Microsoft .Net - C# - Customized

Welcome

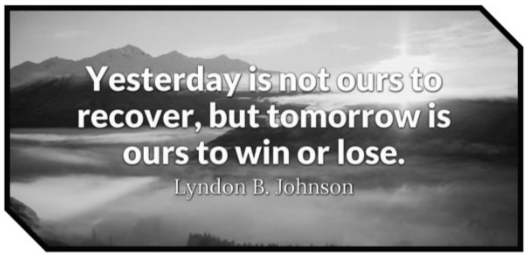
C# .NET (Customized)
Duration – 10 Days

Presented by
Ranjan Bhatnagar

Introduction

- Instructor: Ranjan Bhatnagar
- Microsoft Certified Trainer
- Freelancer Representing
- Client-driven training professional offering 23+ years of progressive experience in training.
- More than 30 corporates & 30000+ Trainees
- Known for outstanding ability to create dynamic, eye-opening, highly interactive, and fun educational experiences for clients that exceed their highest expectations.

❁ ❁ ❁ Thought of the Day ❁ ❁ ❁


Yesterday is not ours to recover, but tomorrow is ours to win or lose.
Lyndon B. Johnson

Problem with Data Access

- At first, there were no common interfaces for accessing data.
- Each data provider exposed an API or other means of accessing its data.
- The developer only had to be familiar with the API of the data provider he or she used. When companies switched to a new database system, any knowledge of how to use the old system became worthless and the developer had to learn a new system from scratch.
- As time went on, more data providers became available and developers were expected to have intimate knowledge of several forms of data access.
- Something needed to be done to standardize the way in which data was retrieved from various sources.

Evolution of ADO.NET

- Native Drivers
  - Available in the form of Win32 Library
  - Specific to a given database
  - Different API for every new database
- ODBC Drivers
  - Available in the form of Win32 Libraries
  - Database independent API
  - Same API implemented differently for every database
- DAO / RDO
  - Object based approach for operating with Database
  - DAO – Data Access Objects were used for MS.Access
  - RDO – Remote Data Access Objects were used for remote databases like Oracle and SQL server.
  - Used only for database backend
  - Cannot be used with Excel, CSV, Email etc.. backend

Evolution of ADO.NET Cont.

- ADO (ActiveX Data Objects)
  - Common Object Model for all databases
  - Uses Ole-Db Providers instead of ODBC
  - It's a compact and efficient Object Model
  - Can be used for all types of backend including databases, excel files, csv files, emails etc...
- ADO.NET
- ADO.NET is a family of technologies that allows .NET developers to interact with data in standard, structured, and primarily disconnected ways. DB Server Managed Provider It is a .Net component implementing certain standard set of interfaces provided by Microsoft, for a specific type of backend.

Presented by
Ranjan Bhatnagar

What is ADO.NET?

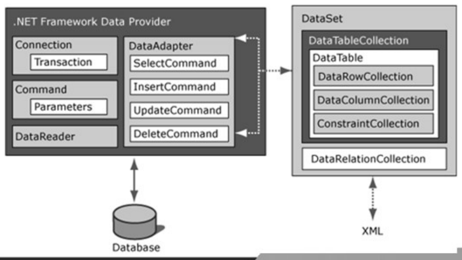
- Applications written using the .NET Framework depend on .NET class libraries, which exist in special DLL files that encapsulate common programming functionality in an easy-to-access format.
- ADO.NET, expressed through the System.Data namespace, implements a small set of libraries that makes consuming and manipulating large amounts of data simple and straightforward.
- ADO.NET provides consistent access to data sources such as SQL Server, XML and other data sources to retrieve, handle, and update the data that they contain.

Major Components of ADO.NET

- The System.Data namespace includes many distinct ADO.NET classes that work together to provide access to tabular data.
- The two main components of ADO.NET for accessing and manipulating data are the .NET Framework data providers and the DataSet.
- The .NET Framework Data Providers are components that have been explicitly designed for data manipulation and fast, forward-only, read-only access to data.
- The ADO.NET DataSet is explicitly designed for data access independent of any data source. As a result, it can be used with multiple and differing data sources, used with XML data, or used to manage data local to the application.

Components of ADO.NET

- The following diagram illustrates the relationship between a .NET Framework data provider and a DataSet.



Managed Providers

- Following are the four managed providers which are build in along with MS.NET Framework.
  - System.Data.SqlClient Manage Provider for SqlServer
  - System.Data.OracleClient- Manage Provider for Oracle
  - System.Data.Oledb -Manage Provider- Can be used for any database which has Oledb provider
  - System.Data.Odbc - Manage Provider for Odbc drivers – This can be used for all databases but it would be slow compared to a managed provider specific to a given database.

.NET Framework Data Providers

- The **Connection** object provides connectivity to a data source.
- The **Command** object enables access to database commands to return data, modify data, run stored procedures, and send or retrieve parameter information.
- The **DataReader** provides a high-performance stream of data from the data source.
- The **DataAdapter** provides the bridge between the DataSet object and the data source.

Core Objects of an ADO.NET Data Provider

Type of Object	Base Class	Relevant Interfaces	Meaning in Life
Connection	DbConnection	IDbConnection	Provides the ability to connect to and disconnect from the data store. Connection objects also provide access to a related transaction object.
Command	DbCommand	IDbCommand	Represents a SQL query or a stored procedure. Command objects also provide access to the provider's data reader object.
DataReader	DbDataReader	IDataReader, IDataRecord	Provides forward-only, read-only access to data using a server-side cursor.

Presented by
Ranjan Bhatnagar

Core Objects of an ADO.NET Data Provider

Type of Object	Base Class	Relevant Interfaces	Meaning in Life
DataAdapter	DbDataAdapter	IDataAdapter, IDbDataAdapter	Transfers DataSets between the caller and the data store. Data adapters contain a connection and a set of four internal command objects used to select, insert, update, and delete information from the data store.
Parameter	DbParameter	IDataParameter, IDbDataParameter	Represents a named parameter within a parameterized query.
Transaction	DbTransaction	IDbTransaction	Encapsulates a database transaction.

DataSet

- **DataSet** can be used with multiple and differing data sources, used with XML data, or used to manage data local to the application.
- The DataSet contains a collection of one or more **DataTable** objects consisting of **DataRows**, **DataColumns**, **Constraints**, and also **Relation** information about the data in the DataTable objects.
- The **DataAdapter** uses Command objects to execute SQL commands at the data source to both load the DataSet with data and reconcile changes that were made to the data in the DataSet back to the data source.

Choosing a DataReader or a DataSet

- When you decide whether your application should use a DataReader or a DataSet consider the type of functionality that your application requires.
- Use a DataSet to do the following:
  - Cache data locally in your application so that you can manipulate it. If you only need to read the results of a query, the DataReader is the better choice.
  - Remote data between tiers or from an XML Web service.
  - Interact with data dynamically such as binding to a Windows Forms control or combining and relating data from multiple sources.
  - Perform extensive processing on data without requiring an open connection to the data source, which frees the connection to be used by other clients.

DataReader for Performance

- If you do not require the functionality provided by the **DataSet**, you can improve the performance of your application by using the **DataReader** to return your data in a forward-only, read-only manner.
- Although the **DataAdapter** uses the **DataReader** to fill the contents of a **DataSet**.
- By using the **DataReader**, you can boost performance because you will save memory that would be consumed by the **DataSet**, and avoid the processing that is required to create and fill the contents of the **DataSet**.

Core Members of the System.Data Namespace

Type	Meaning
Constraint	Represents a constraint for a given DataColumn object.
DataColumn	Represents a single column within a DataTable object.
DataRelation	Represents a parent/child relationship between two DataTable objects.
DataRow	Represents a single row within a DataTable object.
DataSet	Represents an in-memory cache of data consisting of any number of interrelated DataTable objects.
DataTable	Represents a tabular block of in-memory data.
DataTableReader	Allows you to treat a DataTable as a fire-hose cursor (forward only, readonly data access).
DataView	Represents a customized view of a DataTable for sorting, filtering, searching, editing, and navigation.

Core Members of the System.Data Namespace

Type	Meaning
IDataAdapter	Defines the core behavior of a data adapter object.
IDataParameter	Defines the core behavior of a parameter object.
IDataReader	Defines the core behavior of a data reader object.
IDbCommand	Defines the core behavior of a command object.
IDbDataAdapter	Extends IDataAdapter to provide additional functionality of a data adapter object.
IDbTransaction	Defines the core behavior of a transaction object.

System.Data contains types that represent various database primitives as well as the common interfaces implemented by data provider objects. In addition to all above, number of database-centric exceptions e.g. NoNullAllowedException, RowNotInTableException, and MissingPrimaryKeyException are also defined under the namespace.

The Role of the IDbConnection Interface

- The IDbConnection type is implemented by a data provider's connection object.
- This interface defines a set of members used to configure a connection to a specific data store. It also allows you to obtain the data provider's transaction object.
- Here is the formal definition of IDbConnection:

```
public interface IDbConnection : IDisposable
{
    string ConnectionString { get; set; }
    int ConnectionTimeout { get; }
    string Database { get; }
    ConnectionState State { get; }
    IDbTransaction BeginTransaction();
    IDbTransaction BeginTransaction(IsolationLevel il);
    void ChangeDatabase(string databaseName);
    void Close();
    IDbCommand CreateCommand();
    void Open();
}
```

The Role of the IDbCommand Interface

- The IDbCommand interface, which will be implemented by a data provider's command object.
- Like other data access object models, command objects allow programmatic manipulation of SQL statements, stored procedures, and parameterized queries.
- Command objects also provide access to the data provider's data reader type through the overloaded ExecuteReader() method.

Formal Declaration of IDbCommand

```
public interface IDbCommand : IDisposable
{
    string CommandText { get; set; }
    int CommandTimeout { get; set; }
    CommandType CommandType { get; set; }
    IDbConnection Connection { get; set; }
    IDataParameterCollection Parameters { get; }
    IDbTransaction Transaction { get; set; }
    UpdateRowSource UpdatedRowSource { get; set; }
    void Cancel();
    IDbDataParameter CreateParameter();
    int ExecuteNonQuery();
    IDataReader ExecuteReader();
    IDataReader ExecuteReader(CommandBehavior behavior);
    object ExecuteScalar();
    void Prepare();
}
```

The Role of the IDbTransaction Interface

- The overloaded BeginTransaction() method defined by IDbConnection provides access to the provider's transaction object.
- You can use the members defined by IDbTransaction to interact programmatically with a transactional session and the underlying data store:

```
public interface IDbTransaction : IDisposable
{
    IDbConnection Connection { get; }
    IsolationLevel IsolationLevel { get; }
    void Commit();
    void Rollback();
}
```

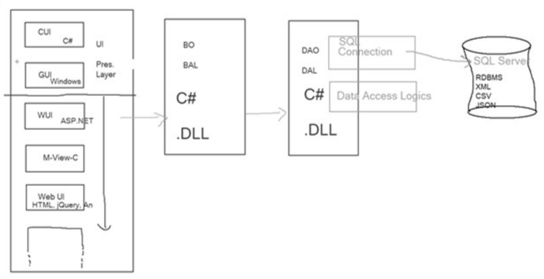
IDbDataParameter & IDataParameter

```
public interface IDbDataParameter
                 : IDataParameter
{
    byte Precision { get; set; }
    byte Scale { get; set; }
    int Size { get; set; }
}
```

- The Parameters property of IDbCommand returns a strongly typed collection that implements IDataParameterCollection.
- This interface provides access to a set of IDbDataParameter-compliant class types.
- IDbDataParameter extends the IDataParameter interface to obtain the following additional behaviors:

```
public interface IDataParameter
{
    DbType DbType { get; set; }
    ParameterDirection Direction { get; set; }
    bool IsNullable { get; }
    string ParameterName { get; set; }
    string SourceColumn { get; set; }
    DataRowVersion SourceVersion { get; set; }
    object Value { get; set; }
}
```

N-Tier Architecture



Presented by
Ranjan Bhatnagar

Let's Do Some Practical Work

- Create a Database using SQL Server.
- You can use SQL Server Management Studio or Visual Studio Server Explorer.
- Give Server Name as .\SqlExpress or . to access local instance of SQL Server.
- You can use Server Name suggested by Database Admin also.
- Give New Database name as MSNETDB
- Create the following table in the database
  - Emp (EmpId, EmpName, EmpSalary)

Creating Table

- Add Columns
  - EmpId – Int, PrimaryKey and also set Identity True in Property of EmpId field
  - EmpName – Varchar(50) - AllowNull = False (Uncheck)
  - EmpSalary – Money – AllowNull = True (Check)
- Right Click on EMP table and select Show Table Data.
- Enter some data rows in table.

Configuring VST Application

- Launch Visual Studio
- Create Windows Forms Application
- Add to the project a file by name App.config (Application Configuration File)

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <add
      name="csMSNETDB"
      connectionString="Server=.; database=MSNETDB; Integrated
Security=True"/>
  </connectionStrings>
</configuration>
```

Connection String

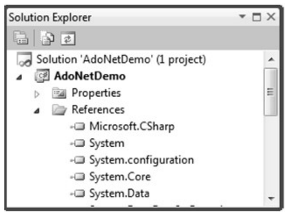
- A connection string contains initialization information that is passed as a parameter from a data provider to a data source.
- The syntax depends on the data provider, and the connection string is parsed during the attempt to open a connection.
- The format of a connection string is a semicolon-delimited list of key/value parameter pairs:
  - keyword1=value; keyword2=value;
- Keywords are not case sensitive, spaces between key/value pairs are ignored. However, values may be case sensitive, depending on the data source.

Few Example

- Various connection string formats for SQL Server
  - "Data Source=.\sqlexpress; Initial Catalog = MSNETDemoDb; Integrated Security=True"
  - "Data Source=.\sqlexpress; Database = MSNETDemoDb; User Id=sa; Password=sa"
  - "Server=.\sqlexpress; Database = MSNETDemoDb; uid=sa; pwd=sa"
- Mostly, Data Source or Server are same, Initial Catalog or Database are same, User Id or uid are same, Password or pwd are same
- We should either give "Integrated Security" or uid / pwd
- You can find many connection strings from a website http://www.connectionstrings.com for all types of databases.

Adding Reference

- Add Reference to System.Configuration
- Project → Reference → .NET Tab → Select System.Configuration → Ok



Helper Class

- Add to the project a class by name Helper (Helper.cs)

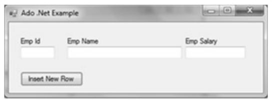
```
class Helper
{
    public static string ConnectionString
    {
        get
        {
            return System.Configuration.
                   ConfigurationManager.
                   ConnectionStrings["csMSNETDB"].
                   ConnectionString;
        }
    }
}
```

Seven Steps

- Steps to execute any SQL Statement
  1. Create a Connection Object
  2. Create a Command Object
  3. Bind the Command to Connection
  4. Initialize Command with SQL Statement
  5. Open the Connection
  6. Execute the Command
  7. Close the Connection

Design a Form

- Place three Labels
  - lblEmpId
  - lblEmpName
  - lblEmpSalary
- Place three TextBoxes
  - txtEmpId
  - txtEmpName
  - txtEmpSalary
- Place a Buttons on form
  - btnInsert

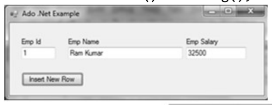


Seven Step Implementation

```
private void btnInsert_Click(object sender, EventArgs e)
{
   //1. Create a Connection
   SqlConnection con = new SqlConnection(Helper.ConnectionString);
   //2. Create a Command
   SqlCommand cmd = new SqlCommand();
   //3. Bind the Command to Connection
   cmd.Connection = con;
   //4. Initialize Command with SQL Statement to execute
   cmd.CommandText = "insert into emp(EmpName, EmpSalary)
   values('" + txtEmpName.Text + "'," + txtEmpSalary.Text + ")";
   cmd.CommandType = CommandType.Text;
   //5. Open the Connection
   con.Open();
   //6. Execute the Command
   cmd.ExecuteNonQuery();
   MessageBox.Show("Inserted...");
   //7. Close the Connection
   con.Close();
}
```

Get EmpId after Insert

```
private void btnInsert_Click(object sender, EventArgs e)
{
   SqlConnection con = new SqlConnection(Helper.ConnectionString);
   SqlCommand cmd = new SqlCommand();
   cmd.Connection = con;
   cmd.CommandText = "insert into emp(EmpName, EmpSalary)
   values('" + txtEmpName.Text + "'," + txtEmpSalary.Text + ")";
   cmd.CommandType = CommandType.Text;
   con.Open();
   cmd.ExecuteNonQuery();
   MessageBox.Show("Inserted...");
   //Retrieve last inserted Identity value
   cmd.CommandText = "Select @@Identity";
   txtEmpId.Text = cmd.ExecuteScalar().ToString();
   con.Close();
}
```



Emp Id: 1
Emp Name: Ram Kumar
Emp Salary: 32500

Caution!

- In any SQL Statement the value of every Varchar field must be enclosed in single quotes. But if the value itself has single quote then it should be replaced with two single quotes.
- You must prepare query string as:

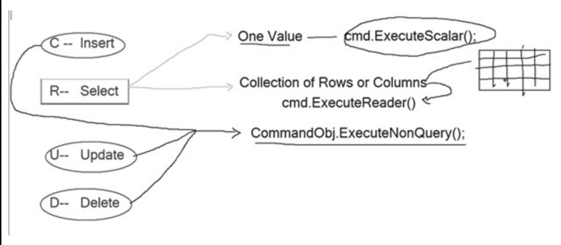
```
string name = txtName.Text.Replace(" ' ", " ' ' ");
decimal salary = decimal.Parse(txtSalary.Text);
cmd.CommandText = "Insert into Emp(EmpName, EmpSalary)
                        Values('" + name + "'," + salary + ")";
```

Presented by
Ranjan Bhatnagar

SQL Command Execution Methods

- int ExecuteNonQuery()
  - Used for execution of those statements which do not return any data from backend to frontend. It returns an integer i.e number of rows affected.
- object ExecuteScalar()
  - Used for execution of statements which return only one value.- returns an Object
- SqlDataReader ExecuteReader()
  - Used for execution of statements which return multiple rows and columns i.e. a set of records. It return SqlDataReader collection.

Three Methods



Update Record

- Add one more button btnUpdate and code
- Record will be updated for entered EmpId

```
private void btnUpdate_Click(object sender, EventArgs e)
{
        SqlConnection con = new
                SqlConnection(Helper.ConnectionString);
        SqlCommand cmd = new SqlCommand();
        cmd.Connection = con;
        cmd.CommandText = "Update Emp set EmpName='" +
                txtEmpName.Text + "', EmpSalary=" +
                txtEmpSalary.Text + " where EmpId=" + txtEmpId.Text;
        MessageBox.Show(cmd.CommandText);
        cmd.CommandType = CommandType.Text;
        con.Open();
        cmd.ExecuteNonQuery();
        MessageBox.Show("Row Updated...");
        con.Close();
}
```

Delete Record

- Add btnDelete and add code
- Record deleted for entered EmpId

```
private void btnDelete_Click(object sender, EventArgs e)
{
            SqlConnection con = new
                      SqlConnection(Helper.ConnectionString);
            SqlCommand cmd = new SqlCommand();
            cmd.Connection = con;
            cmd.CommandText = "Delete from Emp where EmpId=" +
                            txtEmpId.Text;
            MessageBox.Show(cmd.CommandText);
            cmd.CommandType = CommandType.Text;
            con.Open();
            cmd.ExecuteNonQuery();
            MessageBox.Show("Row Deleted...");
            con.Close();
}
```

SqlDataReader

- A forward-only stream that allows reading rows returned by as result of executing query.
- **bool HasRows**: Gets a value that indicates whether the SqlDataReader contains one or more rows
- **int FieldCount**: Gets the number of columns in the current row
- **bool Read()**: returns true if there are more rows; otherwise false. When the SqlDataReader object is returned (as a result of ExecuteReader()) the cursor is placed before the first record. Therefore, call to Read () is necessary to begin accessing any data.

More on SqlDataReader

- This class defines indexers that returns the value of each column in the result set in the current row.
- **void Close()**: closes datareader.
- **XXX GetXXX(int i):** where XXX is Int16, Int32, Int64, Float, Double, DateTime, Char, Boolean, string. Gets the value of the specified column as XXX. And i denotes column index that is zero-based.

IDataReader and IDataRecord Interfaces

- IDataReader represents the common behaviors supported by a given data reader object.
- When you obtain an IDataReader-compatible type, you can iterate over the result set in a forward-only, read-only manner.

```
public interface IDataReader : IDisposable, IDataRecord
{
    int Depth { get; }
    bool IsClosed { get; }
    int RecordsAffected { get; }
    void Close();
    DataTable GetSchemaTable();
    bool NextResult();
    bool Read();
}
```

IDataRecord- a Partial list

- IDataReader extends IDataRecord, which defines many members that allow you to extract a strongly typed value from the stream, rather than casting the generic System.Object retrieved from the data reader's overloaded indexer method.

```
public interface IDataRecord
{
    int FieldCount { get; }
    object this[string name] { get; }
    object this[int i] { get; }
    bool GetBoolean(int i);
    byte GetByte(int i);
    char GetChar(int i);
    DateTime GetDateTime(int i);
    decimal GetDecimal(int i);
    float GetFloat(int i);
    short GetInt16(int i);
    int GetInt32(int i);
    long GetInt64(int i);
        …
    bool IsDBNull(int i);
}
```

Reading Record

```
private void btnGetAllEmps_Click(object sender, EventArgs e)
{
   SqlConnection con = new SqlConnection(Helper.ConnectionString);
   SqlCommand cmd = new SqlCommand("Select * from Emp", con);
   cmd.CommandType = CommandType.Text;
   con.Open();
   SqlDataReader dr = cmd.ExecuteReader();
   string str = "Emp Id \t EmpName \t    EmpSalary\n";
   while (dr.Read())
   {
      str += dr[0].ToString() + "\t";
      str += dr["EmpName"] + "\t";
      int indSalary = dr.GetOrdinal("EmpSalary");
      if (dr.IsDBNull(indSalary))
         str += "----\n";
      else
         str += dr.GetDecimal(indSalary) + "\n";
   }
   MessageBox.Show(str);
   con.Close();
}
```

Multiple select statements

- Multiple SQL command can be supplied to the Command object by separating the sql statements by a semicolon.
  - Select * from Employee; Select * from Dept
- The DataReader returns the first result set automatically.
- To access next resultset, NextResult() method must be called.

NextResult()

```
private void btnNextResult_Click(object sender, EventArgs e)
{
   SqlConnection con = new SqlConnection(Helper.ConnectionString);
   SqlCommand cmd = new SqlCommand("Select Count(*) from Emp;
         Select EmpId, EmpName, EmpSalary from Emp", con);
   cmd.CommandType = CommandType.Text;   con.Open();
   SqlDataReader dr = cmd.ExecuteReader();  dr.Read();
   MessageBox.Show("Total "+ dr[0].ToString()+ " Records Fetched.");
   dr.NextResult();
   string str = "Emp Id \t EmpName \t    EmpSalary\n";
   while (dr.Read())    {
      str += dr[0].ToString() + "\t";
      int indName = dr.GetOrdinal("EmpName");
      str += dr.GetString(indName) + "\t\t";
      int indSalary = dr.GetOrdinal("EmpSalary");
      if (dr.IsDBNull(indSalary))
         str += "----\n";
      else str += dr.GetDecimal(indSalary) + "\n";
   }
   MessageBox.Show(str);  dr.Close();  con.Close();
}
```

Using statement

- Using statement automatically takes care of closing/disposing the object that is used with it. No explicit close statement required.

```
using (SqlConnection con = new SqlConnection(Helper.ConnectionString))
{
   con.Open();
   using (SqlCommand cmd = new SqlCommand("Select * from Emp", con))
   {
      using (SqlDataReader dr = cmd.ExecuteReader())
      {
         while (dr.Read())
         {
            //do something with the data
         }
      }
   }
}
```

Parameterized Query

- If the query or dml statement that has a condition(where clause) and needs to be executed several times with different values for parameters, then parameterized query is preferred way.
- Parameters properties of **SQLCommand** represents **SqlParameterCollection** object
- SqlParameterCollection methods
  - SqlParameter Add(SqlParameter value)
  - SqlParameter Add(string parameterName, SqlDbType sqlDbType)
  - SqlParameter AddWithValue(string parameterName, Object value)

Example

```
private void btnSqlParameter_Click(object sender, EventArgs e)
{
   using (SqlConnection con = new
                  SqlConnection(Helper.ConnectionString))
   {
      con.Open();
      using (SqlCommand cmd = new
            SqlCommand("Select * from Emp where EmpId=@Id", con))
      {
         cmd.Parameters.Add("@id", SqlDbType.Int);
         cmd.Parameters["@id"].Value = txtEmpId.Text;
         using (SqlDataReader dr = cmd.ExecuteReader())
         {
            dr.Read();
            MessageBox.Show("Name: "+dr[1].ToString()
               +"\nSalary: "+ dr[2].ToString());
         }
      }
   }
}
```

Prepared Statement

- Sql Statement on its first execution is compiled, optimized and an execution plan is created for it.
- If the statement is marked as prepared then for further usage this execution plan is cached by the backend, so that the same plan can be reused for the subsequent request for the same statement but different data.
- The prepared statement must be framed using the paramenters. But every parametized statement need not be prepared.
- Command.Prepare() :

Using Prepare()

```
class Emp
{
        public string Name;
        public decimal Salary;
        public Emp(string name, decimal salary)
        {
                this.Name = name;
                this.Salary = salary;
        }
}
private void btnInsPrepared_Click(object sender, EventArgs e)
{
        List<Emp> lstEmps = new List<Emp>();
        lstEmps.Add(new Emp("A1", 10000));
        lstEmps.Add(new Emp("A2", 20000));
        lstEmps.Add(new Emp("A3", 30000));
        lstEmps.Add(new Emp("A4", 40000));
        lstEmps.Add(new Emp("A5", 50000));
```

Using Prepare() Cont.

```
   SqlConnection con = new SqlConnection(Helper.ConnectionString);
   SqlCommand cmd = new SqlCommand();
   cmd.Connection = con;
   cmd.CommandText = "insert into Emp(EmpName,EmpSalary)
           Values(@Name,@Salary)";
   SqlParameter parName =
           new SqlParameter("@Name", SqlDbType.VarChar, 50);
   SqlParameter parSalary =
           new SqlParameter("@Salary", SqlDbType.Money);
   cmd.Parameters.Add(parName); cmd.Parameters.Add(parSalary);
   con.Open(); cmd.Prepare();
   foreach (Emp emp in lstEmps)
   {
      parName.Value = emp.Name;
      parSalary.Value = emp.Salary;
      cmd.ExecuteNonQuery();
   }
   con.Close();
}
```

Cautions!!

- In Sql Server the parameters must begin with @, In Oracle just the name is sufficient but for OleDb and Odbc Providers we need to use "?" as place holder for all the parameters and while adding the parameters a proper order must be maintained.
- If the SQL Statement has to be executed only once, don't mark is Prepared because the backend takes extra time for preparation of execution plan and caching it and this would become overhead if the plan is not reused.
- If the parameter is of type varchar or string and if the value of it has single quote, don't replace that single quote with two single quotes (unlike in a simple statement with data directly embed into the statement).

Review

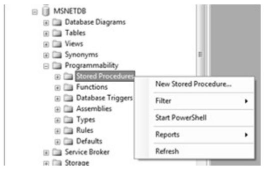
1. Create a Connection object and set its ConnectionString Property
2. Create a Command object and set its Connection, CommandType and CommandText Properties.
3. The CommandText must include place holders for the parameters
4. For every placeholder in the statement, create a Parameter Object.
5. Add all the parameters to the Parameters collection of command object.
6. Open the Connection
7. Prepare the execution plan using Prepare() method of command object.
8. Set the values for parameters and Execute the Command
9. Repeat step 8 for different values of parameters
10. Close the connection.

Using Stored Procedures

- It is a precompiled set of SQL statement which are compiled in native form and stored in the backend.
- Advantages:
  - They are very fast in execution because they are precompiled and stored in backend in native form of that backend.
  - Reduces network traffic because they are executed in backend the data used by them is also in backend.
  - Its easy to update logic/code in them because its stored only at one place i.e in database.

Creating Stored Procedure

- User SQL Server
- Create a Stored Procedure



```
CREATE PROCEDURE GetSalary(@Id int,@Sal money output )
AS
Begin
Select @Sal=EmpSalary from Emp where EmpId=@Id
End
```

Get Salary Using Stored Procedure

```
private void btnGetSalarySP_Click(object sender, EventArgs e)
{
   SqlConnection con = new SqlConnection(Helper.ConnectionString);
   SqlCommand cmd = new SqlCommand();
   cmd.Connection = con;
   cmd.CommandText = "GetSalary";
   cmd.CommandType = CommandType.StoredProcedure;
   SqlParameter parId, parSalary;
   parId = new SqlParameter("@Id", SqlDbType.Int);
   parSalary = new SqlParameter("@Sal", SqlDbType.Money);
   parSalary.Direction = ParameterDirection.Output;
   cmd.Parameters.Add(parId);
   cmd.Parameters.Add(parSalary);
   parId.Value = int.Parse(txtEmpId.Text);
   con.Open(); cmd.ExecuteNonQuery(); con.Close();
   if (parSalary.Value == DBNull.Value)
      txtEmpSalary.Text = "";
   else
      txtEmpSalary.Text = parSalary.Value.ToString();
}
```

Insert with Stored Procedure

```
private void btnInsertSP_Click(object sender, EventArgs e)
{
        SqlConnection con = new
                        SqlConnection(Helper.ConnectionString);
        SqlCommand cmd = new SqlCommand("InsertEmp", con);
        cmd.CommandType = CommandType.StoredProcedure;
        SqlParameter parId, parName, parSalary;
        parId = cmd.Parameters.Add("@Id", SqlDbType.Int);
        parId.Direction = ParameterDirection.Output;
        parName = cmd.Parameters.Add("@name", SqlDbType.VarChar, 50);
        parSalary = cmd.Parameters.Add("@salary", SqlDbType.Money);
        parName.Value = txtEmpName.Text;
        parSalary.Value = txtEmpSalary.Text;
        con.Open();
        cmd.ExecuteNonQuery();
        con.Close();
        txtEmpId.Text = parId.Value.ToString();
}
```

Stored Procedure - Multiple Result Sets

```
private void btnGetEmpCountandList_Click(object sender, EventArgs e)
{
        SqlConnection con = new SqlConnection();
        con.ConnectionString = Helper.ConnectionString;
        SqlCommand cmd = new SqlCommand("GetEmployees", con);
        cmd.CommandType = CommandType.StoredProcedure;
        con.Open(); SqlDataReader dr = cmd.ExecuteReader();
        dr.Read(); MessageBox.Show(dr[0].ToString());
        dr.NextResult(); string str = "";
        while (dr.Read())
        {
                str += dr[0].ToString() + "\t";
                int indName = dr.GetOrdinal("EmpName");
                str += dr.GetString(indName) + "\t";
                int indSalary = dr.GetOrdinal("EmpSalary");
                if (dr.IsDBNull(indSalary))
                        str += "----\n";
                else
                        str += dr.GetDecimal(indSalary) + "\n";
        }
        MessageBox.Show(str); dr.Close(); con.Close();
}
```