



Database Access Technology

ADO.NET

Lesson №1

Connected mode

Contents

ADO.NET overview.....	3
Connected mode	7
Database creation	7
Database Connection	14
Query creation and execution (DbCommand)	19
Obtaining and processing the query results (DbDataReader).....	24
Batch processing of queries	31
Configuration file.....	35
Parameterized queries in DbCommand.....	37
Stored procedures in DbCommand	41
Homework.....	47

ADO.NET overview

You already know what a relational database is. Now you should add the ability to write programs to this knowledge, and the relational databases will use these programs for their needs. Within this course you will get acquainted with the ADO.NET framework, designed for working with the databases in .NET Framework. ADO.NET is a set of classes, combined in the System.Data namespace and allowing you to carry out the operation with such data providers as MS SQL Server, OLE DB, ODBC и Oracle. In other words, you will learn to write programs in C# which will be able to work with these relational databases.

What stages does the database operation consist of? First of all, the application should connect to a data provider, which contains the necessary database. The program product that is called the database server or the DBMS, which is a container for the user databases, can be a database provider. To connect to such a server, you should know the server address, name and password for a server connection, the necessary database name, sometimes, some values. Once connected to a server and opened your database, you should have tools that allow you to send your SQL queries to a server and to receive the results of these queries from the server. So your programs will have to be able to connect to servers with the database and execute the database queries. Moreover, your programs will have to be able to store the data received from the database. These two main tasks: access to the databases and storing the

information from the tables, are solved by the classes that ADO.NET comprises.

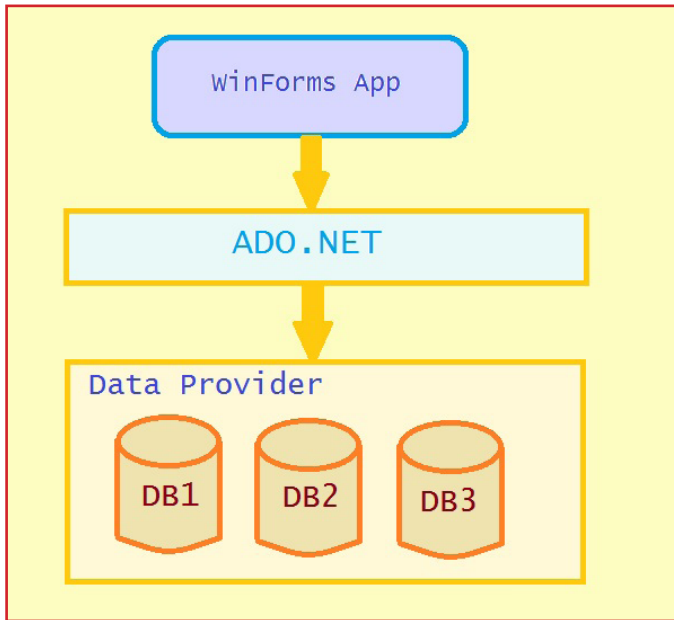


Fig.1. Interaction between the application and database server

To connect to a database and execute the SQL queries the following classes are used:

- `DbConnection;`
- `DbCommand;`
- `DbDataReader;`
- `DbDataAdapter;`

To store the database data, such classes are used:

- `DataTable;`
- `DataSet;`
- Others.

From the beginning of our course you should remember that there are two principally different ways of the application operation with the database: connected mode and disconnected mode. Below we are going to talk about them in more detail.

Within the connected mode an application connects to the database and remains connected for a long time. During this time the application can refer to a database and execute any queries. At the same time the application can do its own affairs without resorting to the database but holding the database connection open.

Within the disconnected mode, an application connects to a database, performs the necessary action, for example, reads data from one or several tables, and immediately disconnects from the database keeping the read data in the relevant classes for a local operation with it. If it is necessary to execute a new query, an application connects to the database again, executes the query and disconnects immediately.

In other words, within the connected mode, an application remains connected to a database for a long time, while within the disconnected mode an application connects to a database in the discrete mode only for a time of executing each query.

Each of these ways of operation has its own advantages and disadvantages and you should learn to use an optimal method in each specific situation. Or combine both of these ways of database operation correctly.

At this stage let's note the main features of these modes. Since the server connection is a labor-intensive and time consuming procedure, therefore within the connected mode the queries are carried out faster, because each subsequent

query shouldn't connect to the database. This connection was executed and remains open. But this method of operation overloads the server. Because the server has to keep in touch with many clients, even with those, that aren't working with it at the moment but keep their connection open. The second mode overloads the server much less, since each client connects only for a microscopic period of time while the query is being executed, and then immediately disconnects. But in this case each query has to connect "from scratch", that slows down the application operation. Let's focus on these differences. We are going to talk about others later.

Let's proceed to the consideration of the main stages of the ADO.NET application.

Connected mode

Database creation

Let's create a simple database named Library, and create two related tables Books and Authors in it. The database can be placed either on the allocated server or on the local server. The principles of the database operation don't depend on its location.

We run Visual Studio 2015 and create a console application with the random name, for example, AdoNetLibrary1. First of all, we should create a database which will be used for our application operation. It can be done in Visual Studio 2015 directly. Navigate to the View menu and activate the SQL Server Object Explorer option. Then select the SQL Server node, activate the context menu and select the Add SQL Server option.

In the appeared window you'll see the attributes of the created

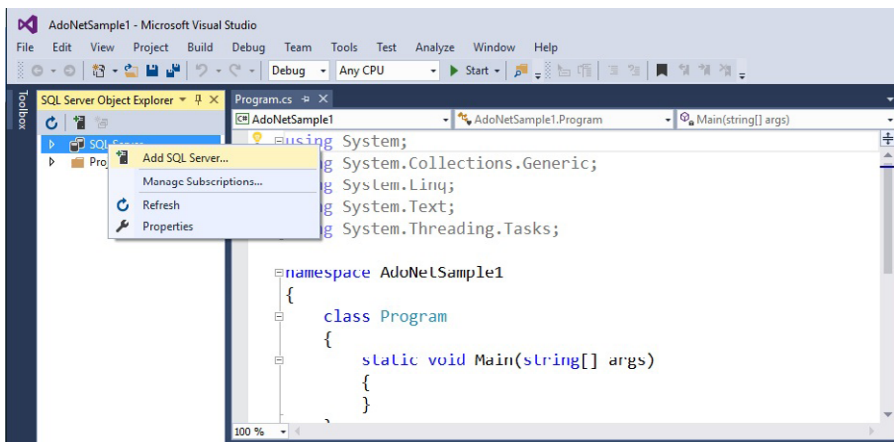


Fig.2. Adding a new SQL Server

server. Visual Studio 2015 creates the LocalDB server by default. LocalDB is a lighter version of SQL Server Express Edition. It supports all the SQL Server Express Edition functionality (except for FileStream) but it's installed faster and is executed in the user mode, but not like a service. LocalDB is not designed for scenarios with the remote connection, but, since the database itself is stored in a file (with the .mdf extension), allows you to transfer the application from the computer to another computer easily. In other words, LocalDB is a perfect tool for debug (and not only), it will be useful for everyone to learn to work with it. Much more information about LocalDB you can search on the Web, for example, [here](#).

This option remains unchanged. The authentication method to access the server also remains unchanged. Click Connect.

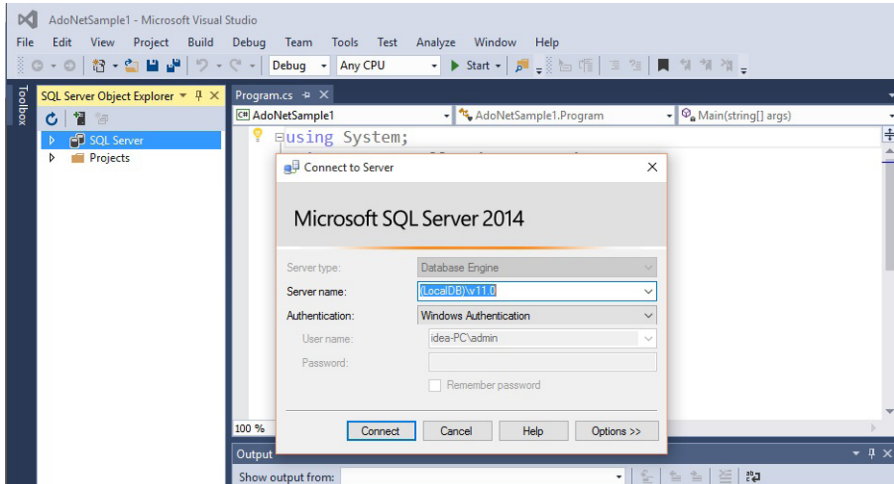


Fig.3. Creating a new SQL Server

After performing these actions, we have a data provider that can store our user databases and control them.

Now, one should create a new database. Our database will consist of two related tables Books and Authors. It will be enough for us to have such a simple database at this stage in order to get acquainted with the base opportunities of the ADO.NET framework. To create a new database, navigate to the SQL Server Object Explorer window again. Expand the node of the created server. Select the Databases node and activate the context menu for it. Activate the Add New Database option in the context menu.

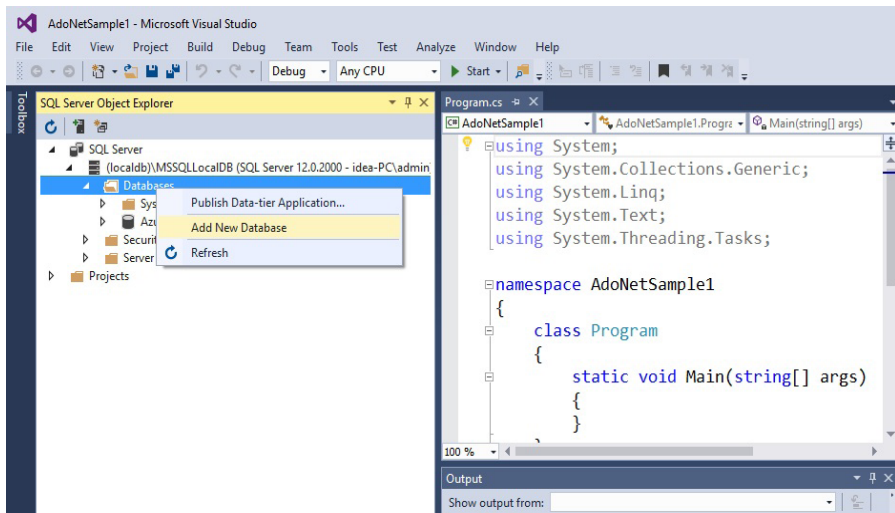


Fig. 4. Adding a new database

In the appeared window, you can specify the name of the database that is created and its location. Enter the Library name for our database, but its location remains unchanged.

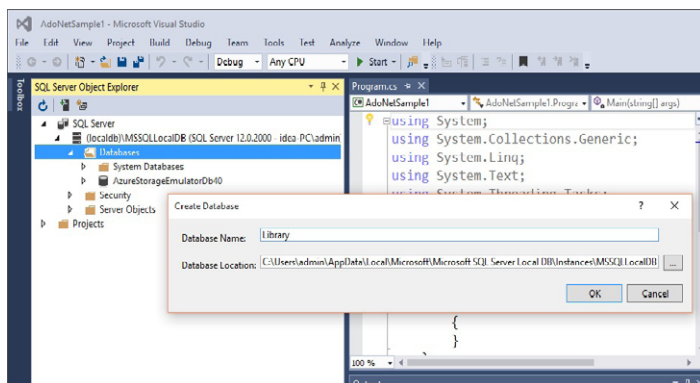


Fig. 5. Creating a new Library database

Now we can proceed to creating the tables in our database. To do it one should navigate to the Tables node of the created database and activate the Add New Table option from the context menu.

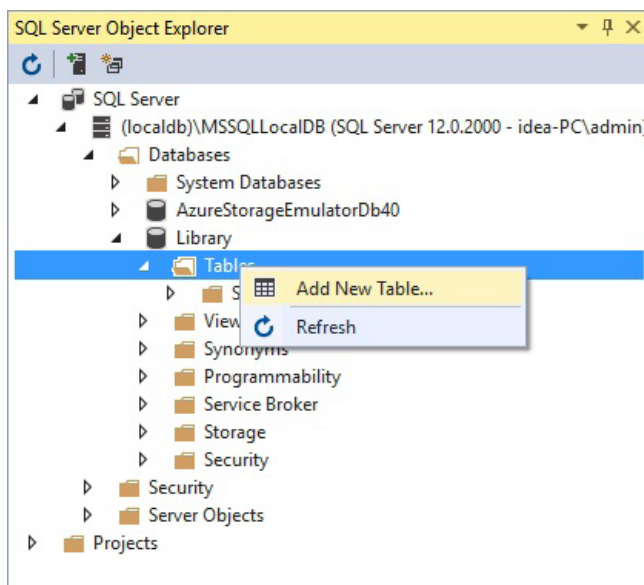


Fig. 6. Adding the new table to the Library database

After performing this action, Visual Studio 2015 will bring you to the database structure change mode. In this mode one can create the tables. For creating the tables one can use the opportunities of the built-in graphics editor. We are going to create our tables using the DDL SQL queries. Below there are queries for creating our two tables and relation between them. At first, one should create the Authors table, and then the Books one.

```
CREATE TABLE [dbo].[Authors]
(
    Id INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
    FirstName VARCHAR(100) NOT NULL,
    LastName VARCHAR(100) NOT NULL
)

CREATE TABLE [dbo].[Books]
(
    Id INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
    AuthorId INT NOT NULL,
    FOREIGN KEY (AuthorId) REFERENCES AUTHORS (Id),
    Title VARCHAR(100) NOT NULL,
    PRICE INT,
    PAGES INT
)
```

Copy the query for creating the Authors table to the T-SQL window in the bottom of the screen, check that the query was copied without bugs and click the Update button on the top toolbar.

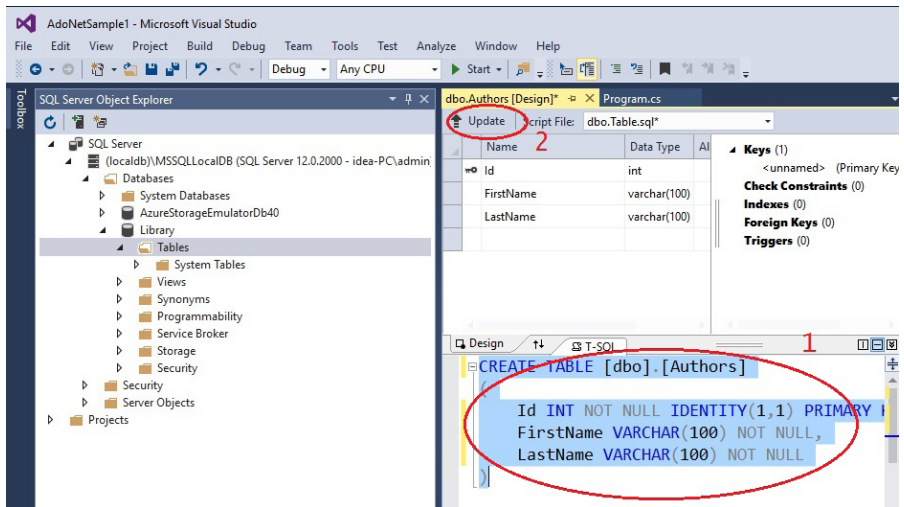


Fig. 7. Creating the Author table in the Library database

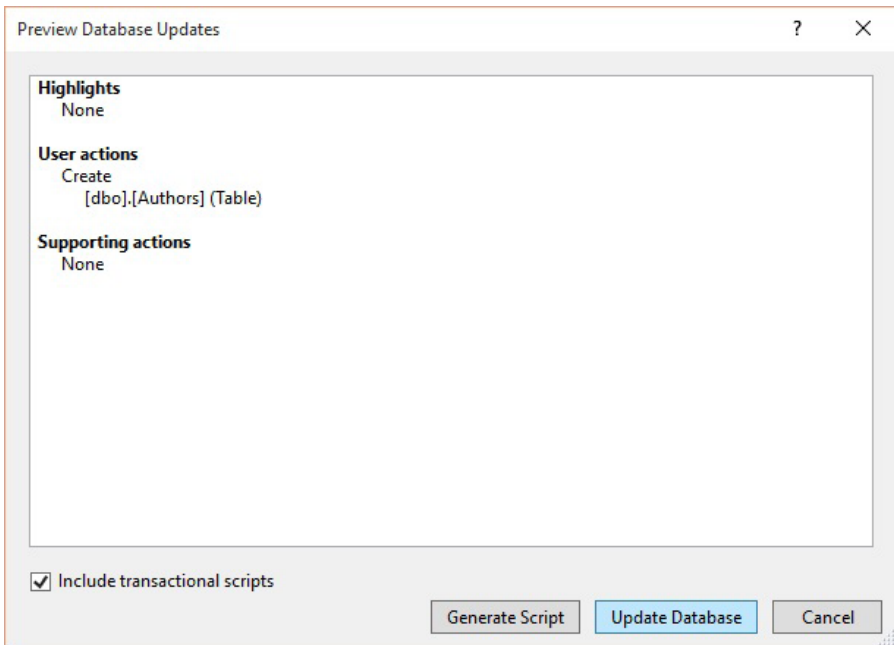


Fig. 8. Confirming the query execution

After clicking the Update button, Visual Studio 2015 will prompt you to confirm the action leading to the database structure change. (Fig. 8).

In the appeared window one should click the Update Database button and a new database will be created. To be more precise, after clicking this button, a query that is positioned in the T-SQL window will be executed. In our case this query creates a new table.

Similarly copy the query for creating a Books table and execute it. Now in our database all the necessary tables and relations between them are created. Expand the Tables node in our database and you will see the created tables there.

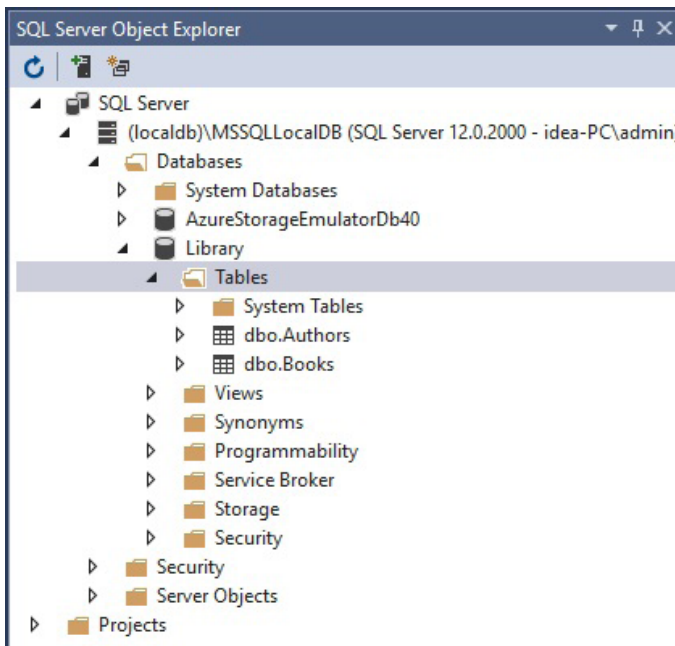


Fig. 9. The created tables

Database Connection

Now we start to consider the specific methods of applying the ADO.NET framework classes. Note that classes listed above, whose names begin with Db, are defined in the System.Data. Common namespace and they are abstract classes. Each of them is a base type for a number of the derived classes. Let's consider the hierarchy of these classes.

Base class	Derived classes
DbConnection	SqlConnection OleDbConnection OdbcConnection OracleConnection
DbCommand	SqlCommand OleDbCommand OdbcCommand OracleCommand
DbDataReader	SqlDataReader OleDbDataReader OdbcDataReader OracleDataReader
DbDataAdapter	SqlDataAdapter OleDbDataAdapter OdbcDataAdapter OracleDataAdapter

As you understand, each of the descendant classes is designed to operate with the relevant specific database. The existence of these hierarchies makes possible to create a polymorphic code that allows the application to operate with different databases

It should be noted that some time ago, Microsoft stopped to support the ADO.NET classes designed to operate with Oracle. We are going to talk about it in more detail in the next sections of our lesson.

The MS SQL Server database connection will be considered as an example. Herewith the sequence of the performed actions will be the same as when connecting to any of the available databases. Only the used classes will be changed.

We have already talked about the need to know some constant values to connect to the server. When operating with the databases, a set of these values is specified in so-called connection string. The connection strings combine the values in a special format; these values allow you to connect to a specific server.

Therefore, for a server connection, one should know its connection string. How to find out the connection string for each server? One can ask Google.

We are going to connect to the MS SQL Server, and the connection string plays an important role in such a connection. At first you should read about the connection string in this lesson. Then, if you have an initial understanding about this object, it will be useful for you to read about the connection string on the MSDN.

There is a mandatory set of values that one should know for a server connection. In addition to these values there are a number of others for the finer tuning. The mandatory values include:

- Server name;
- Database name;

- authentication method on the server;
- name and password to login to the server in the case of SQL Authentication.

Let's consider what these values should be for our case.

The server name is (localdb)\v11.0. It's a registered name of the LocalDB server. It should be specified accurate to each character, as it's specified here. The character register in the localdb value is unimportant. If we worked with the remote allocated server, its name would look like [\\10.3.0.10\MyServer](#) or simply as [\\10.3.0.20](#). If you don't know the server name, ask the system administrator.

The server name is specified by the Data Source attribute in the connection string:

```
Data Source=(localdb)\v11.0;
```

Note that the attribute value is not enclosed in quotes and semicolon is specified at the end of the value.

The database name is specified by the Initial Catalog attribute in the connection string:

```
Initial Catalog=Library;
```

Note that the attribute value is not enclosed in quotes and semicolon is specified at the end of the value.

The connection string appearance depends on the authentication method on the server. If the server is configured for the Windows Authentication, one shouldn't specify the user name and password for a server connection. If the Windows Authentication is disabled, one should specify the user name and password in the connection string for a server connection. .

The authentication method is specified by the Integrated Security attribute in the connection string. But the values of this attribute differ for the different servers. For the MS SQL Server the Windows Authentication can be specified in the connection string in two ways:

```
Integrated Security=true;
```

or

```
Integrated Security=SSPI;
```

Note again that the attribute value is not enclosed in quotes and semicolon is specified at the end of the value. We are going to consider the values of this attribute for other server later.

If there is no Integrated Security attribute in the connection string, it means that the server logon is performed in the mode of the SQL Server Authentication and in this case in the connection string one should specify the user name registered on the server and password of this user. These values are specified in the connection string with the User ID and Password attributes:

```
User ID=имя_пользователя; Password=пароль;
```

Note that the attribute values are not enclosed in quotes and semicolon is specified at the end of the value. Also pay attention that the user name and password are specified in the connection string in the open form. We are going to talk about the problems that arise due to this and about methods of their solving later.

So, we have listed the main values for the connection string and in our case the connection string should look like this:

```
Data Source=(localdb)\v11.0; Initial Catalog=Library;  
Integrated Security=SSPI;
```

If we had to login to our server with the "manager" name and password "123456", then our connecting string would look as follows:

```
Data Source=(localdb)\v11.0; Initial Catalog=Library;  
User ID=manager; Password=123456;
```

Let's proceed to the creation of our database connection. The connection is executed using the `DbConnection` class, more precisely – by any class, that is derived from `DbConnection`. We will use the `SqlConnection` class for our server connection. Because the MS SQL Server is our data provider.

There are several constructors for creating the objects of the `SqlConnection` class. But anyway you should know the connection string to your database. We know the connection string. Therefore we can create an object of the `SqlConnection` class by one of the following methods:

```
SqlConnection conn = null;  
conn = new SqlConnection();  
conn.ConnectionString = @"Data Source=(localdb)\v11.0;  
Initial Catalog=Library; Integrated Security=SSPI;";  
  
//or  
  
SqlConnection conn = null;  
conn = new SqlConnection(@"Data Source=(localdb)\v11.0;  
Initial Catalog=Library; Integrated Security=SSPI;");
```

As you see, the difference is in the initialization method of the `ConnectionString` property of the `SqlConnection` instance by the value of the connection string.

One should take into account that the `SqlConnection` object creation doesn't lead to the automatic execution of the database connection. In order to execute the connection, one should call the `conn.Open()` method, and in order to execute the server disconnection, one should call the `conn.Close()` method. Now we are ready to go on. Let's look how the queries can be transferred to a database and how they can be executed. For this purpose one should use the `DbCommand` class.

Query creation and execution (`DbCommand`)

You already understand that in our application we are going to use the `SqlCommand` class that is derived from the `DbCommand` class. In this class there are two properties that should be filled mandatory. These are `Connection` and `CommandText` properties. The `Connection` property has a `DbConnection` type and should be initialized by the object of this type. We have already such an object – it's our `conn`. The `CommandText` property has a `String` type and should be initialized by any SQL query. These two properties can be initialized when creating the `SqlCommand` object, using the constructor with two parameters, and one can create the `SqlCommand` object by the constructor without parameters, and then initialize these properties separately:

```
string insertString = @"insert into Authors
                        (FirstName, LastName)
                        values ('Roger', 'Zelazny')";
SqlCommand cmd = new SqlCommand(insertString, conn);
or
```

```
string insertString = @"insert into Authors
                        (FirstName, LastName)
                        values ('Roger', 'Zelazny')";
SqlCommand cmd = new SqlCommand();
cmd.Connection = conn;
cmd.CommandText = insertString;
```

Similarly like the SqlConnection object creation doesn't lead to the automatic execution of the database connection, the SqlCommand object creation doesn't lead to the immediate execution of the query, which is included to the CommandText property. For execution of the prepared query one should call one of the special methods. There are several such methods. They are split into synchronous and asynchronous.

Now we are going to use the synchronous query execution to a database. There are several methods for synchronous queries execution. Let's consider these methods that are designed for execution of queries of different types.

ExecuteScalar() is designed to execute the queries that return any value, but herewith this method returns only the first field of the first row of the result. That is, only one value. This method can be used, for example, to execute the queries with the aggregate functions.

ExecuteNonQuery() is designed to execute the insert, update and delete queries. This method returns the number of rows processed by the query into the table, although this returned value is often ignored.

ExecuteReader() is designed to execute the select queries. Returns the result of the query execution and places it in the object of the DbDataReader type.

Based on the foregoing, in order to execute our insert query we should use the `ExecuteNonQuery()` method. For example, like:

```
cmd.ExecuteNonQuery();
```

Let's create the `InsertQuery()` method and implement the insert query execution in it. Now the full code of our application can look as follows:

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

namespace AdoNetSample1
{
    class Program
    {
        SqlConnection conn=null;
        public Program()
        {
            conn = new SqlConnection();
            conn.ConnectionString = @"Data
Source=(localdb)\MSSQLLocalDB;
Initial Catalog=Library;
Integrated Security=SSPI;";
        }

        static void Main(string[] args)
        {
            Program pr = new Program();
            pr.InsertQuery();
        }
    }
}
```

```

public void InsertQuery()
{
    try
    {
        //open the connection
        conn.Open();
        //prepare the insert query
        //in the variable of the string type
        string insertString = @"insert into
                                Authors (FirstName, LastName)
                                values ('Roger', 'Zelazny')";
        //create the command object
        //by initializing both properties
        SqlCommand cmd =
            new SqlCommand (insertString, conn);

        //execute the query that
        //is included in the command object
        cmd.ExecuteNonQuery();
    }
    finally
    {
        // close the connection
        if (conn != null)
        {
            conn.Close();
        }
    }
}
}

```

Execute this application. Now let's make sure that the application was executed successfully and the information about the new author was added to the Authors table. In order to do this one should navigate to the SQL Server

Object Explorer window. Then one should expand the node of our server, database and the Tables node successively. Select the Authors table and activate the View Data option from the context menu (Fig. 10).

If everything is done correctly, you will see the entered information in the table. Enter some more entries into the Authors table in such a way in order to be able to look at something in the select query in the next part of our lesson.

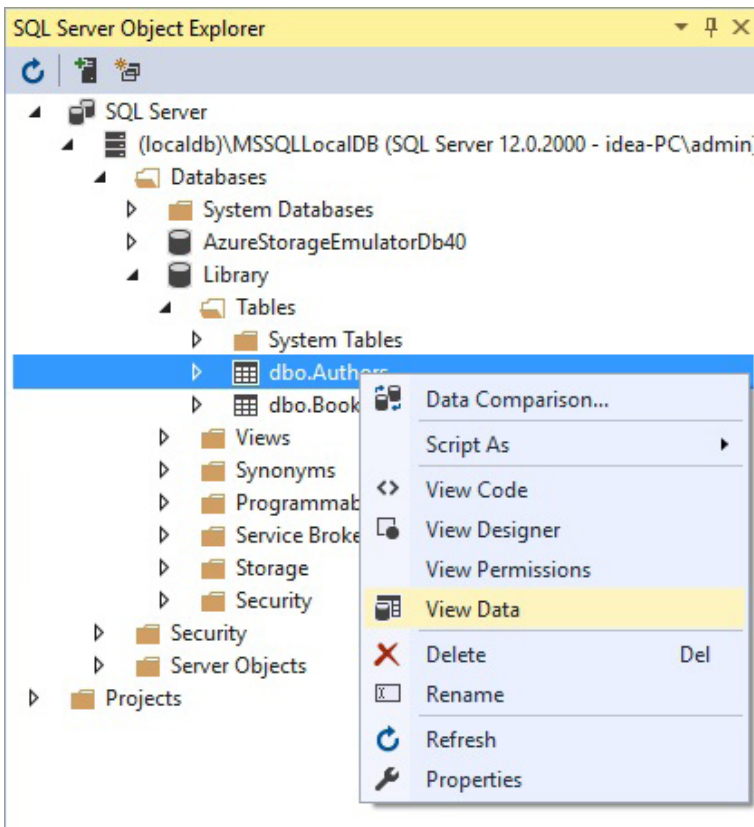


Fig. 10. View the table data

Let's sum up our acquaintance with the SqlCommand class. This class allows you to store the database queries, send these queries to the server and accept the execution results of these queries. The CommandText property is used to store the queries. To execute the queries the SqlCommand class contains three methods:

- *ExecuteReader() is designed to process the select queries and returns the execution results of these queries into the SqlDataReader object.*
- *ExecuteNonQuery() is designed to execute the insert, update and delete queries.*
- *ExecuteScalar() is designed to execute the queries that return only one aggregate value.*

Obtaining and processing the query results (DbDataReader)

Now, when our table is filled, we can proceed to the consideration of how to execute the select query. You certainly understand the specifics of this query that it returns the read strings as a result of its execution. We should learn to accept and process the execution result of the select query. For this purpose we are going to use the DbDataReader class. More precisely, SqlDataReader that is derived from it.

This time we are going to create a SqlCommand object, transferring the "select * from Authors" query to it. To execute this query let's call the ExecuteReader() method, accepting its returned value into the SqlDataReader object.


```
SqlDataReader rdr = null;  
SqlCommand cmd = new SqlCommand("select *  
                                from Authors", conn);  
rdr = cmd.ExecuteReader();
```

An object of the `SqlDataReader` class is a cursor that contains the strings obtained as a result of the select query execution. The operation with this object looks like this. Most often we don't know how many strings were returned by the select query that was executed. But we should know that there is an inner pointer in the `SqlDataReader` object. Initially it points to the first string in this object. The `Read()` method plays the main role in the processing of the obtained data. It extracts the string which the pointer points to and converts the `SqlDataReader` object to an array that contains the extracted string. After this, the `Read()` method moves the pointer to the next string. So, we can call the `Read()` method in the loop and at each iteration, the `SqlDataReader` object will be an array, whose elements are the fields of the current string of our table. In the case of our `Authors` table and using the select query, which returns all the fields of the table, this table will contain three elements – the first one will store the author's id, the second and the third one will store the name and last name. When the strings run out in the cursor, the `Read()` method will return false. Processing the information entered into an `SqlDataReader` object can be executed in such a loop:

```
while (rdr.Read())
{
    Console.WriteLine(rdr[1] + " " + rdr[2]);
}
```

In the SqlDataReader class there are a number of other methods that allow you to process the obtained data

more finely and according to the type. We are going to consider these methods in our lesson later. After processing the obtained data, the SqlDataReader object should be closed to release the resources occupied by it. The full code of the ReadData() method can look so:

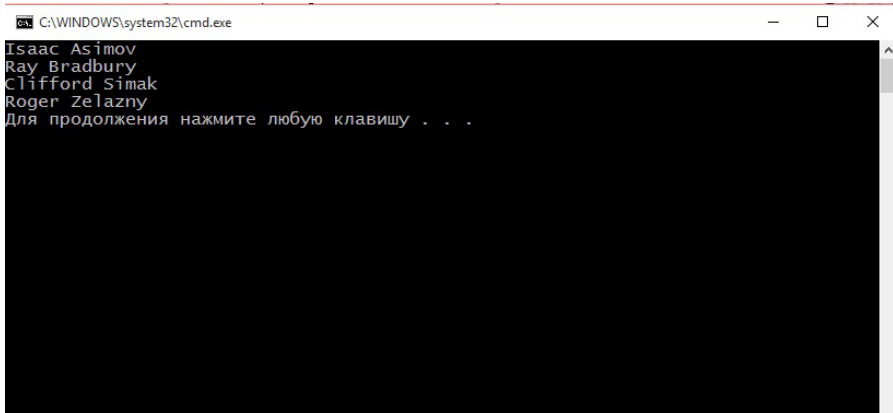
```
public void ReadData()
{
    SqlDataReader rdr = null;
    try
    {
        // open the connection
        conn.Open();
        //create a new command object with the select
        //query
        SqlCommand cmd = new SqlCommand("select *
                                         from Authors", conn);
        //execute the select query by saving the
        //returned result
        rdr = cmd.ExecuteReader();

        //extract the obtained strings
        while (rdr.Read())
        {
            Console.WriteLine(rdr[1] + " " + rdr[2]);
        }
    }
    finally
    {

```

```
//close reader
if (rdr != null)
{
    rdr.Close();
}
//close the connection
if (conn != null)
{
    conn.Close();
}
}
```

Execute our application by calling the `ReadData()` method. If everything is executed successfully, you will see the information from your Authors table in the console window. In my case it looked like this:



```
C:\WINDOWS\system32\cmd.exe
Isaac Asimov
Ray Bradbury
Clifford Simak
Roger Zelazny
Для продолжения нажмите любую клавишу . . .
```

Fig. 11. The select query execution

Let's look at the `SqlDataReader` object more attentively. We used only the `Read()` method in our example. You have already understood that this method reads every string that

was returned by the select query, and converts it to an array and enters it into a SqlDataReader object. At each iteration this array is overwritten by the data of the next string. One should remember that in the SqlDataReader object we have the information only about one string from the table at each moment of time. So we should process each read string in the current iteration.

Does SqlDataReader give us any information about an array that was formed out of the current string? Yes, it does. For example, from the FieldCount property we can find out the number of fields that were returned by the select query. If we call the GetName(index) method and transfer to it an index ranging from 0 to FieldCount-1, we will get the table field name that corresponds to the specified index. In other words, the SqlDataReader object stores the information about the table field names that were read by the select query, and this information is available for us.

In our example we referred to the object as to an indexed array. But it's not the only possibility. If you know the field names of the read table, you can extract the data by the names of these fields. In other words, in the string

```
Console.WriteLine(rdr[1] + " " + rdr[2]);
```

it could be written as:

```
Console.WriteLine(rdr["FirstName"] + " "  
+ rdr["LastName"]);
```

Let's change the code of our ReadData() method taking into account new information and make this method output the names of the read fields of the table.

```
public void ReadData ()
{
    SqlDataReader rdr = null;
    try
    {
        //open the connection
        conn.Open();

        //create a new command object with the select
        //query
        SqlCommand cmd = new SqlCommand("select *
                                         from Authors", conn);

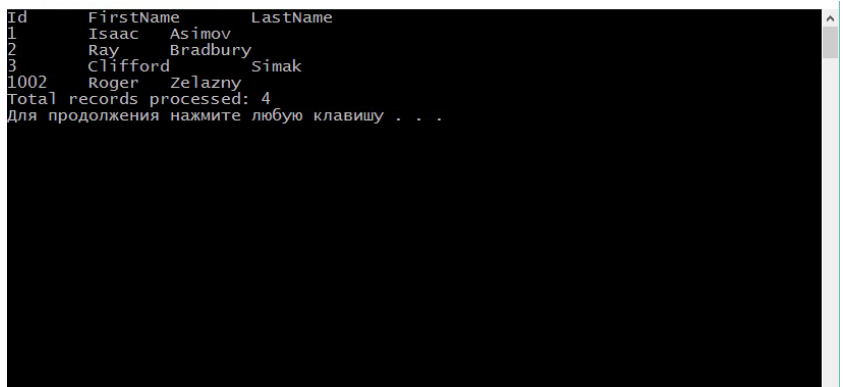
        //execute the select query by saving the
        //returned result
        rdr = cmd.ExecuteReader();
        int line = 0; //string counter
        //extract the obtained strings
        while (rdr.Read())
        {
            //form a table header before outputting
            //the first string
            if (line == 0)
            {
                //a loop through the number of the
                //read fields
                for (int i = 0; i < reader.FieldCount; i++)
                {
                    //output the field names into the
                    //console window
                    Console.Write(rdr.GetName(i) .
                                ToString()+" ");
                }
            }
        }
    }
}
```

```

        }
        Console.WriteLine();
        line++;
        Console.WriteLine(rdr[1] + " " + rdr[2]);
    }
    Console.WriteLine("Handled records: " +
        line.ToString());
}

finally
{
    //close reader
    if (rdr != null)
    {
        rdr.Close();
    }
    //close the connection
    if (conn != null)
    {
        conn.Close();
    }
}
}

```



```

Id      FirstName  LastName
1       Isaac     Asimov
2       Ray       Bradbury
3       Clifford  Simak
1002    Roger     Zelazny
Total records processed: 4
Для продолжения нажмите любую клавишу . . .

```

Fig. 12. ReadData() modified method execution

Let's sum up our acquaintance with the `SqlDataReader` class. This class allows you to store the execution results of the select queries and extract them one after the other successively in the direction from the first to the last one. We can't return to the extracted strings. When the `SqlDataReader` object processing is finished, it should be closed by calling the `Close()` method.

Batch processing of queries

Let's talk about what will happen if not only one but several queries are inserted into a `CommandText` property of the `SqlCommand` object.

It turns out, that `SqlDataReader` can easily execute several queries and return their results. The only condition, that should be fulfilled is to separate the query texts being entered into the `CommandText` property with a semicolon.

If you looked at the properties and methods of the `SqlDataReader` object in a studio attentively, you could notice the `NextResult()` method there. This method is designed to process the results of the set of queries that are entered into the `CommandText` property of the `SqlCommand` object. Let's look at our code in which the select query result is processed. The strings are processed in a while loop until the `Read()` method returns false. Only the results of the first query are processed in such a way. If there are several queries, our code simply will not know about this. In order to be able to process the results of several queries, the while loop should be inserted into another do-while loop, that is controlled by the `NextResult()` method:

```

int line = 0; // string counter
// extract the obtained strings
do {
    while (rdr.Read())
    {
        if (line == 0) // form a table header before
                        // outputting the first string
        {
            // a loop through a number of the read
            // fields
            for (int i = 0; i < rdr.FieldCount; i++)
            {
                //output the field names to the console
                //window
                Console.Write(rdr.GetName(i).
                            ToString()+" ");
            }
        }

        Console.WriteLine();
        line++;
        Console.WriteLine(rdr[1] +" "+ rdr[2]);
    }
    Console.WriteLine("Handled records:
                        " + line.ToString());
} while (rdr.NextResult());

```

Now you can add one more query or several queries to our select query and don't forget to separate them using a semicolon and run the application. In the console window you will see the results returned by all the queries.


```

Id      FirstName  LastName
1       Isaac     Asimov
2       Ray       Bradbury
3       Clifford  Simak
1002    Roger      Zelazny
Total records processed: 4
2       I       I Robot
Total records processed: 5
Для продолжения нажмите любую клавишу . . .

```

Fig. 13. Example of the package query execution

However, this approach has one limitation. If all the select queries entered into a package return the same number of fields, there will be no problems. But if the number of the returned fields differs, at first one should execute a query that returns a maximum number of fields. And then somehow one should solve the problems related to referring to the specific elements of the obtained array. We are going to return to this issue in the second lesson in our next application.

Below there is a full code of the method that executes several select queries simultaneously.

```

public void ReadData2()
{
    SqlDataReader rdr = null;
    try
    {
        //Open the connection
        conn.Open();
        SqlCommand cmd = new SqlCommand("select *
                                         from Authors; select *
                                         from Books", conn);
    }
}

```

```

rdr = cmd.ExecuteReader();
int line = 0;
// extract the obtained strings
do
{
    while (rdr.Read())
    {
        if (line == 0) //form a table header
            // before outputting the first string
        {
            // a loop through a number of the
            // read fields
            for (int i = 0; i < rdr.
                FieldCount; i++)
            {
                //output the field names to the
                // console window
                Console.Write(rdr.GetName(i).
                    ToString() + "\t");
            }
            Console.WriteLine();
        }
        line++;
        Console.WriteLine(rdr[0] + "\t" +
            rdr[1] + "\t" + rdr[2]);
    }
    Console.WriteLine("Total records
        processed: " + line.ToString());
} while (rdr.NextResult());
}
finally
{
    //close the reader
    if (rdr != null)
    {
        rdr.Close();
    }
    // Close the connection
    if (conn != null)
    {
        conn.Close();
    }
}
}

```

Configuration file

In our example there are several deficiencies that should be corrected. One of them is storage of data for server access in the code. You understand that we are talking about the connection string that is stored in the string variable. This is unacceptable both in terms of security and in terms of code changes and rebuilding our application when changing the server address, database name or any other value from the connection string.

How can we solve this problem? For example, one can offer the user to enter the data for connect in a dialog box. It's a very good solution in many cases. However, a standard way of solving this problem is to store the connection string in the application configuration file. The application configuration file is xml file that is added to the application by Visual Studio 2015 automatically. We have a set of classes that allow you to operate with the application configuration file. Let's get acquainted with how to store the connection string in this file and how to extract the necessary information from it.

By default the application configuration file looks as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0"
      sku=".NETFramework,Version=v4.5.2" />
  </startup>
</configuration>
```

The database connection string in this file should be placed in the connectionStrings element inside the configuration

element. Change the application configuration file so that it will look like as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0"
      sku=".NETFramework,Version=v4.5.2" />
  </startup>
  <connectionStrings>
    <add name="MyConnString"
      connectionString="Data Source=(localdb)\v11.0;
      Initial Catalog=Library;
      Integrated Security=SSPI;" />
  </connectionStrings>
</configuration>
```

To access the application configuration file from our code we will use the `ConfigurationManager` class that is defined in the `System.Configuration` namespace. By default this namespace is not added to the project. Therefore one should execute the `Add Link` command. Using this class we can extract the connection string into the `string` variable in such a way:

```
string connectionString = ConfigurationManager.
  ConnectionStrings["MyConnString"].ConnectionString;
```

Pay attention that as an index for a `ConnectionStrings` property we specify the add element name, which our connection string is added to. Since one application can operate with several databases simultaneously, there can be several connection strings in one project and each of these strings should have a unique name in the `name` attribute.

Let's change the constructor of our class, in which the `SqlConnection` object was initialized and the connection string was entered into its `ConnectionString` property:

```
public Program()
{
    conn = new SqlConnection();
    conn.ConnectionString =
        ConfigurationManager.
            ConnectionStrings["MyConnString"].
            ConnectionString;
}
```

Rebuild an application and run it. You will see the result of the `ReadData()` method operation in the console window. Thus, our application operates but herewith the connection string is already outside of the application code in the configuration file.

Parameterized queries in `DbCommand`

Let's return to the `DbCommand` class and get acquainted with its other very useful opportunities. Look at the strings of our `InsertQuery()` method wherein we execute the insert query.

```
//prepare an insert query in the variable of the
//string type
string insertString = @"insert into Authors
    (FirstName, LastName) values ('Roger', 'Zelazny')";

//create a command object by initializing both
//properties
SqlCommand cmd = new SqlCommand(insertString, conn);

//execute the query entered into the command object
cmd.ExecuteNonQuery();
```

You understand that the data transfer for recording into a table in the insert query in the form of literals is a bad solution. One should implement some mechanism that allows the user to transfer the data for inputting to a database dynamically.

Please, never do something like this:

```
string userName = // read from console
                  // or from TextBox;
string sql = @"select * from users where name =
              '"+ userName +"'";
```

Such a style of programming is a huge hole in the security of your computer. It's impossible to check the data which will be entered by the user into a console or into a TextBox element for recording to a database in such a way. What happens if the user enters into TextBox the following string instead of the desired name for userName:

```
"user5'; drop table users;'"?
```

Substitute this string in the specified select query by concatenating. You will obtain such a query:

```
select * from users where name = 'user5';
drop table users;
```

As you understand, this query, more precisely these two queries will delete the users table from your database. And one can come up with many such tricky queries.

One should use another mechanism for data transferring to a query in order to avoid such troubles. And there is such a mechanism. This mechanism is provided to us by the DbCommand class. There is a property with the Parameters

name in this class. According to the property name in the plural we can guess that this property is a collection. The elements of this collection are objects of the DbParameter type, more precisely, – of the type that is derived from it, for example, SqlParameter.

The DbParameter class allows us to create the parameterized queries to a database by ensuring the security of such queries. Let's rewrite our select query in the form of the parameterized query.

```
//prepare a select query in the variable of the
//string type
string sql = @"select * from users where name = @p1";
```

In this code @p1 is a parameter of our query. The "@" character is a sign of the parameter. The parameter names and their number are arbitrary. Now we should create an object of the SqlParameter type for every parameter that is specified in the query and enter this object into a Parameters property of the SqlCommand object. There are several methods for creating a SqlParameter object. Let's consider these method.

```
SqlParameter param1 = new SqlParameter();
param1.ParameterName = "@p1"; //comparing with the
                               //parameter in the query
param.SqlDbType = System.Data.SqlDbType.NVarChar;
                               //parameter type
param1.Value = firstName; //parameter value
```

A param1 object that was created in such a way can be added to the Parameters collection of the SqlCommand object.

```
cmd.Parameters.Add(param1);
```

Similarly one should create an object and enter it into a Parameters collection for each parameter that is specified in the query. Pay attention to a large variety of data types in the SqlDbType enumeration, it's necessary for the most exact match between the .NET Framework data types and the MS SQL Server data types. Some other properties of the SqlParameter object will be considered in the next section of our lesson, where we are going to consider the work with the stored procedures.

The above example of defining the parameters for queries can seem cumbersome. It's actually so. If somebody prefers the shorter encoding forms, in this case I can make you happy. The fact is that the Add() method that is defined in the Parameters collection has one interesting feature. It returns the element that was added to the collection. If one use this feature of the Add() method and its overloaded options, one will be able to write a shorter code option:

```
cmd.Parameters.Add("@p1", SqlDbType.NVarChar).  
Value = firstName;
```

This one string of the code replaces four expanded strings that we have considered before. In this string a new SqlParameter object is created "on the uptake" thanks to using the overloaded Add() method. When creating, the ParameterName and DbType properties are initialized in this object. And then the Value property is initialized. Thus, all the necessary properties are initialized in one string.

There are more other Add() method types, for example, AddWithValue() that initializes the Value property immediately:


```
cmd.Parameters.AddWithValue("@p1", firstName);
```

Do you remember that we have talked about the dynamic query creation by concatenating the string with parameters – is it a bad idea? So, using SqlParameter, an automatic check for a harmful content presence is executed in the parameter values. If any hacker decides to transfer a harmful script to your application, this script will not be activated. And this hacker will be disappointed. What will happen with the above "tricky" query, if the option is transferred by SqlParameter for it? It will turn into such a query:

```
select * from users where name = 'user5';  
drop table users;
```

Everything that is placed after the equal sign "=" in this query is interpreted as a string constant, and this query will not cause any harm to your database.

Stored procedures in DbCommand

We can use not only queries but stored procedures in our applications. Let's consider the use of the stored procedures in the ADO.NET technology. First of all we should create a stored procedure. It's desirable with parameters. And even better – with the input and output parameters! Don't forget that there are the output parameters. Let's create a stored procedure with two parameters. The procedure will take the author's id in the first parameter, and will return the number of books of this author in the Books table through the second parameter. To create a stored procedure in our database, navigate to the SQL Server Object Explorer window, expand the node of our database, next, expand

the Programmability node, next, select the Stored Procedures node. Activate the Add New Stored Procedure option from the context menu.

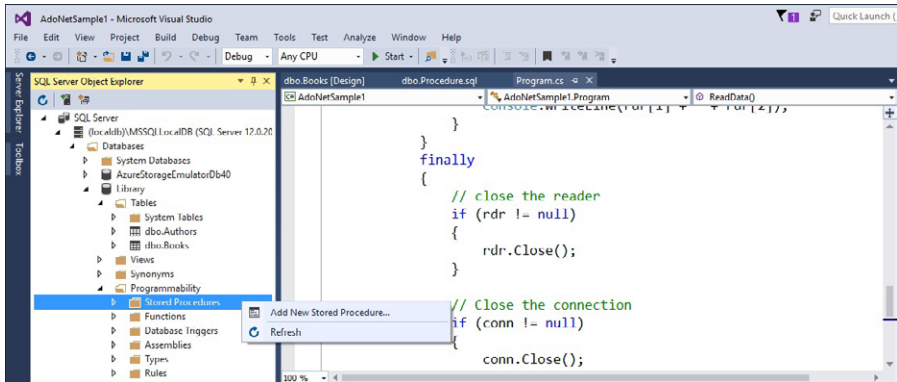


Fig. 14. Adding a new stored procedure

In the right window, a text snippet of the stored procedure will appear. Delete it and insert the code of our stored procedure instead of it.

```
CREATE PROCEDURE getBooksNumber
@AuthorId int,
@BookCount int OUTPUT
AS
BEGIN
    SET NOCOUNT ON;
    SELECT @BookCount = count(b.id)
    FROM Books b, Authors a
    WHERE b.Authorid = a.id AND
    a.id = @ AuthorId;
END;
```

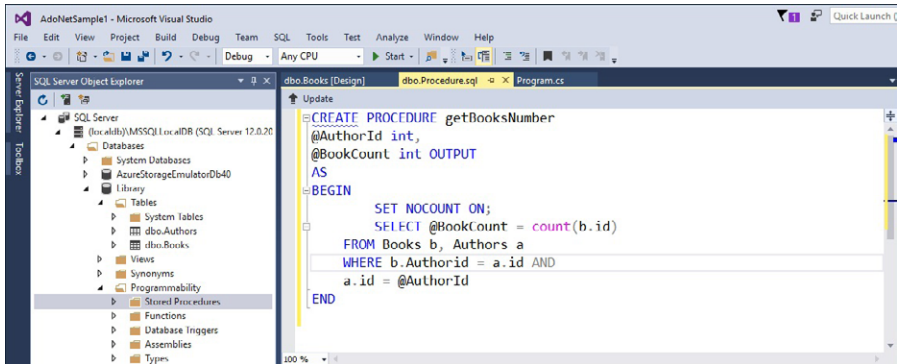


Fig. 15. Creating a new stored procedure

We create a stored procedure with the `getBooksNumber` name. When calling, we will transfer the author's id to the procedure through the first `@AuthorId` parameter of the `int` type. The second `@BookCount` parameter of the `int` type is described as output. When calling the procedure, we will substitute the uninitialized parameter in it. After calling and executing the procedure, we will be able to read the entered value from the second parameter.

After inserting the code of the stored procedure that is created, click on the Update button on the top toolbar. In the appeared window, click the Update Database button. Now a new stored procedure is added to our database. Let's look how we can call the stored procedure from the C# code.

If earlier we have entered the query text into the `CommandText` property of the `SqlCommand` object, then in the case when we want to execute the stored procedure, one should enter only the stored procedure name into the `CommandText` property:

```
SqlCommand cmd = new SqlCommand("getBooksNumber", conn);
```

However, in this case we have to use another property that has not been used before. We should specify that namely the stored procedure is called. To do this, one should initialize the CommandType property of the SqlCommand object. By default this property contains the reference to the query, and we should enter the StoredProcedure value into it:

```
cmd.CommandType = CommandType.StoredProcedure;
```

Since our stored procedure has parameters, they should be described correctly and transferred to the SqlCommand object. As for the first parameter, everything is done traditionally. Specify the name, type, value and enter into the Parameters collection:

```
cmd.Parameters.Add("@AuthorId", System.Data.  
    SqlDbType.Int).Value = 1;
```

As for the second output parameter, the situation is different. It should be created in a separate variable of the SqlParameter type. Specify a name and type for it. The value (the value property) shouldn't be specified. But for this parameter one should fill the Direction property wherein one should specify that this is an output parameter. Then, the created parameter should be entered into the Parameters collection.

```
SqlParameter outputParam = new SqlParameter("@  
    BookCount", System.Data.SqlDbType.Int);  
outputParam.Direction = ParameterDirection.Output;  
//outputParam.Value = 0; //value shouldn't be filled!  
cmd.Parameters.Add(outputParam);
```

If we combine all this together, we will get a code of the `ExecStoredProcedure()` method, that is designed for calling our stored procedure. Pay attention how the value is extracted from the output parameter in this code after the procedure execution.

```
public void ExecStoredProcedure()
{
    conn.Open();
    SqlCommand cmd = new SqlCommand("getBooksNumber",
        conn);
    cmd.CommandType = CommandType.StoredProcedure;

    cmd.Parameters.Add("@AuthorId", System.Data.
        SqlDbType.Int).Value = 1;

    SqlParameter outputParam = new SqlParameter("@
        BookCount", System.Data.SqlDbType.Int);
    outputParam.Direction = ParameterDirection.
        Output;
    //outputParam.Value = 0;
    //value shouldn't be filled!
    cmd.Parameters.Add(outputParam);

    cmd.ExecuteNonQuery();
    Console.WriteLine(cmd.Parameters["@BookCount"].
        Value.ToString());
}
```

Let's sum up our acquaintance with ADO.NET. You have created the application in which you have considered the use of the main classes which are included to this framework:

- *DbConnection* – for a database connection;
- *DbCommand* – for executing the database queries and stored procedures, and for getting the results from the database
- *DbDataReader* – for extracting the results of the select queries execution;
- *DbParameter* – for creating the parameterized references to a database;

In addition to this, you have got acquainted with the use of the configuration file. Everything that we have done in this lesson was based on using the connected mode of operation with the server.

In the next lesson we are going to consider other useful classes and proceed to study the disconnected mode of operation with the server.

Homework

For our database one should write a console application that should calculate the sum of prices of all the books and sum of pages of all the books in the Books table. But at first one should find out the number of books in this table, in order to do this, we offer to execute such a query:

```
Select count(id) from Books;
```

Think about what method of the DbCommand class (ExecuteNonQuery(), ExecuteScalar() or ExecuteReader()) can be used for the most convenient execution of this query. Save the value returned by this query into any int num variable.

Then execute the query:

```
Select * from Books;
```

To break the monotony, don't repeat the code written by us in this lesson, and extract the results of this query in the for loop, using the num variable that is initialized by the first query as a loop limiter. At each iteration of this for loop, extract the values from the Price and Pages fields as the int values (look what methods the DbDataReader class has for this purpose) and sum up them. The values of other fields should be extracted according to their type in a database. Once again look attentively what methods in the DbDataReader class allow you to get the read value from the table and convert it to the string, or int, or double type. Output to the console

all the field values at each iteration. After finishing the loop, output the total price values for all the books and number of pages for all the books.

The purpose of home work:

- Repeat all the actions that were considered in the lesson;
- Learn to extract the data of the different types from the table fields;
- Use not only the `ExecuteQuery()` method of the `DbCommand` class.