var* kulcsszót használjuk. A *var*-t csupán értékadással együtt lehet használni, ugyanis a fordítás közben a fordító az értékadó művelet jobb oldalán lévő kifejezés által eredményezett (legalkalmasabb) típust rendeli hozzá a *var* kulcsszóval jelzett változóhoz. A Visual Studio azonnal feltérképezi, hogy milyen típusról van szó, így az IntelliSense támogatás ugyanúgy biztosított az ilyen értékadásoknál. **Az erős típusosság megmarad.**

Vizsgáljuk meg, milyen típusú lesz az értékadás bal oldalán álló változó!

var i = 5; var s = "string"; var d = i + 0.15;

var a = new[] { 1, 2, 3, 4 }; var a2 = new[] { 1, 2.222, 3, 4 };

A *var* kulcsszó használható *for*, *foreach* és *using* kifejezésekben is.

## Auto-implementált tulajdonságok

Az auto-implementált tulajdonságokkal könnyen definiálhatunk tulajdonságokat, melyek getter/setter metódusai nem kívánnak saját logikát.

Hozzunk létre egy új code file-t, nevezzük Person.cs-nek.

public class Person

{

public Guid Id { get; private set; } // csak olvashato

public int Age { get; set; }

public string Name { get; set; }

}

***Tipp****:Használjuk a prop code snippetet!*

***Tipp****:A példa könnyebben érthető, ha először egy hagyományos tulajdonságot hozunk létre (propfull code snipet), majd rámutatunk, hogy ezt hogyan írhatjuk „tömörebben” az új szintaktikával*

## Objektum és gyűjtemény inicializálók

Az objektum inicializálók lehetőséget nyújtanak arra, hogy az objektum konstruktorának explicit meghívása nélkül létrehozzunk egy példányt egy adott típusú objektumból, és az elérhető mezőinek és tulajdonságainak értéket adjunk.

Person p = new Person { Age = 900, Name = "Yoda" };

A gyűjtemény inicializálók tetszőleges „gyűjtemény” feltöltésére használhatóak, mely implementálja az IEnumerable interfészt és van publikus Add metódusa.

List<Person> list = new List<Person>()

{

p,

new Person() { Age = 25, Name = "Luke" },

new Person() { Age = 50, Name = "Obi-Wan" }

};

***Tipp****:Adjunk hozzá szándékosan egy hibás típusú objektumot (pl. egy int-et) a gyűjtemény inicializálóhoz. Figyeljük meg, hogy a fordító hibaüzenet a megfelelő Add metódus hiányára fog panaszkodni*

## Anonim típusok

Az anonim típus lehetőséget teremt arra, hogy beágyazzunk néhány csak-olvasható (read-only) tulajdonságot egy objektumba anélkül, hogy előtte expliciten definiáltuk volna annak típusát. A típus nevét a fordító generálja, ez forráskódból nem elérhető. A tulajdonságok típusait szintén a fordító határozza meg. Az anonim típusok referencia típusok, csak osztályok (class) lehetnek, melyek csak publikus, csak-olvasható tulajdonságokat tartalmazhatnak, más tagokat (eseményeket, metódusokat) nem.

Ha egy változó egy anonim típusú objektum értékét kapja, az forráskód szinten csak a *var* kulcsszó használatával oldható meg. Ha több anonim típus ugyanazokat a tulajdonságokat tartalmazza ugyanolyan sorrendben, a fordító egyként kezeli őket.

A példában mutassuk be, hogy azonos típust generál a fordító, ha a tulajdonságok nevei és típusai megegyeznek.

Mutassunk rá, hogy minden esetben generikus típust generál a fordító (hatékonysági megfontolásokból).

var anonType = new { myProp = 1, myProp2 = "string" };

var anonType2 = new { myProp = 2, myProp2 = 3 };

var anonType3 = new { };

var anonType4 = new { myProp = 4, myProp2 = "string" };

var anonType5 = new { i = 5, s = "string" };

***Tipp****:A Ha sok idő marad a mérés végén, Reflector-ral vizsgáljuk meg azt a szerelvényt, amelybe a forráskódot írtuk.*

Az anonim típusú osztály láthatósága *internal*, nem származtatható belőle másik típus, valamint a fordító egy *DebuggerDisplay* illetve egy *CompilerGenerated* osztályszintű attribútummal látja el, míg az osztály tagjai közül a két tulajdonságon kívül mindegyiket elrejti még a debugger elől is (*DebuggerHidden* attribútum).

## Bővítő metódusok

A bővítő metódusok lehetőséget nyújtanak arra, hogy már meglévő típusokhoz hozzárendelhessünk saját metódusokat, mint tagmetódusokat anélkül, hogy származtassunk, újrafordítsunk, vagy esetleg változtassunk az eredeti típuson. Tehát látszólag kibővíthetünk egy már meglévő típust egy metódussal. Akár olyan típust is, melynek a forráskódja nem áll a rendelkezésünkre.

A bővítő metódusok olyan speciális statikus metódusok, melyeket úgy hívhatunk, mintha egy adott típus tagmetódusát hívnánk. Egy bővítő metódust csak statikus osztályban definiálhatunk, használva a *this* kulcsszót.

A példában egy bővítő metódust definiálunk a List<Person> típusnak (hogy ne kelljen minden helyen megírni a név kezdőbetűje szerinti szűrést).

Hozzunk létre egy új osztályt, nevezzük PersonHelper-nek (az osztály legyen statikus). Az osztályban definiáljuk a következő függvényt:

public static List<Person> NameStartsWith( this List<Person> list, string startString )

{

return list.FindAll( delegate( Person p )

{

return p.Name.StartsWith( startString, true, CultureInfo.InvariantCulture );

} );

}

Mutassuk meg, hogy ezek után egy Person listánál az Intellisense felajánlja az adott metódust (míg más típusú listánál nem):

foreach ( var person in list.NameStartsWith("y") )

{

Console.WriteLine("{0} [{1}]", person.Name, person.Age);

}

## Lambda kifejezések és kifejezésfák

Mondhatjuk azt, hogy a 2.0-ás anonim metódusok szintaxisának továbbfejlesztésével egy új, funkcionális nyelvi szintaxishoz hasonló eszközökkel definiálhatunk anomim metódusokat.

Egy példán keresztül mutassunk be egy lambda kifejezést:

var adultsA = list.FindAll( delegate( Person p ) { return p.Age >= 18; } );

var adultsB = list.FindAll( p => p.Age >= 18 );

Az p => p.Age >= 18 kifejezés egy eldöntentő kifejezés, mely megállapítja, hogy a “paraméterként kapott” p.Age érték nagyobb-e 18-nál vagy nem. Amint látható, egy lambda kifejezés bal oldalán a paraméter, jobb oldalán egy utasítás/feltétel/stb. áll, míg közöttük egy => szimbólum szerepel (melyet az angol “goes to”-ként olvas).

Az egy paraméterű lambda kifejezés generikus reprezentációja a *Func<T, TResult>* típus, mely *TResult* típusú adatot ad vissza és egy *T* típusú paramétert vár.

***Tipp****: A lambda kifejezések szintaktikája rendkívül szokatlan az imperatív programozási nyelvekben, és a hallgatók többsége korábban nem találkozott vele. Szánjunk kis időt ennek a résznek a tárgyalására.*

A lambda kifejezések futtatható kóddá fordulnak, IL szinten megjelennek. Viszont kifejezésfaként teljesen másképp funkcionálnak. A kifejezésfák lehetővé teszik, hogy a lambda kifejezések futtatható kód helyett adatstruktúraként legyenek jelen a memóriában.

Az a lambda kifejezés, melyet *D* típusú delegate-té lehet konvertálni, *System.Query.Expression<D>* típusú kifejezéssé is konvertálható. Míg delegate-té konvertálva futtatható kód jön létre, kifejezésfa esetén egy kifejezés példány jön létre, mely később kiértékelhető. A kifejezésfák hatékony memóriabeli reprezentációi a lambda kifejezéseknek, melyek biztosítják, hogy a kifejezés struktúrája transzparens és explicit.

public class ExprTest

{

Func<int, int> f = x => x + 1; // Code

Expression<Func<int, int>> e = x => x + 1; // Data

public void Test()

{

Console.WriteLine( f.Method );

Console.WriteLine( Environment.NewLine );

Console.WriteLine( "body: {0} \t|\t body nodetype: {1}", e.Body, e.Body.NodeType );

Console.WriteLine( "nodeType: " + e.NodeType );

e.Parameters.ToList().ForEach( delegate( ParameterExpression pExpr ) { Console.WriteLine( "param: {0} | parNodeType: {1}", pExpr.Name, pExpr.NodeType ); } );

}

}

## LINQ

A Language Integrated Query lehetővé teszi, hogy C# szerű szintaktikával, közvetlenül a C# kódunkba beágyazva fogalmazhassunk meg lekérdezéseket. A LINQ kulcsszavai valójában bővítő metódusokra fordulnak le.

A LINQ-nek különböző változatai léteznek, és a provider-modellnek köszönhetően előreláthatólag számuk növekedni fog. Az „alapértelmezett” változat a *Linq-To-Objects*, mely a memóriában tárolt objektumokon keresztül képes adatot kezelni. Két, nagyon elterjedt változat még a *Linq-To-Sql* és a *Linq-To-Xml*, melyek adatbázistáblákat vagy XML formátumban tárolt információt transzformálnak memóriabeli struktúrákra.

LINQ lekérdezéseket hajthatunk végre olyan gyűjteményosztályokon, mint a List<T>, Array, vagy Dictionary<TKey, TValue>.

Példa (a createList metódus törzsébe másoljuk át a gyűjtemény-inicializáló kódot):

var list = createList();

var query = from p in list

where p.Age < 30

select p;

foreach ( var item in query )

{

Console.WriteLine(item.Name);

}

A kifejezések egyik nagy előnye (pl. az SQL-lel szemben), hogy erősen típusosak. Itt a mérésvezető bemutathatja, hogy a változó-átnevezés utáni refaktorálás a LINQ kifejezésekre is működik (p átnevezése person –re).

Fontos megemlíteni, hogy nem-generikus IEnumerable gyűjtemények esetén a változó típusát ki kell írni:

var query = from Person p in arrayList ...

Lekérdezési kulcsszavak: from, where, join, select, group, into – ezek jelentését a felhasználáskor mondjuk el.

Az alap lekérdező operátorok IEnumerable<T> és IQueryable<T> interfészeket implementáló szekvenciákon hajtódnak végre. Támogatják a szűrést, a projekciót, az aggregációt, és a rendezést.

A különböző interfészeken operáló lekérdező operátorok bővítő metódusként vannak implementálva.

Időzítés szempontjából kétféle végrehajtás lehetséges:

* az egyedi visszatérésű végrehajtás azonnal megtörténik (pl. Average, Sum)
* szekvencia visszatérésű végrehajtás lazy módon történik (csak az első hozzáféréskor értékelődik ki) és csak egy enumerálható objektumot ad vissza

### IEnumerable<T> vs. IQueryable<T>

**IEnumerable<T>:**

A memóriában tartott (in-memory) gyűjteményeken operáló metódusok egy enumerálható objektumot adnak vissza. Amikor az szekvencia elemeit el akarjuk érni, akkor értékelődik ki.

**IQueryable<T>:**

Ezen metódusok egy kifejezésfát építenek, melyek a lekérdezést reprezentálják. A lekérdezés kiértékelését a forrás objektum végzi el (mely az interfészt implementálja).

A lekérdező metódusok hívásait össze lehet láncolni, így egy, bonyolult lekérdező kifejezéssé lehet alakítani a több, egyszerűbb lekérdezést.

A következő példán két lekérdezést összefűzünk, majd egy csoportosítást végzünk a név tulajdonság hossza alapján.

var baseList = from p in list

where p.Age < 100

select p;

var q = from p in baseList

group p by p.Name.Length into \_group

orderby \_group.Key

select new { Length = \_group.Key, Persons = \_group };

foreach ( var item in q )

{

Console.WriteLine("Names with length {0}:", item.Length);

foreach ( var p in item.Persons )

{

Console.WriteLine(p.Name);

}

}

Using-ok rendezése, eltávolítása: a mérés végén a forrásfájlokon végigmehetünk, és a szerkesztőfelületen használjuk a jobbklikk – Organize usings funkciót.

Ha még marad idő, mutathatunk néhány LINQ lekérdezést:

// cross join -- kiirjuk a nala fiatalabbakat

var q = from p in list

from p2 in list

where p2.Age < p.Age

select new { current = p.Name, younger = p2.Name };

foreach ( var item in q )

{

Console.WriteLine(item.current + " > " + item.younger);

}

Ha még marad idő, Reflector-ral megvizsgálhatjuk az eredményezett szerelvények belső felépítését.

## Parciális metódusok

***Tipp****: idő szűkében ezt az alfejezetet érdemes kihagyni. A LINQ és a lambda kifejezések sokkal fontosabbak a későbbiek szempontjából*

A mérésvezető – ha szükségesnek tartja – beszélhet a parciális osztályokról (2.0-ás újdonság).

A parciális metódusok a parciális típusok analógiájára jöttek létre. Csak parciális típusok rendelkezhetnek parciális metódusokkal. Az egyik forrásfájl tartalmazza a parciális metódus szignatúráját, míg egy másikban (vagy ugyanabban) található a parciális metódus implementációja. Azért parciális, mert ha nem található meg az adott (parciális) osztályon belül az implementáció, akkor fordítási időben mind a metódus definíciója, mind annak összes hívása elveszik (nem kerül bele a fordított kódba). Egy parciális metódus csak *void* lehet.

Módosítsuk a Person osztályunkat:

* definiáljunk parciális metódust:
* a Name property getter/setter-ét implementáljuk
* hozzunk létre egy másik code file-t Person.Logic.cs néven, és implementáljuk a parciális metódusokat
* figyeljünk a partial kulcsszó megadására az osztálynév definiálásakor

partial void OnCreated();

partial void OnNameChanged();

public string Name

{

get { return this.name; }

set

{

this.name = value;

OnNameChanged();

}

}

public Person()

{

this.Id = Guid.NewGuid();

OnCreated();

}

Valamint a Person.Logic.cs-ben implementáljuk a parciális metódusokat:

partial void OnCreated()

{

Console.WriteLine( "Person instance created with ID: " + this.Id );

}

partial void OnNameChanged()

{

Console.WriteLine( "Name changed to: " + this.name );

}

Ha a mérés végén sok idő marad, Reflector-ral mutassuk meg, hogy ha egy parciális metódust nem implementálunk, az nem jelenik meg a fordított szerelvényben.