



Database Access Technology

ADO.NET

Lesson №5

Introduction to LINQ

Contents

LINQ	3
Overview	3
LINQ to Objects in the query syntax	4
LINQ to Objects in the method syntax.....	11
LINQ To SQL overview	25
LINQ To XML overview	32
Conclusions	39

Overview

You have already almost mastered the ADO.NET technology and understood its role. This technology is designed to access data stored in different relational databases. SQL is used to communicate with relational databases. It's a very effective programming language distinguished by the high speed of data processing and versatility. On the other hand, C# was used in the applications, which we wrote. In C#, we created objects that allow you to connect to data providers, pass database queries, get query results, and process the received results.

At some point of time, C# developers faced some questions:

- Can we access datasets and process this data in C#?
- Can we make an access to datasets homogeneous regardless of where this data is stored: either in a database or in an XML-file or in a RAM collection?
- Can we make our work with data the most effective?

LINQ (Language Integrated Query) became an answer to all these questions, it's a C# extension appeared in .NET Framework 3.5. We will get acquainted with it in our lesson. The LINQ creation was largely inspired by SQL, which approved itself by working with data in relational databases.

LINQ is designed to work with very large collections of objects. To do this, it offers to create queries in C#, and introduces a whole number of extension methods. This allows you not only to select the required data from a collection,

but execute such data operations as sorting, grouping, and aggregate actions as well.

Nowadays, there are several LINQ varieties in C#. The most widely used are the following ones:

- LINQ to Objects is designed to work with collections stored in RAM;
- LINQ to SQL is designed to work with data in relational databases;
- LINQ to XML is designed to work with data in XML-files.

Now, we will consider the main LINQ features using an example of LINQ to Objects, and after that, we will consider other LINQ features.

LINQ to Objects in the query syntax

Let's create a console application, where we will consider the use of LINQ to Objects. You can use any collection as a data source, for example, an array or a list. Under real-life conditions, LINQ puts its best footforward by working with very large collections. The larger the collection is, the more preferable it is to use LINQ. In our example, we will work with a modest size collection. At first, it will be a string array. After that, we will consider how to work with a collection of objects.

Make the code of the created application look like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```

namespace LinqTest1
{
    class Program
    {
        static void Main(string[] args)
        {
            LinqMethod1();
        }

        private static void LinqMethod1()
        {
            //an array is a data source
            string[] countries = {"Albania", "UK",
                                "Lithuania", "Andorra", "Austria",
                                "Latvia", "Liechtenstein", "Switzerland",
                                "Ireland", "Sweden", "Italy", "France",
                                "Liechtenstein", "Spain", "Turkey", "Germany",
                                "Switzerland", "Monaco", "Montenegro",
                                "Norway", "Finland", "Turkey", "UK", "Poland",
                                "Portugal", "Lithuania", "Luxembourg"
                                };
            //LINQ query
            var result = from c in countries
                        where c.StartsWith("L")
                        select c;

            Console.WriteLine("Countries beginning with L:");
            //continue to work with a query
            foreach (var item in result)
            {
                Console.WriteLine(item);
            }
            Console.Write("Press Enter to complete");
            Console.ReadLine();
        }
    }
}

```

Please note that some countries are repeated in our array with names of countries. We will use this point soon. Also note the fact that our project contains the System.Linq namespace connection initially. It tells us about the importance of LINQ developers pay particular attention to. This namespace includes classes and extension methods, which LINQ consists of. Let's talk about the most unusual string of the LinqMethod1() code:

```
var result = from c in countries where c.StartsWith("L")
              select c;
```

This is a LINQ query written in C#. The "from ... where ... select" structure of this string catches our eye immediately because it reminds us of SQL-queries, but it's still C#. The result of this query is entered into the result variable, which type is described by the var specifier. Var is a new specifier, indicating the generalized data type, that is, the data type, which is defined dynamically by assigning a result. It's a very convenient way to describe types, which are defined dynamically. In our case, the only thing that can be said about the result variable type is that it will be derived from the IEnumerable<> interface. Further detailing of the type depends on the data source type and the query itself.

Let's make a small digression, and talk about imperative and declarative programming. Let's begin with an example.

You have MyTable with a number of strings controlled by the MS SQL Server. In addition to the table, the program in C# includes a collection of objects of any class. You should display all the table strings and all the collection objects

on the screen. For a table, you will write the `select * from MyTable` query without hesitation. And for a collection, you will write the `foreach` loop. Or `while`? Or `for`? But let's think a little bit here. How does the `select` query differ from any loop? The query contains the description of what we want to get. But herewith, the query doesn't specify how you want to get the result. We are accustomed to the fact that details of the SQL query execution depend on how these queries are implemented by the DBMS developers. We suggest that, perhaps, the `select` query is executed on the MS SQL Server in a way different from Oracle one. The result is the same in both cases. And what about loops? Here, we should do everything by ourselves: select a loop, decide how to display objects of our class and overload `ToString()` in this class, define, on what condition the loop execution should be stopped and so on. In other words, we write a step by step set of statements in C# in order to display elements of our collection. In the case of using a loop, we define by ourselves, how the required result should be got. In the case of table, we used declarative programming by specifying what we need but not explaining to the DBMS, how it should get the required result. In the case of collection, we used imperative programming by describing data, which we wanted to be got. The HTML, XML, XAML, SQL markup languages are typical representatives of the declarative programming. The C++, C#, and Java, traditional programming languages, are representatives of the imperative programming. But, this division is conditional. Because, C# comprises many declarative programming elements. For example, the `Array.Sort(coll)` string that sorts the `coll`

collection has all features of declarative programming. We don't specify, what algorithm should execute a sort, in what direction, but we rely on the Array class settings. And if we write a function for sorting such a collection by ourselves, it will be an imperative approach.

You have to get used to the fact that LINQ is declarative. We will repeat once again, that according to the declarative programming, we specify, "what" we want to do, and the system decides how the specified action should be executed by itself. The System.Linq namespace contains many specialized classes for storing the query results, and LINQ will define the best result variable type by itself in our example. Therefore, if some of you want to specify the type of this variable explicitly, you should accept that LINQ will do it better than you.

For the curious, we should inform you in secret, that the type of this variable will be always derived from the `IEnumerable<>` type. In the case of selecting from a database, this type will be derived from the `IQueryable<>` type. Further detailing of the type depends on the type of selected elements, collection type, statements used in a query, and even on the LINQ variety. For example, if you sort your selection, the type will be derived from `IOrderedQueryable<>`. And this type can be changed by the compiler when optimizing. In any case, you should never rely on the accurate type of this variable! If you want to look at this type, you can use `GetType()`.

We continue to analyze the query. The data source should be specified after the "in" specifier in the "from c in countries" string. In our case, it is the countries string array. What object can be a data source for LINQ to Objects? The only requirement

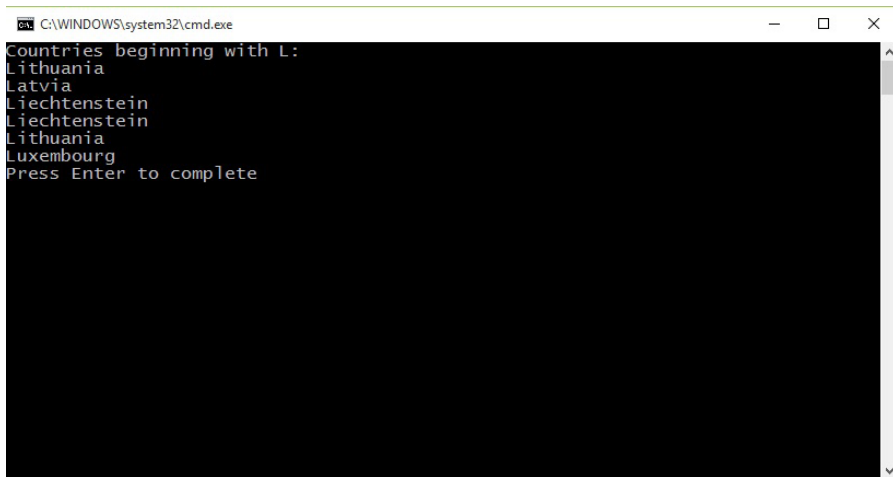
for this object is to be derived from the `IEnumerable<>` interface. Arrays and any other collections meet this requirement. Since the data source is a collection, which elements are sorted out at the query execution, the "c" variable is designed to store the current collection element at each iteration. It is clear, that this variable can have any name.

Then after the "where" specifier, we specify a condition, according to which we select elements of the viewed collection. In our example, from the array, we select names of countries, which begin with "L". Note that in order to form the condition, we use the `StartsWith()` standard method defined for the string type in C#.

And finally, the "select" specifier indicates that we select the current element of the "c" variable into the query result if this element meets the specified selection condition.

And now, some words about the foreach loop, wherein we output the query result. The fact is that this loop is a part of query. It implements the so-called "deferred execution". Note that specifically this loop actually executes the query! Query data will not be retrieved from the data source until we refer to the LINQ query results. And we refer to query results specifically in the foreach loop.

Run the application and look at the result. In my case, it looks as follow:



```
C:\WINDOWS\system32\cmd.exe
Countries beginning with L:
Lithuania
Latvia
Liechtenstein
Liechtenstein
Lithuania
Luxembourg
Press Enter to complete
```

Fig. 1. LINQ query result

Change the query condition to the following one:

```
var result = from c in countries where c.Length >
              10 select c
```

and execute the application once again. In this case, you will see only the countries, which names are longer than 10 characters in the result-set.

At the beginning of this topic, we noted the fact that the array of countries contains repeated names. You see them in the received collection. Rewrite the query as follow:

```
var result = (from c in countries where c.StartsWith("L")
              orderby c.Length descending select c).Distinct();
```

Predictably, repeats disappeared.

LINQ to Objects in the method syntax

Now you should keep in mind that the above query is written using the syntax of LINQ queries. The query syntax is not the only way to write the LINQ queries. The method syntax is another way to do it. Let's create one more method in our application, where we will write the same query to our array, but using the method syntax. Then, we will compare these two ways of writing queries by changing and complementing our methods.

LINQ is implemented as a set of extension methods for objects derived from the `IEnumerable` interface (arrays, collections and so on). If you are in Visual Studio, type the name of our countries array, and then enter the point and look at the IntelliSense list, you will see the number of generic methods, such as `Where<>`, `Union<>`, `Takes<>`, and others. They are LINQ methods. When you write the LINQ query, specifying specifiers like "from ... where ... select", the compiler replaces them with appropriate extension methods.

Let's rewrite our query in the method syntax. Generally speaking, the query syntax is more preferable when writing LINQ queries. It looks more readable. However, it's useful to have an idea of the method syntax, especially because sometimes, we can face the situations, when the method syntax can do something that can't be done in the query syntax.

For example, you want to receive a collection element with a maximum value of the `Price` property. Or you want to receive a number of elements with prices in the range from `Price1` to `Price2`. Have you already guessed what this is about? We talk about aggregate functions. If you want to execute the

query with the aggregate result, you should use the method syntax. Let's write the following queries:

```
//there is a class, which describes a product
class Product
{
    public string Name { get; set; }
    public int Price { get; set; }
}
//create a collection of products, using the
//GetProducts() method
List<Product> products = GetProducts();
```

Will this query solve the task we face?

```
var maxPrice = products.Select(p => p.Price).Max();
```

It will not, of course. From our collection of products, this query will return the maximum price but not an object with the maximum price. The object can be obtained as follows:

```
int maxPrice = products.Max(p => p.Price);
var item = products.First(x => x.Price == maxPrice);
```

We received the result, but at the cost of two queries. Let's consider one more solution. For LINQ, you can connect the morelinq extension package. In order to install it, you should enter the Install-Package morelinq command into the Package Manager console. This extension contains the MaxBy() useful method:

```
var maxItem = products.MaxBy(p => p.Price);
```

In order to define the number of products with prices in the range from 100 to 1000, you can execute the following query:

```
var prodCount = products.Count(p => p.Price > 100 &&  
                                p.Price <= 1000);
```

One detail is related to the method syntax that often discourages programmers from applying this kind of writing queries. The fact is that most LINQ methods use methods or functions as parameters. It's necessary to use delegates in order to pass these parameters, but in this case, LINQ prefers to use lambda-expressions.

Lambda-expressions are a short way to write anonymous methods and functions. The `=>` operator is a distinctive symbol of lambda-expressions, which separates parameters and selection condition. The lambda-expression takes the following basic form:

(parameters) => expression

For example:

- `x => x > 100`, describes the function, which returns true, if the parameter is more than 100, and false if it's otherwise;
- `(x,y) => x+y`, describes the function which returns the sum of its parameters;
- `x => x.StartsWith("L")`, do you recognize? This function returns true, if the parameter begins with the "L" string, and false if it's otherwise.

The lambda-expression can have an arbitrary number of parameters, and as a rule, they should be enclosed in parentheses. Parentheses can be omitted only in the case of one parameter. But it would be better to use them in this case as well. As a rule, the compiler itself always tries to define the type of lambda-expression parameters, but if this is difficult for it, you can specify the parameter type explicitly:

- `(int x, string s) => s.Length > x`, this function selects strings, which length is more than `x`.

In order to initialize delegates, you can use lambda-expressions not only in LINQ but in a regular code as well:

```
delegate int del(int i);
static void TestLambda()
{
    del myDelegate = (x) => x * x;
    int sq = myDelegate(5); //sq = 25
}
```

We will focus on the use of lambda-expressions in the LINQ queries.

Add the following method to our project and call it in `Main()` instead of the previous method:

```
private static void LinqMethod2()
{
    //the array is a data source
    string[] countries = { "Albania", "UK",
        "Lithuania", "Andorra", "Austria", "Latvia",
        "Liechtenstein", "Switzerland", "Ireland",
        "Sweden", "Italy", "France", "Liechtenstein",
        "Spain", "Turkey", "Germany", "Switzerland",
        "Monaco", "Montenegro", "Norway", "Finland",
        "Turkey", "UK", "Poland", "Portugal",
        "Lithuania", "Luxembourg"
    };

    //LINQ query
    var result = countries.Where(c => c.StartsWith("L"));

    Console.WriteLine("Countries beginning with L:");
    //continue to work with the query
    foreach (var item in result)
    {
```

```

        Console.WriteLine(item);
    }
    Console.Write("Press Enter to complete");
    Console.ReadLine();
}

```

The result of this method will be similar to the result of the previous one. This is how it should be since we have written the same query in another form. In the highlighted string, you see the `Where()` method call on behalf of the data source (our array). The lambda-expression, which contains an anonymous function-predicate for selecting values is transferred to the `Where()` method as a parameter. The compiler converts it into an anonymous method, which is executed by the `Where()` method on each collection element. If this anonymous method returns true, the element will be included in the result-set, if false, the element will not be included in the result-set.

Make the query code in the first method look like this and run the application:

```

var result = from c in countries where c.StartsWith("L")
              orderby c.Length select c;

```

We added the requirement of sorting the resulting dataset by the length of the country name, and as you can see, the result-set is sorted. If you want to sort it in descending order, you should add the descending specifier:

```

var result = from c in countries where c.StartsWith("L")
              orderby c.Length descending select c;

```

In the method syntax, this sorting will look as follows:

```
var result = countries.OrderBy(c => c.Length) .  
    Where(c => c.StartsWith("L"));
```

or, to sort in descending order:

```
var result = countries.OrderBy(c => c.Length) .  
    Where(c => c.StartsWith("L"));
```

Is there Distinct() in the method syntax? Here you go:

```
var result = countries.OrderBy(c => c.Length) .  
    Where(c => c.StartsWith("L")).Distinct();
```

LINQ supports the execution of standard aggregate operations. In order to do this, you should call relevant methods on behalf of the result-set. For example:

```
Console.WriteLine(result.Max());  
Console.WriteLine(result.Min());  
Console.WriteLine(result.Count());  
Console.WriteLine(result.Average());  
Console.WriteLine(result.Sum());
```

However, Average() and Sum() make sense only for numerical collections. We considered the LINQ to Objects basic features for a simple dataset. Let's now use a collection of some objects as a data source.

Add the following class to the project:


```

class Product
{
    public string Name { set; get; }
    public int Price { set; get; }
    public string Manufacturer { set; get; }
    public int Count { set; get; }

    public override string ToString()
    {
        return String.Format("{0} {1} {2} {3}",
                               this.Name,
                               this.Price, this.Manufacturer, this.Count);
    }
}

```

After that, add one more method to the main class of the application:

```

private static void LinqMethod3()
{
    //use this array once again but not as a data
    //source
    string[] countries = { "Albania", "UK",
                           "Lithuania",
                           "Andorra", "Austria", "Latvia",
                           "Liechtenstein", "Switzerland",
                           "Ireland", "Sweden", "Italy", "France",
                           "Liechtenstein", "Spain", "Turkey",
                           "Germany", "Switzerland", "Monaco",
                           "Montenegro", "Norway", "Finland",
                           "Turkey", "UK", "Poland", "Portugal",
                           "Lithuania", "Luxembourg"
    };
    //now, the list of objects will be a data source
    List<Product> products = new List<Product>();
    //generate the list of 100 random objects
    for (int i= 0; i<100; i++)
    {

```

```

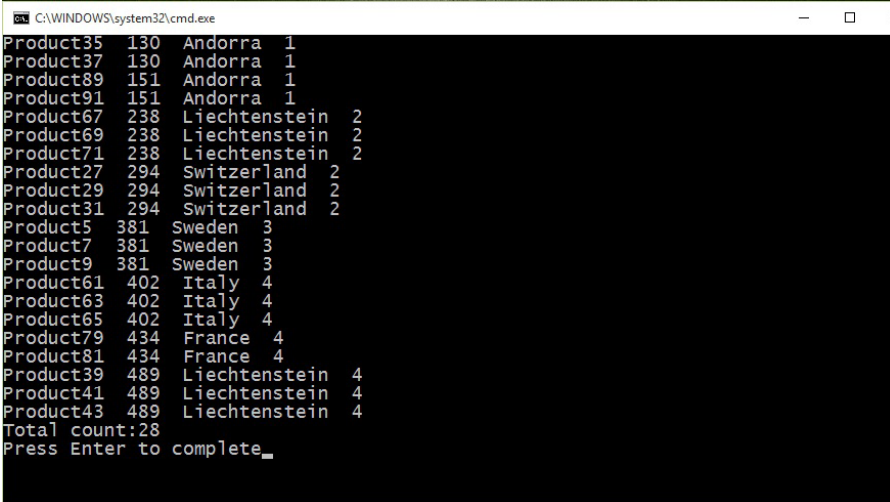
//compiler needs Sleep() to create
//a new Random object at each
//iteration, otherwise, the compiler uses
//optimization and doesn't change the Random
//object for all iterations
Thread.Sleep(5);
products.Add(new Product { Name = "Product" + (++i),
    Price = (new Random()).Next(0,1000),
    Manufacturer =
        countries[(new Random()).
            Next(0, countries.Length - 1)],
    Count = (new Random()).Next(0, 10) });
}
//if you want to view the created list
Console.WriteLine("All products:");
foreach (var item in products)
{
    Console.WriteLine(item);
}

//LINQ query to the list of objects
var result = from c in products
    where c.Price<500
    orderby c.Price
    select c;

Console.WriteLine("\nProducts with price less than 500:");
//continue to work with a query
foreach (var item in result)
{
    Console.WriteLine(item);
}
Console.WriteLine("Total count:"+result.Count());
Console.Write("Press Enter to complete");
Console.ReadLine();
}

```

Run the application by calling a new method in Main(). I got the following result:



```

C:\WINDOWS\system32\cmd.exe
Product35 130 Andorra 1
Product37 130 Andorra 1
Product89 151 Andorra 1
Product91 151 Andorra 1
Product67 238 Liechtenstein 2
Product69 238 Liechtenstein 2
Product71 238 Liechtenstein 2
Product27 294 Switzerland 2
Product29 294 Switzerland 2
Product31 294 Switzerland 2
Product5 381 Sweden 3
Product7 381 Sweden 3
Product9 381 Sweden 3
Product61 402 Italy 4
Product63 402 Italy 4
Product65 402 Italy 4
Product79 434 France 4
Product81 434 France 4
Product39 489 Liechtenstein 4
Product41 489 Liechtenstein 4
Product43 489 Liechtenstein 4
Total count:28
Press Enter to complete_

```

Fig. 2. LINQ query result to the list of objects

Play around with different selection conditions. Try to execute a query to our list, written in the method syntax. You will succeed. Now, we can sum up that LINQ queries work with collections of user objects very well. But specifically in this case, LINQ allows you to do even more. Let's continue to consider our example and pay more attention to the select specifier. Change the query as follows:

```

var result = from c in products where c.Price<500
              orderby c.Price
              select c.Name;

```

Now in the final loop, you will see only names of products. You can even apply acceptable transformations to the query results:

```
var result = from c in products where c.Price<500
              orderby c.Price
              select c.Name.ToUpper();
```

In this case, names of products will be displayed in uppercase. Let's try such a query, wherein we want to see names and prices of selected products:

```
var result = from c in products where c.Price<500
              orderby c.Price
              select c.Name, c.Price;
```

Oops. This query caused a compilation error. The fact is that you can specify only one element after the select specifier. But if you rewrite the query as follows:

```
var result = from c in products where c.Price<500
              orderby c.Price
              select new { c.Name, c.Price };
```

there will be one object in select. Try to run the application with this query. I got the following result (Fig. 3).

In this case, the LINQ query returned a collection of new objects created according to the instance specified in select. You can guess about this even by the appearance of the output objects. Try to write the same query but in the method syntax:

```
var result = products.Where(p => p.Price < 500).
    Select(p => new { p.Name, p.Price });
```

Run the application and make sure that this query operates like the previous LINQ query in the query syntax.

In the SQL syntax, some databases support such a statement

```

C:\WINDOWS\system32\cmd.exe
Name = Product33, Price = 34
Name = Product7, Price = 121
Name = Product9, Price = 121
Name = Product11, Price = 121
Name = Product61, Price = 142
Name = Product63, Price = 142
Name = Product65, Price = 142
Name = Product79, Price = 173
Name = Product81, Price = 173
Name = Product83, Price = 173
Name = Product41, Price = 229
Name = Product43, Price = 229
Name = Product45, Price = 229
Name = Product3, Price = 285
Name = Product5, Price = 285
Name = Product73, Price = 337
Name = Product75, Price = 337
Name = Product77, Price = 337
Name = Product53, Price = 424
Name = Product55, Price = 424
Name = Product13, Price = 480
Name = Product15, Price = 480
Name = Product17, Price = 480
Total count:25
Press Enter to complete

```

Fig. 3. LINQ query execution with creating new objects

as TOP N, which allows you to select the first N strings from the select query result. For example, you have a Books table with lots of strings but you are only interested in the first 10 strings from the select * from Books query result. So, you can write the following query:

```
select TOP 10 * from Books
```

and you will get only 10 strings from the Books table. It's especially convenient, if you want, for example, to get ten cheapest books. To do this, write the following query:

```
select TOP 10 * from Books order by Price
```

In this case, you will get the first ten strings from the list of books sorted by prices. But you should consider that it may be not ten books with ten different prices. It will be exactly ten strings, and all of them can be either with the same minimum price or with different prices

LINQ comprises methods, which execute the similar work. These are Take() and Skip() methods. These methods relate to partitioning operations and allow you to make selections from the result set in opposite ways. Both of these methods are applied to the result set. The Take() method allows selecting the specified number of elements, but the Skip() method allows you to miss the specified number and select other elements. Let's look at how this works. Change the last method of our application to the following form:

```
private static void LingMethod3()
{
    //the array is a data source
    string[] countries = { "Albania", "UK", "Lithuania",
        "Andorra", "Austria", "Latvia",
        "Liechtenstein", "Switzerland", "Ireland",
        "Sweden", "Italy", "France", "Liechtenstein",
        "Spain", "Turkey", "Germany", "Switzerland",
        "Monaco", "Montenegro", "Norway", "Finland",
        "Turkey", "UK", "Poland", "Portugal",
        "Lithuania", "Luxembourg"
    };
    List<Product> products = new List<Product>();
    for (int i= 0; i<100; i++)
    {
        Thread.Sleep(5);
        products.Add(new Product { Name = "Product" + (++i),
            Price = (new Random()).Next(0,1000),
            Manufacturer = countries[(new Random()).
                Next(0, countries.Length - 1)],
            Count = (new Random()).Next(0, 10) });
    }
}
```

```

var result = products.Where(p => p.Price > 800)
    Select(p => new { p.Name, p.Price });

Console.WriteLine("\nProducts with price greater
                    than 800:");
//continue to work with the query
foreach (var item in result)
{
    Console.WriteLine(item);
}

Console.WriteLine("Total count:"+result.Count());
Console.WriteLine("\nTop 5:\n");

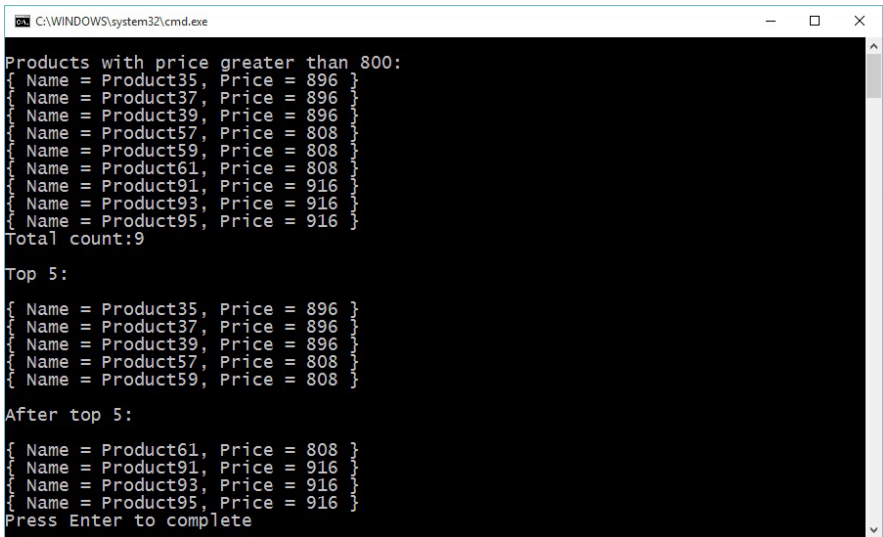
foreach (var item in result.Take(5))
{
    Console.WriteLine(item);
}
Console.WriteLine("\nAfter top 5:\n");
foreach (var item in result.Skip(5))
{
    Console.WriteLine(item);
}

Console.Write("Press Enter to complete");
Console.ReadLine();
}

```

As you see, the results output was mainly changed. Now, we output results in three loops. In the first one, we output the result set, in the second one, we output the first five strings, in the third one, we output other strings that follow after the first five strings. We use `Take()` and `Skip()` methods specifically in the second and third loops. We changed the selection condition a bit in order to reduce a number of objects in the output collection and place all results in one window.

I got the following result:



```

C:\WINDOWS\system32\cmd.exe

Products with price greater than 800:
{ Name = Product35, Price = 896 }
{ Name = Product37, Price = 896 }
{ Name = Product39, Price = 896 }
{ Name = Product57, Price = 808 }
{ Name = Product59, Price = 808 }
{ Name = Product61, Price = 808 }
{ Name = Product91, Price = 916 }
{ Name = Product93, Price = 916 }
{ Name = Product95, Price = 916 }
Total count:9

Top 5:
{ Name = Product35, Price = 896 }
{ Name = Product37, Price = 896 }
{ Name = Product39, Price = 896 }
{ Name = Product57, Price = 808 }
{ Name = Product59, Price = 808 }

After top 5:
{ Name = Product61, Price = 808 }
{ Name = Product91, Price = 916 }
{ Name = Product93, Price = 916 }
{ Name = Product95, Price = 916 }
Press Enter to complete
  
```

Fig. 4 Use of Take() and Skip()

There are other useful methods that allow selecting the first element that was met in the set and corresponds to the specified condition. These are First() and FirstOrDefault() methods. Add the following strings to the last method:

```

var pr = products.First(p => p.Price > 500 &&
    p.Manufacturer.StartsWith("L"));
Console.WriteLine("\nFirst:"+pr);
  
```

While this code is executed, you can see the found element description on the screen, but if not a single element in products corresponds to the specified conditions, you will get an exception.

But if you add the following strings:


```
var pr = products.FirstOrDefault(p => p.Price > 500 &&  
    p.Manufacturer.StartsWith("L"));  
Console.WriteLine("\nFirst:" + pr);
```

you will not get the exception. You will see either the found element description or null, if there is no such element in products.

LINQ comprises methods, which execute the similar work. These are Take() and Skip() methods. These methods relate to partitioning operations and allow you to make selections from the result set in opposite ways. Both of these methods are applied to the result set. The Take() method allows selecting the specified number of elements, but the Skip() method allows you to miss the specified number and select other elements. Let's look at how this works. Change the last method of our application to the following form.

LINQ To SQL overview

LINQ To SQL translates LINQ queries into SQL queries automatically and allows you to work with such databases as MS SQL Server and Oracle. To do this, it is necessary to solve ORM tasks, when you create the application classes that correspond to database tables. All these issues will be considered in detail in our following lessons when studying Entity Framework. Today, Entity Framework overshadowed the LINQ To SQL technology. Although, it will be more correctly to say that it integrated this technology into itself and developed it even more.

Let's consider an example of using LINQ To SQL for our database. Create a console application named LinqToSqlTest.

Activate the Server Explorer option in the View menu of this project. Select the Data Connections option and Add Connection command from the context menu in the opened Server Explorer window.

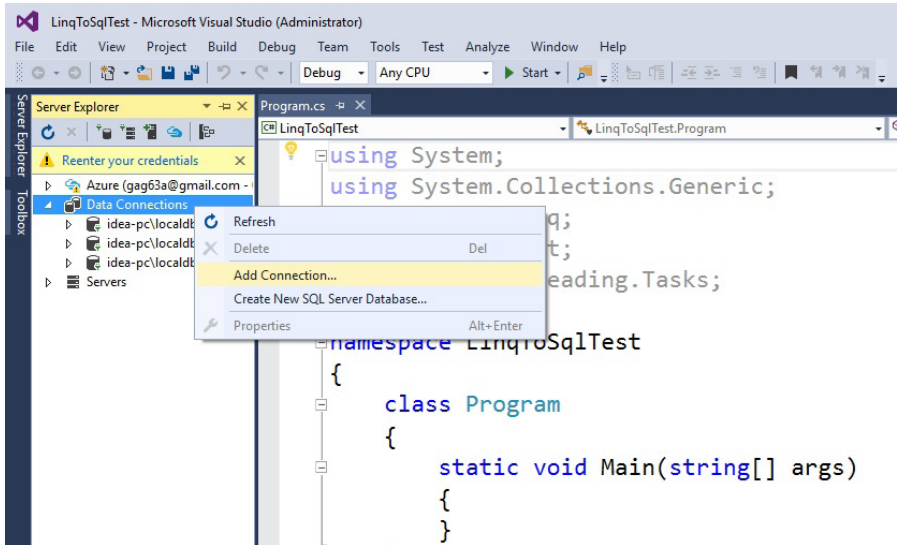


Fig. 5. Server connection

After that, execute the Add New Connection action. In the opened window, specify the server address as (localdb)\v11.0 and select our database. That's all. Communication between the database and application is installed (Fig. 6).

Add Connection

Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.

Data source:
Microsoft SQL Server (SqlClient) Change...

Server name:
(localdb)\v11.0 Refresh

Log on to the server

☒ Use Windows Authentication
☐ Use SQL Server Authentication

User name:
Password:
☐ Save my password

Connect to a database

☒ Select or enter a database name:
LibraryDb

☐ Attach a database file:
 Browse...
Logical name:

Advanced...

Test Connection OK Cancel

Fig. 6. Database connection

Now, select our application in the Solution Explorer window and click the right mouse button. Select actions Add – New Item – LINQ to SQL Classes. It will be an object with the dbml extension, which will help us to create classes that correspond to database tables in the application (Fig. 7).

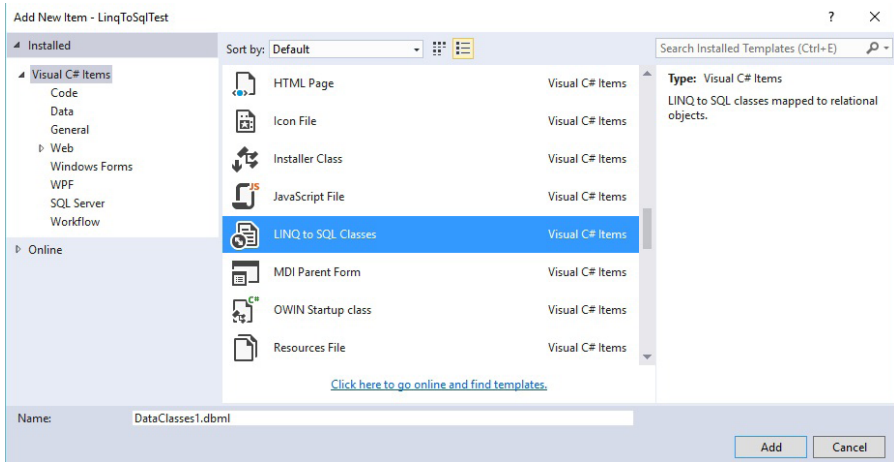


Fig. 7. Adding LINQ to SQL Classes

When this object is created, change nothing in the Visual Studio windows and return to the Server Explorer window. Expand the node of the created connection, after that, expand the node of our database, and after that, expand the Tables node, in order to see our tables. Drag from our database and throw the Books and Authors tables on the added DataClasses1.dbml element (which is open now in the central window). After these actions, you will see the database diagram. Fields of all tables and relationships between them will be displayed on the diagram. Now, you can close the DataClasses1.dbml element by saving it.

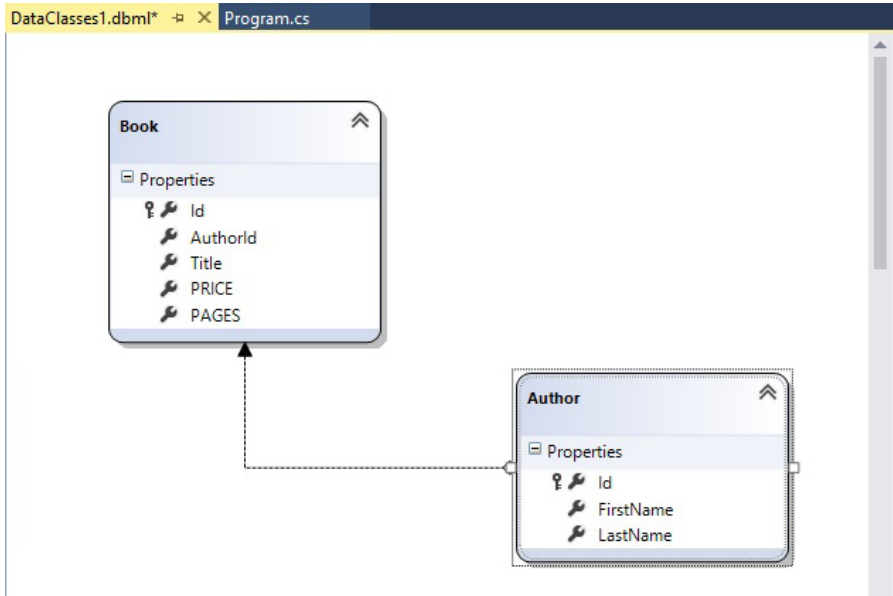


Fig. 8. Database diagram

Now, navigate to the Solution Explorer window, expand the `DataClasses1.dbml` element in the project, and select the `DataClasses1.designer.cs` element in it. This element contains classes, which the studio created automatically in our application while we have dragged tables from a database. Pay more attention to the following classes:

`DataClasses1DataContext` is derived from `System.Data.Linq.DataContext`, this class is usually called a database context and is a mean to connect to a database, it is a tool to pass queries to a database and get results of these queries;

`Book` is derived from the `INotifyPropertyChanged` interface, this class is the Books database table template in the application; properties of this class correspond to the table fields. We will work with objects of this class in the application, and LINQ

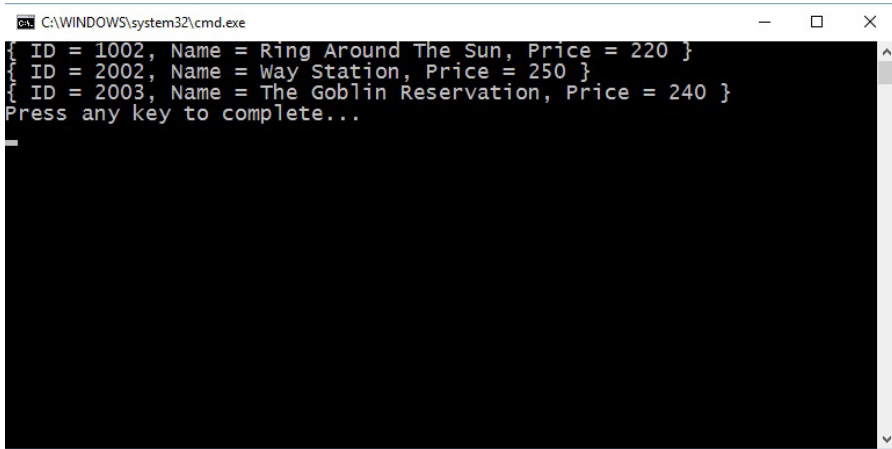
To SQL will translate our actions into the database using a database context.

The **Author** class, which corresponds to the Author table, is created similarly to the Books table class in our application.

Now, our application is ready to work with the database. Enter the following code into the Main() method:

```
static void Main(string[] args)
{
    DataClasses1DataContext db =
        new DataClasses1DataContext();
    var queryResults = from c in db.Books
                        where c.PRICE > 200
                        select new
                        {
                            ID = c.Id,
                            Name = c.Title,
                            Price = c.PRICE
                        };
    foreach (var item in queryResults)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine("Press any key to complete...");
    Console.ReadLine();
}
```

Now, execute the created application. In my case, results look as follows:

A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window has a black background with white text. It displays the results of a LINQ query, showing three objects in curly braces. Each object contains three fields: ID, Name, and Price. The first object has ID 1002, Name "Ring Around The Sun", and Price 220. The second object has ID 2002, Name "Way Station", and Price 250. The third object has ID 2003, Name "The Goblin Reservation", and Price 240. Below the objects, the text "Press any key to complete..." is visible. The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

```
C:\WINDOWS\system32\cmd.exe
{ ID = 1002, Name = Ring Around The Sun, Price = 220 }
{ ID = 2002, Name = Way Station, Price = 250 }
{ ID = 2003, Name = The Goblin Reservation, Price = 240 }
Press any key to complete...
```

Fig. 9. Program execution results

Let's discuss the received results. When we added the LINQ to SQL Class object with the dbml extension to the project, we had an opportunity to execute a whole number of important actions using this object. At first, we create an application class that is derived from `System.Data.Linq.DataContext`, which will be an access tool to a database for the application. When dragging the database tables into the dbml object, the Book and Author classes are created in our application, and they correspond to tables. Properties of the created classes correspond to the table fields.

The code located in the `Main()` method is an example of using LINQ To SQL in a query format. First of all, we create an object of the database context. We refer to the Books table through this object, select books with the price of more than 200 from it, and form new objects, which are entered into the `queryResults` variable and output them. After that, we output the received results in the loop.

So, LINQ To SQL allows us to work with data in the relational database tables in the style of OOP. We communicate with classes that correspond to database tables. All actions applied to objects of these classes lead to changing the data state in database. And all of this is carried out in the style of OOP with the opportunity to write LINQ queries in C#.

LINQ To XML overview

Why is LINQ To XML used? This extension is not designed to solve tasks, which such API as XML DOM, XPath, XQuery, XSLT and others face. LINQ To XML simply offers additional opportunities to create and survey data in the XML format. Yes, you understood everything correctly. LINQ To XML allows you to create XML documents and work with them after that. And you can, of course, upload the existing XML-files.

In C#, create a new console project, wherein we will consider the LINQ To XML main features. At first, we will add a method to the project, wherein we will create a new XML document from scratch. In order to create such documents, it's necessary to add the System.Xml.Linq namespace reference to the project. This namespace contains the so-called functional constructors for creating XML objects. The feature of these constructors is the clarity of their use. Calls of these constructors are inserted one into another, repeating the structure of the created XML object. So, the object creation allows you to see its structure immediately. Add the following method to the created project:


```

static void CreateXmlDocument()
{
    XDocument xmldoc = new XDocument(
        new XElement("computers",
            new XAttribute("Price", "800"),
            new XAttribute("Warranty", "2 years"),
            new XElement("CPU",
                new XAttribute("Name", "Intel Core i7-6700K"),
                new XAttribute("GHz", 2.5)
            ),
            new XElement("HDD",
                new XAttribute("Name", "Samsung 850 PRO"),
                new XAttribute("Size", 1.0)
            ),
            new XElement("computer",
                new XAttribute("Price", "900"),
                new XAttribute("Warranty", "2 years"),
                new XElement("CPU",
                    new XAttribute("Name", "AMD A10-5800K"),
                    new XAttribute("GHz", 2.5)
                ),
                new XElement("HDD",
                    new XAttribute("Name", "Transcend
ESD400"),
                    new XAttribute("Size", 1.0)
                )
            )
        );
    Console.WriteLine(xmldoc);
}

```

Call this method in Main() and execute the created project. You will see the structure of the created XML object on the screen. The object itself call the XDocument () constructor. Since in

accordance with the XML laws, an object should contain a root element, the call of the XElement () constructor follows further. XElement () constructor creates both the root element and elements that are nested into the root element. As a parameter, this constructor can accept one more XAttribute() constructor that is designed to create attributes. This last constructor accepts an attribute name and its value as a parameter. After that, the content of the created object is output to the screen. The above code speaks for itself, and there is nothing to explain additionally here. The only thing that should be said is that if it's necessary to create an element text node, the content of this node should be passed to the XElement () constructor as a parameter. For example,

```
new XElement("computer",
    "This is not expensive and reliable
computer",
    new XAttribute("Price", "800"),
    new XAttribute("Warranty", "2 years"),
    new XElement("CPU",
        new XAttribute("Name", "Intel Core i7-
6700K"),
        new XAttribute("GHz", 2.5)
    )
)
```

Let's look at how the created object can be recorded into a file. To do this, add the following strings to the created method after outputting the XML object to the screen:

```
xmlFilePath = @"example.xml";
xmldoc.Save(xmlFilePath);
```

Execute the project and navigate to a root folder of the application. You will see the created XML-file there. In order to

verify that this file is well formed, open it in Internet Explorer. I saw the following picture:

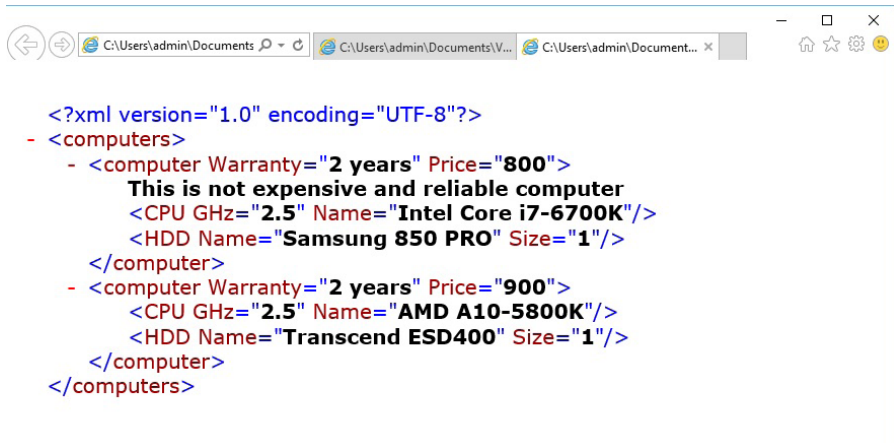


Fig. 10. Created XML-file

In order to read the existing XML-file, it's necessary to execute the following actions:

```
XDocument xmlDoc2 = XDocument.Load(xmlFilePath);
```

That is, it's necessary to create an XML object of the XDocument type, call the static Load() method on behalf of the XDocument class by specifying a path to the existing XML-file as a parameter, and assign the result of this method execution to the created object. There is a possibility to create an XML file from the string that contains XML data. To do this, use the Parse() method:

```
XDocument xmlDoc3 = XDocument.Parse(xmlString);
```

In this example, xmlString is a string with the XML data.

As you understand, `XDocument` is an XML object, which we want to use as a data source. The `XDocument` class has several very important properties, which will help us to access the content. These are `DescendantNodes` and `Descendants` properties. The first property is a collection of all nested nodes (including elements, text strings, comments), and the second one is a collection of nested elements only. Namely these two collections often act as data sources, by working with XML objects. Let's look at the example of the use of these properties. Add the following method to our project:

```
static void ReadXmlDocument ()
{
    string xmlFilePath = @"example.xml";
    XDocument xmldoc2 = XDocument.Load(xmlFilePath);

    var result = from c in xmldoc2.Descendants(XName.
        Get("computer"))
        where Convert.ToInt32(c.Attribute(XName.
            Get("Price")).Value) < 850
        select c;

    foreach (var item in result)
    {
        Console.WriteLine(item);
    }
}
```

Insert the call of this method into `Main()` and execute our application. In my case, the result looked as follows:

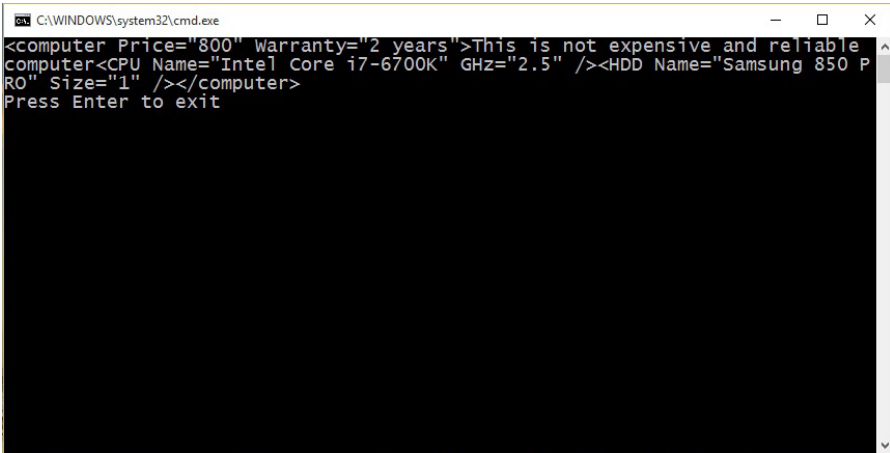


Fig. 11. Selection from the XML file

There is little beauty, but the sense is clear. We have got a list of elements from our XML file that correspond to the specified condition – the price is less than 850. You should keep in mind that LINQ is especially good for working with very large collections, and you should use it only as a demo-example for our little file.

This method contains the known LINQ query in the query syntax, which selects computer elements with the Price attribute value less than 850 from all the nodes of the specified file. Along the way, we convert the attribute value from a string to an integer.

It's clear that this query can be rewritten in the method syntax. It can look as follows:

```
var result = xmldoc2.Descendants(XName.Get("computer")).
    Where(c => Convert.ToInt32(c.Attribute(XName.
        Get("Price")).Value) < 850);
```

Pay attention to the lambda expression in the Where() method. The result of this query will be similar to the result of the previous one. It will be a useful exercise for you to find a fairly large XML file that contains as much nested elements as possible, and make several different selections from this file using LINQ To XML.

Conclusions

Now you have considered all the modern means to access data sources that are in .NET for C#. One of the tasks, which you will face in the future by working with data providers, is to select the technology, which will be the most suitable for solving the specified task. You will get acquainted with a very effective tool, which can be an alternative to everything, what you have already learned so far.