Ръководство за C# 3.0 и LINQ

Галин Илиев
www.galcho.com

Въведение

За пет години (официално – седем от обявяването) съществуване .NET Framework и създаденият специално език C# промениха много начина на разработване на софтуер. Може дори да се каже, че направиха революция като обединиха няколко важни възможности, които до онзи момент съществуваха накуп само в Java, а именно:

- Силно типизирани данни/променливи
- Улеснено управление на паметта, чрез „събирач на боклук“ (Garbage Collector)
- Архитектура, базирана на виртуална машина
- Управление на грешки базирано на обработване на изключения
- Обширна библиотеки, които покриват голям диапазон от функционалност.

Със втората версия на .NET Framework (пусната на пазара през декември 2004) бяхме свидетели на множество нови допълнения към C#, които позволяват създаването на много по-елегантни (и по-лесно читаеми) сорсове. Въпреки, че много от вас са запознати с тях нека ги обобщим:

- Частични класове (`partial classes`) – позволяват разделянето на един клас в няколко файла
- Generics или параметризирани типове
- Нова форма на цикли, въвеждаща yield оператора, подобно на езика Python
- Анонимни методи
- Модификатори за достъп могат да бъдат задавани независимо към помощните get и set методи на свойствата.
- Стойностни типове, които могат да приемат стойност `null`
- Обединяващ оператор ??, който връща първият от операндите, който не е `null` и др.

(Вече има достатъчно ресурси по темата, така че няма да продължавам с описанието на C# 2.0)

Нека да продължим към

Visual Studio 2008 и C# 3.0

Предполагам, че много от вас са чували за Visual Studio 2008 (доскоро познато под кодовото име ”Orcas”) и за LINQ. Това е следващата версия на популярното Visual Studio (версия 9), което ще бъде пуснато на пазара с новата версия 3.0 на C#. Както може да се очаква с тази версия на C# ще се добави още функционалност, която още повече ще увеличи производителността на софтуерните инженери. Някои от тези допълнение са:

- Подобрена инициализация на обектите

C# 2.0	C# 3.0

`Movie movie = new Movie();`
`movie.Name = "Space wars";`
`movie.Director = "George Lucas";`
	`Movie movie = new Movie {Name="Space wars", Director = "George Lucas"};`

- Инициализация на колекции подобно на масиви

C# 2.0	C# 3.0

`SomeList list = new SomeList();`
`List.Add(1);`
`List.Add(2);`
	`SomeList list = new SomeList {1, 2};`

- Подразбиране на типа на променливите

```
var text = "my text";          =          String text = "my text";
```

- Имплицитно типизиране на масивите, в зависимост от данните

```
int[] arr=new int[] { 1, 2, 3 }    =  var  arr = new int[] { 1, 2, 3 }
```

- Ламбда изрази

```
listOfFoo.Where(delegate(Foo x) { return x.size > 10; })
е еднозначно на
listOfFoo.Where(x => x.size > 10);
```

- Компилатора превежда ламбда изразите към силно типизирани делегати или към дървета от изрази (expression trees)
- Анонимни типове

```
var x = new { Name = "Galcho" }
```

- Разширения на методи
- LINQ – Language INtegrated Query или като опит за превод – заявки вградени в езика.

Всяка от тези нови функции ще разгледаме в детайли в отделна секция на това ръководство.

Както виждате Майкрософт работят по интересни възможности, които ще пренесат C# на едно ново ниво. Големият въпрос тук е "Откъде да започнем?". Този път нека започнем отзад напред. Така ще се разкрие най-голямото предимство на C# 3.0, а именно LINQ, като после ще покажем в детайли останалите нововъведения, които правят LINQ възможно.

Ако в началото не са ви познати езиковите конструкции – не се притеснявайте – ще ги обясним в детайли в следващи секции.

Въведение в LINQ и основни възможности

Както споменахме по-горе зад абревиатурата LINQ стои **L**anguage **IN**tegrated **Q**uery. Това е проект на Майкрософт за добавяне синтаксис за заявки, характерен за T-SQL, в езиците от .NET Framework. Първоначално този синтаксис ще се поддържа от C# и Visual Basic, но се очаква да бъде въведен и в останалите езици на по-късен етап.

LINQ дефинира стандартни оператори за заявки, които позволяват на езиците с поддръжка на LINQ, да филтрират, изброяват и създават проекции на няколко типа колекции, като използват единен синтаксис. Основните източници на данни към момента са масиви, изброими класове (имплементиращи интерфейсите `ICollection` и `IEnumerable`), XML, `DataSet`-ове от релационни бази данни. Както в повечето си продукти Майкрософт са осигурили API (Application Program Interface), чрез който трети страни могат да осигурят поддръжка на свои източници на данни в LINQ.

За да стане по-ясно какви са ползите от LINQ и ще направим паралел със SQL заявките.

Предполагам повечето от вас са запознати със заявките в SQL, но и да греша това няма да попречи да разберете извършваните операции.

Филтриране в SQL

Нека имаме таблица Contact в MS SQL Server със следните колони и данни:

Contact	
ID	**ContactName**
1	Galcho
2	George
3	Trifon
4	Kiril
5	Shumi

За да изведем всички записи записи от колоната ContactName, които започват започват с буквата “G” използваме следната заявка:

```
SELECT [ContactName]
FROM [Contact]
WHERE [ContactName] LIKE 'G%'
```

И като резултат получаваме:

```
Galcho
George
```

Достатъчно просто, нали?

Но това е когато изпълняваме заявките на SQL сървър. А какво се получава, когато вече сме извлекли данните от сървъра и искаме само да приложим допълнителен филтър!? Тогава вариантите са:

- Изпълняваме нова заявка към SQL сървъра заедно с новите условия за филтриране
- Пишем допълнителен код за филтриране на резултатите.

Уловката тук е, че не винаги се налага да филтрираме данни, които са взети от SQL сървър. Затова нека разгледаме филтрирането на масиви и колекции в дотук познатите ни версии на C#.

За всеки пример ще филтрираме този масив:

```
string[] contacts = { "Galcho", "George", "Trifon", "Kiril", "Shumi" };
```

като ще използваме същото условие – имената да започват с буквата 'G'

Филтриране в C# 1.1

За да върнем същият резултат е необходимо да изпълним следният код:

```
private void FilterWithCSharp1_1()
{
    string[] contacts = { "Galcho", "George", "Trifon", "Kiril", "Shumi" };

    //display result
    foreach (string name in FilterResults(contacts))
    {
        Console.WriteLine(name);
    }
}

private IEnumerable FilterResults(string[] Data)
{
    //result collection
    StringCollection results = new StringCollection();

    //filter array
    foreach (string name in Data)
    {
        if (name.StartsWith("G"))
            results.Add(name);
    }
    return results;
}
```

В този код след дефиницията на входните данни имаме само една **foreach** конструкция, която извежда последователно всеки елемент върнат от функцията **FilterResults,** на която подаваме като параметър масива с входни данни.

В нея декларираме и инициализираме променлива **results** от тип **StringCollection**.След това с помощта на **foreach** конструкция, проверяваме всеки елемент дали отговаря на

зададеното условие (започва с буква “G”) и ако отговаря го добавяме в колекцията с резултатите **results**. Накрая връщаме **results** като резултат.

Филтриране в C# 2.0

Нека да разгледаме как бихме реализирали тази функционалност със средствата, които предоставя C# 2.0

```
private void FilterWithCSharp2()
{
    string[] contacts = { "Galcho", "George", "Trifon", "Kiril", "Shumi" };

    //display result
    foreach (string name in FilterResults2(contacts))
    {
        Console.WriteLine(name);
    }
}
private IEnumerable FilterResults2(string[] Data)
{
    //filter array
    foreach (string name in Data)
    {
        if (name.StartsWith("G"))
            yield return name;
    }
}
```

В този код, подобно на предишният пример, имаме само една **foreach** конструкция, която извежда последователно всеки елемент върнат от функцията **FilterResults2,** на която подаваме като параметър масива с входни данни.

Тънкият момент е във функцията **FilterResults2**. Там се осъществява филтрирането на елементите и ако отговарят на условието се изпълнява оператора **yield return**, който връща управлението на извикващата функция за всеки запис по отделно.

Забележка: Препоръчвам да обходите двата примера в Debug Mode и стъпка по стъпка и да обърнете внимание, че във функцията **FilterResults** първо се обхождат всички елементи, а след това започва отпечатването на екран. Както посочихме във функцията **FilterResults2** не е така. Това оказва голямо влияние при обработка на големи колекции, тъй не е необходимо да се изчаква обработката на всички елементи преди да продължи работата.

Повече за оператора **yield** може да прочетете на адрес http://msdn2.microsoft.com/en-us/library/9k7k7cf0.aspx

Филтриране с LINQ и C# 3.0

Как става това в C# 3.0? Нека разгледаме следващият код:

```
 private static void FilterWithCSharp3() {
    string[] contacts = { "Galcho", "George", "Trifon", "Kiril", "Shumi" };

    var result = from s in contacts
                 where s.StartsWith("G")
                 select s;

    //display result
    foreach (string name in result) {
        Console.WriteLine(name);
    }
}
```

След първоначално необичайния синтасис изглежда много елегантно, нали?! Много прилича на SQL заявката.

Същият резултат може да бъде постигнат и по алтернативен начин като използваме новите методи дефинирани в LINQ асемблитата.

```
private static void FilterWithCSharp3_2() {
    string[] contacts = { "Galcho", "George", "Trifon", "Kiril", "Shumi" };

    var result = contacts.Where<string>(x => x.StartsWith("G"));

    //display result
    foreach (string name in result) {
        Console.WriteLine(name);
    }
}
```

По този начин става дори още по-елегантно.

Този пример е достатъчно кратък, за да покаже измененията в синтаксиса, обема код и читаемостта. Показахме само използването само на един от операторите за заявки – **`where`**.

Нека да направим нещата малко по-сложни и да групираме данните.

Групиране с LINQ и C# 3.0

За да групираме данни ще имаме нужда от по-сложни данни. Нека да дефинираме клас **`Employee`** по следният начин:

```
public class Employee {
    public string Name;
    public string Department;
    public double Salary;
}
```

Обърнете внимание, че не дефинираме конструктор и компилатора добавя конструктор по подразбиране (без параметри).

Забележка: В реални проекти е добре да дефинираме свойства, които да достъпват тези полета, но с цел опростяване на кода сега ги пропускаме.

Дефинираме и инициализираме колекция от обекти от тип **`Employee`** със следният код:

```
Employee[] employees = {
   new Employee{Name="Joe", Department="IT", Salary=1800d},
   new Employee{Name="Galcho", Department="IT", Salary=2000d},
   new Employee{Name="Jana", Department="Sales", Salary=900d},
   new Employee{Name="Schumacher", Department="Motor Sport", Salary=1000000d},
   new Employee{Name="Mary", Department="Sales", Salary=700d},
   new Employee{Name="July", Department="Marketing", Salary=2800d},
};
```

В следващият код ще групираме служителите според техният отдел и за да направим нещата по-интересни, ще сумираме заплатите на всички служители в отдела. Вече не е толкова лесно да го направим със средствата на C# 2.0, нали?!

```
var result = from e in employees
             group e by e.Department into g
             select new {
                        Department = g.Key,
                        Employees = g,
                        SumSalaries = g.Sum(e => e.Salary)};

//display result
foreach (var dept in result) {
    //display department name
    Console.WriteLine("Department Name:{0} Sum of Salaries:{1}",
                          dept.Department, dept.SumSalaries);

    //display employees in current department
    foreach (Employee empl in dept.Employees) {
        Console.CursorLeft = 4;
        Console.WriteLine("Name:{0},   Salary:{1}", empl.Name, empl.Salary);
    }
}
```

В тази заявка се изпълняват две основни функции:

- Групиране на служителите по отдели и
- Сумиране на заплатите на служителите за всеки отдел.

Като резултат се връща колекция от нов тип обекти със свойства:

- **`Department`** - текстов низ с името на отдела
- **`Employees`** - колекция от обекти тип със служителите за конкретният отдел
- **`SumSalaries`** - число съдържащо сумите на заплатите на служителите в отдела

Понеже имаме две (вложени) колекции, за да визуализираме резултата трябва да реализираме два цикъла (също вложени). И ето какъв е изхода на екран:

```
Department Name:IT          Sum of Salaries:3800
    Name:Joe,    Salary:1800
    Name:Galcho,    Salary:2000
Department Name:Sales          Sum of Salaries:1600
    Name:Jana,    Salary:900
    Name:Mary,    Salary:700
Department Name:MotoSport          Sum of Salaries:1000000
    Name:Schumacher,    Salary:1000000
Department Name:Marketing          Sum of Salaries:2800
    Name:July,    Salary:2800
```

Съвет: Преминете през кода стъпка по стъпка като проверявате състоянието на променливите. По този начин ще вникнете в детайли какво става на всеки ред от кода.

До момента разгледахме два от операторите за заявки - **`where`** и **`group by`**, А има още достатъчно много (**`order`**, **`join`**, **`union`**, **`distinct`**, **`except`**, **`intersect`**), чието имплементиране в .NET 1.1 и .NET 2.0 не е толкова лесно. Тяхното използване ще покажем по-нататък в това ръководство. Нека сега да се върнем малко назад и да разгледаме разширенията в синтаксиса на C#, които правят LINQ възможно.

Проекти с примерите
Проект за VS 2005 + LINQ May 2006 Preview (301KB)
Проект за VS 2008 Beta 2 (34.3KB)

Инициализиране на обекти и колекции

Преди да започнем да използваме обектите трябва да създадем инстанция и да ги инициализираме. Разбира се, че от това няма как да избягаме ако искаме да следваме добър стил и да създадем функциониращо приложение.

Да приемем, че сме направили дизайна на класовете, с които ще работи приложението, написали сме член променливите, обвили сме ги в свойства (properties) с подходящи **`get`** и **`set`** методи.

Нека да разгледаме декларацията на следният клас

```
public class Customer
{
    private int id;
    public int Id
    {
        get { return id; }
        set { id = value; }
    }

    private string name;
    public string Name
    {
```

```
        get { return name; }
        set { name = value; }
    }

    private string country;
    public string Country
    {
        get { return country; }
        set { country = value; }
    }

    private string city;
    public string City
    {
        get { return city; }
        set { city = value; }
    }

    private string address;
    public string Address
    {
        get { return address; }
        set { address = value; }
    }

    public Customer() { }

}
```

В този клас са декларирани пет член-променливи със съответните им свойства. Класът вече може да бъде използван в приложението, но инициализацята му няма да е много удобна. За да зададем стойности на всички свойства трябва да:

- Създадем инстанция на класа извиквайки конструктура му и
- Присвоим стойност на всяко свойство поотделно.

Това цялото действие с този конструктор ще заеме 6 реда и не е много елегантно решение (особено с нововъведенията в C# 3.0). Като възможност може да разширим конструктора да приема като параметри стойностите на свойствата, които искаме да зададем при създаването на обекта. Не трябва да забравяме основните предназначения на конструктора:

- Създава инстанция на обекта, като инициализира променливи и ги подготвя за работа.
- Приема като параметри минимум информация, която е необходима за създаване на обекта. Например за при създаване на обект от тип **`Customer`** може подадем **`ID`** или **`Name`**, в зависимост от изискванията към този бизнес обект. Няма логика, обаче, напишем конструктор, който за да приема като параметър само стойност за свойството **`Address`**, тъй като то не може да служи за уникално характеризиране на обекта. (Разбира има и изключения, когато се налага използването на подобен подход, но трябва да се използва внимателно)

Посоченият пример е сравнително малък, но ако разгледаме клас със 20 свойства и по-сложна бизнес логика ще видим, че трудно може да се създадат конструктори, които да обхващат всички възможни начини на инициализация.

Усложнение, което носят множеството конструктори е, че всеки един от тях съдържа логика по инициализиране на обекта, която е различна в зависимост от параметрите, но и много подобна, защото става въпрос за един и същи тип. Това може да довете до сложно навързване на конструкторите и повтаряне на програмен код, което както знаем е индикация на лошо построяване на кода и лош дизайн.

Точно по тези причини не може напълно да избягаме от шаблона:

```
Customer cust = new Customer();
cust.Id = 1;
cust.Name = "John Atanasov";
```

Този шаблон носи още по-голямо усложнение, когато трябва да инициализираме колекция от такива обекти. В такъв случай следват дълги и монотонни конструкции като горният пример.

За да решим този проблем може да използваме новите изрази за инициализиране на обекти в C# 3.0. Така имаме възможност за създадем обект и да инициализираме свойствата му със следният код:

```
Customer cust = new Customer { Id=1, Name="John Atanasov" };
```

Нека да видим какво става зад кулисите. За целта ще използваме популярният декомпилатор Reflector, за да видим кода до който се компилира приложението. (Компилаторите на .NET езиците компилират кода до Common Intermediate Language – CIL, а използва CIL, за да генерира C# код).

За последният обект компилатора (от LINQ Preview (May 2006)) е генерирал следният код:

```
Program.Customer customer2 = new Program.Customer();
customer2.Id = 1;
customer2.Name = "John Atanasov";
Program.Customer customer1 = customer2;
```

Както се вижда компилатора създава временен обект ,чиито свойства се инициализират по стандартният начин и после временният обект се присвоява на обекта, който сме дефинирани (последното би се видяло при използване на обекта). Обърнете внимание, че имената на променливите не отговарят на дефинираните в сорс кода, но това е един от недостатъците на декомпилаторите.

Освен, че е с по-малко код постигаме същият резултат каква е практическата полза от тази конструкция?

Най-голямото предимство при този начин на инициализиране на обектите се вижда при инициализация на колекции.

```
List<Customer> customers = new List<Customer>{
    new Customer { Id=1, Name="John Atanasov" },
    new Customer { Id=2, Name="Galin Iliev" },
    new Customer { Id=3, Name="Bill Gates", Country="USA", Address="Seattle" }
};
```

Така елегантно може да се инициализира колекция от обекти. Може да си представите как би изглеждал кода за инициализация по стандартния начин (без да се променя декларацията на класа `Customer`).

Обърнете внимание на последния обект от тип `Customer` - при него са зададени стойности на други две допълнителни свойства. Това показва,че може произволно да избираме на кои свойства да задаваме стойности.

Подразбиране на типа на променливите

Предполагам, че забелязахте в частта Филтриране с LINQ и C# 3.0, че използвахме ключовата дума `var` вместо познатите ни типове. В тази и последващата част, че обясним подробно защо.

Като разработчици на C# знаем, че променливи се декларират по този начин

```
<име на типа>  <име на променлива> [ = инициализация];
```

Където:

- **<име на типа>** - тип на променливата (`string, int, SqlCommand` или друго име на клас), който показва какви данни ще се съдържат в декларираната променлива
- **<име на променлива>** - наименование на променливата, което ще използваме по-късно в кода. Има много литература, които описват как трябва да се кръщават променливите и всички се съгласяват, че името трябва да е показва значението на данните, които се съхраняват в променливата. (Пример за добри имена – `customerDiscount, databaseConnection`. Лоши имена – `alabala, rf0j;`)
- **[= инициализация]** – това е частта, където се извикват конструкторите на класовете или данните на променливите.

Както, сигурно, се досещате ключовата дума `var` замества името на типа. Това означава, че променливите декларирани по този начин:

```
var text1 = "This is text. ";
var num =  12;
```

са идентични на променливите декларирани по познатият ни начин:

```
string text1 = "This is text. ";
int num =  12;
```

Тук по-напредналите читатели може би си задават вълна от въпроси: Не е ли същото ако вместо `var` използваме `object`? Или Variant за дните на VB6 и COM? А какво стана с идеята за силно типизираните променливи?

Отговорите на тези въпроси ще потърсим с вече познатият ни инструмент Reflector. Нека да декомпилираме асемблии, в което сме декларирали променливи с **var**. За да направим нещата по-интересни ще добавим декларация на клас:

```
var text1 = "This is text. ";
var num =  12;
var connection = new SqlConnection("connection string here");
```

След компилацията (с компилатора от LINQ Preview (May 2006)) и разглеждане на асемблито с Reflector, виждаме следното:

```
string text1 = "This is text. ";
int num =  12;
SqlConnection connection = new SqlConnection("connection string here");
```

Интересно... Както бихме го написали и в C#2.0. Това означава, че компилатора разпознава данните и поставя правилният тип. Това действия не е по време на изпълнение (за разлика от **Variant**) или по време на създаване на приложението (design time), а по време на компилация.

По тази причина, когато се използва var променливите трябва да се инициализират на същият ред. В противен случай компилаторът няма да има представа за данните, които ще се съхраняват. Тази малка хитрина само привидно размива идеята за силно типизирани променливи.

Както изглежда тази нова езикова конструкция въвежда голямо улеснение за разработчиците. Ако си представим метод с двадесетина променливи, декларирани с **var**, ще е трудно да запомним типа на всяка една от тях, а и няма да може да го прочетем ведната, а трябва да погледнем и инициализиращата секция. Това би направило кода нечетим.

Декларирането на променливи с ключовата дума `var` трябва да става само за Анонимни типове и колекции връщани от методите на LINQ. Ненужното използване може да довете до намаляване качеството на програмният код и последваща промяна (maintenance).

На пръв поглед тази малка ключова думичка има много малък принос. Но това е само на пръв поглед, както ще видим в следващата част – Анонимни типове.

Анонимни типове

Характерно за програмните езици до момента е следният начин за работа с данни:

- Дефиниране на типовете данни – деклариране на класове, структури
- Инициализиране на променлива от тип дефиниран в предходната точка
- Използване на данните – попълване, четене, промяна на свойствата на обектите.

По този начин се създават обектите в т.нар бизнес слой (business layer). Както винаги, обаче, нещата не спират до тук и следва не малко усложнение – данните се съхраняват в отделен слой (SQL Server), който използва различни начини за съхранение на данните. Това налага свързване

между данните в слоя за съхранение (data tier) с бизнес слоя. (По-късно ще разгледаме подробно как се осъществява това свързване с помощта на C# 3.0 и Visual Studio 2008).

Не бива да се пренегрегва момента, в който се налага да се променят използваните данни. Тогава се налага промяна на дефинираните класове и кода, където се използват те. А ако са свързани със слой за съхранение, то тогава се налагат промени и в него.

Забележка: По-често в практиката промените в бизнес слоя се налагат заради промени слоя с данни, но за целта на абстракцията разглеждаме двата варианта – с и без слой за съхранение.

Както може да се предположи, изискването за някаква промяна в данните на приложението изисква множество промени в кода. Докато не може да се избяга (все още) от промяната на потребителския интерфейс, в зависимост от данните, то анонимните типове предоставят удобен и лесен начин за работа с данни, които често се променят.

За момента ще представим как се постига това, а практическата полза ще излезе наяве, когато разглеждаме LINQ в детайли.

Нека се върнем на предишният пример с класа **Customer**. С помощта на новите езикови конструкции може да се инициализира по този начин:

```
Customer cust = new Customer { Id=1, Name="John Atanasov" };
```

Това улеснява много инициализацията на обекта, но все още изисква декларация на класа **Customer**. Ако искаме да използваме данни за служител без да дефинираме клас за него това може да стане така:

```
var employee = new {Id=15, Name="Galin", Experience = 8, ManagerId=2};
```

и може да използваме така създадената и инициализирана променлива в същият метод:

```
Console.WriteLine(employee.Name);
```

или по начин, който намерим за добре. Това е изключително удобно в LINQ заявки, които сортират, групират и връщат нови структури от данни. Ако трябва да се дефинират предварително всички класове, това би отнело много време. (виж примера от Групиране с LINQ и C# 3.0)

Нека да навлезем в дълбините на тази функционалност и да видим как това става възможно. Отново използваме Reflector и при разглеждане на компилираното асембли откриваме следният клас:

```
[CompilerGenerated]
public sealed class <Projection>f__0
{
	// Methods
	public <Projection>f__0();
	public override bool Equals(object);
	public override int GetHashCode();
	public override string ToString();

	// Properties
	public int Experience { get; set; }
	public int Id { get; set; }
	public int ManagerId { get; set; }
	public string Name { get; set; }

	// Fields
	private int _Experience;
	private int _Id;
	private int _ManagerId;
	private string _Name;
}
```

Вече става по-ясно, нали? Както изглежда отново компилатора върши неприятната част от работата, като генерира клас, в зависимост от данните/свойствата от които имаме нужда. По този начин, когато се наложи да променяме класа, просто трябва да променим инициализацията (и използването) на класа, но не трябва да се грижим да промяна на дефиницията.

Компилатора има и механизъм за оптимизация, който използва вече генерираният клас ако използваме две променливи с еднакви свойства. Ако променим свойствата, обаче, нов анонимен клас се генерира.

Автоматично изграждане на свойства

Едно от най-новите синтактични подобрения е автоматичното изграждане на свойства. Въпреки, че това не е радикално нововъведение е подходящо да му отделим малко внимание.

Когато създаваме класове добрият стил изисква всички полета/променливи да са капсулирани и маркирани като частни или защитени (`private/protected`). Ако съдържанието на някое поле трябва да бъде използвано от външни за класа методи, то трябва да се обвие в свойство. За свойства, които имат сложни getters и setters (напр. Преди са се върне стойността на полето се създава запис в лога) това е удобен начин да построим класа, но за останалите свойства е малко досадно да създаваме множество от подобни двойки поле/свойство:

```
private string fullName;
public string FullName
{
    get { return fullName; }
    set { fullName = value; }
}
```

Последното въведение в C# 3.0 позволява да съкратим кода, който пишем до:

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

и вече в методите на класа (а и извън тях) може да достъпваме декларираното свойство. По разбираеми причини вече не може да се объркаме и да използваме полето вместо свойството, както беше възможно в по-горният случай.

Нека обаче да погледнем какво е генерирал компилатора от този код :

```
[CompilerGenerated]
private string <>k__AutomaticallyGeneratedPropertyField0;

public string FullName
{
    [CompilerGenerated]
    get
    {
        return this.<>k__AutomaticallyGeneratedPropertyField0;
    }
    [CompilerGenerated]
    set
    {
        this.<>k__AutomaticallyGeneratedPropertyField0 = value;
    }
}
```

С изключение на странното име на полето, останалата част изглежда така както я бихме написали сами.

Както виждате това само по себе си не е революционно нововъведение, но то би могло да съкрати кода, който пишем оставяйки ни повече време за мислене върху по-сериозни проблеми.

Както навярно сами се досещате, това може да бъде приложено само в свойства без допълнителна логика в getters и setters.

Разширяващи методи (Extension Methods)

Разширяващите методи са една от новите възможности в C# 3.0 и позволяват разширяването на съществуващ тип с допълнителни методи без да се налага промяна в сорса на типа или прекомпилиране на асемблито, което го съдържа. Терминът разширяване в този случай означава, че разширяващите методи могат да се извикват от инстанция на разширения тип, въпреки че тези методи не са дефинирани във въпросният клас.

Разширяващите методи могат да се извикват само когато се включи пространството от имена с ключовата дума ***`using`***.

Най-широко разпространетото използване на разширяващите методи към момента е в LINQ, където се добавят допълнителни методи към типовете **`IEnumerable<T>`** и **`IQuerable<T>`**.

Разширяващите методи се дефинират като статични, но се извикват чрез инстанция на разширеният тип. Техният първи параметър показва кой тип се разширява и се предхожда от ключовата дума **`this`**. Освен това първият параметър преоставя достъп до инстанцията, която

нормално се достъпва с this. Малко объркващо звучи, но ето един пример (от документацията на VS 2008):

```
public static int WordCount(this System.String str)
{
    return str.Split(null).Length;
}
```

Единствената разлика от дефинирането на нормален статичен метод е ключовата дума **this** преди първият параметър. Това обаче позволява много удобно да извикаме разширяващият метод от инстанция на класа, който в горният случай е **string**. Този пример преброява думите в даден текстов низ. Ето как може да се използва:

```
string test = "Hello from C# tutorial";
int wordCount = test.WordCount(); // wordCount = 4
```

Разширяващите методи могат да се използват, за да се добави функционалност към клас или интерфейс, но не и да се замени функционалност (както се прави с **override**). Разширяващ метод със същото име и параметри както метод на клас/интерфейс никога няма да бъде извикан. По време на компилация, разширяващите методи имат по-нисък приоритет от методите, които са дефинирани в самият тип. Ако типа не дефинира метод, който се извиква, компилатора търси разширяващ метод с подадените параметри и използва първият, който бъде намерен. За да разберем по-подробно по какъв начин компилатора решава кой метод да бъде използван ще разгледаме следващият пример.

Нека да дефинираме интерфейс **IMyInterface**, който дефинира един метод **MethodB**. След това дефинираме три класа, който имплементират посоченият метод по различен начин и два от тях дефинират допълнителен метод **MethodA** с различни параметри.

```
public interface IMyInterface
{
    void MethodB();
}

class A : IMyInterface
{
    public void MethodB() { Console.WriteLine("A.MethodB()"); }
}
class B : IMyInterface
{
    public void MethodB() { Console.WriteLine("B.MethodB()"); }
    public void MethodA(int i) { Console.WriteLine("B.MethodA(int i)"); }
}
class C : IMyInterface
{
    public void MethodB() { Console.WriteLine("C.MethodB()"); }
    public void MethodA(object obj) { Console.WriteLine("C.MethodA(object
obj)"); }
}
```

Класът **B** дефинира **MethodA**, който приема един параметър от тип **int**, а класът **C** дефинира **MethodA**, който приема един параметър от тип **object**. За да разберем кои методи имат по-голям приоритет ще дефинираме три разширяващи метода за интерфейса **IMyInterface**:

```
// Define extension methods for any type that implements IMyInterface.
public static class Extensions
{
    public static void MethodA(this IMyInterface myInterface, int i)
    {
        Console.WriteLine("Extension.MethodA(this IMyInterface myInterface,
int i)");
    }
    public static void MethodA(this IMyInterface myInterface, string s)
    {
        Console.WriteLine("Extension.MethodA(this IMyInterface myInterface,
string s)");
    }
    // This method is never called, because the three classes
    public static void MethodB(this IMyInterface myInterface)
    {
        Console.WriteLine("Extension.MethodB(this IMyInterface myInterface,
int i)");
    }
}
```

Първият метод - **MethodA** – разширява интерфейса **IMyInterface** като изисква един параметър от тип **int**. Обърнете внимание, че класът **A** не дефинира метод **MethodA**;класът **B** дефинира **MethodA** със същите параметри, а класът **C** дефинира **MethodA**,който приема параметър от тип **object**.

Вторият метод - **MethodA** – разширява интерфейса **IMyInterface** като изисква един параметър от тип **string**. Забележете, че никой от класовете не дефинира **MethodA** с такъв параметър.

Третият метод - **MethodB** – разширява интерфейса **IMyInterface** като не изисква параметри за да се извика. Важното тук е, че интерфейса и всички класове имат метод **MethodB**, който не приема параметри.

Нека да изпълним един малък пример, за да видим кои методи се изпълняват:

```
static void Main(string[] args)
{
    A a = new A();
    B b = new B();
    C c = new C();
    TestMethodBinding(a, b, c);
}

static void TestMethodBinding(A a, B b, C c)
{
    // A has no methods, so each call resolves to
    // the extension methods whose signatures match.
    a.MethodA(1);  // Extension.MethodA(object, int)
    a.MethodA("hello");  // Extension.MethodA(object, string)
    a.MethodB();

    // B itself has a method with this signature.
    b.MethodA(1);  // B.MethodA(int)
    b.MethodB();

    // B has no matching method, but E does.
    b.MethodA("hello");  // Extension.MethodA(object, string)

    // In each case C has a matching instance method.
    c.MethodA(1);  // C.MethodA(object)
    c.MethodA("hello");  // C.MethodA(object)
    c.MethodB();
}
```

В този пример създаваме три инстанции съответно на класовете **A**, **B**, **C** и ги подаваме на метода **TestMethodBinding**, който извиква съответните методи на всеки клас.

За класът **A**:

- Всяко извикване на **MethodA** се свързва с дефинираните разширяващи методи, защото клас A не дефинира **MethodA**.
- Извикването на **MethodB** се изпълнява от дефинираният метод в тялото на класа.

За класът **B**:

- Извикването на **MethodA** с параметър **1** се изпълнява от дефинираният метод в тялото на класа.
- Извикването на **MethodB** също се изпълнява от дефинираният метод в тялото на класа.
- Извикването на **MethodA** с параметър `"hello"` се изпълнява от разширяващият метод, защото няма подходящ метод в тялото на класа, който да приема параметър от този тип.

За класът **C**:

- Извикването на **MethodA** с параметър **1** се изпълнява от дефинираният метод в тялото на класа, защото типът **int** може да се преобразува към **object**.
- Извикването на **MethodA** с параметър `"hello"` се изпълнява изпълнява от дефинираният метод в тялото на класа, защото типът **string** може да се преобразува към **object**.

- Извикването на **MethodB** също се изпълнява от дефинираният метод в тялото на класа.

Извод: Разширяващите методи имат най-нисък приоритет и се изпълняват само ако няма метод със същото име в тялото на разширяваният клас/интрефейс и същите параметри или параметрите не могат да се преубразуват до изискваните.

Разширяващите методи са мощен инструмент за добавяне на функционалност към компилирани обекти, но трябва са се използва внимателно.

Основното предимство се разкрива в следната ситуация: Нека си представив, че нашата компания използва външни библиотеки (3rd party libraries), но те нямат малка функционалност, от която ние имаме нужда. Да приемем, че разполагаме със сорс кода на библиотеката. В този случай можем да променим сорса в съответствие с нашите нужди (и в зависимост от лиценза) и да прекомпилираме библиотеката. Дотук добре, но при следваща версия на библиотеката промените, които сме направили трябва да се прилагат отново. А това може да отнеме същото време както първоначалното модифициране. В този случай е много удачно да използваме разширяващи медоти.

Ламбда изрази

Ламбда изразите са естественото продължение на анонимните методи, които бяха въведени в C# 2.0. За да проследим развитието ще започнем от самото начало като обясним накартко делегатите.

Още с първата си версия C# предостави специален вид променлива, която сочи към методи – делегатите. Това позволява делегати да се предават като параметри, да се променят и да се извиква сочената функция. Използването на делегати позволява елегантно промяна на програмния поток в зависимост от функциите, към които сочат.

Делегати

В следващият пример са извършени следните операции:

- Дефиниране на делегат **Operation**, който приема два параметъра.
- Реализиран метод **Multiply** със същите параметри като делегата **Operation**.
- В метода се създава променлива **deleg**, сочеща към инстанция на делегата **Operation**, която сочи към метода **Multiply**. Променливата **deleg** се извиква като нормален метод.

```
public delegate long Operation(long first, int second);

static long Multiply(long i, int j){
    return i * j;
}

static void Main(string[] args){
    Operation deleg = new Operation(Multiply);
    long res = deleg(12, 13);
}
```

Анонимни методи

С помощта на анонмните методи, които бяха въведени в C# 2.0 горният код може да се съкрати до.

```
public delegate long Operation(long first, int second);

static void Main(string[] args){
    deleg2 = (Operation)delegate(long i, int j) { return i * j; };
    long res = deleg2(12, 13);
}
```

Разликата от предишният код е, че тук няма метод **Multiply**, а е осъществен без да е дадено изрично име на метода. Предимствата на анонимните методи са:

- Класът остава стегнат и не се разпокъсва от малки **private** методи.
- Кодът се поставя там, където се използва вместо някъде другаде в класа.
- Не трябва да се измисля подходящо име на метода, но трябва да се поставят подходящи коментари.
- Не се указва тип на резултата – той се подразбира.
- Може да се използват променливи от външният метод.

От анонимни методи към ламбда изрази

Ламбда изразите правят синтаксиса още по сбит. Прототипът е следният:

```
(Списък с параметри) => тяло на метода
```

Типът на параметрите не е задължителен и може да се пропусне:

```
(x, string s) => s.Length > x
```

Ако тялото на метода съдържа повече от един ред тогава цялото тяло се огражда с {}.

Използвайки лабда изразите горният пример изглежда така:

```
static void Main(string[] args){
    Operation deleg3 = (long i, int j) => {return i * j; };
    long res = deleg2(14, 13);
}
```

Правила за използване на ламбда изрази:

- Скобите на списъка с параметри може да се пропусне, ако параметъра е един - **n => n<10**, освен ако не се указва типа на параметрите – **(string n) => n<10**.
- Скобите на списъка с параметри е задължителен ако параметрите са повече от един или няма входни параметри.
- Може да се достъпват променливи от външният метод също като при анонимните методи.

Ламбда изразите са много често използвани в LINQ, както ще видим по късно в това ръководство. За да покажем силата на тази конструкция ще разгледаме следният код:

```
List<int> numbers = (new int[] { 1, 2, 3, 4, 5, 21, 22, 14, 16, 18
}).ToList<int>();

//print all the numbers in the list to the console
numbers.ForEach(n => Console.WriteLine(n));
```

В първият ред инициализираме масив от целочислени числа и го конвертираме към `List<int>`. На следващият ред подаваме на разширяващият метод **`ForEach`** ламбда израз, който е извикван за всеки елемент от списъка. Като резултат в конзолата се отпечатват всички елементи на списъка.

Дървета от изрази (expression trees)

Както видяхме в предишната точка, ламбда изразите са разширение на анонимните методи. А анонимните методи, като всеки метод, могат да се сведат до променлива чрез използване на делегати. В този ред на мисли, ако имаме променлива, която представлява метод, то съответно можем да имаме и масиви от методи. А защо не и колекции от методи (делегати)!?

По този начин може да се зададе път на обработка на определени данни по време на изпълнение на програмата. А след като имаме масиви и колекции от методи, защо да не може да се построи дърво от методи!? Всъщност точно това представляват дърветата от изрази: познатото дърво, като структура от данни, но вместо данни разполагаме с методи.

Както повечето от вас вече сигурно се досещат след като стигнахме до дърветата, върху тях може да приложим всички алгоритми, оптимизации, обхождания и др., които съществуват в теорията и практиката. (Това е добра причина изтърскаме праха от основните алгоритми за работа в дървета, с които ни занимаваха в университета ☺)

LINQ предлага клас, с който методите вече се третират като данни от компилатора. Например ако напишем следният код:

```
public delegate long Operation(long first, int second);

private void Test ()
{
    Operation op = (x,y) => x+y;
    long res = op.Invoke(2,5);
}
```

То компилаторът ще генерира:

```
public delegate long Operation(long first, int second);

private void Test()
{
    Operation op = delegate (long x, int y) {
        return x + y;
    };
    long res = op((long) 2, 5);
}
```

Това изглежда само като синтактично подобрение. Ако обаче използваме класът **Expression<T>** (**System.Expressions.Expression<T>** от асемблито **System.Query.dll**), за да обвием метода и напишем:

```
static Expression<Func<int, int, long>> sumExpr = ( x, y) => x + y;
```

То след компилация ще имаме:

```
private static Expression<Func<int, int, long>> sumExpr;

static Form1()
{
    ParameterExpression x;
    ParameterExpression y;
    sumExpr = Expression.Lambda<Func<int, int, long>>(
        Expression.Convert(
            Expression.Add(
                x = Expression.Parameter(typeof(int), "x"),
                y = Expression.Parameter(typeof(int), "y")),
            typeof(long)),
            new ParameterExpression[] { x, y });
}
```

Малко по-различно от горният пример, нали!? Нека да обясним какво написахме, та компилатора го разбра по такъв начин.

Нека да започнем с типа, който подахме на **Expression<T>: Func<int, int, long>**. След като имаме структури от данни, които всъщност съдържат методи, то трябва по някакъв начин да разбираме методите, които имаме – трябва да знаем броя на параметрите, които се приемат, техният тип, и типа на връщаният резултат.

За целите на LINQ в асемблито **System.Query.dll** (пространството от имена **System.Query)** има дефинирани следните делегати:

```
public delegate T Func<T>()

public delegate T Func<A0, T>(A0 arg0)

public delegate T Func<A0, A1, T>(A0 arg0, A1 arg1)

public delegate T Func<A0, A1, A2, T>(A0 arg0, A1 arg1, A2 arg2)

public delegate T Func<A0, A1, A2, A3, T>(A0 arg0, A1 arg1, A2 arg2, A3 arg3)
```

Както забелязвате във триъгълните скоби са типовете на параметрите (**A0, A1,**…), последният е типа на връщаният резултат. Изглежда това са дефинициите на най-използваните методи, но това не означава, че не може да подадете на **Expression<T>** делегат от ваш тип.

Нека да разгледаме какво се случва в статичният конструктор (това е генериран от компилатора код):

- На първите два реда се декларират параметрите (**x** и **y**) от тип **ParameterExpression**.

- На следващият ред се задава израз от тип **`Expression.Lambda<Func<int, int, long>>`**. За да разберем същността на това действие, ще го разгледаме от вътре навън:
    - Добавя се нов израз от тип **`BinaryExpression`** чрез **`Expression.Add`**, като се задават параметрите и техният тип.
    - След това с помощта на метода **`Expression.Convert expr1`** се конвертира до обект от тип **`Expression`** като се задава и тип на резултата.
    - И накрая чрез **`Expression.Lambda<Func<int, int, long>>`** се създава обект от тип **`LambdaExpression`**, като се подават инстанциите на създадените параметри.

За да е по-лесно за четене същият код може да се престави и така:

```
ParameterExpression x;
ParameterExpression y;

BinaryExpression expr1 = Expression.Add(
            x = Expression.Parameter(typeof(int), "x"),
            y = Expression.Parameter(typeof(int), "y"));

Expression expr2 = Expression.Convert(expr1,typeof(long));

sumExpr1 = Expression.Lambda<Func<int, int, long>>(
    expr2, new ParameterExpression[] { x, y });
```

И накрая изразите могат да се използват по следният начин:

```
Func<int, int, long> func = sumExpr.Compile();
long res = func(3,5);
```

Както, може би, се досещате дърветата от изрази могат да се използват за динамично генериране на изпълним код.

В тази точка не разгледахме конкретни практически примери, но дърветата от изрази са важна част от LINQ и C# 3.0 и без тях голяма част от нововъведенията не биха били възможни. При работа с голяма част от разширяващите методи и LINQ to SQL, LINQ to Object и др. може да забележим, че заявките се преубразуват от дървета от изрази, които се изпълняват в момента на извикване.

LINQ в детайли

След като разгледахме в детайли синтактичните допълнения към C# във версия 3.0 е време да разгледаме в детайли как може да се построяват заявки в C# - или с други думи казано – да

разгледаме LINQ. Преди да започнем нека да направим малко уточнение – LINQ синтаксиса е само едно улеснение с цел прегледност на кода и по-лесна читаемост. LINQ използва основно разширяващи методи, ламбда изрази и дървета от изрази, за да се постигне желаната фунционалност. За да стане по-ясно нека да разгледаме малко код – нека се върнем малко назад и да разгледаме в детайли кода от точката Филтриране с LINQ и C# 3.0. Както казахме тогава кода от методите **FilterWithCSharp3** и **FilterWithCSharp3_2** извършват идентични действия. Това е възможно, понеже компилаторът взема синтаксиса и го конвертира към код, извикващ методи и използващ разширяващи методи, ламбда изрази и дървета от изрази.

Всяка LINQ заявка започва с **from** и завършва със **select** или **group**, като **from** клаузата указва източника на данни, а **select** указва резултата (както и формата/типа на резулата).

Нека да разгледаме заявката от **FilterWithCSharp3** в детайли:

```
var result = from s in contacts
             where s.StartsWith("G")
             select s;
```

На първият ред се указва, че ще се използва елементите от променливата **contacts** (която трябва да бъде от тип **IEnumerable**) за източник на данни. При обработката на елементите всеки от тях ще бъде достъпен през променливата **s**, като това е произволно избрано – може да бъде **t, m, f** или всяка друга незаета променлива (като правило се е наложило променливите към елементите да бъдат кратки, за да не се бъркат с типовете или колекциите). Или накратко първият ред означава – за всеки елемент **s** в колекцията **contacts**….

На вторият ред е филтъра, както подсказва ключовата дума **where**. След нея се поставя условие, резултата от което трябва да е от тип **bool**. Могат да се поставят множество условия като се използват операторите за логически операции || и &&.

На последният ред опрераторът **select** връща обект като резултат. Не е задължително да бъде същият елемент, който е взет от входящите данни – може да бъде свойство на елемента (написано като “**select e.Name;**” ако обекта е от тип **Employee** – виж точка Групиране с LINQ и C# 3.0) или изцяло нов обект от анонимен тип.

За тези, които имат опит със SQL този начин на изписване едва ли изглежда много различно и привикването към него е изключително лесно. Още повече, че Visual Studio предоставя intellisense.

Основни оператори

Тайната на LINQ е, всъщност, в използваните оператори. Както и в SQL без операторите **where, group by, join (inner, outer…)** възможностите биха били доста ограничени. Нека да разгледаме подробно с какви оператори разполагаме в LINQ и как се работи с тях:

where

Where оператора служи за филтриране на елементите във входната колекция по зададени критерии аналогично в T-SQL заявките. Предимството тук е, че се използва C# синтаксис, за задаване на критериите, а вътрешните класове на LINQ to SQL се грижат за коректния превод към

T-SQL. Това ни позволява да изпишем критериите по същият начин, по който бихме ги изписали и в **if** условие. Ако вземем примерът по-горе и добавим още едно условие, което да връща елементите започващи с “G” и дължина по-голяма от 4 символа, то ще се получи следното:

```
var result = from s in contacts
             where s.StartsWith("G") && s.Lenght > 4
             select s;
```

Възможно е използването на разширяващия метод **Where< TSource >** дефиниран за **IEnumerable<TSource>**. Така че следващия код е аналогичен като действия, но използва разширяващи методи и ламбда изрази:

```
var result = contacts.Where<string>(s => s.StartsWith("G") && s.Length > 4);
```

Както виждаме **where** клаузата доближава C# и T-SQL изключително много.

orderby

Точно така! Вече не трябва да имплементираме **IComparer** интерфейса, за да сортираме колекция по зададен критерий. Операторът **orderby** позволява сортиране по подадени критерии аналогично на T-SQL. В следващият пример освен филтриране на входната колекция извършваме и сортиране:

```
var result = from s in contacts
             where s.StartsWith("G") && s.Lenght > 4
             orderby s ascending
             select s;
```

По този начин като резултата се сортира във възходящ ред.

За сортирането имаме нужда от допълнителни оператори, които да указват посоката на сортиране:

- **ascending** – указва сортиране във възходящ ред (A-Z, 1-5), като това е посоката по подразбиране. Ако не се укаже посоката в оператора **orderby** се използва тази посока.
- **descending** – низходящо сортиране (Z-A, 5-1)

Най-хубавото нещо тук е възможността да използваме оператора **orderby** независимо от **where**. По този начин може само да сортираме колекцията, без да я филтрираме. В примера по-долу сортираме цялата колекция в низходящ ред и я връщаме като резултат:

```
var result = from s in contacts
             orderby s ascending
             select s;
```

Възможно е също подреждане по няколко критерия, като критериите се отделят със запетая.

group

Тази функционалност – групирането – също е взаимствана от T-SQL. По своята същност, обаче, тя е малко по-различна от предходните два оператора и се нуждае от различна имплементация в LINQ. При групирането се създават два вида колекции:

- Колекция от групи
- Колекции от елементи за всяка група

По тази причина при обхождането на резултата трябва да го реализираме с помощта на два вложени цикъла. Нека да разгледаме примера, който групира данни от точка Групиране с LINQ и C# 3.0:

```
var result = from e in employees
             group e by e.Department into g
             select g;
```

Нека да разгледаме синтаксисът на командата: след оператора **group** указваме кои елементи да се групират, последвани от ключовата дума **by**, след която се указва критерят по който ще се групират елементите и след тях оператора **into** указващ променливата за групата.

Обърнете внимание, че на последния ред връщаме като резултат групата - **g**, а не елемента – **e**, както в останалите примери. Това е необходимо понеже трябва да върнем колекция от групи, което споменахме в началото. Самите елементи се намират в колекция в дадена група.

Нека да разгледаме кода за обхождане на резултата, понеже и той си има своята специфика:

```
foreach (IGrouping<string, Employee> group in result) {
    Console.WriteLine("Group:{0}", group.Key);
    foreach (Employee empl in group) {
        Console.WriteLine("   {0} {1}", empl.Name, empl.Salary);
    }

    Console.WriteLine();
}
```

Кодът се състои от два вложени цикъла – по един за всеки тип колекция- един за групите и един за елементите. Интересната част тук е типа на групата – това generic интерфейса **IGrouping<TKey, TElement>** - като първият тип указва типа на критерия, по който се групира, а вторият – типа на елементите в групата. Този интерфейс дефинира свойство **Key**, от което може да вземем стойноста на критерият за групиране за контретната група. Разбира се може да заменим този тип с вълшебният оператор **var**.

Вътрешният цикъл обхожда елементите, които се съдържат в инстанцията на **IGrouping<TKey, TElement>.** Това е възможно, защото **IGrouping<TKey, TElement>** наследява интерфейсите **IEnumerable<TElement>** и **IEnumerable** като всъщност представлява изброяем тип.

select

На този оператор следва да му се обърне малко повечко внимание, понеже той указва какво всъщност да се върне като резултат от операцията. До момента връщахме само един тип елементи в заявките, като този тип беше предварително дефиниран. С помощта на **select** оператора и анонимните типове може да върнем анонимен тип създаден от свойствата на един или повече елементи. Това става по следният начин:

```
var result = from e in employees
             select new { EmployeeName = e.Name, Department = e.Department };
```

По този начин може да върнем като резултат нов тип образуван от свойствата на елемнети от няколко колекции. В следващата точка ще покажем как **select** оператора играе важна роля при свързванията (**join**).

join

join оператора служи за обединяване/свързване на резултатите на две колекции, като свързването се осъществява по поле, което съдържа общи данни (т.нар ключ). За да опишем какви са възможностите в LINQ нека да разгледаме в T-SQL теорията какви видове свързвания съществуват:

- **inner join** – вътрешно свързване – при това свързване за всеки елемент от първата колекция се намира само един съответстващ ред във втората колекция. Елементите, които нямат съотвестващ елемент в насрещната колекция се изключват от резултата.
- **outer join** – външно свързване – при този вид свързване се връщат всички елементи от едната колекция и само съответстващите елементи. Съществуват следните подвидове:
    - **left outer join** – връщат се всички елементи от лявата (първата) колекция и само съответстващите елементи от дясната (втората) колекция.
    - **right outer join** – връщат се всички елементи от дясната (втората) колекция и само съответстващите елементи от лявата (първата) колекция.
    - **full outer join** – връщат се всички елементи от двете колекции като съответстващите елементи се засичат.
- **cross join** – пълна комбинация. При този вид свързване за всеки елемент от двете колекции се изпълнява пълно комбиниране на насрещните елементи. Например ако в първата колекция имаме 10 елемента, а 5 във втората то като резултат ще получим 50 елемента. Този вид свързване не е много често използван.

В LINQ са реализирани само най-често употребяваните видове свързвания:

- **inner join** – или в документацията на LINQ просто **join**
- **left/right outer join** – или в документацията на LINQ **group join**. Наречен е така, заради начина по-който се връща резултата – под формата на група по подобие на **group** оператора. Всъщност е реализиран само **left join**, но сменяйки местата на входните колекции може да изпълним логически и **right join**.

За да представим подходящи примери ще дефинираме нова масив от тип **Departments**. Дефиницията на класа е:

```
public class Department {
    public string Name;
    public string Building;
}
```

А самият масив е инициализиран по този начин:

```
Department[] departments = {
            new Department{Name="IT", Building="Building A"},
            new Department{Name="Sales", Building="Building B"}
            };
```

Обърнете внимание, че според колекцията дефинирана в точка Групиране с LINQ и C# 3.0 нямаме създаден елемент за отделите “Motor Sport” и “Marketing”.

C# join

Нека да разгледаме как се реализира inner join с помощта на C# синтаксис. Самото изписване е по следният начин:

```
join … in … on … equals
```

Със следващият пример ще върнем като резултат всички служители, за които има запис в масива **departments**:

```
var result = from d in departments
             join e in employees on d.Name equals e.Department
             select new { EmployeeName = e.Name, Building = d.Building };
```

Начинът на изписване на join е подобен на from, но с малко допълнения. **join e in employees** указва входяща колекция и променлива за елементите в нея; **on d.Name equals e.Department** указва ключа, по който се свързват елементите. Като резултат получаваме елементите от колекцията **departments**, за които има съответстващ елемент в **employees**, но като резултат получаваме анонимен тип, съдържащ името на служителя и съответстващата сграда, в която се помещава.Така се обединява информацията от двете колекции от входни данни, за да се получи:

```
Joe - Building A
Galcho - Building A
Jana - Building B
Mary - Building B
```

Забележете, че в резулата не са включени всички записи от колекцията **employees**.

Съответно същият резултат може да бъде получен чрез извикване на разширяващият метод **IEnumerable<T>.Join()**:

```
var result = departments.Join(employees,
    d => d.Name,
    e => e.Department,
    (d, e) => new { Building = d.Building, EmployeeName = e.Name });
```

C# group join

Изписването на **group join** е по следният начин:

```
join … in … on … equals … into …
```

Нека да преработим примерът от предишният пример в **group join**:

```
var result = from d in departments
             join e in employees on d.Name equals e.Department into emp
             select new { DepartmentName = d.Name, Building = d.Building,
Employees = emp };
```

Освен ключовите думи, които описахме в предходната точка тук срещаме още една – **into emp**. Този израз указва променливата в която ще се поставят елементите (съдържа повече от един елемент и затова е колекция) от колекцията **employees**, които отговарят на условието. След това използвайки оператора select указваме, че в резултата ще се състои от нов, анонимен тип, който съдържа три свойства:

- **DepartmentName** – който се взема от променливата **d**, сочеща към даден отдел,
- **Building** – името на сградата, отново взета от променливата **d**,
- **Employees** – и тук е интересната част: съдържа колекцията от съответстващи елементи от **employees**.

Понеже, както и при групирането имаме два вида колекции, се нуждаем от два вложени цикъла, за да обходим резултата:

```
foreach (var item in result) {
    Console.WriteLine("{0} - {1}", item.DepartmentName, item.Building);

    foreach (var em in item.Employees) {
        Console.WriteLine("   {0} - {1}", em.Name, em.Salary);
    }
}
```

И като резултат получаваме в конзолата:

```
IT - Building A
   Joe - 1800
   Galcho - 2000
Sales - Building B
   Jana - 900
   Mary - 700
```

Проекти с примерите
Проект за VS 2008 Beta 2 (33.3KB)

След като разгледахме основните операции, които могат да се изпълняват с помощта на LINQ, нека е време да се впуснем в клоновете на LINQ. Разликата в отделните клонове не е съществена – поне тази част, която е видима за нас като потребители. Всъщност вътрешната част на всеки клон е различна и може да бъде много сложна (напр при LINQ to SQL, където C# израза трябва да се трансформира в SQL заявка, като резултата да се свърже с обектите), но използването е унифицирано.

LINQ to Objects

Ще започнем с LINQ to Objects, понеже останалите клонове са много подобни.

LINQ to Objects позволява на разработчиците да изпълняват „заявки" върху колекции от обекти, като разполагат със значителен набор от възможности, които могат да се очакват от SQL заявка върху релационни бази данни.

При работа с колекции всеки програмист използва основно **`for`** и **`foreach`** цикли, за обхождане, if оператор за условие и т.н. LINQ ни освобождава от нуждата да пишем тези цикли.

Заявки може да се пишат срещу всяка колекция, която имплементира някой от следните интерфейси: **`IEnumerable`**, **`IEnumerable<T>`**, **`IQuerable`**. Това на практите означава почти всички колекции, предоставени от .NET Framework. Например ако погледнем дефиницията на популярният списък **`List<T>`** ще видим какви интерфейси имплементира:

```
public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>, IList,
ICollection, IEnumerable
```

което на практика означава, че може да се пишат заявки към **`IList<T>`**.

С това се изчерпват изискванията към колекциите. А към обектите няма никакви специфични изисквания, за да може да се работи с тях с помощта на LINQ – това може да са структури от .NET Framework или пък класове, които са дефинирани в нашите проекти – на практика всеки вид клас.

Може би е време да дадем пример, за да стане по-ясно!? Всъщност примерите, които разглеждахме до момента показват точно LINQ to Objects.

Нека все пак да разгледаме пример, в който колекцията от служители от точка Групиране с LINQ и C# 3.0 се сортират според заплатата в низходящ ред:

```
var sorted = from e in employees
             orderby e.Salary descending
             select e;

foreach (Employee e in sorted) {
    Console.WriteLine("{0}\t\t{1}", e.Name, e.Salary);
}
```

И съответно резулата е:

```
Schumacher      1000000
July            2800
Galcho          2000
Joe             1800
Jana            900
Mary            700
```

Както казахме в началото това не единственият начин на изписване. Можем да постигнем същата цел като използваме разширяващите методи на **IEnumerable<T>**. В нашият случай трябва да използваме само метода **IEnumerable<T>.OrderBy()**:

```
var sorted2 = employees.OrderBy(e => e.Salary);
```

Ако имаме и филтър освен сортирането може да се наложи да извикаме **IEnumerable<T>.Where()** и върху резултата **IEnumerable<T>.OrderBy()** последователно:

```
var sorted3 = employees.Where(e=>e.Name.StartsWith("G")).OrderBy(e =>
e.Salary);
```

Вътрешно LINQ заявката се преобразува до последователно извикване на разширяващи методи при LINQ to Objects. Затова няма разлика в резултата при двата начина на изпсиване, но LINQ заявката е по-добре читаем. При останалите клонове LINQ заявката се преобразува до подходяща заявка, в зависимост от конкретната реализация (напр. при LINQ to SQL се генерира SQL израз).

Както ще видим в следващите точки останалите клонове на LINQ се базират върху LINQ to Objects като добавят някаква функционалност.

LINQ to SQL

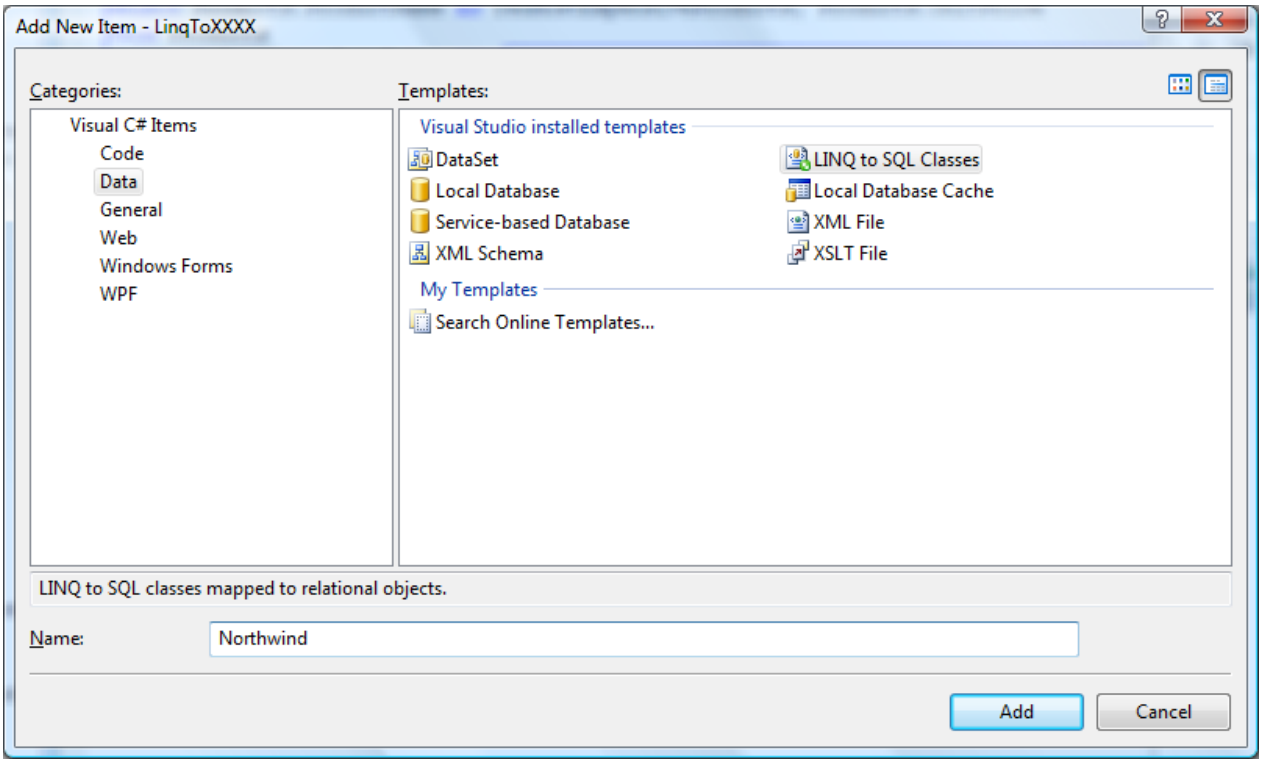
LINQ to SQL е ORM (object relational mapping), който позволява работа с таблици в релационна база данни използвайки класове в .NET. С други думи LINQ to SQL дава възможност за се пишат заявки на предпочитан език на .NET, които да манипулират данни в SQL Server.

Забележка: LINQ to SQL работи само с MS SQL Server. За да се работи с други релационни бази данни е необходим допълнителен LINQ Provider. В уеб пространството вече се говори, че съществуват такива за водещите системи за релационни бази данни.

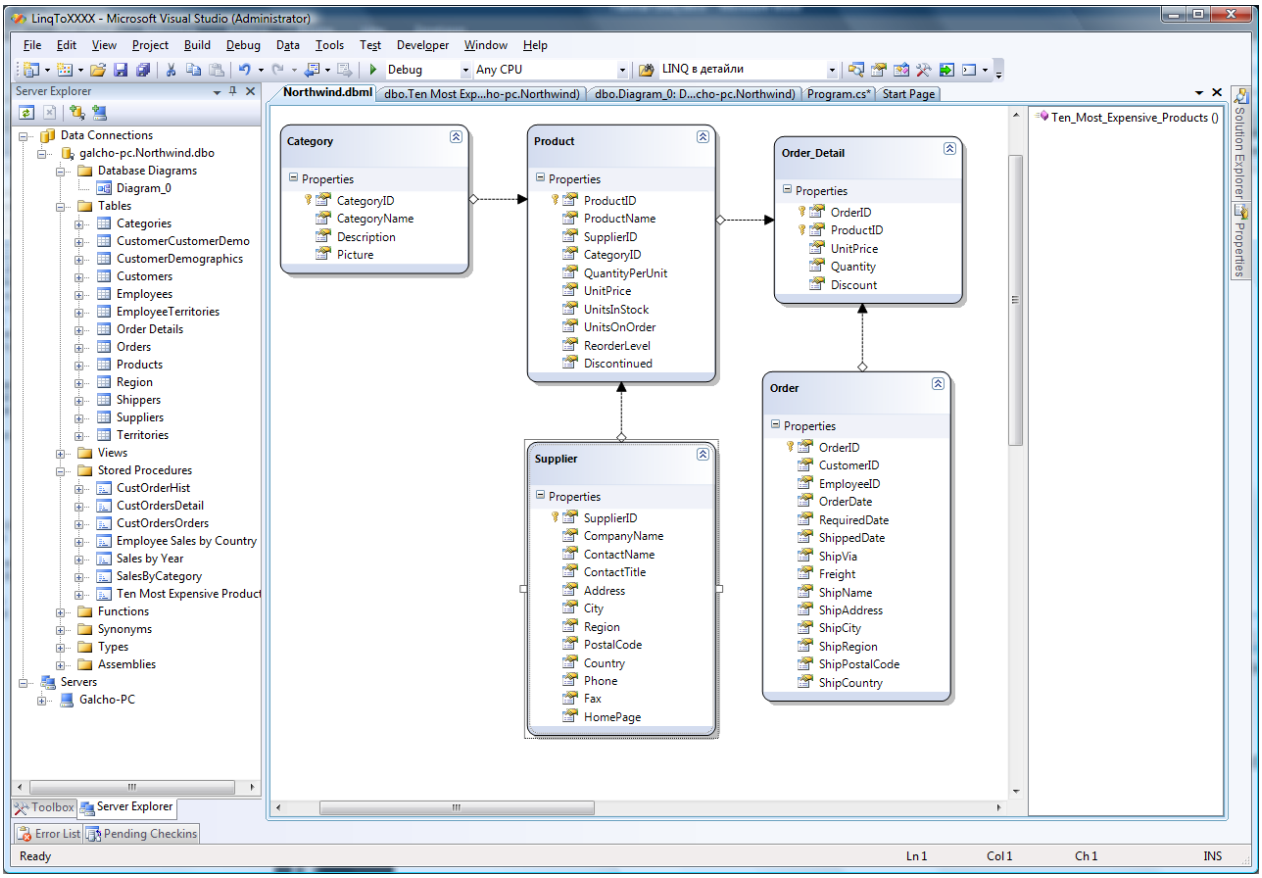
Създаване на модела на базата

За да използваме LINQ to SQL е необходимо да създадем модел на базата в нашия .NET проект. Този модел много наподобява **DataSet**, понеже в него се съдържат специфични класове, представящи конкретните таблици в базата.

1. За целта отваряме нов проект във Visual Studio 2008
2. От менюто Project -> Add New Item избираме Linq to SQL Classes и в полето Name въвеждаме “Northwind”.



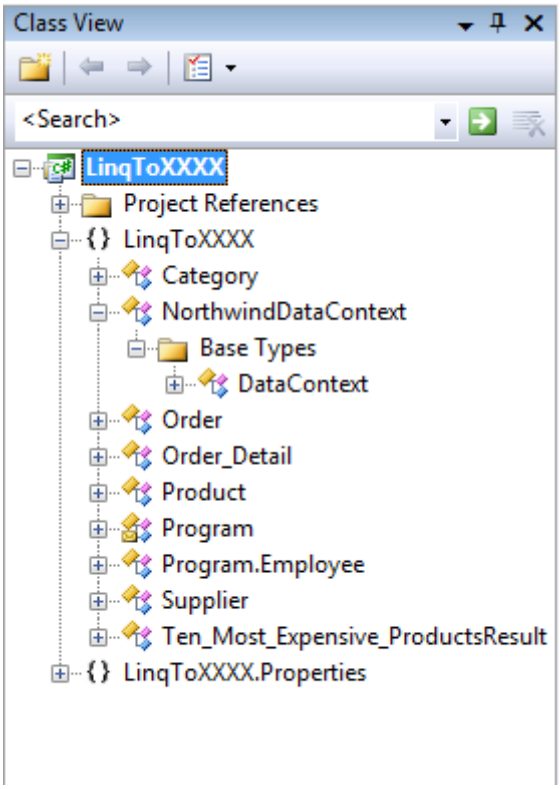
3. Добавяме нова връзка към базата Northwind в Server Explorer (от менюто View -> Server Explorer). За целите на демонстрацията ще използваме примерната база Northwind, а на вас оставям инсталацията на MS SQL Server.
4. Издърпайте последователно таблиците (Categories, Products, Suppliers, Orders, OrderDetails) докато се получи следната диаграма. Забележете, че обектите са в единствено число.



5. От Server Explorer->servername.Northwind.dbo->Stored Procedures придърпайте процедурата "Ten Most Expensive Products" върху панела вдясно.
6. Запишете промените и сме готови за действие.

С горната поредица от действия създадохме набор от класове за работа със съответните таблици и един специален клас за връзка с базата – **`NorthwindDataContext`** наследник на **`DataContext`**. Този клас е основата на всички колекции извлечени от базата и затова преди да започнем за изпълняваме заявки към базата трябва да създадем негова инстанция. При генерирането на **`NorthwindDataContext`** се създават няколко конструктора, които приемат указания за връзка към сървъра като текст (connection string) или инстанция на **`IDbConnection`**, а конструктура по подразбиране (този без параметри) използва връзката записана в конфигурационния файл. Това позволява голяма гъвкавост при посочване местоположението на базата.

Важно: Всички колекции извлечени от една инстанция на **`DataContext`** класа съдържат споделени инстанции на обектите. Например ако извлечем две колекции от продукти – едната с продуктите, започващи с буква “A”, а другата с 10-те най-скъпи продукта и се окаже, че 3 продукта ги има и в двете колекции, то ако вземем инстанцията на такъв продукт от колекцията с имената и му променим цената, то ще се промени и цената на обекта в колекцията с 10-те най-скъпи продукта.



Горните класове може да се генерират и с помощта на инструмента SqlMetal - http://blogs.msdn.com/sanjeets/archive/2007/06/07/how-to-use-sqlmetal-exe-to-create-a-class-from-xml-and-database.aspx

Извличане на данни

Имайки построеният модел и генерираните класове можем да извлечем данни от базата с LINQ заявка, така както бихме го направили с помощта на LINQ to Objects:

```
NorthwindDataContext db = new NorthwindDataContext();
var products = from p in db.Products
               where p.Category.CategoryName == "Seafood"
               select p;

foreach (var p in products) {
    Console.WriteLine("{0}\t\t{1}", p.ProductName, p.UnitPrice);
}
```

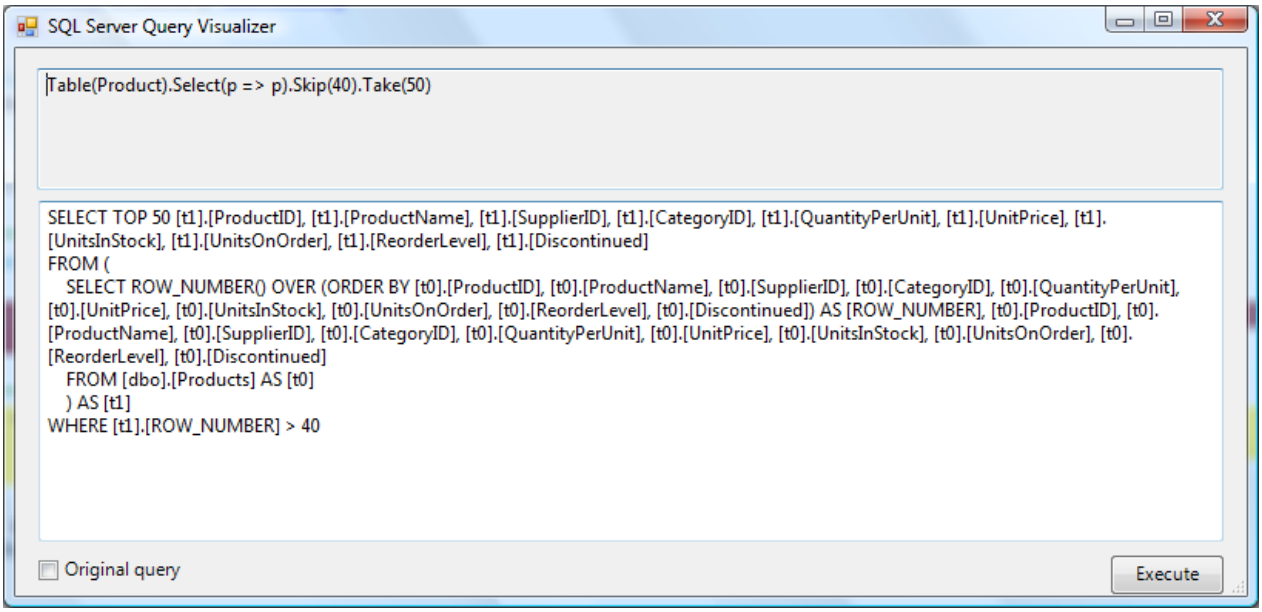
Забележка: Данните не се извличат на реда, започващ с **`var products`**. На този етап LINQ заявката се преобразува в SQL заявка, но се изпълнява едва когато се извлече някой елемент от резулата – в нашият случай – на реда **`foreach (var p in products)`**.

Страниране на данни

Много често в практиката се налага да не извличаме всички редове от таблицата наведнъж, а използвайки страниране ни трябват X реда, от общо Y, започвайки от ред с № Z. Затова нека да покажем как може да извлечем 50 продукта от базата, започвайки от продукт № 40:

```
NorthwindDataContext db = new NorthwindDataContext();
var paging = (from p in db.Products
              select p).Skip(40).Take(50);
```

Изключително лесно и удобно, нали? А ето и каква заявка се генерира от LINQ to SQL (с помощта на SQL Server Query Visualizer)



Генерираната SQL заявка много прилича на заявката, която ние бихме написали ръчно, за да изпълним желаната цел.

Промяна на данни

Промяната на данни също е толкова лесна – в следващият пример създаваме заявка, от която извличаме един елемент и му променяме някои от свойствата:

```
NorthwindDataContext db = new NorthwindDataContext();
var productChai = (from p in db.Products
                   where p.ProductName == "Chai"
                   select p).Single();

productChai.UnitsInStock = 55;
productChai.UnitPrice = 2.55;

db.SubmitChanges();
```

Забележете последният ред - **`db.SubmitChanges();`** всъщност записва промените в базата. Преди този ред промените са само в паметта.

Вмъкване на данни

Вмъкването на данни става на два етапа – първо се създава инстанция на обекта, после се добавя в колекцията. (Подобно на добавяне на **DataRow** в **DataTable**). В следващият пример добавяме два нови продукта и един доставчик, като присвояваме доставчика на едниният продукт:

```
NorthwindDataContext db = new NorthwindDataContext();

Product bicycle = new Product();
bicycle.ProductName = "Bicycle";
bicycle.UnitPrice = 160.5M;

Supplier supp = new Supplier { CompanyName = "Supplier 1", ContactName = "Bai
Spiro" };

Product chair = new Product { ProductName = "Chair", UnitsInStock = 56 };
chair.Supplier = supp;

db.Suppliers.Add(supp);
db.Products.Add(bicycle);
db.Products.Add(chair);
```

Изтриване на данни

Изтриването е подобно –премахваме елементи от колекцията. В примера избираме колекция от продукти, които после изтриваме с помощта на метода RemoveAll:

```
var productForDelete = from p in db.Products
                       where p.ProductName.Contains("Chef")
                       select p;

db.Products.RemoveAll(productForDelete);
```

С това разгледахме CRUD (Create, Read, Update, Delete) операциите и видяхме по какъв начин се извършва всяка една от тях. С това, обаче, не се изчерпват възможностите на LINQ to SQL – същите операции могат да бъдат извършени по няколко начина, а тук разгледахме само по един. За повече информация може да разгледате връзките в края на този документ. Те могат да бъдат доста полезни в изучаването на C# 3.0 и LINQ to SQL.

Извикване на съхранени процедури (stored procedures)

До момента разгледахме начините за работа с данни през код написан на C#, за самите SQL заявки се генерираха от LINQ. Функционалността на LINQ to SQL не би била пълна ако не предоставяше възможност за извикване на съхранени процедури. Дали е по-добре да се изпълняват заявки от кода или да се извикват съхранени процедури? Има много изписано по темата, но накратко казано съхранените процедури са за предпочитане, защото са компилирани на SQL сървъра и той е подготвен за изпълнението им (това е наистина кратко обяснение – ако се нуждаете от повече се обърнете към документацията на MS SQL Server).

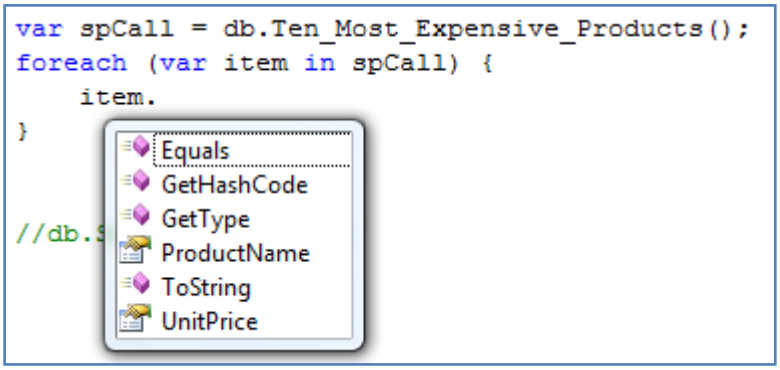
Забележете, че при създаване на модела на базата, в стъпка 5 добавихме съхранена процедура. Макар и малко прибързано това беше с цел да покажем къде се разполагат в диаграмата, но ще свърши работа, за да покажем как се извикват съхранени процедури.

Съхранените процедури се извикват като метод на **DataContext** класа, след като са добавени в модела на базата:

```
NorthwindDataContext db = new NorthwindDataContext();

var spCall = db.Ten_Most_Expensive_Products();
```

Но нека да видим какво се получава при обхождането на резулата:



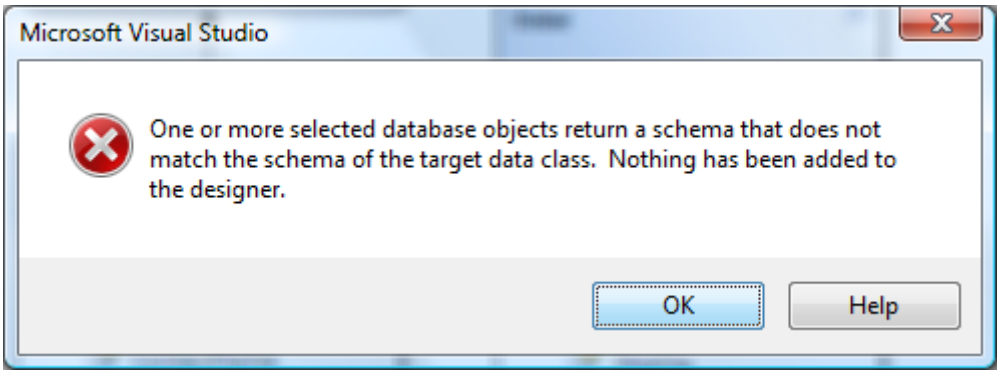
Но това не са свойствата на класа **Product**, които очакваме!

Това се получава по две причини: първата е самото тяло на съхранената процедура:

```
ALTER procedure "Ten Most Expensive Products" AS
SET ROWCOUNT 10
SELECT ProductName, UnitPrice
FROM Products
ORDER BY Products.UnitPrice DESC
```

Всяка съхранена процедура може да върне произволен брой полета от произволен брой таблици. Затова по подразбиране при пускането на съхранената процедура върху бялата повърхност на модела се генерира метод за извикването ѝ, но и нов клас за резултата – погледнете на снимката на прозореца Class View по-горе.

За да получим резултат от тип, който вече имаме в диаграмата трябва да пуснем процедурата върху типа, от който желаем да е резулата. Нека да изтрием съдържанието на панела вдясно на модела (Northwind.dbml). Придърпваме "**Ten Most Expensive Products**" върху формата на класа **Product** и

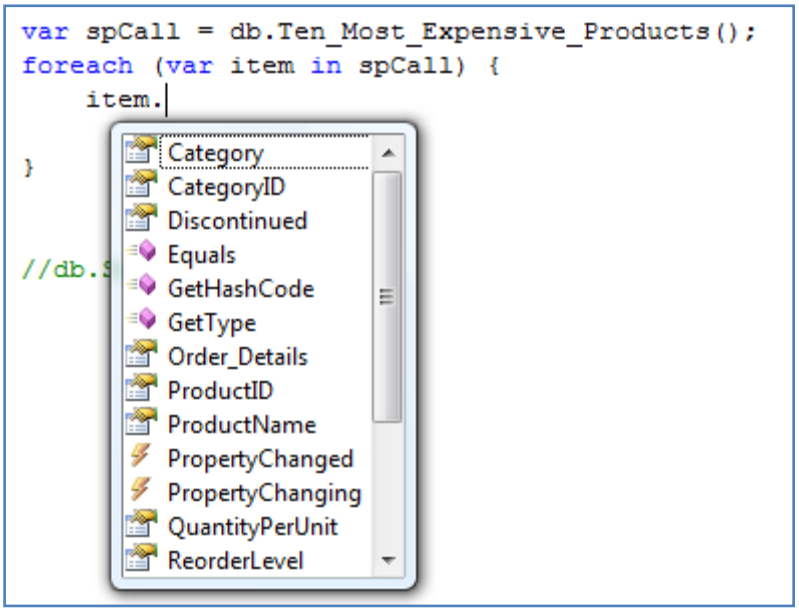


Visual Studio 2008 проверява резултата на процедурата и ни уведомява, че схемата се различава и операцията не може да бъде изпълнена, както и че модела не е променен.

Налага се да променим връщаните полета от съхранената процедура:

```
ALTER procedure "Ten Most Expensive Products" AS
SET ROWCOUNT 10
SELECT *
FROM Products
ORDER BY Products.UnitPrice DESC
```

Записваме промените в процедурата, опитваме отново да я пуснем върху формата на класа **Product** и този път е успешно.



Извикване на потребителски функции (UDF)

Често в при работа данни създаваме потребителски функции, които могат да се ползват в SQL заявки и съхранени процедури. Въпреки, че пренасяме голяма част от работата с данни в C#, не е оправдано да изхвърляме или пренаписваме работещ код, още повече че той се изпълнява на сървъра и може да намали натоварването на клиента. Затова нека да разгледаме как става извикване на такава потребителска функция.

За целта ще създадем една проста функция, която умножава две числа:

```
CREATE FUNCTION dbo.udfMultiply (@num1 INT,  @num2 INT)
RETURNS INT
AS
BEGIN
RETURN (@num1 * @num2)
END
```

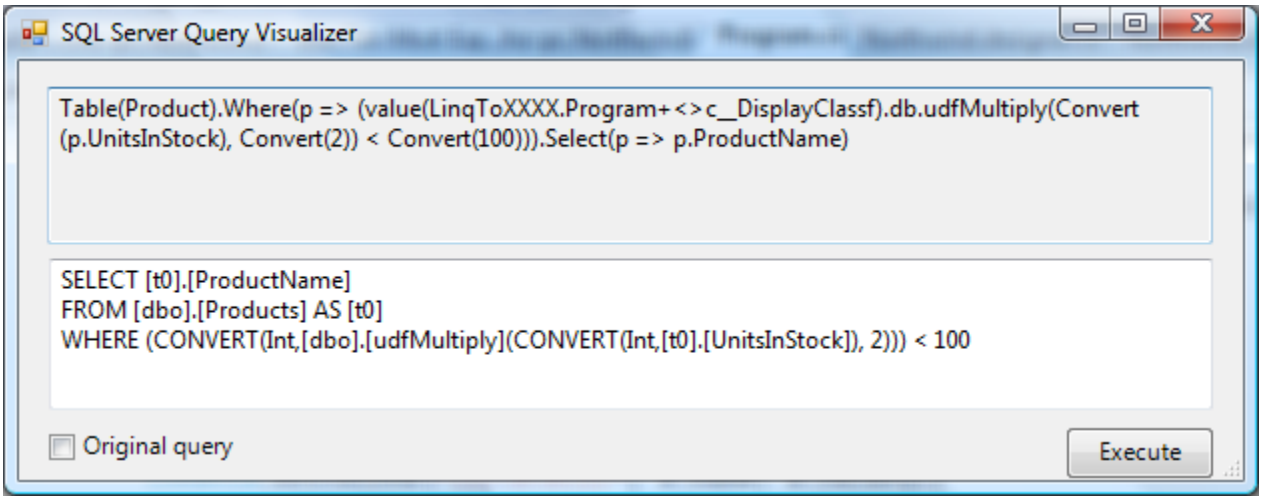
Подобно на добавянето на съхранена процедура я добавяме в модела на базата в нашия проект (чрез влачене и пускане от Server Explorer върху десния панел на Northwind.dbml)

След това можем да я извикваме като метод на наследника на `DataContext` класа, подобно на съхранената процедура. Нека да видим SQL кода, който се генерира за следната LINQ заявка (която връща имената на тези продукти, чието количество умножено по 2 е по-малка от 100):

```
NorthwindDataContext db = new NorthwindDataContext();

var udfCall = from p in db.Products
              where db.udfMultiply(p.UnitsInStock, 2) < 100
              select p.ProductName;
```

Стартираме проекта в режим на debug и показваме кода в SQL Server Query Visualizer:



Забележете извикването на функцията в `where` клаузата на заявката на последния ред.

С това разгледахме основните възможности на LINQ to SQL. Съществуват още тънкости, които тук не са описани като извикване на процедури, които връщат колекции от различни таблици, работа с `output` параметри, както и обработване останалите видове резултат от функции. За повече информация се обърнете към източниците, посочени в края на документа.

Макар да не наблегнахме ние използвахме в примерите връзки между обектите от тип едно-към-много. За съжаление връзки от тип много-към-много не се поддържа от LINQ to SQL, но това не означава, че не е възможно използване му. С малко повече код от наша страна може да се постигне – за повече информация вижте Create many-to-many relations with LINQ to SQL.

LINQ to Entities

Следащият голям клон в LINQ е LINQ to Entities (или още познат като ADO.NET vNext или ADO.NET Entity Framework). Неговата имплементация изисква много повече ресурси от тази на LINQ to SQL, което в комбинация с недостатъчното време до официалното пускане на пазара на VS 2008 принуди екипа да извади LINQ to Entities от пакета възможности на VS 2008. Това не означава, че ще трябва да чакаме до следващата версия на Visual Studio, за да го използваме – след като бъде завършено LINQ to Entities ще бъде пуснато като допълнение към Visual Studio.

Ако обърнахте внимание при работа с LINQ to SQL всеки клас от C# е директно свързан към една таблица/изглед от MS SQL Server – или имаме директно свързване (mapping) на бизнес обекта със механизма за съхранение. Някои проекти с по-сложна схема на базата данни изискват съхранение на един обект в няколко таблици, което изисква по-сложно свързване (mapping). По този начин логическата схема на бизнес обектите може да се различава значително от схемата на базата данни. Това прави LINQ to Entities много по-завършен ORM инструмент от LINQ to SQL.

За да сравним по-добре LINQ to Entities с LINQ to SQL нека да разгледаме следната таблица с основни възможности[6]:

Feature	LINQ to SQL	LINQ to Entities
Language Extensions Support	Y	Y
Language Integrated Database Queries	Y	Y
Many-to-Many (3way Join/Payload relationship)	N	N
Many-to-Many (No payload)	N	Y
Stored Procedures	Y	N (to be added)
Entity Inheritance	N	Y
Single Entity From Multiple Tables	N	Y
Identity Management / CRUD features	Y	Y

Самата архитектура на LINQ to Entity добавя допълнителен слой между езиковата конструкция, определяща заявката (код на C#) и заявката към слоя за съхранение на данните (MS SQL Server, Oracle и др.). В този слой EntityProvider изплозвайки дефинирата схема за свързване (mapping) преобразува заявката в подходяща за схемата и езика на слоя за съхранение на данните.

За да работим с LINQ to Entity трябва да създадем Entity Data Model (EDM), подобно на Създаване на модела на базата в LINQ to SQL с тази разлика, че освен генерирани класове трябва да се направи файл, който да указва свързването. Този файл е в XML формат. След това може да работим с данните по същият начин, по който работихме и в LINQ to SQL, защото вътрешните детайли остават скрити за нас.

LINQ to Entity (Entity Framework), както показва и името, е цяла нова система за работа с данни – нов framework и цялостното му обхващане може да се събере в отделна книга. Затова ще спрем до тук с описанието и ще изчакаме следващите етапи в развитието му.

LINQ to DataSet

LINQ to DataSet представлява възможност да се пишат заявки към обекти, съдържащи се в DataSet – това са познатите ни обекти от ADO.NET – **`DataTable, DataView, DataRow`**. Реално много прилича на LINQ to Objects, защото все пак работим с обекти. Това ни позволява да напишем следната заявка):

```
Northwind.ProductsDataTable productsTable = new Northwind.ProductsDataTable();

var products = from dr in productsTable
               where dr.CategoriesRow.CategoryName == "Seafood"
               select dr;
```

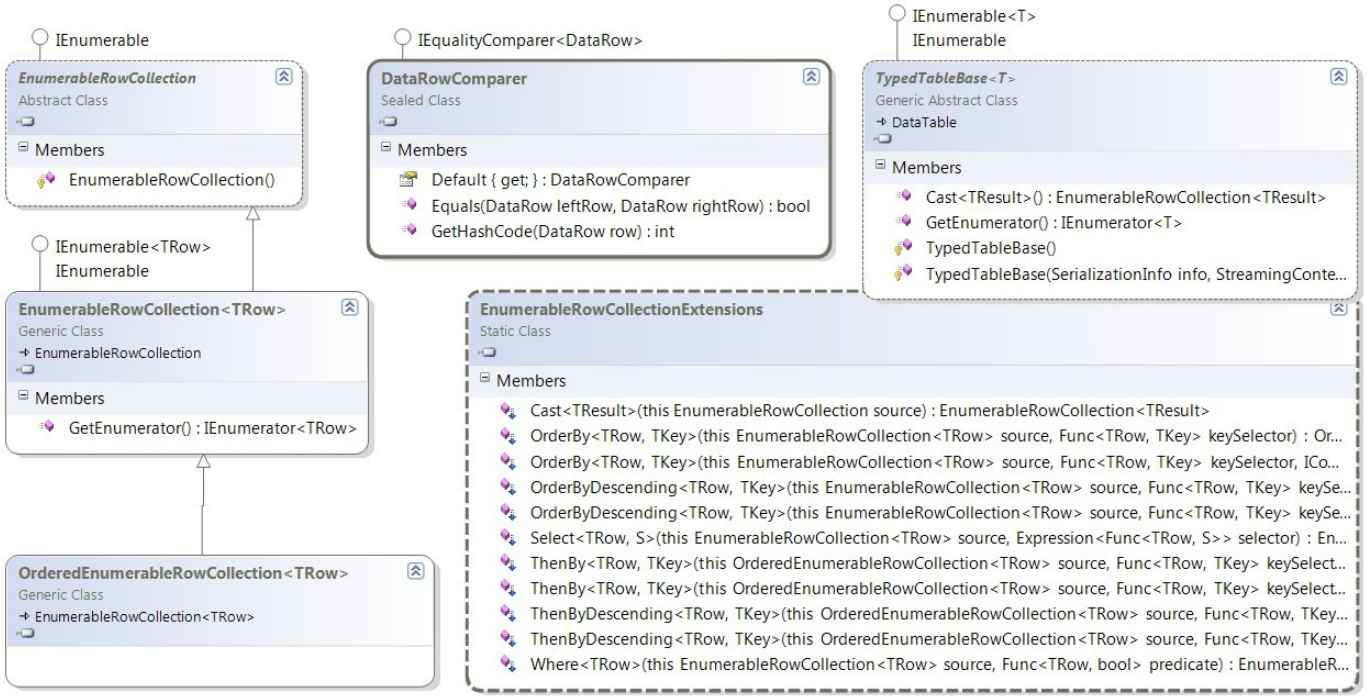
Забележете, че тук връзките между таблиците остават както в нормален обект от тип **`DataSet`**, така че имаме достъп до **`parent`** обекта чрез свойството **`CategoriesRow`**. Обектите **`dr`** и

`CategoriesRow` са наследници на класа `DataRow`. Още една малка подробност: за свойствата на обект от тип `DataRow` е характерно, че `get` може да хвърли изключение, което от своя страна не е добра практика за работа с LINQ. Затова е добре да използваме метода `DataRow.IsNull()` при работа със свойства на `DataRow` в LINQ to DataSet.

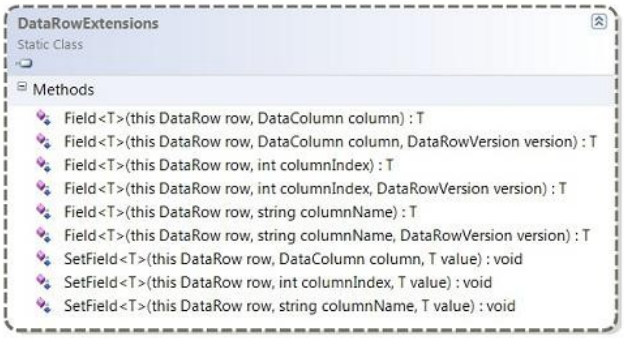
На пръв поглед това подобрение не е голямо, но самата комбинация на `DataSet` класовете с възможностите на LINQ дава много възможности, които ги няма при работа само с `DataSet` класовете. Ето някои от тях:

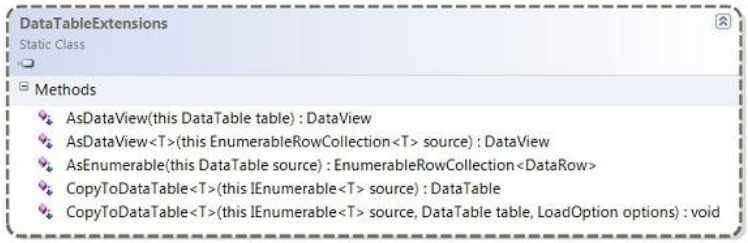
- Създаване на заявка, като резултата връща нова колона, която съдържа изчисление
- Създаване на изглед в смисъла на SQL Server – съдържащ колони от няколко таблици и др.

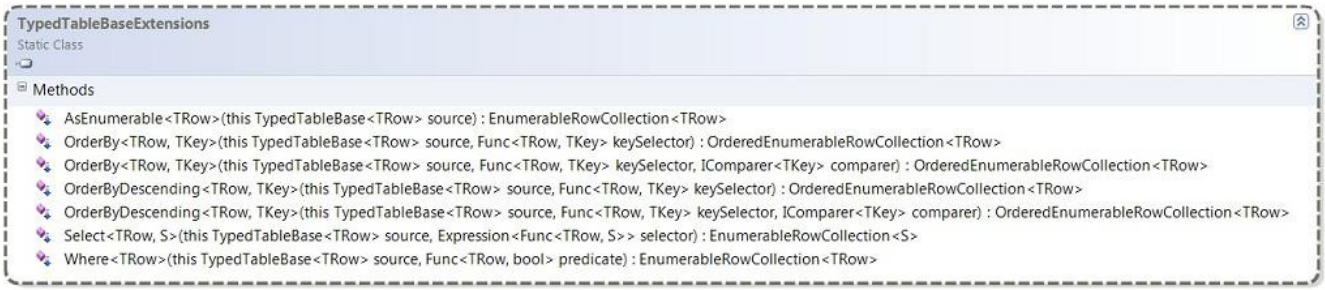
Ако погледнем следната диаграма[7], ще видим класовете от асемблито System.Data.DataSetExtensions.dll, които подпомагат реализацията на LINQ to DataSet:



Както казахме и по-горе, LINQ заявката се преобразува до разширяващи методи, и затова нека погледнем какви разширяващи методи са създадени към типовете от ADO.NET. Следната диаграма[7] показва дефиницията на тези методи, които правят LINQ to DataSet възможно:







LINQ to XML

LINQ to XML предоставя нов програмен модел за четене, записване и конструиране на XML. Този модел е много по-удобен от DOM API (на **`XmlDocument`**), което се използва до момента, а и използва много по-малко памет. Освен това е много по-лесен за използване от **`XmlReader/XmlWriter`**. За да се постигне това улеснение са създадени нов набор от класове за работа с XML в пространството от имена **`System.Xml.Linq`** в асемблито System.Xml.Linq.dll.

Четене и обхождане на XML

Нека да разгледаме следният xml документ:

```
<?xml version="1.0" encoding="utf-8"?>
<Products>
  <Product ProductName="Alice Mutton">
    <ProductID>17</ProductID>
    <UnitPrice>39.0000</UnitPrice>
    <UnitsInStock>0</UnitsInStock>
  </Product>
  <Product ProductName="Aniseed Syrup">
    <ProductID>3</ProductID>
    <UnitPrice>10.0000</UnitPrice>
    <UnitsInStock>13</UnitsInStock>
  </Product>
</Products>
```

Този фрагмент съдържа основен (root) елемент <Products>, който съдържа два поделемента от тип <Product>. Данните от елементите <Product> са съхранени по два начина – като атрибути и като поделементи. С помощта на следния израз можем да прочетем документа и да го покажем на в конзолата:

```
XElement xml = XElement.Load(@"D:\LINQTutorial\Products.xml");

var products = from p in xml.Descendants("Product")
               orderby p.Element("ProductID").Value descending
               select new {
                   ProductName = p.Attribute("ProductName").Value,
                   ProductID = p.Element("ProductID").Value,
                   UnitPrice = p.Element("UnitPrice").Value,
                   UnitsInStock = p.Element("UnitsInStock").Value
               };

foreach (var item in products) {
    Console.WriteLine("{0}\t{1}\t{2}\t{3}", item.ProductID, item.ProductName,
item.UnitsInStock, item.UnitPrice);
}
```

На първият ред създаваме инстанция от тип **XElement**, зареждайки XML документа от файловата система. Следва по-интересната част – самата LINQ заявка.

От първият ред на заявката определяме основната променлива **p** (отново от тип **XElement**), която ще представлява всеки елемент върнат от израза **xml.Descendants("Product")**. Метода връща всички поделементи от тип <Product>. За да извлечем данните използваме два метода – **XElement.Element()** и **XElement.Attribute()**, съответно, за да извлечем данните съответно от поделементите и от атрибутите. Забележете, че тези два метода отново връщат резултат от тип **XElement** и за да извлечем данните трябва да използваме свойството **Value**.

Тук обаче се крие опасност – тъй като подаваме имената на елементите и атрибутите като текст има възможност за грешка, а е възможно също така даден елемент да не съдържа всички поделементи и атрибути. В този случай **XElement.Element()** и **XElement.Attribute()** ще няма да намерят търсеният елемент/атрибут и ще върнат **null**. Когато се опитаме да вземем стойността на свойството **Value** ще получим **NullReferenceException**. За да избегнем подобна ситуация може да използваме помощен метод:

```
var products = from p in xml.Descendants("Product")
               orderby p.Element("ProductID").Value descending
               select new {
                   ProductName = GetXElementValue(p.Attribute("ProductName")),
                   ProductID = GetXElementValue(p.Element("ProductID")),
                   UnitPrice = GetXElementValue(p.Element("UnitPrice")),
                   UnitsInStock = GetXElementValue(p.Element("UnitsInStock")),
               };

private string GetXElementValue(XElement el) {
    if (null != el) return el.Value;
    return string.Empty;
}
```

По този начин се подсигуряваме, че няма да възникне изключение.

Създаване на XML

Освен прочитането и обхождането на XML документ, можем много лесно да генерираме такъв. Понеже класът **XElement** е много гъвкав позволява генерирането на цял документ с един израз:

```
XElement e = new XElement("Products",
                 new XElement("Product",
                    new XAttribute("ProductName", "Product1"),
                    new XElement("ProductID", 1),
                    new XElement("UnitPrice", 11.5),
                    new XElement("UnitsInStock", 12)),
                 new XElement("Product",
                    new XAttribute("ProductName", "Product2"),
                    new XElement("ProductID", 2),
                    new XElement("UnitPrice", 2.45),
                    new XElement("UnitsInStock", 89))
                  );
```

Въпреки, че класът **XElement** е нов горната конструкция не е нова – в нея използваме комбинация от два конструктура на **XElement**, за да постигнем целта:

```
public XElement(XName name, params object[] content)
public XElement(XName name, object content)
```

Тези два конструктура ни позволяват да комбинираме LINQ to XML и LINQ to SQL, за да експортираме данни към XML. В следващият пример комбинираме двата конструктура на **XElement** с зявка от LINQ to SQL, за да получим XML документ със същата структура, какъвто показахме в началото на тази точка:

```
NorthwindDataContext db = new NorthwindDataContext();

XElement xml = new XElement("Products",
        from p in db.Products
        orderby p.ProductName
        select new XElement("Product",
                  new XAttribute("ProductName", p.ProductName),
                  new XElement("ProductID", p.ProductID),
                  new XElement("UnitPrice", p.UnitPrice),
                  new XElement("UnitsInStock", p.UnitsInStock))
        );

xml.Save(@"D:\LINQTutorial\Products.xml");
```

В този пример вместо да подадем с код поредицата от параметри за конструктура, изискващ **params object[] content**, подаваме резултат от LINQ to SQL заявка, която връща инстанции на **XElement**.

Това показва силата на LINQ и неговите клонове. Със сигурност ако трябваше да изпълним тази задача с помощта на C# 2.0 кодът нямаше да е толкова малко и така подреден.

LINQ to XSD

Преди да започнем с описанието на LINQ to XSD нека да споменем какво стои зад абревиатурата XSD (XML schema definition) – това е схемата, която описва XML документите. Въпреки, че XML документи могат да съществуват и без да има дефинирана схема, то наличието на такава позволява валидиране на структурата на документа и въведените данни. Чрез валидация осигуряваме данните, които са записани в чист текстов вид (какъвто представлява XML) са такива, каквито трябва да бъдат.

Разполагайки със схемата можем с инструментариума на LINQ to XSD, се улеснява значително писането на зявки към данните в XML. Така кода от Четене и обхождане на XML би изглеждал по този начин:

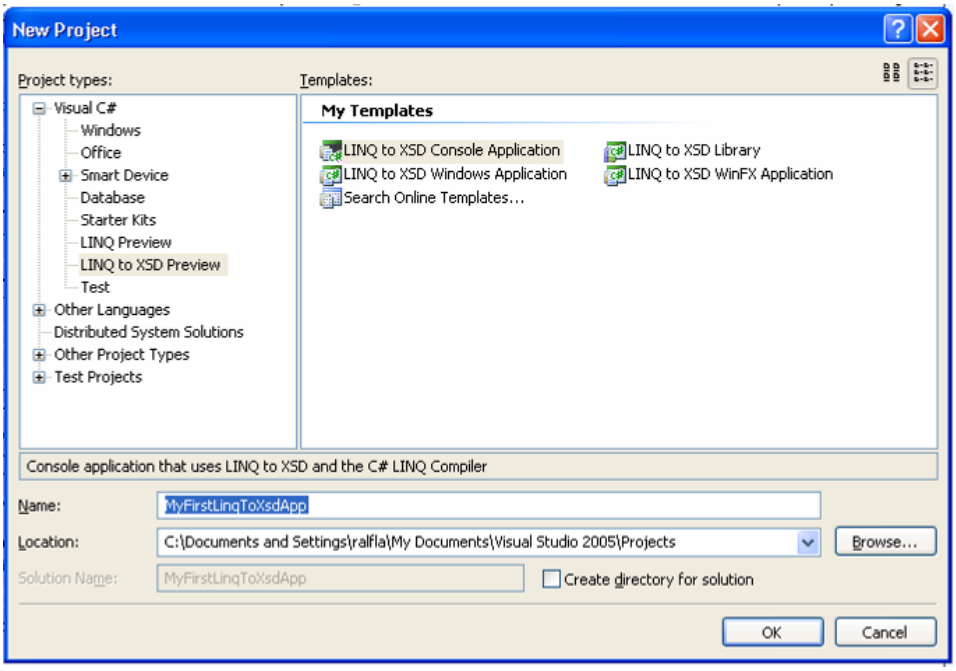
```
var products = from p in xml.Product
               orderby p.ProductID descending
               select new {
                   ProductName = p.ProductName,
                   ProductID = p.ProductID,
                   UnitPrice = p.UnitPrice,
                   UnitsInStock = p.UnitsInStock
               };
```

Както се вижда значително се намалява кода, а и с това възможността да се получи изключение по време на изпълнение. За да се постигне подобен начин на изписване, обаче, трябва да се изпълнят няколко неща:

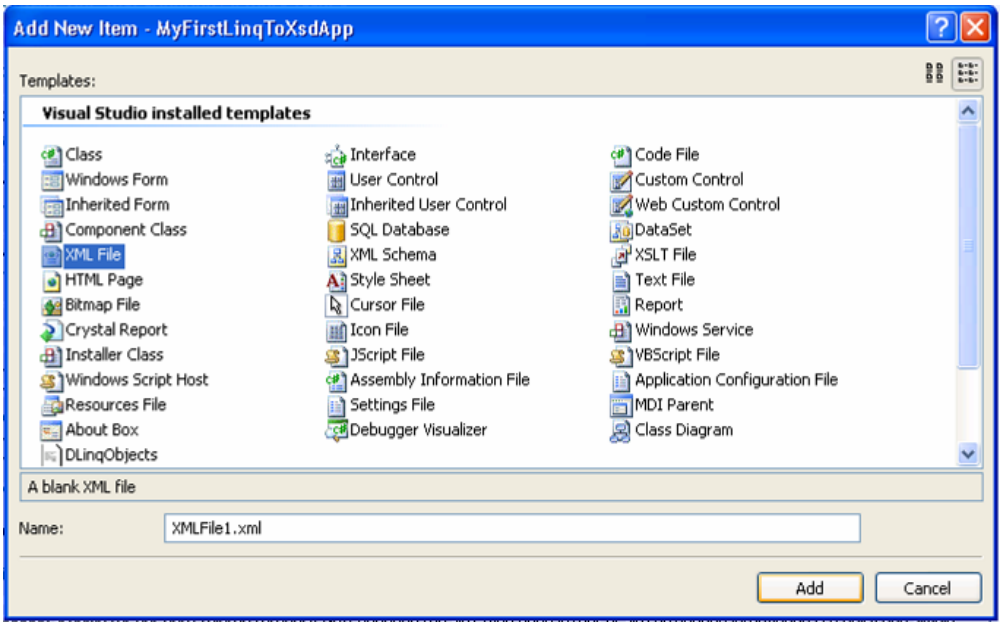
- Да разполагаме с XML документа,
- Да разполагаме с XSD схемата,
- VS да генерира съответните класове за данните спрямо схемата.

Точно тук се крие разликата в изграждането на LINQ to XSD. Нека да разгледаме последователно как може да постигнем това в среда на Visual Studio[12]:

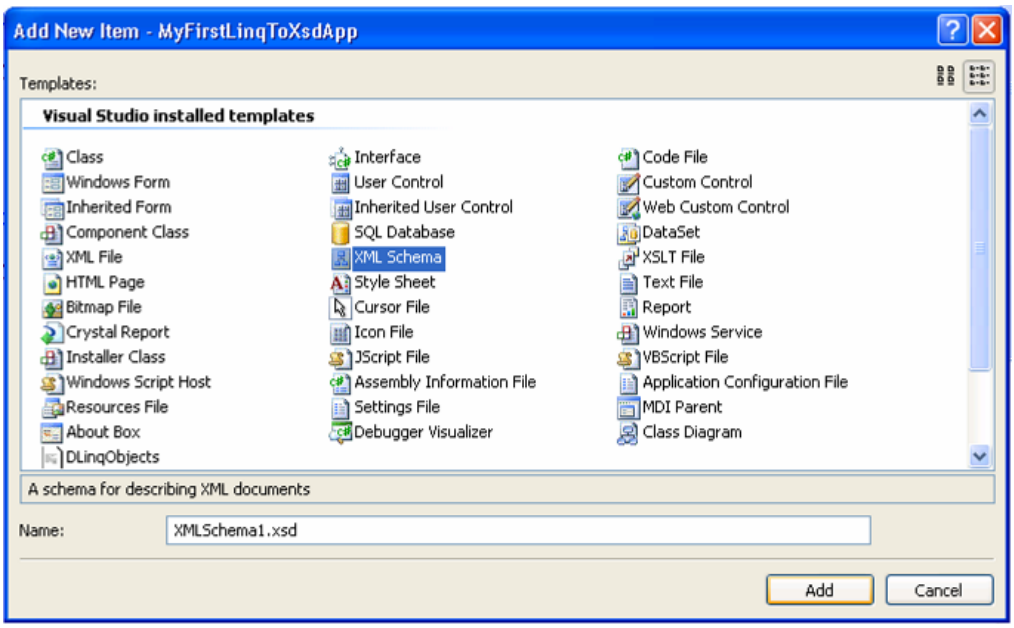
1. Създаваме нов проект от тип “LINQ to XSD Console Application”



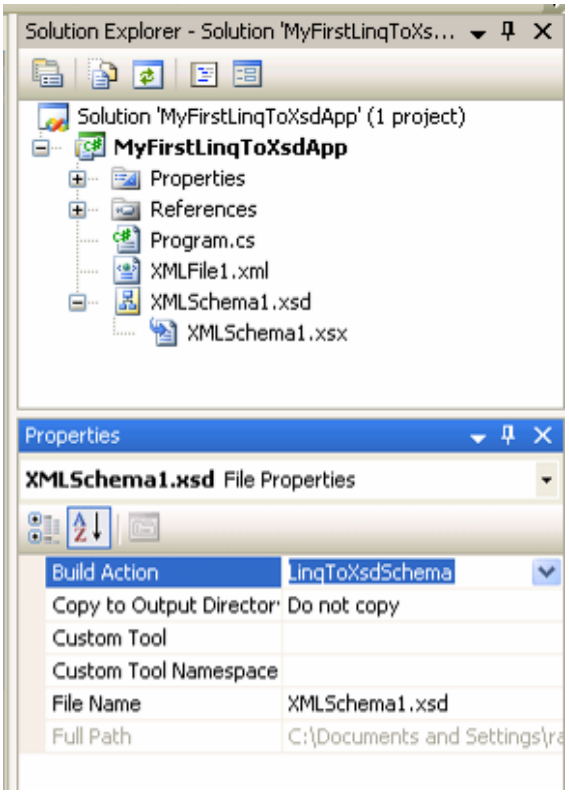
2. Добавяме нов елемент към проекта от тип XML File



3. Добавяме нов елемент към проекта от тип XML Schema

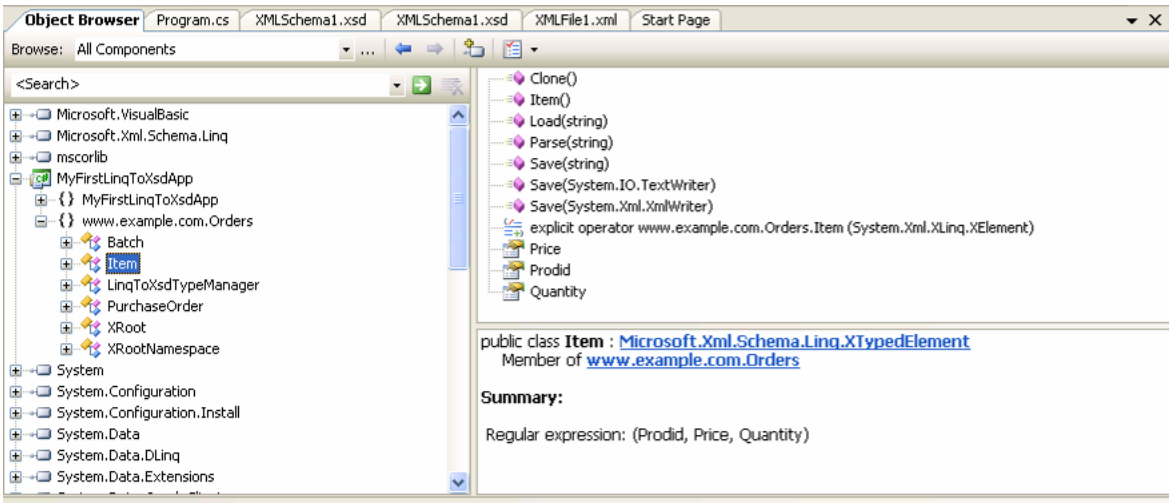


4. Задаваме при компилиране XML схемата да се използва за генериране на класове като задаваме Build Action “LinqToXsdSchema”

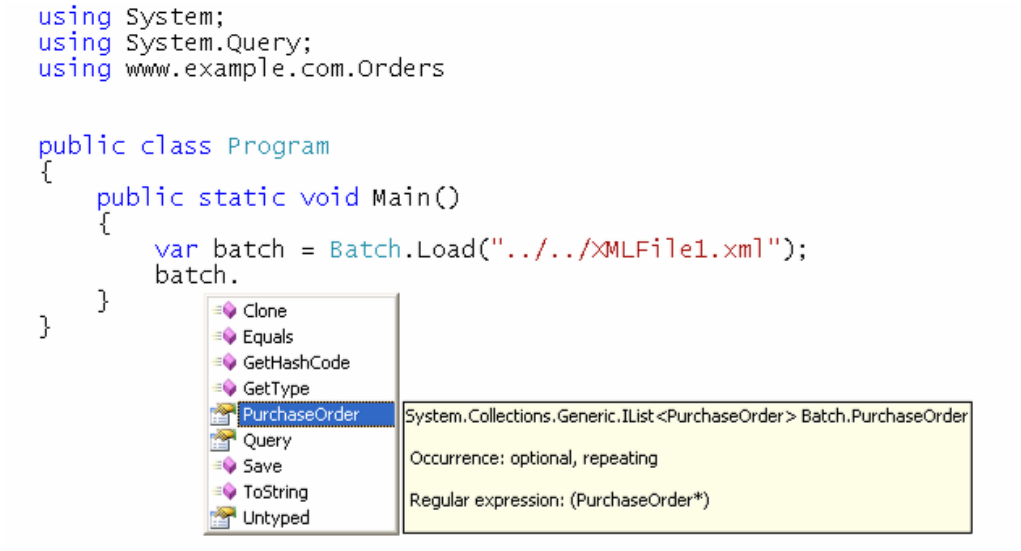


5. Компилираме проекта, за да се генерират класовете от схемата, с което IntelliSense на Visual Studio ще има възможност да помага при писане на заявките. След компилация

може да разгледаме класовете в прозореца Object Browser (View -> Object Browser)



6. Добавяме в кода пространството от имена на генерираните класове и можем да пишем заявки, като разполагаме и с IntelliSense



За съжаление LINQ to XSD изглежда е изключено от Visual Studio 2008 Beta 2 както и LINQ to Entities, което означава, че двете ще бъдат пуснати по-късно като допълнение на Visual Studio 2008. Дотогава ще трябва да се задоволим с възможностите на LINQ to XML.

Заключение

Новите възможности на езика C# и LINQ повишават значително продуктивността на разработчиците на софтуер. Както с всеки мощен инструмен и към тези трябва да се подходи внимателно и обмислено, за да може като краен резултат да се получи функционален (и функциониращ), бърз, лесен за промяна и разширение продукт. Последното е в ръцете на разработчиците!

Използвана литература

1. Microsoft Development Network (MSDN) - http://msdn2.microsoft.com/en-us/netframework/aa904594.aspx
2. Блога на Scott Guthrie – General Manager on Developer Division - http://weblogs.asp.net/scottgu/archive/tags/LINQ/default.aspx
3. Hooked on LINQ - Developers' Wiki - http://www.hookedonlinq.com/
4. Блога на Rico Mariani – Chief Architect of Visual Studio - http://blogs.msdn.com/ricom/default.aspx
5. Блога на Sahil Malik – http://blah.winsmarts.com
6. Блога на Kevin Hoffman - http://dotnetaddict.dotnetdevelopersjournal.com/adoef_vs_linqsql.htm
7. Блога на Daniel Moth - http://www.danielmoth.com/Blog/2007/07/linq-to-dataset.html
8. MSDN Magazine - http://msdn.microsoft.com/msdnmag/issues/07/07/DataPoints/default.aspx
9. Блога на Mike Taulty - http://mtaulty.com/CommunityServer/blogs/mike_taultys_blog/archive/2007/08/03/9558.aspx
10. Erick Thompson's post series LINQ to DataSet - http://blogs.msdn.com/adonet/archive/2007/01/26/querying-datasets-introduction-to-linq-to-dataset.aspx
11. Блога на Fabrice Margueire - http://weblogs.asp.net/fmarguerie/archive/2007/01/15/linq-to-xsd-typed-xml-programming-with-linq.aspx
12. LINQ to XSD Overview document - http://download.microsoft.com/download/1/5/a/15a59e5d-464e-438e-bc59-9846ab787fd5/LinqToXsdOverview_November_2006.doc