STEP IT ACADEMY

Database Access Technology

ADO.NET

Lesson № 3

Provider factory, review of acynchronous operations

Contents

DbProviderFactory data providers

General principles

You have already got some idea of how one can work with data sources in the ADO.NET. So now let's remember of that we have class hierarchies to access different data sources. These classes are derived from abstract classes. For example, the OdbcConnection, OleDbConnection, OracleConnection and SqlConnection classes are derived from the abstract DbConnection class. In our applications, we have used the derived SqlXXX classes, because we used the MS SQL Server. If we had to work with Access, we would use the OleDbXXX classes, if we wanted to work with Oracle, we would use the OracleXXX classes. Which begs the question, can one write a code that will be the same for different data sources? It is needed in order to avoid rewriting and recompiling of our application, when we have to change the database server. And to avoid a business trip to your client, who uses your application, in order to change the data provider.

So, our objective is to write a code that is invariant with respect to the data source. It's clear that this code should be parameterized. That is, the code should somehow find out, which data source it should work with, and it should continue to execute uniformly all the actions that the application needs. To write this code, we should get acquainted with the DbProviderFactory class.

Each data provider contains a factory class, derived from the DbProviderFactory class. If you get an access to this class factory for a specific data provider, by using it you will be able to create the Connection, Command, Adapter, and other objects for this data provider.

How can the application find out, what data providers are available and how to get the factory? The Windows operating system will help the application here. Each computer running Windows contains a file named machine.config. All the available data providers are registered in this file. You will not be surprised, when you find out that machine.config is an XML file. Consequently, it will be very simple to read this file and extract the necessary information. On my computer, this file has the following path: C:\Windows\Microsoft.NET\Framework\v2.0.50727\CONFIG. The section of this file, which contains the necessary information looks as follows:

```
<system.data>
    <DbProviderFactories>
        <add name="Odbc Data Provider" invariant="System.
Data.Odbc" description=".Net Framework Data Provider
for Odbc" type="System.Data.Odbc.OdbcFactory,
System.Data, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"/>
        <add name="OleDb Data Provider"
invariant="System.Data.OleDb" description=".Net
Framework Data Provider for OleDb" type="System.Data.
OleDb.OleDbFactory, System.Data, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089"/>
        <add name="OracleClient Data Provider"
invariant="System.Data.OracleClient" description=".
Net Framework Data Provider for Oracle" type="System.
Data.OracleClient.OracleClientFactory, System.Data.
OracleClient, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"/>
```

```
        <add name="SqlClient Data Provider"
invariant="System.Data.SqlClient" description=".
Net Framework Data Provider for SqlServer"
type="System.Data.SqlClient.SqlClientFactory,
System.Data, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"/>
        <add name="Microsoft SQL Server Compact Data
Provider 4.0" invariant="System.Data.SqlServerCe.4.0"
description=".NET Framework Data Provider for
Microsoft SQL Server Compact" type="System.Data.
SqlServerCe.SqlCeProviderFactory, System.Data.
SqlServerCe, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=89845dcd8080cc91"/>
    </DbProviderFactories>
</system.data>
```

Example use

Let's create an application that implements the following logic:

- reads a list of available data providers from machine.config and outputs it to the user (in Combobox and DataGridView);
- allows a user to choose the required provider from the list (Combobox) and reads the connection string for this provider (from AppConfig);
- allows a user to input the SQL query for the database (corresponding to the read connection string);
- executes the query and outputs the results (in DataGridView).

In order to show that the application written by us can work with different data providers, let's create one more table in the MS Access. Our application will work with our Library database and with the table in MS Access. If you have other available databases, use them. Let the table in MS Access be named Books as well, although its name can be arbitrary. The table structure and table data will differ from the structure and data of the Library database. Generally speaking, it can be an arbitrary table, its structure and data don't really matter. Before developing the application, let's consider briefly new classes, which we will have to work with.

First of all, it's the DbProviderFactories class. Using its static GetFactoryClasses() method, we can get a list of available data sources. The GetFactoryClasses() method reads the information from the machine.config file and returns the result in the form of the DataTable object.

Using another static GetFactory() method of this class, we will be able to get a factory for a specific data provider. As a parameter of the GetFactory() method, we should specify a provider, whose factory we want to get. For example, in order to get a factory for the MS SQL Server, one should pass the "System.Data.SqlClient" string to this method. You will see later, where these strings for the GetFactory() method come from.

The specific data provider factory is an object of the DbProviderFactory type. Having such a factory, we can use the CreateConnection(), CreateCommand(), and CreateDataAdapter() factory methods and get the DbConnection, DbCommand, and DbDataAdapter objects for the specific data provider. Everything else you already know how to do.

As we have already agreed on, in order to have a data source different from System.Data.SqlClient, let's create a table in MS Access 2007. The table can be arbitrary, we should simply make sure that we will get access to it. We are going to connect to this table through System.Data.OleDb data provider. In my case, the created database is named Library.accdb and for simplicity, this file is located in the root folder of the created application.

- Run Visual Studio 2015 and create a new Windows Forms project with the following window:
- two buttons: "Get All Providers" and "Execute request";
- Combobox, to select a provider;
- Two text fields, to display a connection string (Readonly) and to input queries;
- DataGridView, to display query results.

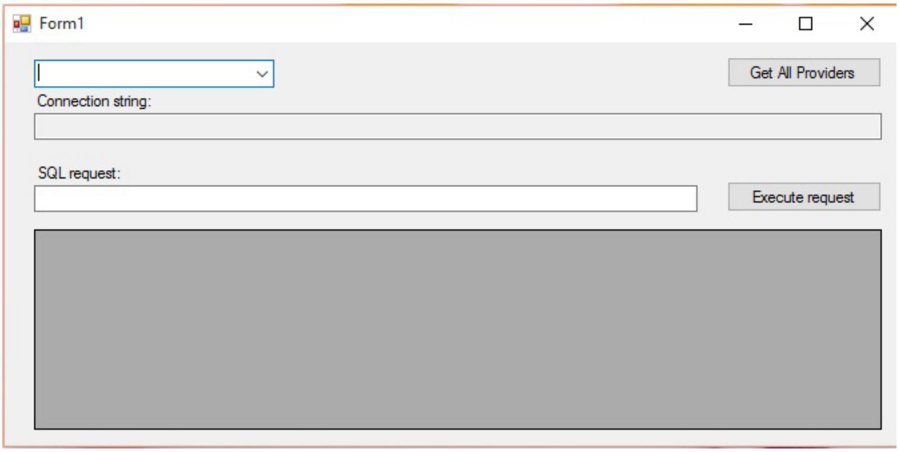


Fig. 1. Main application window

In the configuration file of the created application, add connection strings to the databases, which you plan to work with. In my case, App.config looks as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <clear/>
    <add name="MyLibrary"
     providerName="System.Data.SqlClient"
     connectionString=
     "Data Source=(localdb)\MSSQLLocalDB;Initial
Catalog=Library; Integrated Security=SSPI;"
    />
    <add name="MyAccess"
     providerName="System.Data.OleDb"
     connectionString=
     "Provider=Microsoft.ACE.OLEDB.12.0;Data
Source=Library.accdb;Persist Security Info=False;"
    />
    </connectionStrings>
</configuration>
```

Add the System.Configuration namespace reference and the using System.Configuration directive to the project.

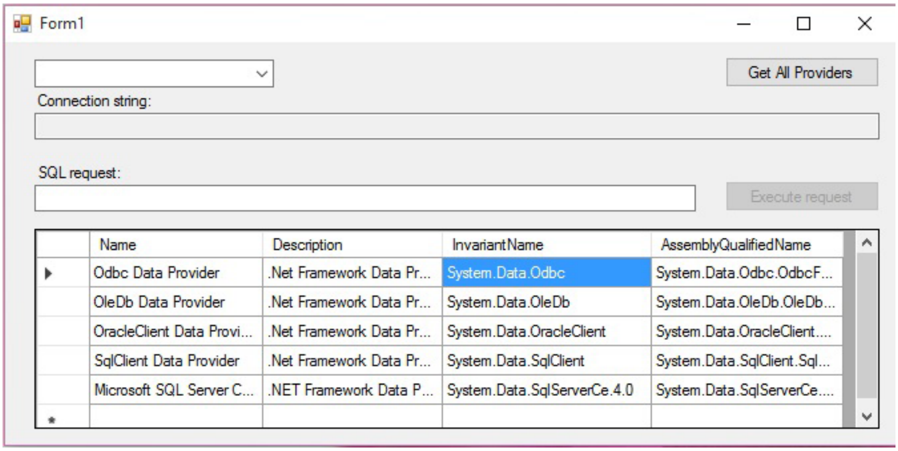
Bring the application code to the following form:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Data.Common;
using System.Configuration;

namespace ADO_ProviderFactory
{
   public partial class Form1 : Form
   {
      DbConnection conn=null;
      DbProviderFactory fact=null;
      string providerName="";

      public Form1()
      {
         InitializeComponent();
         button2.Enabled = false;
      }
      /// <summary>
      /// Read the list of the registered
      /// data providers
      /// return the list in the form of a table
      /// display the list in dataGridView1
      /// enter values of the InvariantName
      /// column of the created table
      /// into comboBox1
      /// </summary>
      /// <param name="sender"></param>
     /// <param name="e"></param>
     private void button1_Click(object sender, EventArgs e)
     {
        DataTable t = DbProviderFactories.GetFactoryClasses();
        dataGridView1.DataSource = t;
        comboBox1.Items.Clear();

     foreach (DataRow dr in t.Rows)
     {
        comboBox1.Items.Add(dr["InvariantName"]);
     }
  }
  /// <summary>
  /// according to value selected in comboBox1c,
  /// using the GetFactory() method
  /// create a factory for the selected provier
  /// using the GetConnectionStringByProvider() method
  /// get the database connection string from App.config
  /// display the connection string in textBox1
  /// and save it in the providerName global string
```

```
 /// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void comboBox1_SelectedIndexChanged(object
                                   sender, EventArgs e)
{
   fact = DbProviderFactories.GetFactory(comboBox1.
        SelectedItem.ToString());
   conn = fact.CreateConnection();
   providerName =
        GetConnectionStringByProvider(comboBox1.
                                 SelectedItem.ToString());
   textBox1.Text = providerName;
}
/// <summary>
/// from App.config, read the connection string
/// with the providerName value,
/// which is that of the providerName parameter
/// return the found connection string or null
/// </summary>
/// <param name="providerName"></param>
/// <returns></returns>
static string GetConnectionStringByProvider(string
   providerName)
{
   string returnValue = null;

   // read all the connection strings of App.config
   ConnectionStringSettingsCollection settings =
      ConfigurationManager.ConnectionStrings;

   // find and return the connection string
   // for providerName
   if (settings != null)
   {
      foreach (ConnectionStringSettings cs in settings)
      {
         if (cs.ProviderName == providerName)
         {
            returnValue = cs.ConnectionString;
```

```
                    break;
                }
            }
        }
            return returnValue;
        }

        /// <summary>
        /// having a factory for the selected provider
        /// execute the standard actions with the database
        /// to demonstrate the code efficiency
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void button2_Click(object sender, EventArgs e)
        {
            conn.ConnectionString = textBox1.Text;
            // create an adapter from the factory
            DbDataAdapter adapter = fact.CreateDataAdapter();
            adapter.SelectCommand = conn.CreateCommand();
             adapter.SelectCommand.CommandText = textBox2.
                   Text.ToString();
             // execute the select query from the adapter
             DataTable table = new DataTable();
             adapter.Fill(table);
             // output query results
             dataGridView1.DataSource = null;
             dataGridView1.DataSource = table;
        }
        private void textBox2_TextChanged(object sender, EventArgs e)
        {
            if (textBox2.Text.Length > 5)
                button2.Enabled = true;
            else
                button2.Enabled = false;
        }
    }
}
```

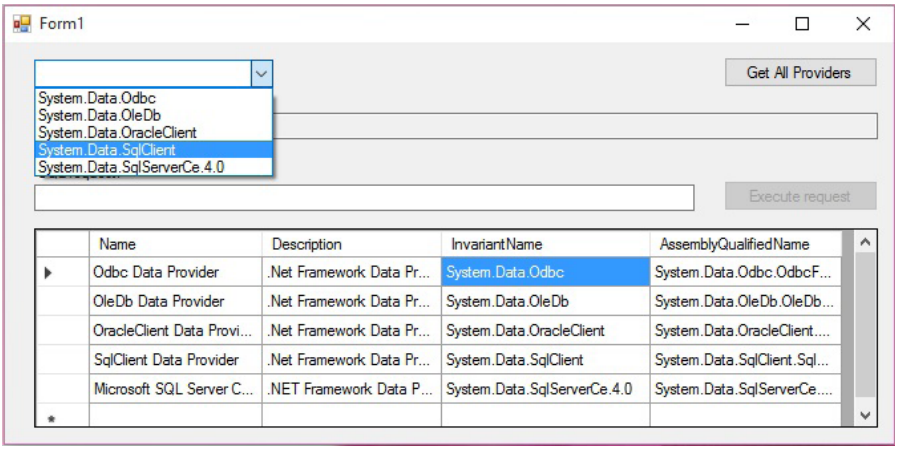
Let's look at how our application works and consider some points in more detail. Run the application and click the Get All Providers button. In the bottom part of the window, you will see a table, which displays the information about the registered data providers from the machine.config file. Pay more attention to the InvariantName column. It contains a unique name for every provider. We will select a specific factory according to this name. Values of this column are always arranged in the connection string of the App.config file in the providerName attribute. Our application enters the InvariantName values for the registered data providers into the combobox.



	Name	Description	InvariantName	AssemblyQualifiedName
▶	Odbc Data Provider	.Net Framework Data Pr...	System.Data.Odbc	System.Data.Odbc.OdbcF...
	OleDb Data Provider	.Net Framework Data Pr...	System.Data.OleDb	System.Data.OleDb.OleDb...
	OracleClient Data Provi...	.Net Framework Data Pr...	System.Data.OracleClient	System.Data.OracleClient....
	SqlClient Data Provider	.Net Framework Data Pr...	System.Data.SqlClient	System.Data.SqlClient.Sql...
	Microsoft SQL Server C...	.NET Framework Data P...	System.Data.SqlServerCe.4.0	System.Data.SqlServerCe....
*				

Fig. 2. Data providers list

Select the required data provider name in the combobox, in our case it's System.Data.SqlClient or System.Data.OleDb. Because we didn't provide connection strings for other providers.

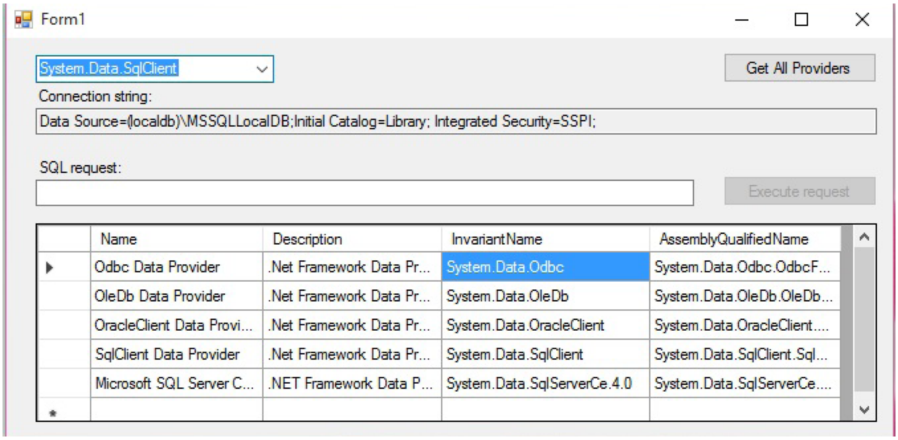


System.Data.Odbc
System.Data.OleDb
System.Data.OracleClient
System.Data.SqlClient
System.Data.SqlServerCe.4.0

	Name	Description	InvariantName	AssemblyQualifiedName
▶	Odbc Data Provider	.Net Framework Data Pr...	System.Data.Odbc	System.Data.Odbc.OdbcF...
	OleDb Data Provider	.Net Framework Data Pr...	System.Data.OleDb	System.Data.OleDb.OleDb...
	OracleClient Data Provi...	.Net Framework Data Pr...	System.Data.OracleClient	System.Data.OracleClient....
	SqlClient Data Provider	.Net Framework Data Pr...	System.Data.SqlClient	System.Data.SqlClient.Sql...
	Microsoft SQL Server C...	.NET Framework Data P...	System.Data.SqlServerCe.4.0	System.Data.SqlServerCe....
*				

Fig. 3. Data provider selection

After selecting a data provider, you will see a connection string in a text field. Note that the query execution button is inactive until you enter the query into the second text field.



System.Data.SqlClient

Connection string:

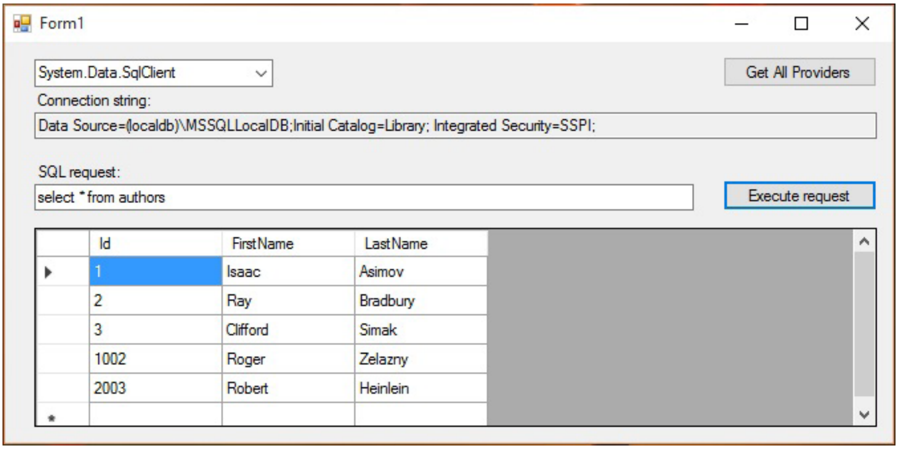
Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=Library; Integrated Security=SSPI;

SQL request:

	Name	Description	InvariantName	AssemblyQualifiedName
▶	Odbc Data Provider	.Net Framework Data Pr...	System.Data.Odbc	System.Data.Odbc.OdbcF...
	OleDb Data Provider	.Net Framework Data Pr...	System.Data.OleDb	System.Data.OleDb.OleDb...
	OracleClient Data Provi...	.Net Framework Data Pr...	System.Data.OracleClient	System.Data.OracleClient....
	SqlClient Data Provider	.Net Framework Data Pr...	System.Data.SqlClient	System.Data.SqlClient.Sql...
	Microsoft SQL Server C...	.NET Framework Data P...	System.Data.SqlServerCe.4.0	System.Data.SqlServerCe....
*				

Fig. 4. Connection string display

Enter any query to your database into the query text field and click the Execute request button. The query will be executed, and its results will be output in dataGridView1. So our code works at least with the System.Data.SqlClient data provider.



System.Data.SqlClient

Connection string:

Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=Library; Integrated Security=SSPI;

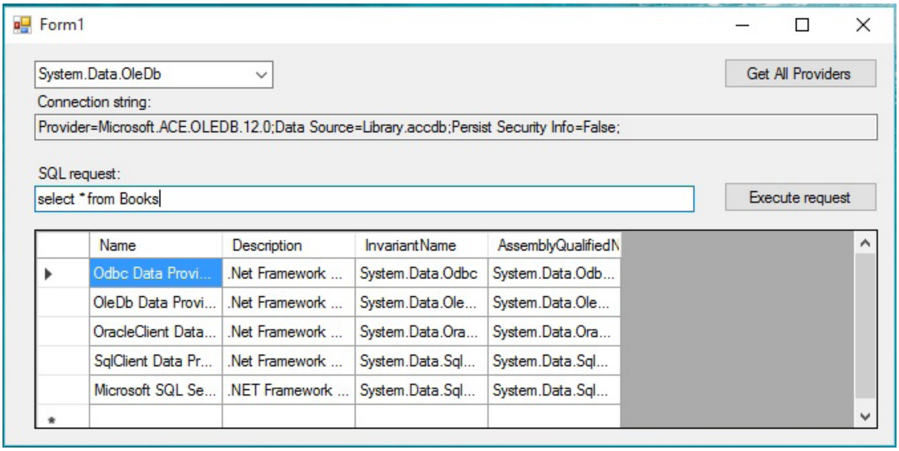
SQL request:

select * from authors

Id	FirstName	LastName
1	Isaac	Asimov
2	Ray	Bradbury
3	Clifford	Simak
1002	Roger	Zelazny
2003	Robert	Heinlein

Fig. 5. Query results display

We move on in order to check whether this code allows you to work with other data providers. Now, let's execute the same actions for the second database in our application. Select the System.Data.OleDb value in the combobox. Select specifically this value because we have a database for this data provider and a connection string of this database in the configuration file of our application. The connection string named MyAccess uses the System.Data.OleDb data provider as the providerName attribute value shows this. Now, let's enter the select query suitable for this database and execute it clicking the Execute request button. In the bottom part of the window we will see the query result. When selecting the System.Data.OleDb data provider, I get the following picture (I entered the select query manually):



System.Data.OleDb

Connection string:

Provider=Microsoft.ACE.OLEDB.12.0;Data Source=Library.accdb;Persist Security Info=False;

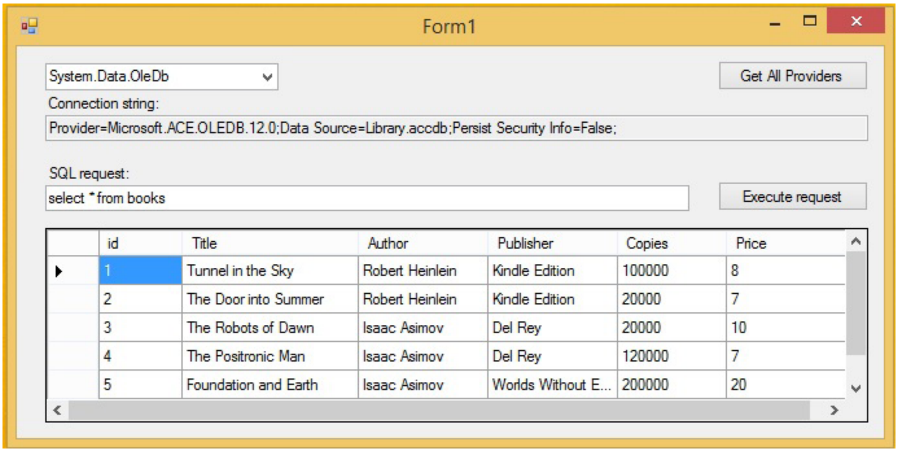
SQL request:

select * from Books

Name	Description	InvariantName	AssemblyQualifiedN
Odbc Data Provi...	.Net Framework ...	System.Data.Odbc	System.Data.Odb...
OleDb Data Provi...	.Net Framework ...	System.Data.Ole...	System.Data.Ole...
OracleClient Data...	.Net Framework ...	System.Data.Ora...	System.Data.Ora...
SqlClient Data Pr...	.Net Framework ...	System.Data.Sql...	System.Data.Sql...
Microsoft SQL Se...	.NET Framework ...	System.Data.Sql...	System.Data.Sql...

Fig. 6. Access to the MS Access 2007 table

After executing the "select * from books" query for this table, I got the following result:



System.Data.OleDb

Connection string:

Provider=Microsoft.ACE.OLEDB.12.0;Data Source=Library.accdb;Persist Security Info=False;

SQL request:

select * from books

id	Title	Author	Publisher	Copies	Price
1	Tunnel in the Sky	Robert Heinlein	Kindle Edition	100000	8
2	The Door into Summer	Robert Heinlein	Kindle Edition	20000	7
3	The Robots of Dawn	Isaac Asimov	Del Rey	20000	10
4	The Positronic Man	Isaac Asimov	Del Rey	120000	7
5	Foundation and Earth	Isaac Asimov	Worlds Without E...	200000	20

Fig. 7. MS Access 2007 table data

This application shows that the same code can work with different data providers. You understand that it's possible thanks to the DbProviderFactory and DbProviderFactories classes. Do you want to try any other provider? Add a connection string to the relevant database in our configuration file and ensure that the required provider is registered on your computer. You should change nothing in the application code.

And one more thing. For you to warm-up. Look closely at MS Access 2007 table data shown in the last picture. Think, why shouldn't this table be used in a real database? The answer is, of course, because the above table is not normalized. It is not even brought to 1NF.

DbDataAdapter additional features

Data Control in DataSet

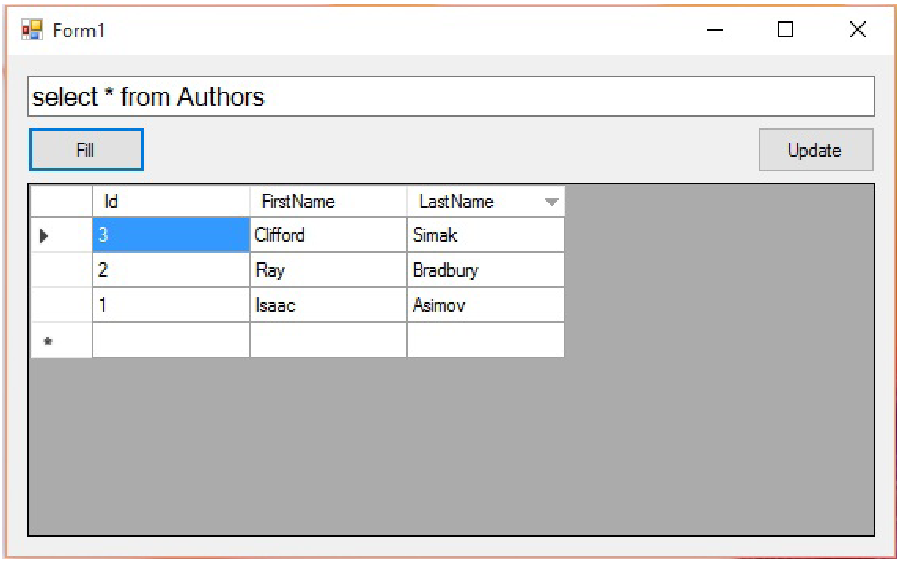
Let's assume that you executed any select query and got the result of this query in DataSet. Then you disconnected from the data source and continue to work with the received data locally. Now, assume that you need to sort or filter the data in your DataSet somehow. How can you do it? Of course, you can execute a new query, wherein you can specify required conditions of sorting and selection. Can we do it without accessing the database once again? Because we already have the required data in DataSet and we only want to represent them in another form.

You can filter and sort local data in DataSet using the DataViewManager class. The DataViewManager object allows you to control sorting and filtration of the local data using its RowFilter and Sort properties. Let's consider how it can be done.

```
// create an adapter  with a query to the Authors
// table
SqlDataAdapter adapter = new SqlDataAdapter("SELECT *
FROM Authors", conn);

// execute the query and enter results into DataSet
adapter.Fill(set, "Authors");

// for DataSet, create a
// DataViewManager object with our results
DataViewManager dvm = new DataViewManager(set);
```

```
// specify conditions of the selection and sorting
// for the required table in DataSet
dvm.DataViewSettings["Authors"].RowFilter = "id < 100";
dvm.DataViewSettings["Authors"].Sort = "LastName ASC";

// create an object containing the selected and
// sorted data and bind this object to dataGridView
DataView dataView1 =
      dvm.CreateDataView(set.Tables["Authors"]);

dataGridView1.DataSource = dataView1;
```

If you insert this code fragment into our third application, you will get the following result on the screen:



Id	FirstName	LastName
3	Clifford	Simak
2	Ray	Bradbury
1	Isaac	Asimov

Fig. 8. Use of filtration and sorting

It's clear that it's not the best solution to insert the filtration and sorting control strictly into the code. It would be better to create a graphic interface that allows you to control the data selection. But it's inexpedient to create such an interface for our demo-exampl.

Now you know how to add a data display control to DataSet using the DataViewManager class in your application. The effectiveness of this approach is due to the fact that you shouldn't execute a new query each time you want to see data in a new form. The query is executed only once, and then different selections are executed from local data.

Transaction support

We have already discussed that saving the database data integrity is one of the most important tasks of any DBMS. Transactions are one of the main tools for this. Let's recall what a transaction is and how it works.

Transaction is a mechanism designed to comply with the database data integrity, when executing any changes of this data. Transaction can be represented as a set of any actions with data designed as a whole, as a command batch. If all single operations contained in a transaction are executed successfully, a database moves to a new, changed but integrated state. Even if one of the actions that the transaction includes can't be executed for any reason, all the actions executed before this action of the current transaction will be canceled. In this case, one says that the transaction rolls back and the database returns to the initial integrated state, in which it was before the transaction execution started.

Formally, in terms of SQL, the transaction begins with the Begin statement and ends either with the Rollback statement or with the Commit statement. Let's consider the money transfer transaction in the amount of @Value from the account with an id equals the @FromId value to the account with an id equals the @ToId value.

```
-- -- transaction begins
BEGIN TRANSACTION
-- add money to the recipient account
UPDATE Account
        SET Balance = Balance + @Value
        WHERE Id = @ToId

    -- check whether money was added to the recipient
    -- account
    IF (@@error <> 0)
        -- cancel the transaction if there are errors
        ROLLBACK TRANSACTION -- if an error occurs,
               -- execute the transaction rollback


-- withdraw money from the sender's account
UPDATE Account
        SET Balance = Balance - @Value
        WHERE Id = @FromId

    -- check whether money was withdrawn from the
    -- sender's account
    IF (@@error <> 0)
        ROLLBACK TRANSACTION -- if an error occurs,
               -- execute the transaction rollback
-- Transaction completes successfully
COMMIT TRANSACTION
```

Of course, this example is conditional, but it demonstrates the transaction use logic.

ADO.NET should provide us with the opportunity to work with transactions. Let's consider what classes are applied for creating transactions and how we should work with them. The Transaction class is used to support transactions, and in this case, together with it the Connection and Command classes are applied and we already know them.

The transaction creation algorithm looks as follows.

- To create a transaction, the Connection.BeginTransaction() method is called;
- The created transaction is entered into the Command. Transaction property;
- A list of queries or calls of the stored procedures, which the transaction should contain, is formed;
- The transaction can be completed, using the Transaction. Commit() or **Transaction.Rollback()** methods.

Now, after the brief theoretic overview, let's add a transaction to our application of the second lesson, to which I assigned the LibraryTest3 name. Let's represent the transaction in the form of a separate method and provide it with its own button. The added method can look as follows:

```
private void btntran_Click(object sender, EventArgs e)
{
    conn = new SqlConnection(cs);
    SqlCommand comm = conn.CreateCommand();
    SqlTransaction tran = null;
    try
    {
        conn.Open();
        tran = conn.BeginTransaction();
        comm = conn.CreateCommand();
        comm.Transaction = tran;
        comm.CommandText = @"create table tmp3(
            id int not null identity(1,1) primary key,
            f1 varchar(20), f2 int)";
        comm.ExecuteNonQuery();
        comm.CommandText = @"insert into tmp3(f1, f2)
            values('Text value 1', 555)";
        comm.ExecuteNonQuery();
        comm.CommandText = @"insert into tmp4(f1, f2)
            values('Text value for second row', 777)";
         comm.ExecuteNonQuery();

         tran.Commit();
    }

    catch (Exception ex)
    {
         MessageBox.Show(ex.Message);
         tran.Rollback();
    }
    finally
    {
         conn.Close();
    }
}
```

Let's consider in detail the above code. We are interested in the strings highlighted in yellow. Create the SqlTransaction type reference and initialize it by calling the Connection. BeginTransaction() method. For the created SqlCommand object, initialize the Transaction property by the newly created transaction. Now enter the necessary queries into the SqlCommand object successively and execute them by calling the ExecuteNonQuery() method since these queries return nothing. Note that in this case the results of each query are NOT applied to the database immediately and they don't change the database state! The results of each query included in the transaction are stored in the RAM either until calling the Commit() method, and then they will be entered into the database, or until calling the Rollback() method, and in this case, they will be ignored.

I deliberately made a mistake in the third query included in the transaction, when I specified the incorrect table name (tmp4) there. If you click the button now and execute the above transaction, you will not see the table named tmp3 in the database. Although the queries for creating a table and for entering the first entry into it are written correctly, they will not be executed because they are included in the transaction, which contains one incorrect statement. One incorrect statement cancels the result of two previous correct statements. The transaction should behave in such a way.

Correct the table name in the third query to tmp3 and execute the transaction once again. In this case, you will not see the exception message. If you enter now the "select * from tmp3" query to the window of our application and click the Fill button, you will see a new table data in our dataGridView1. Transaction works.

Delete the created tmp3 table from the database since we won't need this table. I want to say a few words about what actions should be executed as an explicit transaction. This is usually dictated by the application logic. Classic example is a money transfer from one account to another. Your application should reduce the sender's balance by the specified amount and increase the recipient's balance by the same amount. You withdraw money from the sender's account, and the application completes its work for any reason, without adding money to the recipient's account. The data integrity is compromised. But if the process of money withdrawing from one account and adding it to another one was arranged as a transaction, the data integrity would not be compromised.

So, the following question may arise: if the transaction is so useful in ensuring the data integrity, should all the database actions be arranged as transactions? To answer this question correctly, we should adhere to the special terminology. You should distinguish such concepts as "explicit transaction" and "implicit transaction" (or simply "transaction").

Explicit transaction is a set of SQL statements that starts with the BEGIN TRANSACTION statement and ends with either the ROLLBACK TRANSACTION or COMMIT TRANSACTION statement. If any action included in the explicit transaction ends with an error, ROLLBACK TRANSACTION is executed, and the transaction rolls back all the data modifications right up to the beginning transaction execution state. If all actions are completed successfully, COMMIT TRANSACTION will be executed and data will move to a new integrated state.

Implicit transaction doesn't include the BEGIN TRANSACTION, ROLLBACK TRANSACTION, or COMMIT TRANSACTION statements, but herewith, it is executed as a transaction as well. For example, the DDL SQL queries (select, insert, update, and delete) are executed as implicit transactions. In other words, if you execute the insert query, which should fill twenty fields in any table, this query will not be completed if only twelve or fifteen fields are filled. It either will fill all twenty fields or will not fill any field at all in case of any error. The relevant data provider developers bothered about such a query execution.

Now let's return to our question of whether one should arrange all the database actions as transactions. It would be more correct to formulate this question as follows: "Should all the database actions be arranged as explicit transactions?" The answer is: "No, they should not". The fact is that only the actions related to each other logically should be arranged as an explicit transaction, for example, if they link the processing of several tables into one whole, as in the money transfer example. The explicit transaction is a quite resource-intensive thing and one should use it "without bigotry". Developers of data providers bothered about the transactional execution of other actions in a database.

Working asynchronously with database

It's time to talk about an asynchrony when working with data providers. You understand perfectly well that it would be better to execute some database actions asynchronously. What is asynchrony? It is no synchrony or blocking. Of course, you have faced a situation, when a program begins to execute an action and suddenly, a window of this program becomes inactive, a message "*Program name* is not responding" can appear in the heading, and any user's actions become impossible. After some time, everything restores, and the program behaves itself like nothing happened. It's a typical example of blocking. A blocking like this arises due to the fact that an active action makes everyone wait until it (this action) will be completed. And if this action is executed in the primary thread of the application, it's very bad. The fact is that the primary thread of the application is designed to serve user's actions with controls of the program window. And if the primary thread executes the labor-intensive operation, nobody will serve the program window operation. To avoid such situations, labor-intensive actions should be executed asynchronously, that is, in such a way so they won't cause blocking. One of the methods for ensuring the asynchrony is to execute the blocking action in an additional thread.

Many people believe that asynchrony is a work in an additional thread. It's not so. Asynchrony is no blockings at all. This can be achieved without using additional *threads. You will learn how to do this when considering the async and await specifiers.*

Now, we are going to consider the asynchrony implementation with the use of additional threads. You should understand that the additional threads are very recourse-intensive. Therefore, you should use them carefully. First, let's get acquainted with methods provided by ADO.NET to work with a database in additional threads.

And then we will talk once again about the cases, when one should use asynchrony.

Today, there are two methods for writing an asynchronous code that uses additional threads to execute blocking actions in them: to use the classic BeginXXX() and EndXXX() pattern or new async and await means.

Classic approach to asynchrony

When does the application start to work with the database and when does the data transfer between the application and database server begin? It occurs when executing the database queries. At the moment when the application calls the ExecuteNonQuery(), ExecuteReader() or ExecuteScalar() methods. Thus, we should provide asynchrony namely for these methods. The asynchronous execution of any action is usually provided by two methods. One method, that is conditionally called BeginAction(), begins to execute the necessary action in the additional thread. Another method, that is conditionally called EndAction(), should be called when the action in the additional thread will be completed. You should understand that in the BeginAction() and EndAction() methods the word Action means the name of some specific action. For example, Read and Write. BeginAction() and EndAction() are generalized names of the asynchronous methods, there are no methods with such names really. And there are, for example, the BeginRead() and EndRead(), BeginWrite() and EndWrite() methods, and the asynchronous methods for working with the database, which we are going to get acquainted with now. Let's continue considering the asynchrony implementation schemes.

You should understand two moments in this scheme:

- If the action, executed in an additional thread of the BeginAction() method, should return some result, then this result can be obtained only after calling the EndAction() method;
- If we call the EndAction() method before completing the additional thread action, this call of the EndAction() method will be blocking and will block the thread, which called it.

Thus, the task boils down to how we can find out when the additional thread action was completed in order to call the EndAction() method and not to block the application for obtaining the result. There are three methods for this:

- use of callback delegates;
- use of WaitHandle class;
- additional thread survey.

They are the following asynchronous options of the command methods: BeginExecuteNonQuery() and EndExecuteNonQuery(), BeginExecuteReader() and EndExecuteReader(). There is no asynchronous option for the ExecuteScalar() method.

In addition to these methods, the IAsyncResult interface also plays an important role in the asynchrony implementation.

What should we know about this interface?

- BeginExecuteNonQuery() and BeginExecuteReader() have the IAsyncResult return value type.
- Callback methods that are used in the asynchronous mechanism have a single parameter of the IAsyncResult type.
- The IAsyncResult.AsyncWaitHandle property contains the WaitHandle object that allows you to control the EndAction() method call.
- One can transfer data between the additional and main thread through the IAsyncResult.AsyncState property.

We are going to talk in more detail about all the features of the IAsyncResult type when creating the application code. Consider all three methods of executing the asynchronous actions.

Use of callback methods

Let's create a version of the asynchronous reference to the database in the form of a separate method and use it once again in the LibraryTest3 application of the second lesson. Add one more button to the specified project, sign it, for example, Async Callback, and create a handler for this button. Insert the following code in the created handler:

```
private void btAsync_Click(object sender, EventArgs e)
{
    /// block 1
    const string AsyncEnabled =
              "Asynchronous Processing=true";
    if (!cs.Contains(AsyncEnabled))
    {
        cs = String.Format("{0}; {1}", cs, AsyncEnabled);
    }
    ///

    conn = new SqlConnection(cs);
    SqlCommand comm = conn.CreateCommand();

    /// block 2
    comm.CommandText = "WAITFOR DELAY '00:00:05';
         SELECT * FROM Books;";
    comm.CommandType = CommandType.Text;
    comm.CommandTimeout = 30;
    ///
    try
    {
        conn.Open();
        /// block 3
        AsyncCallback callback =
                  new AsyncCallback(GetDataCallback);
       comm.BeginExecuteReader(callback, comm);
       MessageBox.Show("Added thread is working...");
       ///
    }
    catch (Exception ex)
    {
       MessageBox.Show(ex.Message);
    }
}
```

In this simple handler, there are three code blocks about which I should tell you in detail. In the first block, the connection string is modified. The fact is that when using the asynchronous access to the data source, the connection string should contain the "Asynchronous Processing=true" attribute. Therefore, we check whether there is this attribute in this block, and if there is no attribute in the connection string, it will be added to the read copy of the string. Herewith, the connection string remains unchanged because we don't plan to use asynchrony all the time.

In the second block, we simulate the long operation implementation on the database server. The SQL "WAITFOR DELAY '00:00:05'" query is used for this. This query causes a five-second stop of the database server to give us time to make sure that the window of our application is not blocked when executing a database query. For this purpose, we output a dialog box with the message: "Added thread is working..." in order to show that the application continues working in the primary thread after it started to work in the additional thread. Here, one should pay attention to the initialization of the comm.CommandTimeout property. The default value of this property is equal to 30 seconds. This time is given to the application to connect to the database server and execute the query. If you think that your application might need more time for it, you should change the value of this property. In our case, the default value remains in order to demonstrate the necessity to remember about this property.

And finally, in the third block, a very important part of work is executed. Here, we create a delegate of the AsyncCallback type, into which we enter an address of a callback method named GetDataCallback(). The name of this method is arbitrary, but the signature should be as follows: void is a return value type and one parameter of the IAsyncResult type. Let's look at how this method is used when running the asynchrony operation, and then we will present its code and consider in more detail how it works. The additional thread work starts when calling the comm.BeginExecuteReader(callback, comm) method:

```
comm.BeginExecuteReader(callback, comm);
```

Note that we transfer the created delegate with the callback method address to an additional thread. This delegate will do so that our callback GetDataCallback() method will be called by the system automatically when the additional thread will complete its work. Moreover, we transfer our conn object to the additional thread. Let's consider now the GetDataCallback() method code in detail, which also should be added to our project next to the created handler.

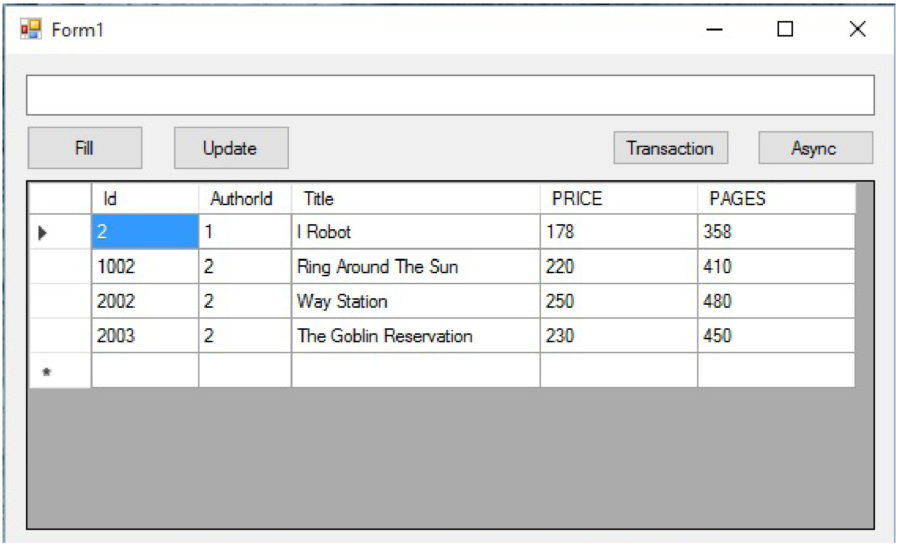
```
private void GetDataCallback(IAsyncResult result)
{
    SqlDataReader reader = null;
    try
    {
        /// block 1
        SqlCommand command = (SqlCommand)result.AsyncState;
        ///
        /// block 2
        reader = command.EndExecuteReader(result);
        ///
        table = new DataTable();

        int line = 0;

        do
        {
            while (reader.Read())
            {
                if (line == 0)
                {
                    for (int i = 0; i <
                              reader.FieldCount; i++)
                   {
                    table.Columns.Add(reader.GetName(i));
                   }
                   line++;
                }
                DataRow row = table.NewRow();
                for (int i = 0; i < reader.FieldCount; i++)
                {
                    row[i] = reader[i];
                }
                table.Rows.Add(row);
               }
      } while (reader.NextResult());
      DgvAction();
    }
    catch (Exception ex)
    {
        MessageBox.Show("From Callback 1:"+ex.Message);
    }
    finally
    {
        try
        {
            if (!reader.IsClosed)
            {
                reader.Close();
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show("From Callback 2:" +
                                ex.Message);
        }
    }
}
```

Since the action is executed in the additional thread of this method, we can't refer to DataGridView from this method directly. Therefore, we have created the DgvAction() method, from which we refer to DataGridView in order to display query results in it. To do this, we should make the table object global in order to provide an access to it from GetDataCallback() and DgvAction().

```
private void DgvAction()
{
    if (dataGridView1.InvokeRequired)
    {
        dataGridView1.Invoke(new Action(DgvAction));
        return;
    }
    dataGridView1.DataSource = table;
}
```

Remember that when calling BeginExecuteReader(), we transferred the comm object to the additional thread in the second parameter. Keep in mind that all you transfer to the callback method will be in the AsyncState property of the parameter of this callback method. The AsyncState property has the Object type; therefore, we can transfer any types to the additional thread. We just take away the transferred comm object from the input parameter in the first block of the highlighted code. You may ask: "Why do we need this object in the additional thread?" The fact is that the BeginExecuteReader() method was called from this object, and we should finish this action by calling the EndExecuteReader() method from the same object. We do this in the second block. Note once again, that one can get the result of the asynchronous action only after calling the EndAction() method. In our case, one should call the EndExecuteReader() method. In our example, the reader object filled with data is a result of the action execution, and we get it after calling the EndExecuteReader() method and work further with it.

Next, we execute the standard SqlDataReader processing you already know: extract data in DataTable and display them in dataGridView1. Note that there are two queries in the command property of the comm object, and both of them are processed normally. When running the application, I obtained the following result. In the above picture I have already closed the dialog box with the message: "Added thread is working...".



Id	AuthorId	Title	PRICE	PAGES
2	1	I Robot	178	358
1002	2	Ring Around The Sun	220	410
2002	2	Way Station	250	480
2003	2	The Goblin Reservation	230	450

Fig. 9. Asynchronous database access

Now, we focus on how our asynchronous query to the database was executed. After calling the BeginExecuteReader() method, a new thread was created automatically and in it, our application has connected to the database server and began to execute the transferred queries on a server. Meanwhile, the main thread of our application has continued working; in particular, it has formed and output the dialog box with the message. At some point of time, the additional thread completed its work. And here, our callback GetDataCallback() method has played its role. This role is that this method was called by the system automatically (therefore, it's a callback method), as soon as the work in the additional thread was completed. We have already discussed everything that occurs in the GetDataCallback() method. We wouldn't have to define the work completion of the additional thread explicitly. The callback GetDataCallback() method has done this instead of us. And note that we called the BeginExecuteReader() method and did not obtain its return value from it. We can use the callback methods without this value.

Use of the WaitHandle class

Now, we will consider another mechanism of the asynchronous database access. We will use an object of the WaitHandle synchronization class in order to find out whether the work was completed in the additional thread and call the EndAction() method. Add a new button in the LibraryTest3 application window again and create a handler for this button:

```
private void btAsync2_Click_1(object sender, EventArgs e)
{
    const string AsyncEnabled =
                        "Asynchronous Processing=true";
    if (!cs.Contains(AsyncEnabled))
    {
        cs = String.Format("{0}; {1}", cs, AsyncEnabled);
    }
```

```
        conn = new SqlConnection(cs);
        SqlCommand comm = conn.CreateCommand();
        comm.CommandText = "WAITFOR DELAY '00:00:05';
                            SELECT * FROM Books;";
        comm.CommandType = CommandType.Text;
        comm.CommandTimeout = 30;
        try
        {
            conn.Open();
            /// block 1
            IAsyncResult iar = comm.BeginExecuteReader();
            ///

            /// block 2
            WaitHandle handle = iar.AsyncWaitHandle;
            ///
            /// block 3
            if(handle.WaitOne(10000))
            {
                /// block 4
                GetData(comm, iar);
                ///
            }
            else
            {
                MessageBox.Show("TimeOut exceeded");
            }
            ///
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }
```

Let's consider the above example in detail. Only those strings are highlighted in it that differ from the previous version. The BeginExecuteReader() method is called in the first block. Now when calling, we get the return value of this method and don't transfer any parameters to it. As you see, there are no delegates and callback methods either.

In the second block, from the value that is returned by the BeginExecuteReader() method, we take an object of the WaitHandle type, with the help of which we find out about the work completion in the additional thread. Don't be fooled by the fact that we obtain the return value from the BeginExecuteReader() method, which still continues to work in the additional flow. The thing is that this method isn't blocking. It starts the work in the additional thread and immediately returns the control to the flow that called it. The WaitHandle object is located in the returned value, in the AsyncWaitHandle property, and it is accessible from the additional thread and will signal to us that the work of the additional thread is completed. In the third block, we check whether the work of the additional thread is completed. When the additional thread will be completed, the WaitOne() method will return true and our code will call the usual (not callback) GetData() method to get a result. Note that the timeout is specified in the WaitOne() method in order to avoid freezing of the application in the case of any error in the additional thread.

You already understand, why the comm and iar objects are transferred to this GetData() method. Below, there is a code of the GetData() method, which is very similar to the callback method of the previous section.

```
private void GetData(SqlCommand command, IAsyncResult ia)
{
    SqlDataReader reader = null;
    try
    {
        reader = command.EndExecuteReader(ia);
        DataTable table = new DataTable();
        dataGridView1.DataSource = null;

        int line = 0;

        do
        {
        while (reader.Read())
        {
            if (line == 0)
            {
               for (int i = 0; i < reader.FieldCount;
                                   i++)
               {
                  table.Columns.Add(reader.GetName(i));
               }
               line++;
            }
            DataRow row = table.NewRow();
            for (int i = 0; i < reader.FieldCount; i++)
            {
               row[i] = reader[i];
            }
            table.Rows.Add(row);
        }
        } while (reader.NextResult());
        dataGridView1.DataSource = table;
    }
    catch (Exception ex)
    {
         MessageBox.Show("From GetData:" + ex.Message);
        }
        finally
        {
            try
            {
                if (!reader.IsClosed)
                {
                    reader.Close();
                }
            }
            catch
            {
            }
        }
    }
```

Run this application and make sure it works. What should one note about this method? You understand that the call of WaitOne() is blocking and our application waits for the main thread completion. It would seem that this is a drawback. But in some cases, the application has nothing to do until the additional thread is completed. Moreover, the main advantage of using the WaitHandle object appears when one should follow the completion of several additional threads. In addition to the WaitOne() object, there are also the WaitAll() and WaitAny() static methods in this class. These methods accept arrays of the WaitHandle objects that correspond to the running additional threads, and can signalize about the completion of all the traceable threads, or about the completion of one of the traceable threads. In order to learn how to use these methods, you will be offered the relevant homework.

Use of the additional thread survey

In two previous examples, you saw how a main thread can find out about the work completion of the additional thread in order to call the EndAction() method and get the result. There is one more way to do this. You need to survey the IsCompleted property of the object returned by the BeginExecuteReader() method:

```
IAsyncResult iar = comm.BeginExecuteReader();
while (!iar.IsCompleted)
{
    Console.WriteLine("Waiting...");
}
```

Use of this method is a "blank run" of the application, or an aimless waste of CPU time.

It is unlikely to use this approach in real applications. Although sometimes you may need such a method.

Home Task

In this homework, you should create an application, which will connect to a database and read data from two different tables simultaneously in two different threads. After of both threads complete their work, they should output some final results of their work. Since our database is small and the work of the threads will be executed very quickly, it's desirable to add a few seconds delay to each query.

Note that results should be output when both threads will complete their work. If one of them will complete its work earlier, it will have to wait until the second thread completes its work. The implementation of this moment is one of the most interesting problems in this task. Another task is to come up with how to create additional threads. Select and implement solutions of these tasks by yourself.

So. Create a Windows Forms application with the "Start" button and with two small fields, wherein results of the threads work will be output. The text fields can be signed, for example, "Flow 1" and "Flow 2". The "Start" button should start a creation of two additional threads. The "select * from Books" query should be executed in one thread, and the "select * from Authors" query should be executed in the second thread. The first thread should calculate a total number of characters for all the strings in the title field of the Books table. The second thread should calculate a total number of characters in the FirsName and LastName fields of the Authors table. These calculated numbers of characters should be output into the relevant text fields.