CSCE 547
Windows Programming

ADO.NET

Department of Computer Science and Engineering
University of South Carolina
Columbia, SC 29208

ADO.NET

Part of the vision behind .NET is to provide sophisticated DB support for web applications.

One of the consequences of .NET's modular design is that the same DB support that exists for Window Forms is also available for Web-based applications.

In fact, as the text book suggests, Amazon.com and Ebay.com are nothing more than web fronts to sophisticated DBMSs.

DB support in .NET is built around ADO.NET.

Most of that support is located within the System.Data name space.

This chapter discusses such topics as DB protocols, connections, commands (transacted and parameterized), data readers, data adapters, stored procedures, data sets, data adapters, and data views.

The common thread is data.

A Milieu of Acronyms

Before ADO.NET we had ODBC, DAO, ADO, RDO, OLE DB, and others.

Therefore the natural question is why yet another API?

The short answer is that ADO.NET is designed to provide DB support for HTTP transactions at the core of the web-based paradigm.

Applications using ADO.NET access data through data providers, in the form of OLE DB or SQL DB providers.

OLE DB provides offer support for legacy applications based on Oracle, Sybase, etc, through ODBC or similar protocols installed in your system.

SQL DB data providers directly interface Microsoft's SQL Server and are faster than the OLE DB counterpart.

Selection of Data Providers (DP)

Before embarking on writing a DB application, developers should consider carefully which DP is most adequate.

MS SQL Server .NET is compatible with MS Server 7.0 or later. This is the fastest DP available on the .NET platform, for the simple reason that it was optimized for .NET

Applications that must use other DPs, such as Oracle, Sybase, etc, are condemned to access data through OLE DB.NET; this is not bad *per se*, the OLEDB.NET DP was written by MS simply to provide a DP available for users of other DBs.

It is expected that in the near future, Oracle, Sybase, etc., will provide their own, optimized versions of .NET DPs.

Probably, as we speak, those DPs are already present in the marketplace. (Apple announced the release of a MacOs version of .NET

Typical Use of Connections and Commands

- Applications using ADO.NET will use two types of objects: **Connection** and **Command objects**, in their SQL or OLEDB flavors.
- The steps that would be typically used with ADO.NET are:
  1. Create a Connection object that contains a connection string.
  2. Use the Connection.Open() method
  3. Create a Command object that links the Command with the connection.
  4. Call one of the Command.Execute() methods to execute a command string
  5. Use the **Connection.Close**() method

Differences between SQL vs OLE DB code

```
using System.Data.SqlClient;
SqlConnection conn = new SqlConnection
    ("server=localhost;database=pubs;uid=sa;pwd=");
try {
    conn.Open ();
    SqlCommand cmd = new SqlCommand ("select * from titles", conn);
    SqlDataReader reader = cmd.ExecuteReader ();
    while (reader.Read ())
        Console.WriteLine (reader["title"]);
}
catch (SqlException ex) {
    Console.WriteLine (ex.Message);
}
finally {
    conn.Close ();
}
```

```
using System.Data.OleDb;
OleDbConnection conn = new OleDbConnection
 ("provider=sqloledb;server=localhost;database=pubs;uid=sa;pwd=");

try {
    conn.Open ();
    OleDbCommand cmd = new OleDbCommand ("select * from titles", conn);
    OleDbDataReader reader = cmd.ExecuteReader ();
    while (reader.Read ())
        Console.WriteLine (reader["title"]);
}
catch (OleDbException ex) {
    Console.WriteLine (ex.Message);
}
finally {
    conn.Close ();
}
```

Two Flavors of the Connection Class

System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Data.SqlClient.SqlConnection

public sealed class SqlConnection : Component, IDbConnection, ICloneable

System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Data.OleDb.OleDBConnection

public sealed class OleDbConnection : Component, IDbConnection, ICloneable

Options to the SQL Connection Class

SQL Connections are created using various flavors of the constructor:

```
SqlConnection conn = new SqlConnection ();
conn.ConnectionString = "server=localhost;database=pubs;uid=sa;pwd=";

SqlConnection conn = new SqlConnection
    ("server=localhost;database=pubs;uid=sa;pwd=");

SqlConnection conn = new SqlConnection
    ("server=hawkeye\wintellect;database=pubs;uid=sa;pwd=");
```

If the parameters are incorrect, the constructor throws a **System.ArgumentException**.

You must become familiar with the various options available in the constructor and the Connection String. The sample below omits security:

```
SqlConnection conn = new SqlConnection
    ("server=hawkeye\wintellect;database=pubs;uid=sa;pwd=;" +
    "min pool size=10;max pool size=50;connect timeout=10");
```

Options to the OLE DB Connection Class

OLE DB Connections also have several flavors of constructors.

Note that the format of the connection string is dependent upon the specific DP being connected

```
OleDbConnection conn = new OleDbConnection
    ("provider=sqloledb;server=localhost;database=pubs;uid=sa;pwd=");

OleDbConnection conn = new OleDbConnection
    ("provider=sqloledb;data source=localhost;" +
    "initial catalog=pubs;user id=sa;password=");

OleDbConnection conn = new OleDbConnection
    ("provider=sqloledb;data source=localhost;OLE DB Services=-2" +
    "initial catalog=pubs;user id=sa;password=");
```

If the parameters are incorrect, these constructors throw a **System.ArgumentException**.

Opening and Closing Connections

Once a connection object is created, it can be open for business. Note the similarities between opening a SQL and an OLE DB connection:

```
SqlConnection conn = new SqlConnection
   ("server=localhost;database=pubs;uid=sa;pwd=");
try {
    conn.Open ();
    // TODO: Use the connection
}
catch (SqlException ex) {
    // TODO: Handle the exception
}
finally {
    conn.Close ();
}
```

```
OleDbConnection conn = new OleDbConnection
   ("provider=sqloledb;server=localhost;database=pubs;uid=sa;pwd=");
try {
    conn.Open ();
    // TODO: Use the connection
}
catch (OleDbException ex) {
    // TODO: Handle the exception
}
finally {
    conn.Close ();
}
```

Command Classes

As before, there are two flavors of command classes.

```
SqlConnection conn = new SqlConnection
    ("server=localhost;database=pubs;uid=sa;pwd=");
try {
    conn.Open ();
    SqlCommand cmd = new SqlCommand ();
    cmd.CommandText = "delete from titles where title_id = 'BU1032'";
    cmd.Connection = conn;
    cmd.ExecuteNonQuery (); // Execute the command
}
catch (SqlException ex) {
    // TODO: Handle the exception
}
finally {
    conn.Close ();
}
```

The other flavor is OleDbCommand. There are a number of parameters and methods that if used properly improve performance.

The text book discusses the methods ExecuteNonQuery, ExecuteScalar, and ExecuteReader.

ExecuteNonQuery

This command is used to execute SQL commands that are not expected to return values, e.g., CREATE DATABASE, CREATE TABLE. In some cases the command returns values, e.g., INSERT, UPDATE, DELETE, return the number of records affected.

```
SqlConnection conn = new SqlConnection
    ("server=localhost;database=pubs;uid=sa;pwd=");
try {
    conn.Open ();
    SqlCommand cmd = new SqlCommand
        ("insert into titles (title_id, title, type, pubdate) " +
        "values ('JP1001', 'Programming Microsoft .NET', " +
        "'business', 'May 2002')", conn);
    cmd.ExecuteNonQuery ();
}
catch (SqlException ex) {
    // TODO: Handle the exception
}
finally {
    conn.Close ();
}
```

The chapter also shows samples with UPDATE and DELETE.

ExecuteScalar Method

This command is used to execute SQL commands that are expected to return scalar values, e.g., COUNT, AVG, MIN, MAX, SUM.

```
SqlConnection conn = new SqlConnection
    ("server=localhost;database=pubs;uid=sa;pwd=");
try {
    conn.Open ();
    SqlCommand cmd = new SqlCommand
        ("select max (advance) from titles", conn);
    decimal amount = (decimal) cmd.ExecuteScalar ();
    Console.WriteLine ("ExecuteScalar returned {0:c}", amount);
}
catch (SqlException ex) {
    Console.WriteLine (ex.Message);
}
finally {
    conn.Close ();
}
```

ExecuteScalar returns an Object, which must be type-casted.

Reading BLOBs with ExecuteScalar

ExecuteScalar can also be used to return BLOBs:

```
using System.IO;
using System.Drawing;
using System.Data.SqlClient;
```

```
MemoryStream stream = new MemoryStream ();
SqlConnection conn = new SqlConnection
    ("server=localhost;database=pubs;uid=sa;pwd=");
try {
    conn.Open ();
    SqlCommand cmd = new SqlCommand
        ("select logo from pub_info where pub_id='0736'", conn);
    byte[] blob = (byte[]) cmd.ExecuteScalar ();
    stream.Write (blob, 0, blob.Length);
    Bitmap bitmap = new Bitmap (stream);
    // Use the darn bitmap . . .
    bitmap.Dispose ();
}
catch (SqlException ex) {
    // Handle exceptions
}
finally {
    stream.Close ();
    conn.Close ();
}
```

Writing BLOBs with ExecuteScalar

ExecuteScalar can also be used to write BLOBs into a DB:

```
using System.IO;
using System.Drawing;
using System.Data.SqlClient;

FileStream stream = new FileStream ("Logo.jpg", FileMode.Open);
byte[] blob = new byte[stream.Length];
stream.Read (blob, 0, (int) stream.Length);
stream.Close ();

SqlConnection conn = new SqlConnection
    ("server=localhost;database=pubs;uid=sa;pwd=");
try {
    conn.Open ();
    SqlCommand cmd = new SqlCommand
      ("insert into pub_info (pub_id, logo) values ('9937', @logo)",conn);
    cmd.Parameters.Add ("@logo", blob);
    cmd.ExecuteNonQuery ();
}
catch (SqlException ex) {
    // TODO: Handle the exception
}
finally {
    conn.Close ();
}
```

ExecuteReader

This method provides an efficient mechanism for getting queries from a DB. The command returns a DataReader, which, as before, come in two flavors.

Data Readers are forward-only, read-only objects that iterate very quickly over the records in a table.

Data Readers are especially efficient for reading remotely records from far away DBs.

```
SqlConnection conn = new SqlConnection
    ("server=localhost;database=pubs;uid=sa;pwd=");
try {
    conn.Open ();
    SqlCommand cmd = new SqlCommand ("select * from titles", conn);
    SqlDataReader reader = cmd.ExecuteReader ();
    while (reader.Read ())
        Console.WriteLine (reader["title"]);
}
catch (SqlException ex) {
    Console.WriteLine (ex.Message);
}
finally {
    conn.Close ();
}
```

Getting DB Schemas with DataReader

Using DataReaders this it is is quite easy to obtain the configuration of a DB.

The code below gets the names of all the fields in table “Titles.” Note that the level of inspection can be extended to inquire about tables their names, etc.

You can even ask for details regarding other info, such as stored procedures, relations, commands, etc.

```
SqlConnection conn = new SqlConnection
    ("server=localhost;database=pubs;uid=sa;pwd=");
try {
    conn.Open ();
    SqlCommand cmd = new SqlCommand ("select * from titles", conn);
    SqlDataReader reader = cmd.ExecuteReader ();
    for (int i=0; i<reader.FieldCount; i++)
        Console.WriteLine (reader.GetName (i));
}
catch (SqlException ex) {
    Console.WriteLine (ex.Message);
}
finally {
    conn.Close ();
}
```

Transactions

A transaction is a set of commands executed in sequence, to form a contingent, logical unit, so that failure of any one step is signaled as a failure of the entire sequence.

Typical examples include getting cash from ATMs, bank deposits, withdraws, transfer of funds, etc.

```
SqlConnection conn = new SqlConnection
    ("server=localhost;database=mybank;uid=sa;pwd=");
try {
    conn.Open ();
    // Debit $1,000 from account 1111
    SqlCommand cmd = new SqlCommand
        ("update accounts set balance = balance - 1000 " +
        "where account_id = '1111'", conn);
    cmd.ExecuteNonQuery ();
    // Credit $1,000 to account 2222
    cmd.CommandText = "update accounts set balance = " +
        "balance + 1000 where account_id = '2222'";
    cmd.ExecuteNonQuery ();
}
```

If one operation fails, the whole transaction should fail

Transactions, II

Using a transaction with two commands

```
SqlConnection conn = new SqlConnection
   ("server=localhost;database=mybank;uid=sa;pwd=");
try {
   conn.Open ();
   // Start a local transaction
   trans = conn.BeginTransaction (IsolationLevel.Serializable);
   // Create and initialize a SqlCommand object
   SqlCommand cmd = new SqlCommand ();
   cmd.Connection = conn;
   cmd.Transaction = trans;
   // Debit $1,000 from account 1111
   cmd.CommandText = "update accounts set balance = " +
      "balance - 1000 where account_id = '1111'";
   cmd.ExecuteNonQuery ();
   // Credit $1,000 to account 2222
   cmd.CommandText = "update accounts set balance = " +
      "balance + 1000 where account_id = '2222'";
   cmd.ExecuteNonQuery ();
   // Commit the transaction (commit changes)
   trans.Commit ();
}
```

Create Transaction

Link Transaction with Command

Step one of Transaction

Step Two of Transaction

Execute Command with Transaction

Commit Transaction

Parameterized Queries

Using parameters makes queries more generic:

```
SqlTransaction trans = null;
SqlConnection conn = new SqlConnection
    ("server=localhost;database=mybank;uid=sa;pwd=");
try {
    conn.Open ();
    // Create and initialize a SqlCommand object
    SqlCommand cmd = new SqlCommand
        ("update accounts set balance = balance + @amount " +
        "where account_id = @id", conn);
    cmd.Parameters.Add ("@amount", SqlDbType.Money);
    cmd.Parameters.Add ("@id", SqlDbType.Char);
    // Debit $1,000 from account 1111
    cmd.Parameters["@amount"].Value = -1000;
    cmd.Parameters["@id"].Value = "1111";
    cmd.ExecuteNonQuery ();

    // Credit $1,000 to account 2222
    cmd.Parameters["@amount"].Value = 1000;
    cmd.Parameters["@id"].Value = "2222";
    cmd.ExecuteNonQuery ();
}
```

Parameters are @amount and @id

MS SQL Specific Syntax.

SqlDbType is MS Specific

Cmd has a Parameters property of type string array.

Stored Procedures

Stored procedures are compiled methods that execute faster and provide a mechanism for adding user-defined commands into a DB.

```
CREATE PROCEDURE proc_TransferFunds
    @Amount money,
    @From char (10),
    @To char (10)
AS
    BEGIN TRANSACTION
    UPDATE Accounts SET Balar
        WHERE Account_ID = @F
    IF @@ROWCOUNT = 0
    BEGIN
        ROLLBACK TRANSACTION
        RETURN
    END
    UPDATE Accounts SET Balar
        WHERE Account_ID = @T
    IF @@ROWCOUNT = 0
    BEGIN
        ROLLBACK TRANSACTION
        RETURN
    END
    COMMIT TRANSACTION
GO
```

```
SqlConnection conn = new SqlConnection
   "server=localhost;database=mybank;uid=sa;pwd=");
try {
    conn.Open ();
    SqlCommand cmd = new SqlCommand
      ("proc_TransferFunds", conn);
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.Parameters.Add ("@amount", 1000);
    cmd.Parameters.Add ("@from", 1111);
    cmd.Parameters.Add ("@to", 2222);
    cmd.ExecuteNonQuery ();
}
catch (SqlException ex) {
    // TODO: Handle the exception
}
finally {
    conn.Close ();
}
```

Stored Procedures that Return Values

Stored procedures can return values to the caller

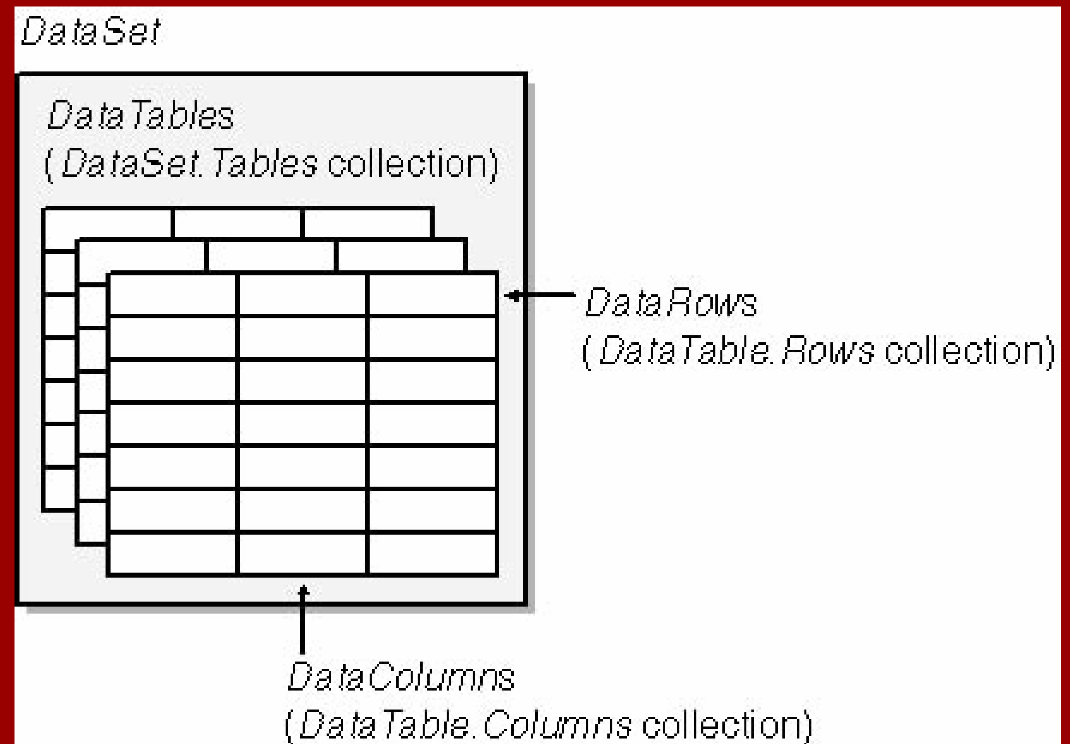
```
SqlConnection conn = new SqlConnection
 ("server=localhost;database=mybank;uid=sa;pwd=");
try {
    SqlCommand cmd = new SqlCommand ("proc_GetBalance", conn);
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.Parameters.Add ("@id", 1111);
    SqlParameter bal =
        cmd.Parameters.Add ("@balance",SqlDbType.Money);
    bal.Direction = ParameterDirection.Output;
    SqlParameter ret = cmd.Parameters.Add ("@return", qlDbType.Int);
    ret.Direction = ParameterDirection.ReturnValue;
    cmd.ExecuteNonQuery ();
    int retval = (int) ret.Value;
    decimal balance = (decimal) bal.Value;
}
```

```
    SELECT @Balance = Balance FROM Accounts WHERE Account_ID = @ID
    IF @@ROWCOUNT = 1
        RETURN 0
    ELSE
    BEGIN
        SET @Balance = 0
        RETURN -1
    END
GO
```

DataSets and DataAdapters

DataSets are objects defined in the System.Data name space. These objects capture in memory the results of a query, support backward and forward traversal through the data, enable data editing, and can propagate the edited changes back to the original data source.

DataSets contain DataTables, these are collections of DataRows and DataColumns, as shown in the figure.



Data Adapters, is a type of object that provide bridges between DataSets and the actual physical data sources.

DataSets vs DataReaders

DataReaders provide read-only, forward only access.

DataSets provide read-write, random, cacheable access

DataReaders are significantly faster than DataSets, but have more limited capabilities.

Most of the time DataSets are created from database queries or from XML documents. In both cases the actual physical work is done by DataAdapters

DataAdapters

Most of the time DataSets are created from database queries or from XML documents.

In both cases the actual physical work is done by DataAdapters

DataAdapters come in two flavors, one for SQL, another for OleDB.

The common base class is **System.Data.Common.DbDataAdapter**

The two most important methods of DataAdapters are **Fill** and **Update**.

DataAdapter.Fill()

```
SqlDataAdapter adapter = new SqlDataAdapter
  ("select * from titles","server=localhost;database=pubs;uid=sa;pwd=");
DataSet ds = new DataSet ();
adapter.Fill (ds, "Titles");
```

1. Fill opens a connection to the Pubs DB using the connection string passed to SqlDataAdapter's constructor.
2. The adapter performs a query on the Pubs DB using the query string passed to the SqlDataAdapter constructor.
3. The adapter creates a DataTable named "Titles" in the DataSet.
4. The adaper initializes the DataTable with a schema that matches that of the "Titles" table in the DB.
5. The adapter retrieves all the records produced by the query and writes them into the DataTable.
6. The adapter closes the connection to the DB.

DataTables

A DataSet is a collection of DataTables, created most of the time by DataAdapters.

Tables can be referenced by name or by zero-based indexes, as shown below:

```
foreach (DataTable table in ds.Tables)
     Console.WriteLine (table.TableName);

DataTable table = ds.Tables[0];
foreach (DataRow row in table.Rows)
    Console.WriteLine (row[0]);

DataTable table = ds.Tables[0];
foreach (DataRow row in table.Rows)
    Console.WriteLine (row["account_id"]);


DataTable table = ds.Tables[0];
foreach (DataColumn col in table.Columns)
    Console.WriteLine ("Name={0}, Type={1}",
        col.ColumnName, col.DataType);
```

Tables can be referenced by name

Tables can be referenced by index

Tables contain collections of rows and columns, which can be accessed by name or index

DataTables

The chapter shows code for inserting, selecting, updating and deleting records from a table.

```
DataSet ds = new DataSet ();
adapter.Fill (ds, "Titles");
// Create a new DataRow
DataTable table = ds.Tables["Titles"];
DataRow row = table.NewRow ();
// Initialize the DataRow
row["title_id"] = "JP1001";
row[". . .] = ". . .";
// Add new row to the Table
table.Rows.Add (row);
```

```
DataRow[] rows = table.Select ("title_id = 'JP1001'");
DataRow[] rows = table.Select ("price < 10.00");
DataRow[] rows = table.Select ("pubdate >= '#1/1/2000#'");
DataRow[] rows = table.Select ("state = 'tn' and zip like '37*'");
```

DataTables

The chapter shows code for updating and deleting records from a table.

```
SqlDataAdapter adapter = new SqlDataAdapter ("select * from titles",
    "server=localhost;database=pubs;uid=sa;pwd=");

DataSet ds = new DataSet ();
adapter.Fill (ds, "Titles");

DataRow[] rows = table.Select ("ytd_sales > 10000");
foreach (DataRow row in rows)
    row["price"] = (decimal) row["price"] + 10.00m;
```

```
SqlDataAdapter adapter = new SqlDataAdapter
  ("select * from titles",server=localhost;database=pubs;uid=sa;pwd=");

DataSet ds = new DataSet ();
adapter.Fill (ds, "Titles");

DataRow[] rows =
    table.Select ("ytd_sales < 10000 OR isnull (ytd_sales, 0) = 0");
foreach (DataRow row in rows)
    row.Delete ();
```

DataAdapter.Update

Inserts, updates, and deletes performed on a DataTable don't automatically propagate back to the physical database unless **update** is called

```
SqlDataAdapter adapter =
    new SqlDataAdapter ("select * from titles",
    "server=localhost;database=pubs;uid=sa;pwd=");
SqlCommandBuilder builder = new SqlCommandBuilder (adapter);
DataSet ds = new DataSet ();
adapter.Fill (ds, "Titles");

// Insert a record into table
DataTable table = ds.Tables["Titles"];
DataRow row = table.NewRow ();
row["title_id"] = "JP1001";
row["title"] = "Programming Microsoft .NET";
row["price"] = 59.99m;
row["ytd_sales"] = 1000000;
row["type"] = "business";
row["pubdate"] = new DateTime (2002, 5, 1);
table.Rows.Add (row);

// Update the database
adapter.Update (table);
```

CommandBuilder

When an update command is executed, rows added to a DataTable are added to the DB via INSERT commands, rows that were modified are updated into the DB via UPDATE commands, and DELETE commands are issued for rows that were deleted.

Those commands are built dynamically by an SqlCommandBuilder object.

A DataAdapter has four properties that control how it communicates with a database:

1. SelectCommand, captures the command that the DataAdapter uses to perform queries
2. InsertCommand, captures the command that the DataAdapter uses to insert rows
3. UpdateCommand, which . . .
4. DeleteCommand, which . . .

DataViews

A view is a logical table that contains rows from one or more physical tables. ADO.NET views are instances of **System.Data.DataView.** Views in ADO.NET support sorting and filtering and are often used to customize controls through data binding

```
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<html>
  <body>
    <form runat="server">
      <asp:DataGrid ID="MyDataGrid" RunAt="server" />
    </form>
  </body>
</html>
<script language="C#" runat="server">
  void Page_Load (object sender, EventArgs e) {
       SqlDataAdapter adapter = new SqlDataAdapter ("select * from titles",
        "server=localhost;database=pubs;uid=sa;pwd=");
       DataSet ds = new DataSet ();
       adapter.Fill (ds, "Titles");
       DataView view = new DataView (ds.Tables["Titles"]);
       view.Sort = "title ASC";
       MyDataGrid.DataSource = view;
       MyDataGrid.DataBind ();
  }
</script>
```