COMMAND OBJECT

- Command object is the biggest object in ADO.NET
- It is the only object which can perform actions with database
- It can be created directly using Command class or can be created using **Connection.Create** command (factory classes also contain creation of Command).

  Ex: SqlConnection sqlCon = new SqlConnection("......");
    SqlCommand sqlCmd = sqlCon.CreateCommand();
- Commands that we run depend upon the kind of query that we want to execute with database. All databases support two types of queries.

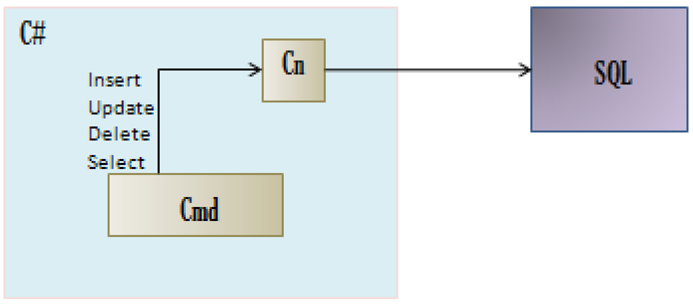
  i. Action Queries

  ii. Non-action Queries

  Action queries are those which change the state of database and which don't return any query results(though they return the number of records affected). Ex: Insert, Delete and Update statements

  Non-action queries are those which don't affect the database but return the results to the user. Ex: Select statement

Method of execution of queries:



Command object provides the following methods to execute queries:

1. ExecuteNonQuery()
2. ExecuteReader()
3. ExecuteScalar()
4. ExecuteXMLReader()

1. **ExecuteNonQuery():** This method is used for executing the queries which perform some action and change the state of the database. This method is used to execute insert, update and delete statements.

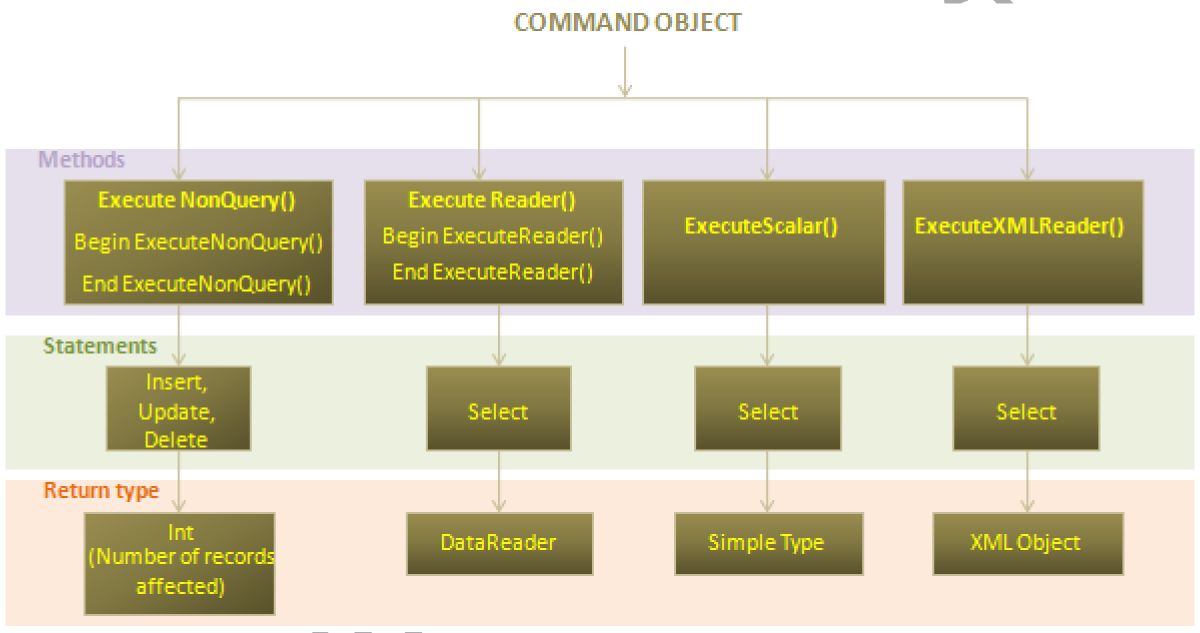
   Ex: Delete * from tmpjobs;
   Insert into jobs values('abc', 100, 120);

2. **ExecuteReader():** This method can run any select statement.

Ex: Select ename from emp; //which returns multiple values
Select count(*) from emp; //which returns a single value
Select E.Empno, E.Ename, D.Deptno, D.Dname from emp E, Dept D where E.Deptno=D.Deptno //which returns values from multiple tables

3. **ExecuteScalar():** This method is used for select statements which return only single values.
   Ex: Select count(*) from emp;

4. **ExecuteXMLReader():** This method is applicable only for SqlClient namespace. It is used for queries which return xml data. But Oledb managed provider does not support this method.
   Ex: Select * from jobs FOR XML AUTO



Note: ExecuteNonQuery(), which is included from ADO.NET Version 1.0/1.1 is used for synchronous execution of queries, whereas BeginExecuteQuery() and EndExecuteQuery() methods which were introduced in ADO.NET Version 2.0 are used for asynchronous execution of queries.

All the methods of Command object can run all the actions. But we have to use this object carefully depending upon our requirements, otherwise performance will go down.

Command Object Demo:

This program demonstrates the use of Command object for a delete statement. In this program when the linklabel is clicked, the corresponding record in the job table of pubs database will be deleted depending on job_id value given in textbox.

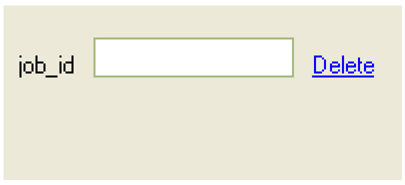
The following connection string is declared in app.config file:

```
<configuration>
  <appSettings>
```

```
    <add key="sqlCnStr" value="data source=mytoy; user id=sa; database=pubs"/>
  </appSettings>
</configuration>
```

Design:

The form is designed with just a textbox and a linklabel as shown below



Code:

```
    public partial class cmdobj : Form
    {
             SqlConnection cn = new SqlConnection(ConfigurationSettings.AppSettings["sqlCnStr"]);
             SqlCommand cmd;
             string stmt;

             public cmdobj()
             {
                 InitializeComponent();
              }

              private void cmdobj_Load(object sender, EventArgs e)
             {
                  cn.Open();

              }

              private void lnklblDelete_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e)
             {
                      stmt = "delete from jobs where job_id=" + txtJobId.Text;
                      cmd = new SqlCommand();
                      cmd.CommandText = stmt;
                      cmd.CommandType = CommandType.Text;
                      cmd.Connection = cn;
                      cmd.CommandTimeout = 20;
                      int i = cmd.ExecuteNonQuery();
                       if (i > 0)
                           MessageBox.Show("Record deleted successfully");
                       else MessageBox.Show("Records not found");
                       MessageBox.Show("No. of records deleted: "+i);

              }

      }
```

When the job id is given in the textbox and delete linklabel is clicked, it should delete the corresponding records from the database. For this purpose ExecuteNonQuery() method of command object is used for executing the delete statement. If the records are successfully deleted it returns the number of records deleted, otherwise returns zero value.

Batch SQL:

We can write more than one SQL statement in a single string and execute them as a group(internally execution takes place one after the other) at the same time. This is called batch SQL.

The statements are separated by a semi-colon and written in the same string.

Ex: “select ename from emp; select * from jobs; select * from dept”

Batch SQL is very effective and saves time because it is not necessary to wait for the execution of first statement, in order to run the next statement. But there is also a great drawback in this approach, which relates to the SQL injection attacks.

SQL Injection attacks:

If even a single statement which is harmful to the database is provided in the batch, it results in stalling of the whole database. This is a great disadvantage of command objects.

For example, take the same demo written earlier. In that program there is a provision for the user to enter the job_id in the given text box.

```
stmt = "delete from jobs where job_id=" + txtJobId.Text;
cmd = new SqlCommand();
cmd.CommandText = stmt;
cmd.CommandType = CommandType.Text;
```

This works fine as long as the user enters the correct job_id. But the drawback here is that the user can include another statement in the same text box after providing the required input. Suppose the job_id of the record to be deleted is 15. So the user is expected to enter 15 in the textbox, but instead if he enters something like this:

15; drop table emp

So the total string becomes:
“delete from jobs where job_id=15; drop table emp”

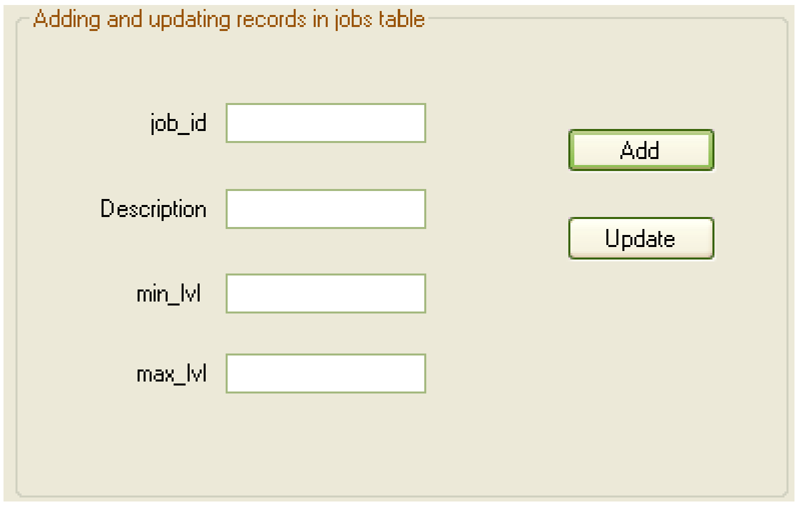
As the command object is capable of performing batch SQL operations, it tries to execute both statements on the database. This results in dropping of the complete emp table which is not desirable.

Some validations, permissions and stored procedures can be used to overcome these problems. Moreover SQL2005 and 2008 also provide some utilities to avoid these scenarios.

Making little changes in the design and code of the application can also help in avoiding SQL injection attacks to some extent.

Demo2: Adding and updating records in a table from front-end

In this program the operations add and update can be performed on the jobs table from the front-end by using the command object. Three different ways of adding the records are shown here. The form design is as follows:



The four text boxes correspond to the four columns in the jobs table. Since job_id is an identity column, there is no need to enter any values for that column while adding records. But while updating records it is necessary to provide job_id value to identify the correct record.

Code:

```
string stat;
SqlConnection cn;
SqlCommand cmdobj;

//to open connection and execute the command statement
public int performAction(string st)
{
   cn = new SqlConnection("data source=mytoy; user id=sa; database=pubs");
   cn.Open();
   cmdobj = new SqlCommand(stat, cn);
   int i = cmdobj.ExecuteNonQuery();
   return i;

}


//To update records
private void btnUpdate_Click(object sender, EventArgs e)
{
   stat = "update jobs set job_desc='" + txtDesc.Text + "', min_lvl=" +  txtMinLvl.Text + ", max_lvl=" +
          txtMaxLvl.Text + " where job_id=" + txtJobId.Text;
   int a = performAction(stat);
   MessageBox.Show("Records updated:" + a.ToString());
}
```

//First Method of adding records

```
private void btnAdd_Click(object sender, EventArgs e)
{
   stat = "insert into jobs values('" + txtDesc.Text + "'," +
             txtMinLvl.Text + "," + txtMaxLvl.Text + ")";

   int a = performAction(stat);
   if (a > 0)
```

```
        MessageBox.Show("Record added sucessfully");
    else
        MessageBox.Show("Unsuccessful operation");
}
```

//Second Method of adding records

```
private void btnAdd_Click(object sender, EventArgs e)
{

    stat=string.Format("Insert into jobs values('{0}',{1},{2})", txtDesc.Text, txtMinLvl.Text, txtMaxLvl.Text);
    int a = performAction(stat);
    if (a > 0)
       MessageBox.Show("Record added sucessfully");
    else
       MessageBox.Show("Unsuccessful operation");

}
```

//Third Method of adding records

```
private void btnAdd_Click(object sender, EventArgs e)
{
    cn = new SqlConnection("data source=mytoy; user id=sa; database=pubs");
    cn.Open();
    stat = "insert into jobs values(@a,@b,@c)";
    cmdobj = new SqlCommand(stat, cn);
    cmdobj.Parameters.AddWithValue("@a", txtDesc.Text);
    cmdobj.Parameters.AddWithValue("@b", txtMinLvl.Text);
    cmdobj.Parameters.AddWithValue("@c", txtMaxLvl.Text);
    int a=cmdobj.ExecuteNonQuery();

    if (a > 0)
       MessageBox.Show("Record added sucessfully");
    else
       MessageBox.Show("Unsuccessful operation");
}
```

In the above form all front-end objectives can be viewed with respect to databasees. A front-end should provide design and that design should be interactive with databases.

Three ways of adding records to the database are shown in this program. In the first one, a command string is provided to the command object with the textbox values directly embedded in it. This string is executed by using ExecuteNonQuery method.

In the second approach also a command string is provided,but the values are inserted into the string as arguments using string.Format method.

In the third approach, command parameters are used to supply the textbox values to the command string which is executed by using the ExecuteNonQuery() method.

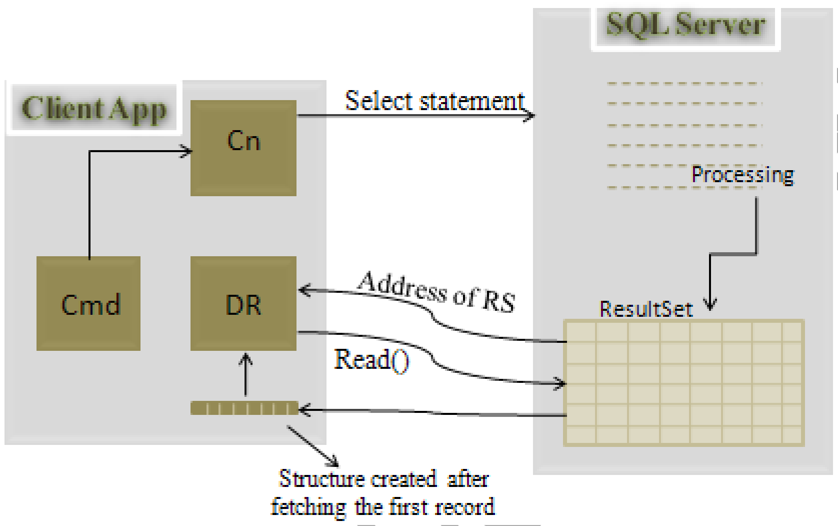
DataReader

In ADO.NET there are only two objects which can store data. One is DataSet while the other is DataReader.

Features of DataReader:

- DataReader is a forward only accessible object
- It is a read-only object
- It provides sequential access for rows and can be extended for sequential column access also.

How DataReader Works:



The select statement from the command object is processed and a ResultSet with the required data is created in the form of rows and columns at the database server. The address of this ResultSet is provided to the DataReader.

cmd=new SqlCommand("select * from emp", cn);

dr=cmd.ExecuteReader(); //provides the address of the ResultSet to the DataReader(dr)

When the read() method of the DataReader is used it returns true if records are present and false if records are not present. At the same time if records are present, a similar structure like the ResultSet is created at the client side and the first record is fetched into it. The DataReader reads the record from this structure.

dr[n] is an indexer to access its created structure.

Ex: dr[0] provides the first column of the retrieved row

dr[1] provides the second column of the retrieved row

A pointer exists on ResultSet to identify the record being read. When the read() method is again called from the DataReader, the pointer moves to the next record if present. The structure at the DataReader now gets overwritten by this record.

Drawbacks:

- It is a forward-only and read-only object
- DataReader is a connection-oriented object, which means that the access to data is possible only as long as the connection exists.
- DataReader works only on databases, but not on other data sources like XML files.

Note: It is always necessary to close the DataReader object after the required data is read.

Inspite of all drawbacks, DataReader usage is in its performance. To access data, DataReader is the fastest object in entire .NET(not only in ADO.NET).

DataReader capabilities are enhanced in .NET 2.0. The new features are:

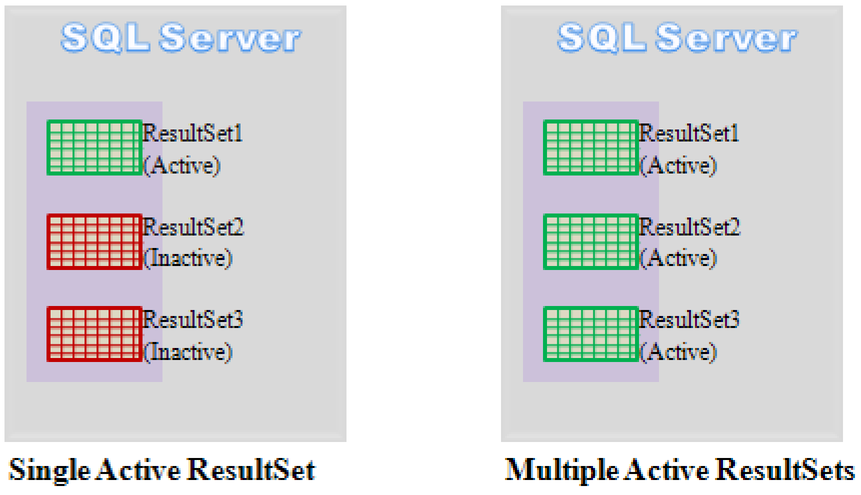
- It can communicate with other objects like DataSet

  DataReader →DataSet(DataTable)

  DataTable →DataReader

- The biggest enhancement is MARS(Multiple Active ResultSets)
  (In earlier versions it was SAR(Single Active ResultSet)

Multiple Active ResultSets(MARS):



Multiple ResultSets are created when Batch SQL is provided. i.e. for each individual statement in the batch, a different ResultSet is created. But normally only one ResultSet is active at a time. If the next ResultSet has to be active, then the previous one should become inactive. This is called Single Active ResultSet eventhough multiple ResultSets are present. But with MARS, more than one ResultSet can be active at a time. This property can be specified in the connection string by giving true or false .

SqlConnection cn=new SqlConnection("data source=...... ; MultipleActiveResultSets=true; ...");

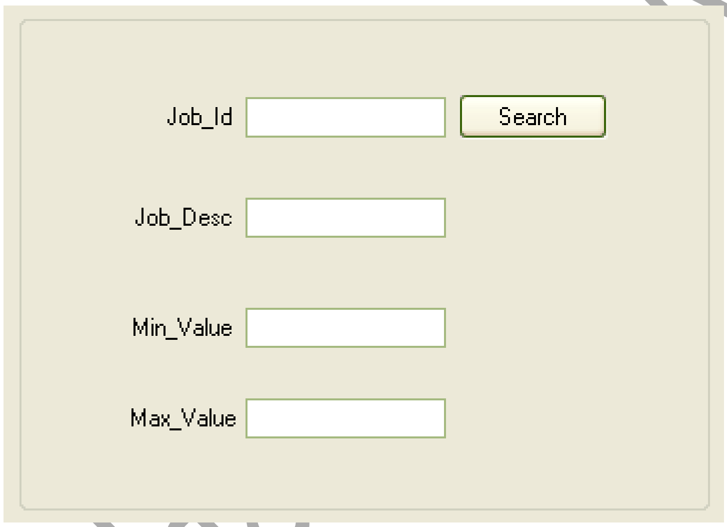
SingleActiveResultSet is the default and so we need to specify MARS explicitly in connection string.

Using MARS eliminates the need to open and close connection for each ResultSet and hence improves the performance. But this is not the case at all times. The performance is improved only when multiple commands have to be executed using a single connection. On the other hand it is not worth using for very few commands and it even results in some overhead in certain situations. So MARS should be used only when it is really needed.

Note:
We can use ExecuteReader() method for action queries also(insert, update, delete). When there is a combination of select and any action query in Batch SQL , we have to use ExecuteReader, which gets address of the ResultSet of select query and also does the action of the action query.

DataReader Demo:

This program shows how a DataReader can be used to read the data fetched from the database. A job_id is provided in the relevant textbox and searched for the record with that job_id. If matching record is present in the database, it is fetched into the datareader, which reads the column values and fills the textboxes provided for the purpose.



```
SqlConnection cn = new SqlConnection("data source=mytoy; user id=sa;
                        database=pubs");
SqlCommand cmd;
SqlDataReader dr;

private void btnSearch_Click(object sender, EventArgs e)
{
    string stat = "select * from jobs where job_id=" + txtJobId.Text;
    if (cn.State.ToString() == "Closed")
      cn.Open();
    cmd = new SqlCommand(stat, cn);
    dr = cmd.ExecuteReader();
    if (dr.Read())
    {
        txtDesc.Text = dr[1].ToString();
        txtMinVal.Text = dr[2].ToString();
        txtMaxVal.Text = dr[3].ToString();
        dr.Close();

      }
      else
        MessageBox.Show("Record not found");

  }
```

More DataReader Methods and Properties:

1.dr.GetString(0): This statement is used to retrieve the data in the required format. Here the requirement is to retrieve the first column value of the DataReader in the form of a string. Normally for this purpose, the below method is adopted:

textBox1.Text=dr[0].ToString();

Here the data is first retrieved and then converted into string format. But if dr.GetString(0) is used, the data is directly retrieved in the form of string, which aids in better performance.

2.dr.GetInt32(1): This works similar to the above method, but fetches data in the form of an integer. Similarly there are methods like GetDouble, GetByte,GetDateTime etc. for different datatypes.

3. HasRows Property: It checks whether records are present or not and returns true or false accordingly. This property provides better performance than Read() because Read method checks the presence of the records and also reads the records. Since HasRows property does not read the records it is faster.

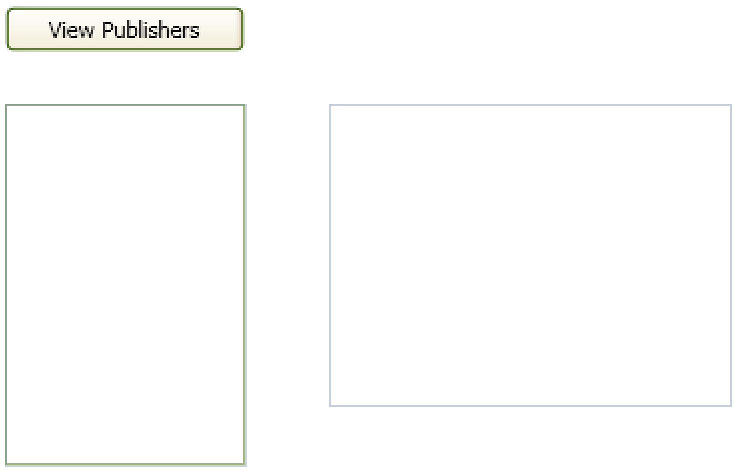
4.IsClosed: It returns true if the DataReader is closed, otherwise returns false.
If cmd.ExecuteReader(CommandBehaviour.CloseConnection) is used then connection gets closed when DataReader is closed. In such cases, the state of the connection can be checked by using IsClosed property.

5.IsDBNull: This property is used to check if a DataRow is null. Returns true if it is null, otherwise returns false.
Ex: if(dr.IsDBNull[0]) checks whether the first row is null or not.

DataReader Demo2:

This wpf program demonstrates the retrieval of columnwise data from database using DataReader. This kind of sequential access is required when there are columns with more volume of data like structures, MS-Word documents, Excel etc.

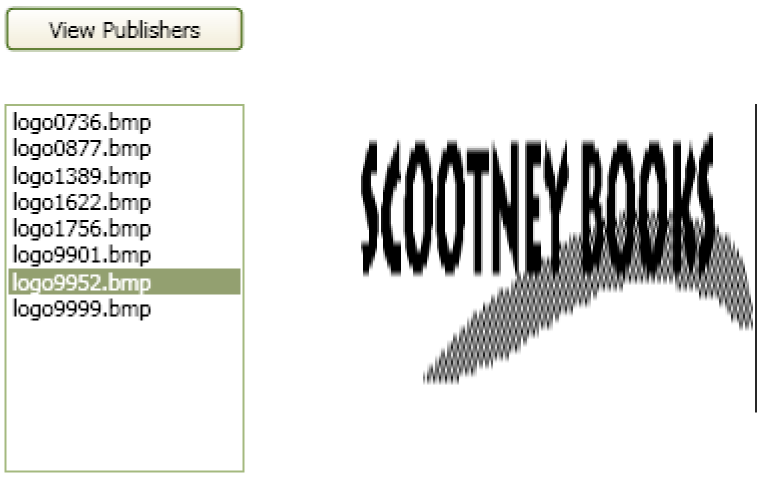
Here a listbox, image and a button are taken in the window. The button code is written so that the image names in the table are shown in the list box. When the required image name is chosen in the listbox, the actual image is shown beside it.



```
SqlConnection cn = new SqlConnection("data source=mytoy; user id=sa;
                       database=pubs");
SqlCommand cmd;
SqlDataReader dr;

private void btnViewPubs_Click(object sender, EventArgs e)
{
    cmd = new SqlCommand("select pub_id, logo from pub_info", cn);
    FileStream stream;
    BinaryWriter writer;
    int bufferSize = 100;
    byte[] outByte = new byte[buffersize];
    long retval;
    long startIndex = 0;
    string pubId = "";
    cn.Open();
    SqlDataReader reader =
                cmd.ExecuteReader(CommandBehavior.SequentialAccess);
    while (reader.Read())
    {
        pubId = reader.GetString(0);
        string fname = "logo" + pubId + ".bmp";
        stream = new FileStream(fname, FileMode.OpenOrCreate,
                                FileAccess.Write);
        writer = new BinaryWriter(stream);
        startIndex = 0;
        retval = reader.GetBytes(1, startIndex, outByte, 0, bufferSize);
        while (retval == bufferSize)
        {
            writer.Write(outByte);
            writer.Flush();
            startIndex += bufferSize;
            retval = reader.GetBytes(1,startIndex,outByte,0,bufferSize);

        }
        writer.Write(outByte, 0, (int)retval - 1);
        writer.Flush();
        writer.Close();
        stream.Close();
        listBox1.Items.Add(fname);
    }
    reader.Close();
    cn.Close();
}

private void listBox1_SelectionChanged(object sender,
                                    SelectionChangedEventArgs e)
{
    BitmapImage bi = new BitmapImage();
    bi.BeginInit();
    bi.UriSource=new
Uri(Environment.CurrentDirectory+@"\"+listBox1.SelectedItem.ToString());
    bi.EndInit();
    image1.Source=bi;
 }
```

The namespace System.IO is needed for file stream operations. Bitmap images are accessed sequentially by using the ExecuteReader method of the command object. Finally the filenames from the table are displayed in the listbox. When image name is selected in the listbox, it displays the image in the image placeholder.

The result will be as shown in this screen shot:



Stored Procedures:

Different methods of writing programs in back-end are:

- Blocks (T-SQL for SQL Server, PL/SQL for Oracle) - Blocks compile everytime when called. They are not stored
- Procedures - Procedures get compiled and stored as part of database objects. So they are faster in execution. They need to be explicitly called using an execute statement
- Functions - They also get compiled and stored like procedures. They can be called with a select statement.
- Triggers – They are executed automatically(implicitly called)

Why use Stored Procedures?

Stored Procedures are used because of the following features they offer:

- ✓ Modular programming
- ✓ Distribution of work
- ✓ Database security
- ✓ Faster execution
- ✓ Network traffic reduction
- ✓ Flexibility

ADO.NET uses Command object to invoke procedures. SQL 2005/.NET 2.0 provided a new feature of writing procedures in C#.NET and storing them in SQL Server. This option is provided as a separate project option. From Visual Studio we can open SQL Server by following the below path:

File→New Project → Database →SQL Server

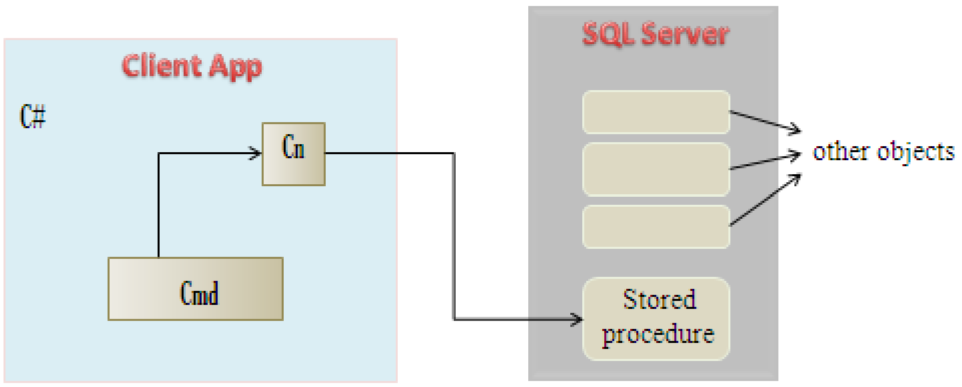
Apart from this, VS.NET provides options like Server Explorer, T-SQL Editor where we can write, compile, execute and store procedures at back-end.

Procedures:

A typical procedure looks like the following one:

```
CREATE PROCEDURE dbo.AddJob
(
        @jobdesc varchar(50),
        @minlvl int,
        @maxlvl int
)

AS
        Insert into jobs values(@jobdesc, @minlvl, @maxlvl)
RETURN
```

Every parameter in a procedure should be preceded by a special character @ to differentiate from the column names of the table. By default all the declared parameters are input parameters. If the procedure has to return values, then the output parameters must be explicitly specified by using OUTPUT keyword for SQL Server and OUT for Oracle.



Calling procedures from client-side:
This is the code for adding records into the jobs table. The three column values are entered into the text boxes(no need to add values to the first column because it is an identity column) and at the click of a button the whole record should be entered into the table. This is done by using the stored procedure shown earlier. To call that procedure, the button click code should be written as follows:

```
cmd = new SqlCommand("AddJob", cn);
cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.AddWithValue("@jobdesc", txtJobDesc.Text);
cmd.Parameters.AddWithValue("@minlvl", txtMinLvl.Text);
cmd.Parameters.AddWithValue("@maxlvl", txtMaxLvl.Text);
cmd.ExecuteNonQuery();
MessageBox.Show("Record Added");
```

Here the textbox values are assigned to the parameters of the procedure by using Parameters.AddWithValue() method using command object. When the command is executed, it executes the procedure at the database server and records are added into the table.

Note: If OleDb is used then the parameters should be in same order as the columns in the table, but for SQL Server this is not required.

The command types that are supported by ADO.NET are :
i. Text: This is default. It can be any query including the table name.
ii. StoredProcedure: To call a procedure at the back-end
iii. TableDirect: This is mainly for OleDb, where the table name has to be specified in place of command text.

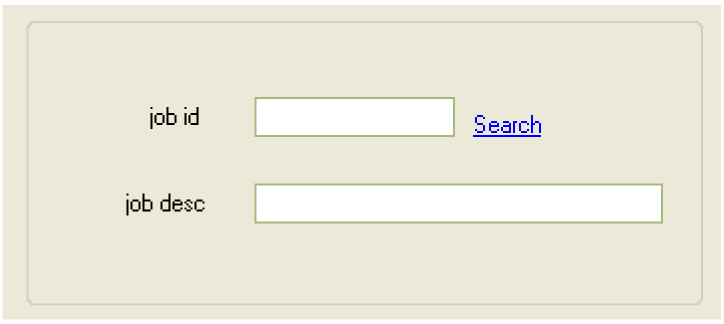
Procedures can be written to return query results also(i.e. select statement results). To execute the procedures that return query results, ExecuteReader should be used from the command object instead of ExecuteNonQuery as shown below:

```
cmd = new SqlCommand("jobsinfo", cn);
cmd.CommandType = CommandType.StoredProcedure;
SqlDataReader dr = cmd.ExecuteReader();
```

But it is not preferred to write such procedures because all databases do not support them. But for returning values from the procedures, nearly all back-ends support directional parameters-IN, OUT, INOUT.
i. IN Parameter: It is read-only in procedure i.e.the procedure can only read the value from it and but cannot return value to it. The direction of this parameter is from the calling program (client) to called program(server).
ii. OUT Parameter: It is write-only which means that the procedure can only return values to it, but cannot read any values from it. Direction is from server to client
iii. INOUT Parameter: It can be used for both Read and Write. It is bi-directional i.e. from server to client as well as from client to server.

Demo-Stored Procedures:

This program demonstrates the use of OUTPUT direction parameter and also introduces the SQL provider specific parameter class for working with stored procedures.



Here when job id is provided and searched, the corresponding job description should be displayed in the second text box. For this purpose a stored procedure is used which takes job id as the input parameter and then gives its job description as the output parameter.

First the stored procedure is created as follows:

```
CREATE PROCEDURE dbo.GetJobInfo

      (
      @jobId int,
      @jobDesc varchar(50)OUTPUT
      )

AS
    select @jobDesc=job_desc from jobs where job_id=@jobId

RETURN
```

When the Search linklabel is clicked the stored procedure has to be called. So the code for calling this procedure is written in the click event of that link label. The code is as follows:

```
SqlConnection cn = new SqlConnection("data source=mytoy; user id=sa;
                                      database=pubs");
SqlCommand cmd;

private void llblSearch_LinkClicked(object sender,
                                        LinkLabelLinkClickedEventArgs e)
{
    SqlParameter pJobId, pJobDesc;
    cn.Open();
    cmd = new SqlCommand("GetJobInfo", cn);
    cmd.CommandType = CommandType.StoredProcedure;
    pJobId = new SqlParameter("@jobId", SqlDbType.Int);
    pJobId.Value = int.Parse(txtJobId.Text);
    cmd.Parameters.Add(pJobId);
    pJobDesc = new SqlParameter("@jobDesc", SqlDbType.VarChar, 50);
    pJobDesc.Direction = ParameterDirection.Output;
    cmd.Parameters.Add(pJobDesc);
    cmd.ExecuteNonQuery();
    MessageBox.Show("Procedure Executed");
    txtJobDesc.Text = cmd.Parameters["@jobDesc"].Value.ToString();

}
```

In this demo SqlParameter is used, which is another provider specific class used to prepare parameters. The created parameters can be added in command object parameters collection.

In addition to the above, parameters can also be prepared as shown below:

(i)
```
cmd.Parameters.Add("@jobId", SqlDbType.VarChar, 50);
```

(ii)
```
cmd.Parameters.Add(new SqlParameter("@jobId",
                   SqlDbType.VarChar,50));
```

Note: For single valued queries ExecuteScalar() should be used, otherwise it takes the first value by default from the retrieved values. ExecuteScalar() provides better performance because of its single return value.