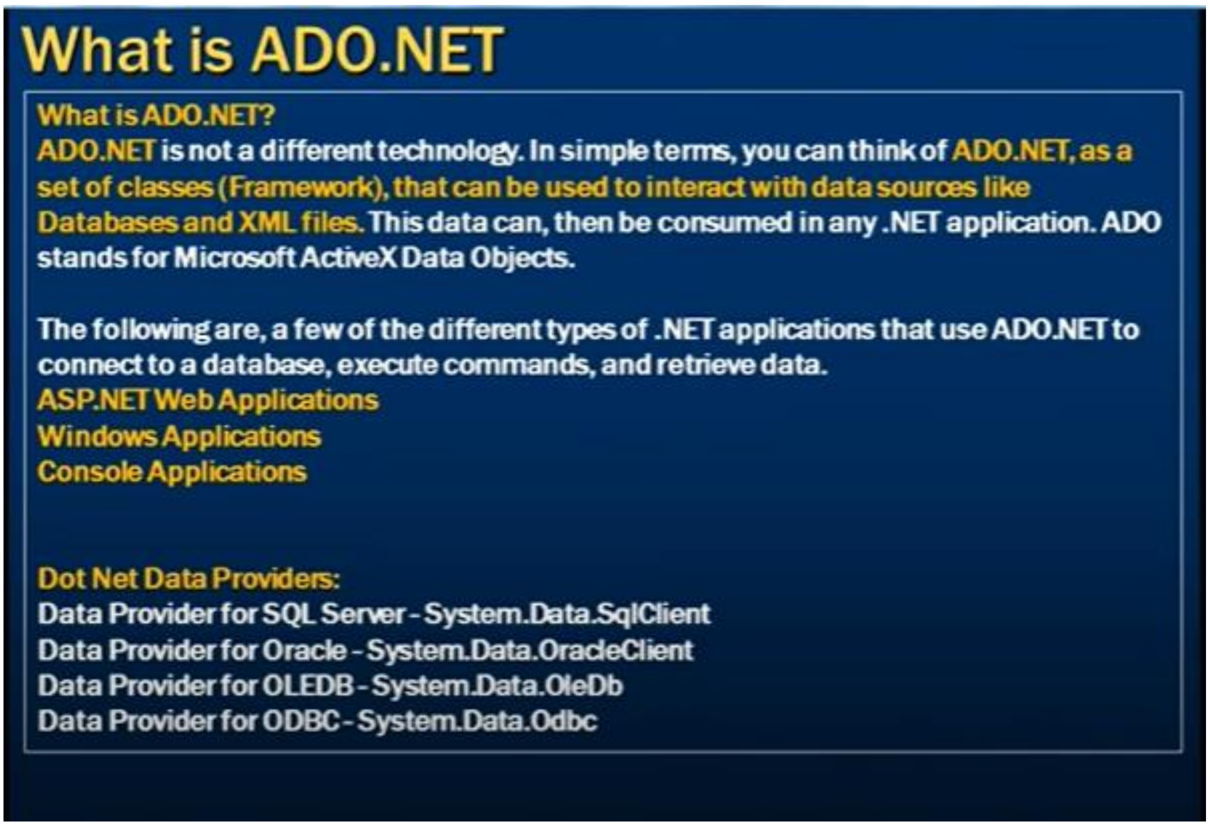
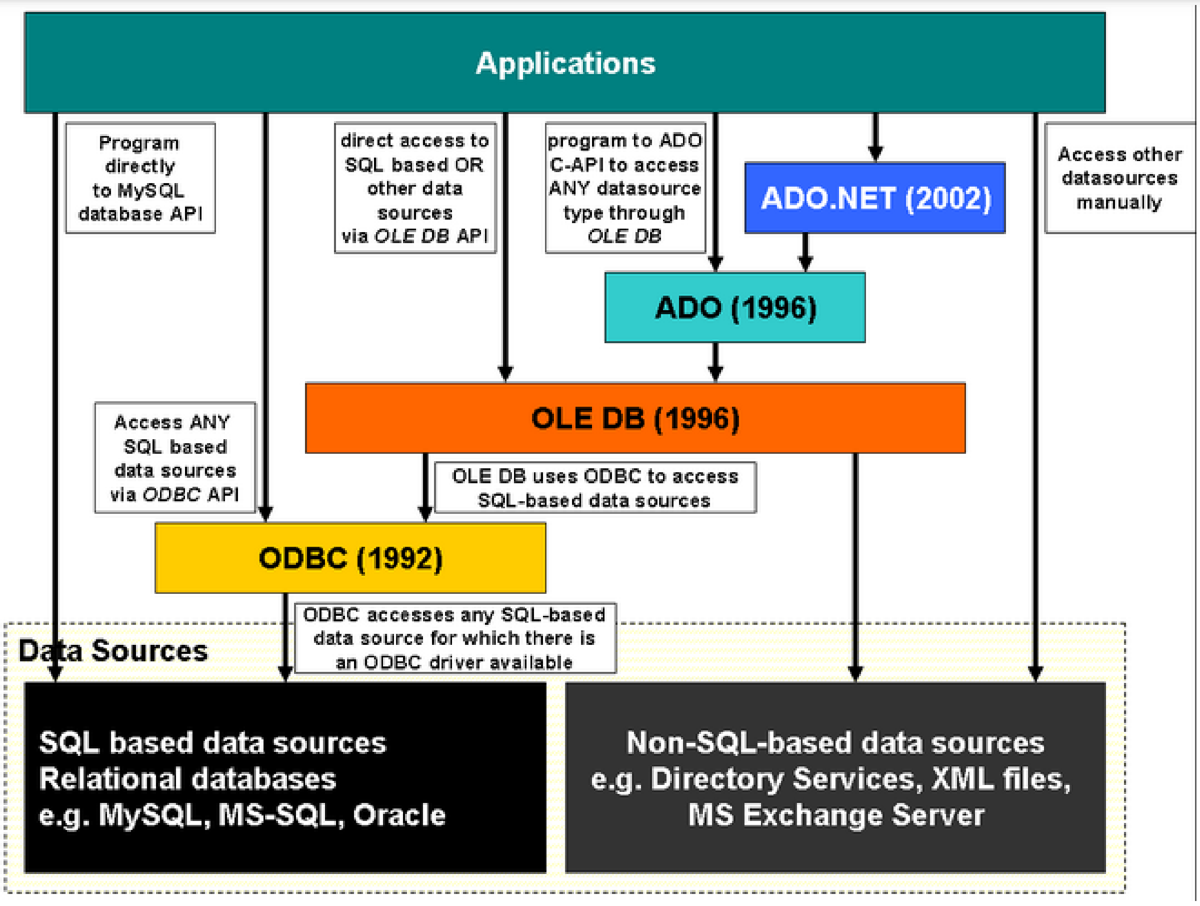


What is ADO.NET

What is ADO.NET?
ADO.NET is not a different technology. In simple terms, you can think of ADO.NET, as a set of classes (Framework), that can be used to interact with data sources like Databases and XML files. This data can, then be consumed in any .NET application. ADO stands for Microsoft ActiveX Data Objects.

The following are, a few of the different types of .NET applications that use ADO.NET to connect to a database, execute commands, and retrieve data.
ASP.NET Web Applications
Windows Applications
Console Applications

Dot Net Data Providers:
Data Provider for SQL Server - System.Data.SqlClient
Data Provider for Oracle - System.Data.OracleClient
Data Provider for OLEDB - System.Data.OleDb
Data Provider for ODBC - System.Data.Odbc

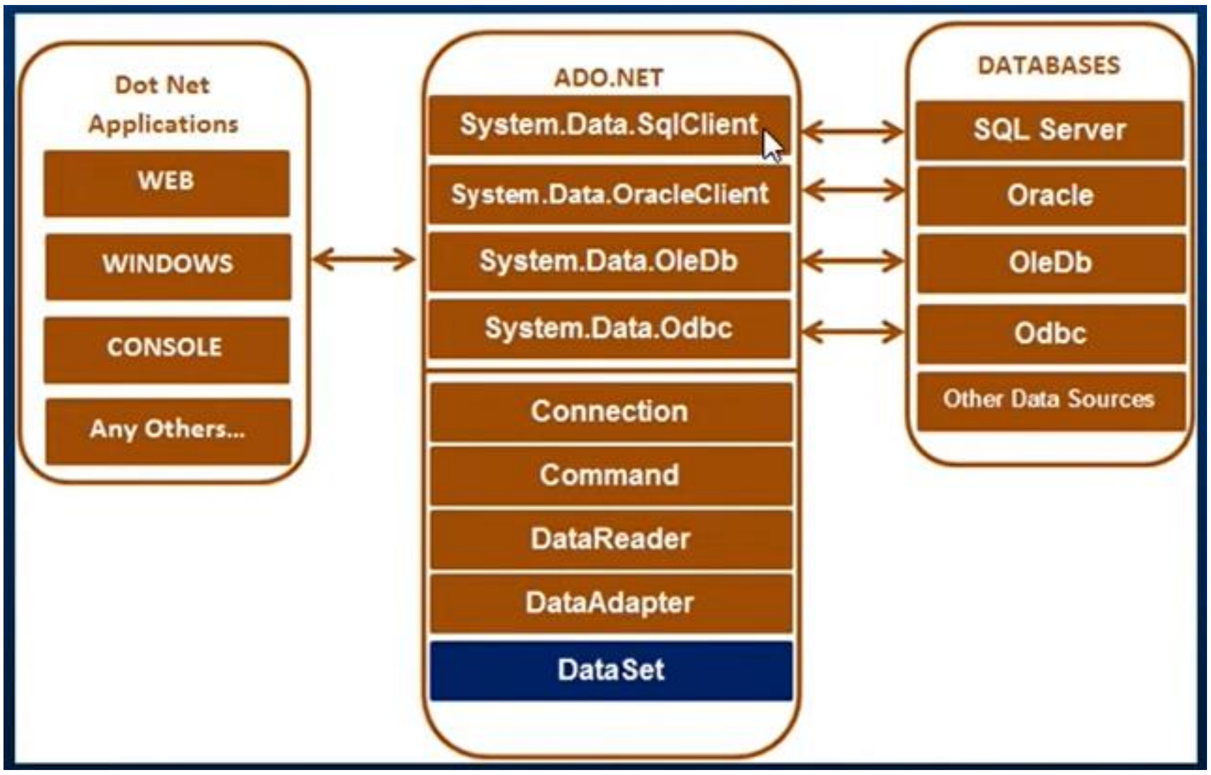

Applications
Program directly to MySQL database API
direct access to SQL based OR other data sources via OLE DB API
program to ADO C-API to access ANY datasource type through OLE DB
ADO.NET (2002)
Access other datasources manually
ADO (1996)
OLE DB (1996)
Access ANY SQL based data sources via ODBC API
OLE DB uses ODBC to access SQL-based data sources
ODBC (1992)
ODBC accesses any SQL-based data source for which there is an ODBC driver available
Data Sources
SQL based data sources Relational databases e.g. MySQL, MS-SQL, Oracle
Non-SQL-based data sources e.g. Directory Services, XML files, MS Exchange Server

Data Providers in ADO.Net

1. SQL Data Provider (System.Data.SqlClient)
2. Oracle Data Provider (System.Data.OracleClient)
3. OleDB Data Provider (System.Data.OleDb) = Mainly Used to access MS Excel, MS Access
4. Entity Client Data Provider (System.Data.EntityClient)
5. ODBC Data Provider (System.Data.Odbc)

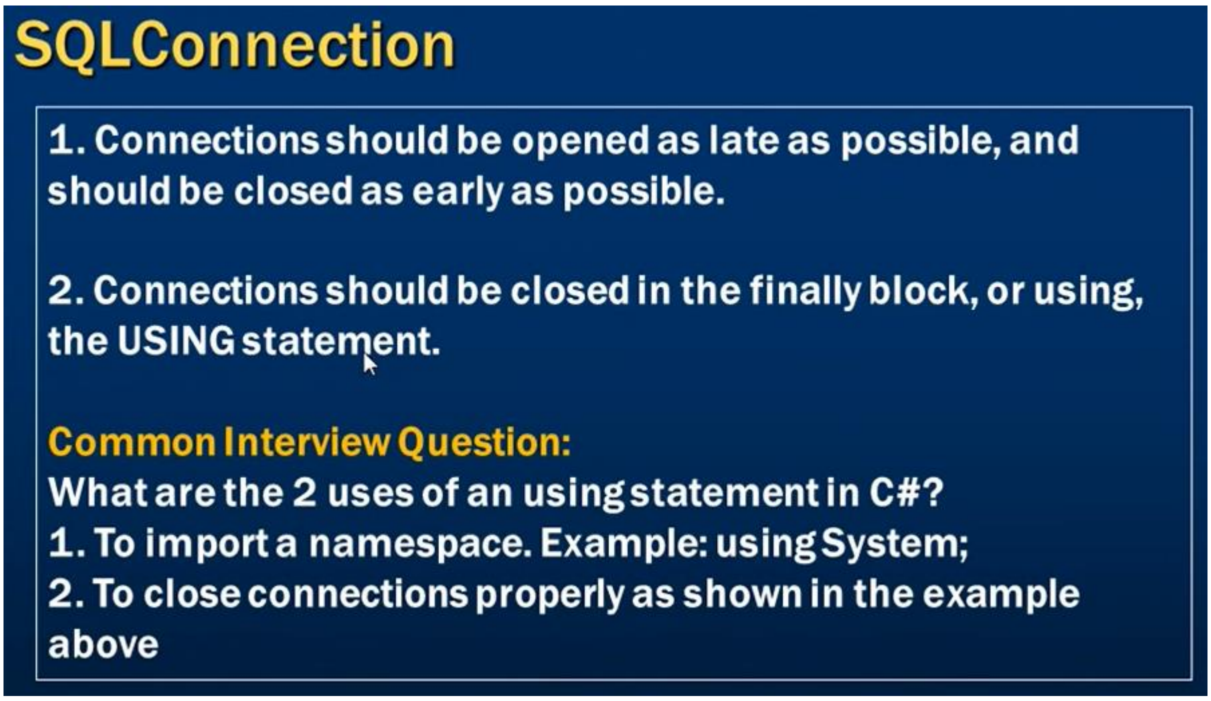
Top Objects

1. Connection
2. Command
3. Data Reader
4. Data Adaptor
5. DataSet (System.Data)



Connection String In case of different Data Providers	`SqlConnection connection = new SqlConnection("data source=.; database=FastPass; integrated security=SSPI;")`
	
	`OdbcConnection connection = new OdbcConnection("Driver={Sql Server};Server=.;Database=FastPass;")`
	
	`OleDbConnection connection = new OleDbConnection("Provider=SQLOLEDB;Data Source=.;Initial Catalog=FastPass; Integrated Security=SSPI;")`

```
string windowsAuthenticationCS = "data source=2212VIKASS0000L; database=FastPass;
integrated security=SSPI";
string sqlAuthenticationCS = "data source=2212VIKASS0000L; database=FastPass; user
id=sa; password=Microsoft#1234";
///We can use try catch finally to handle the connection.close; else we can use the
using statement, as soon as the control left the using block connection will get
closed forcefully by using statement. so either way is fine to handle connection
/// Using statement internally get converted into try catch finally (and disposes
object in finally block)
using (SqlConnection connection = new SqlConnection(sqlAuthenticationCS))
{
    SqlCommand command = new SqlCommand("Select * from Activity", connection);
    connection.Open();
    SqlDataReader reader = command.ExecuteReader();
    while (reader.Read())
    {
         Console.WriteLine("ActivityID is {0}", reader.GetGuid(0));
    }
}
```



SQLConnection

1. Connections should be opened as late as possible, and should be closed as early as possible.

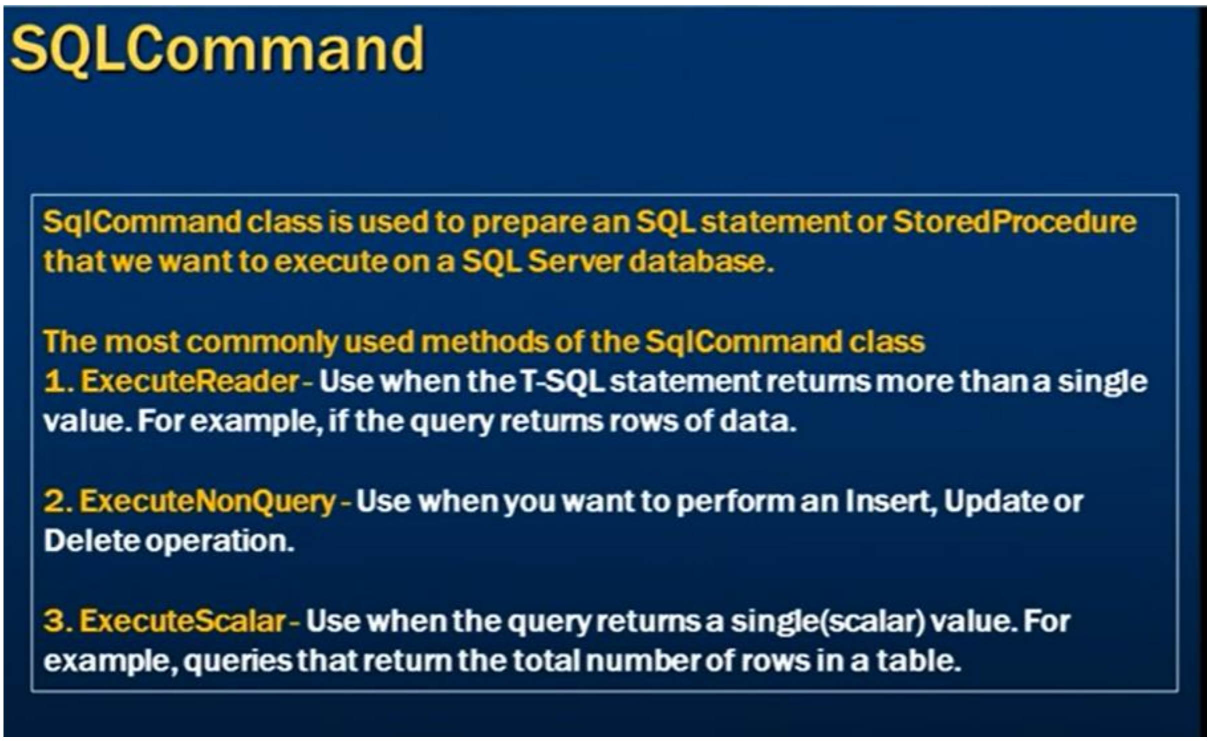
2. Connections should be closed in the finally block, or using, the USING statement.

Common Interview Question:
What are the 2 uses of an using statement in C#?
1. To import a namespace. Example: using System;
2. To close connections properly as shown in the example above

Read Connection String through web.config/app.config

```
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5.2" />
    </startup>
  <connectionStrings>
    <add name="windowsAuthenticationCS"
         connectionString="data source=2212VIKASS0000L; database=FastPass; integrated
security=SSPI"
         providerName="System.Data.SqlClient"/>
    <add name="sqlAuthenticationCS"
         connectionString="data source=2212VIKASS0000L; database=FastPass; user id=sa;
password=Microsoft#1234"
         providerName="System.Data.SqlClient"/>
  </connectionStrings>
</configuration>
```

```
string connectionString = ConfigurationManager.ConnectionStrings["windowsAuthentica-
tionCS"].ConnectionString;
using (SqlConnection connection = new SqlConnection(connectionString))
{

}
```



SQLCommand

SqlCommand class is used to prepare an SQL statement or StoredProcedure that we want to execute on a SQL Server database.

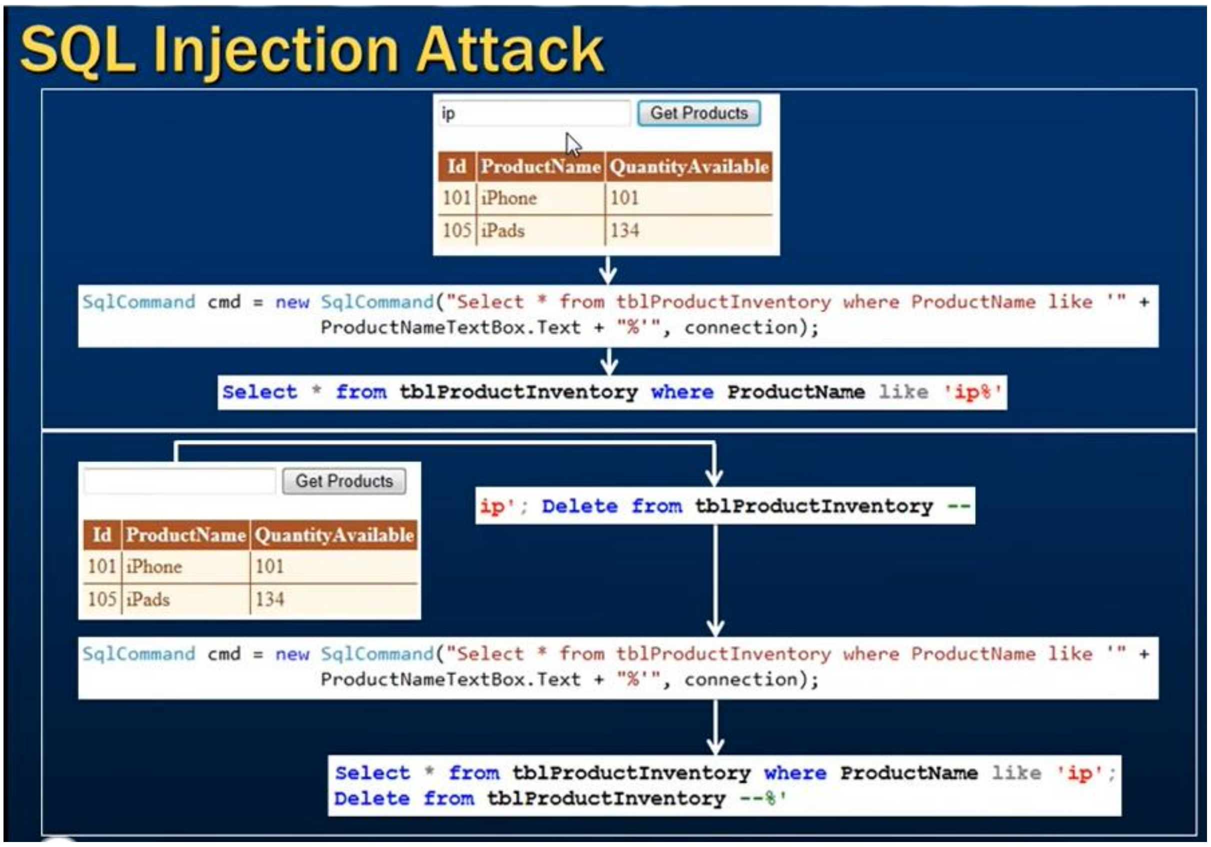
The most commonly used methods of the SqlCommand class
1. ExecuteReader - Use when the T-SQL statement returns more than a single value. For example, if the query returns rows of data.

2. ExecuteNonQuery - Use when you want to perform an Insert, Update or Delete operation.

3. ExecuteScalar - Use when the query returns a single(scalar) value. For example, queries that return the total number of rows in a table.

ExecuteNonQuery: returns int that is how many rows affected

ExecuteScaler: returns first column of first row ignoring every other cols/rows. And return type is object



SQL Injection Attack

Id	ProductName	QuantityAvailable
101	iPhone	101
105	iPads	134

```
SqlCommand cmd = new SqlCommand("Select * from tblProductInventory where ProductName like '" +
                    ProductNameTextBox.Text + "%'", connection);
```

```
Select * from tblProductInventory where ProductName like 'ip%'
```

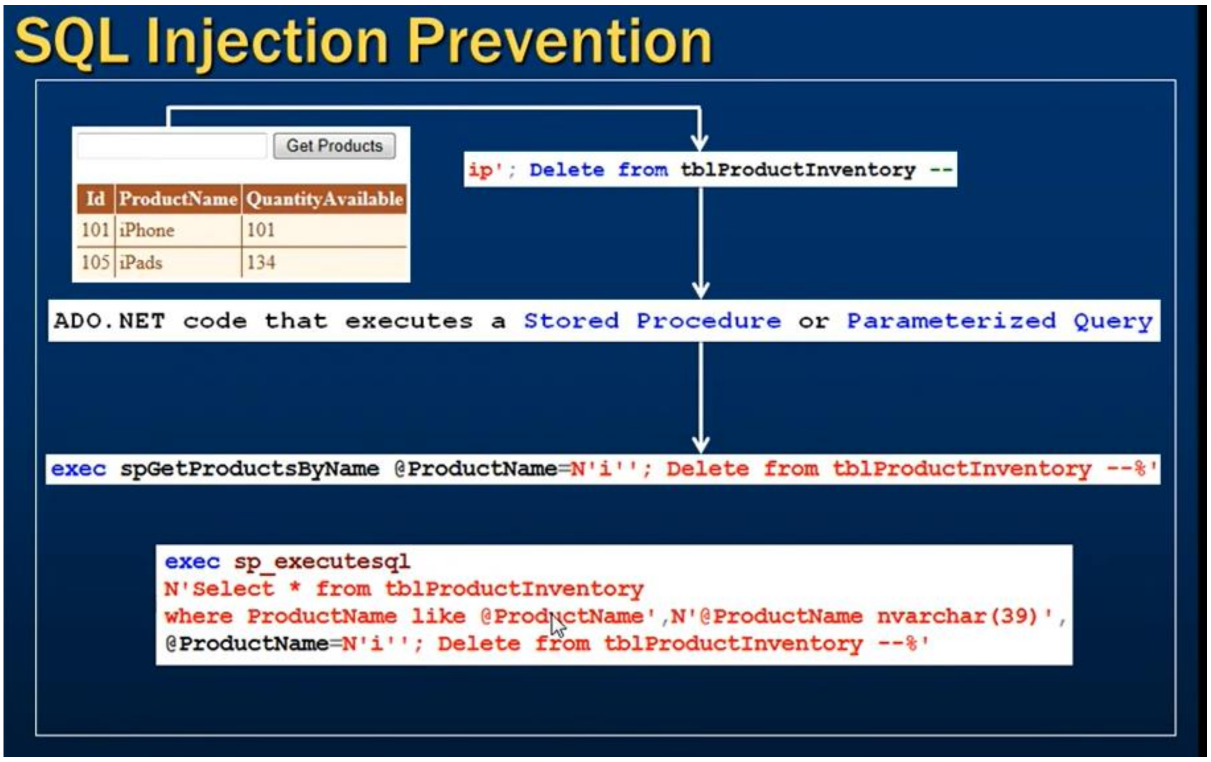
Id	ProductName	QuantityAvailable
101	iPhone	101
105	iPads	134

```
ip'; Delete from tblProductInventory --
```

```
SqlCommand cmd = new SqlCommand("Select * from tblProductInventory where ProductName like '" +
                    ProductNameTextBox.Text + "%'", connection);
```

```
Select * from tblProductInventory where ProductName like 'ip';
Delete from tblProductInventory --%'
```

To avoid this, we should use Parameterized Queries or Stored Procedure



SQL Injection Prevention

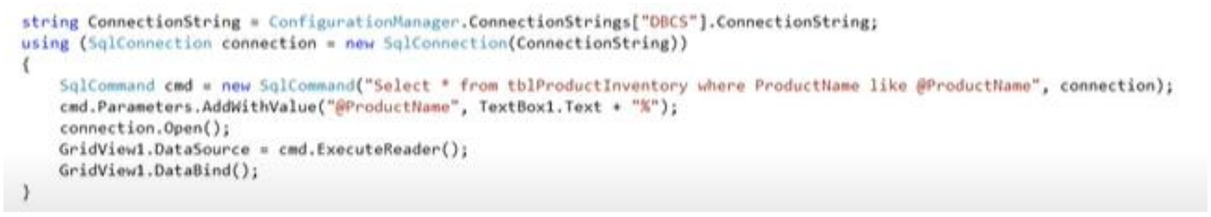
Id	ProductName	QuantityAvailable
101	iPhone	101
105	iPads	134

```
ip'; Delete from tblProductInventory --
```

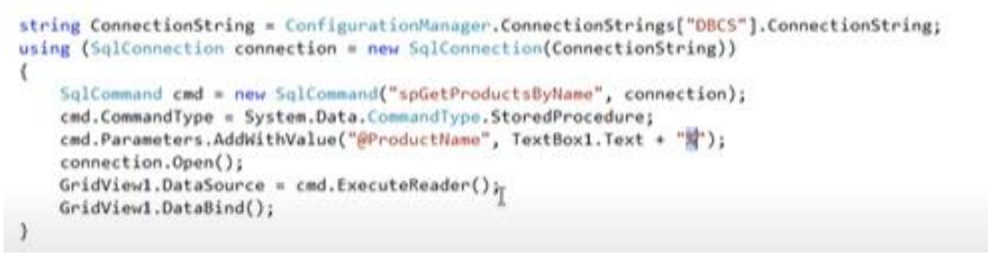
ADO.NET code that executes a Stored Procedure or Parameterized Query

```
exec spGetProductsByName @ProductName=N'i''; Delete from tblProductInventory --%'
```

```
exec sp_executesql
N'Select * from tblProductInventory
where ProductName like @ProductName',N'@ProductName nvarchar(39)',
@ProductName=N'i''; Delete from tblProductInventory --%'
```

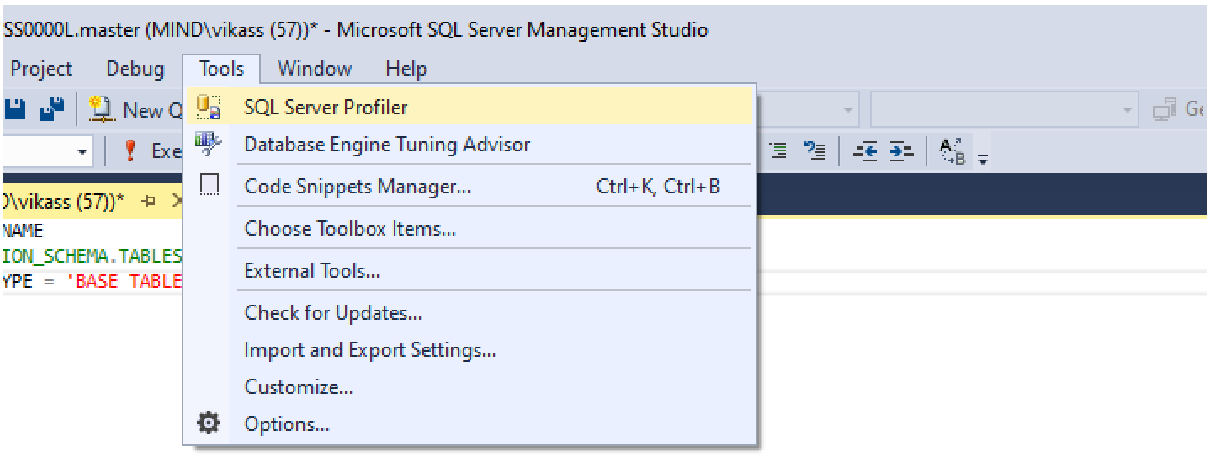


```
string ConnectionString = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
using (SqlConnection connection = new SqlConnection(ConnectionString))
{
    SqlCommand cmd = new SqlCommand("Select * from tblProductInventory where ProductName like @ProductName", connection);
    cmd.Parameters.AddWithValue("@ProductName", TextBox1.Text + "%");
    connection.Open();
    GridView1.DataSource = cmd.ExecuteReader();
    GridView1.DataBind();
}
```

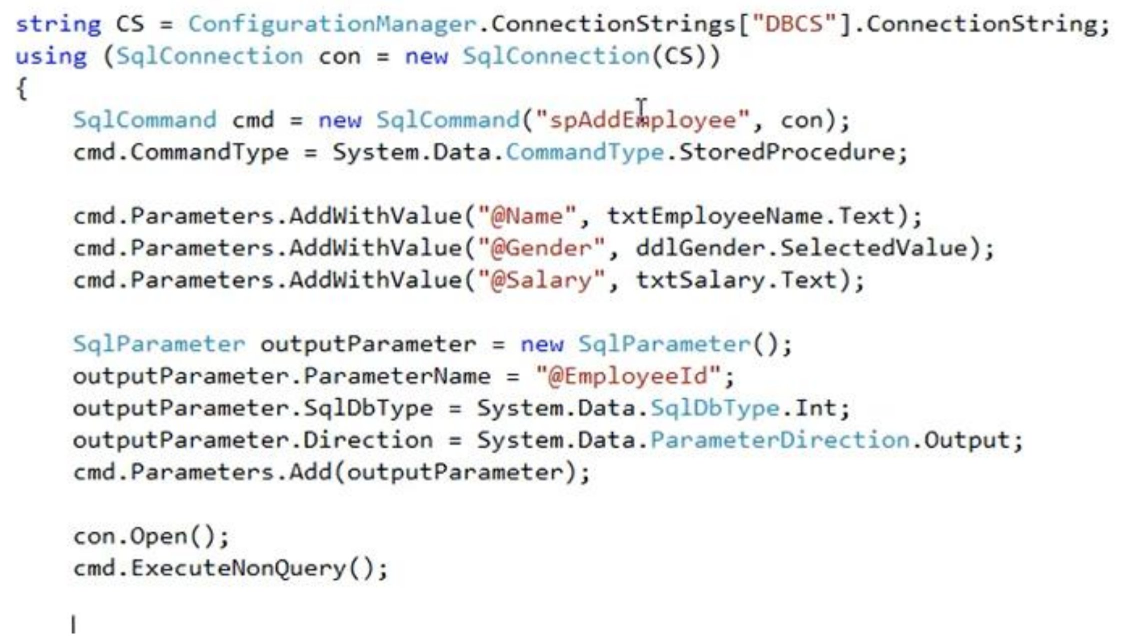


```
string ConnectionString = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
using (SqlConnection connection = new SqlConnection(ConnectionString))
{
    SqlCommand cmd = new SqlCommand("spGetProductsByName", connection);
    cmd.CommandType = System.Data.CommandType.StoredProcedure;
    cmd.Parameters.AddWithValue("@ProductName", TextBox1.Text + "%");
    connection.Open();
    GridView1.DataSource = cmd.ExecuteReader();
    GridView1.DataBind();
}
```

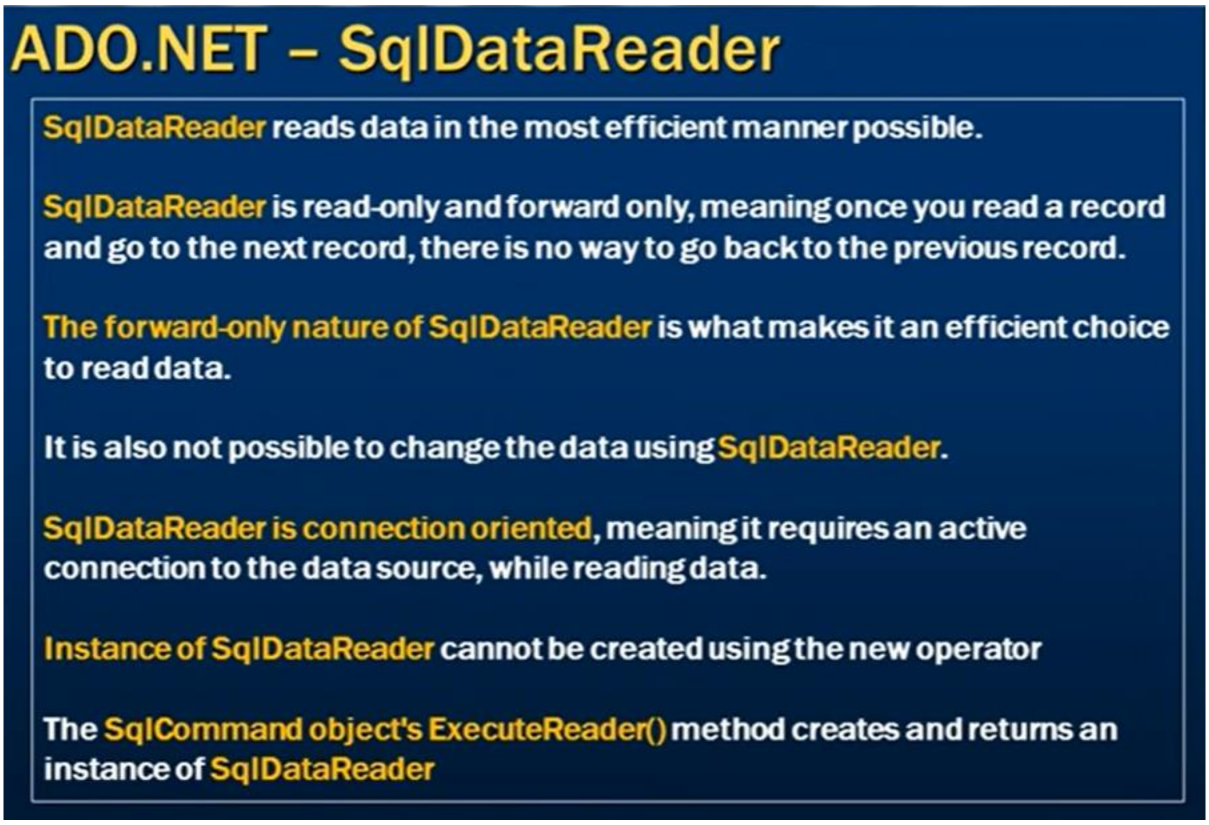
SQL Server Profiler can be found here:



Execute Stored Procedure with “out” parameter:



```
string CS = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
using (SqlConnection con = new SqlConnection(CS))
{
    SqlCommand cmd = new SqlCommand("spAddEmployee", con);
    cmd.CommandType = System.Data.CommandType.StoredProcedure;

    cmd.Parameters.AddWithValue("@Name", txtEmployeeName.Text);
    cmd.Parameters.AddWithValue("@Gender", ddlGender.SelectedValue);
    cmd.Parameters.AddWithValue("@Salary", txtSalary.Text);

    SqlParameter outputParameter = new SqlParameter();
    outputParameter.ParameterName = "@EmployeeId";
    outputParameter.SqlDbType = System.Data.SqlDbType.Int;
    outputParameter.Direction = System.Data.ParameterDirection.Output;
    cmd.Parameters.Add(outputParameter);

    con.Open();
    cmd.ExecuteNonQuery();
```



ADO.NET – SqlDataReader

SqlDataReader reads data in the most efficient manner possible.

SqlDataReader is read-only and forward only, meaning once you read a record and go to the next record, there is no way to go back to the previous record.

The forward-only nature of SqlDataReader is what makes it an efficient choice to read data.

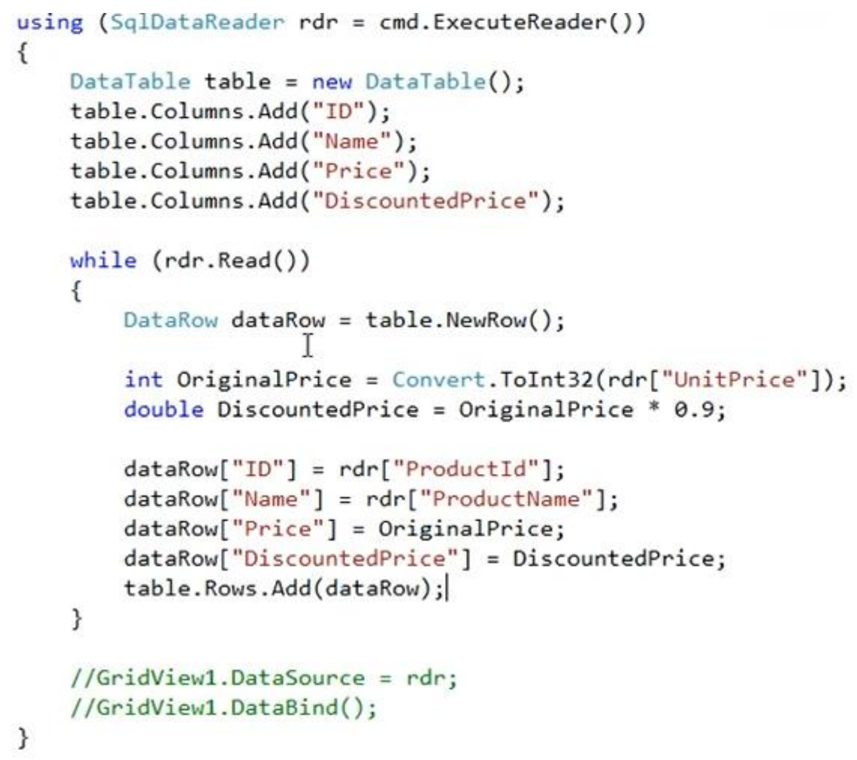
It is also not possible to change the data using SqlDataReader.

SqlDataReader is connection oriented, meaning it requires an active connection to the data source, while reading data.

Instance of SqlDataReader cannot be created using the new operator

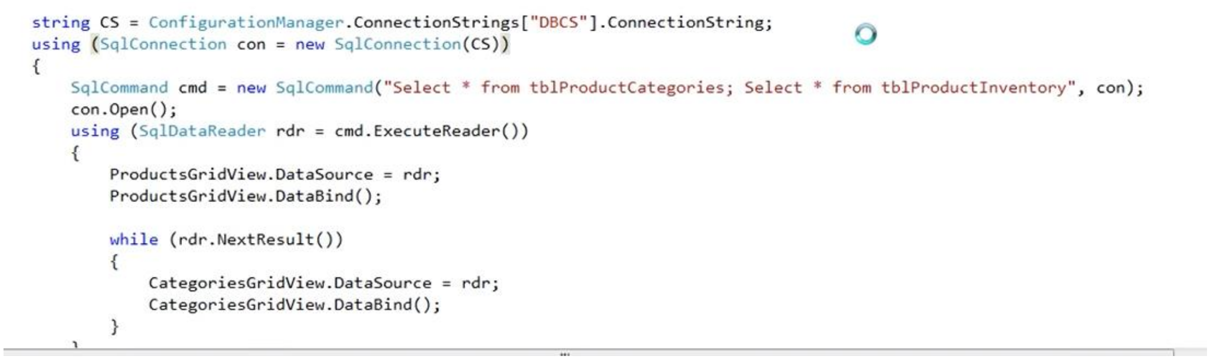
The SqlCommand object's ExecuteReader() method creates and returns an instance of SqlDataReader

Loop through on SqlDataReader:



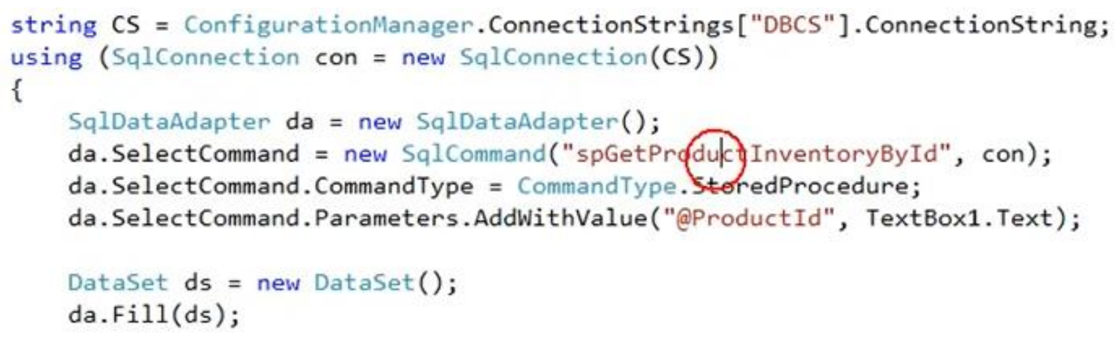
```
using (SqlDataReader rdr = cmd.ExecuteReader())
{
    DataTable table = new DataTable();
    table.Columns.Add("ID");
    table.Columns.Add("Name");
    table.Columns.Add("Price");
    table.Columns.Add("DiscountedPrice");

    while (rdr.Read())
    {
        DataRow dataRow = table.NewRow();

        int OriginalPrice = Convert.ToInt32(rdr["UnitPrice"]);
        double DiscountedPrice = OriginalPrice * 0.9;

        dataRow["ID"] = rdr["ProductId"];
        dataRow["Name"] = rdr["ProductName"];
        dataRow["Price"] = OriginalPrice;
        dataRow["DiscountedPrice"] = DiscountedPrice;
        table.Rows.Add(dataRow);
    }

    //GridView1.DataSource = rdr;
    //GridView1.DataBind();
}
```

Multiple Result Set can be looped through NextResult:



```
string CS = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
using (SqlConnection con = new SqlConnection(CS))
{
    SqlCommand cmd = new SqlCommand("Select * from tblProductCategories; Select * from tblProductInventory", con);
    con.Open();
    using (SqlDataReader rdr = cmd.ExecuteReader())
    {
        ProductsGridView.DataSource = rdr;
        ProductsGridView.DataBind();

        while (rdr.NextResult())
        {
            CategoriesGridView.DataSource = rdr;
            CategoriesGridView.DataBind();
        }
```

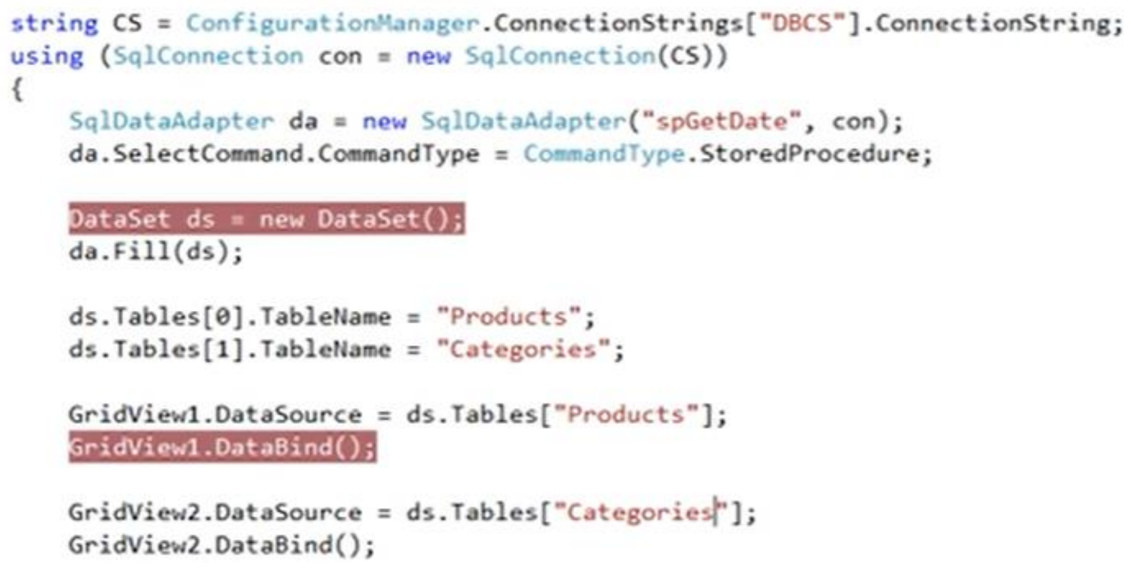
SQLDataAdapter



```
string CS = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
using (SqlConnection con = new SqlConnection(CS))
{
    SqlDataAdapter da = new SqlDataAdapter();
    da.SelectCommand = new SqlCommand("spGetProductInventoryById", con);
    da.SelectCommand.CommandType = CommandType.StoredProcedure;
    da.SelectCommand.Parameters.AddWithValue("@ProductId", TextBox1.Text);

    DataSet ds = new DataSet();
    da.Fill(ds);
```

<u>DataSet:</u>

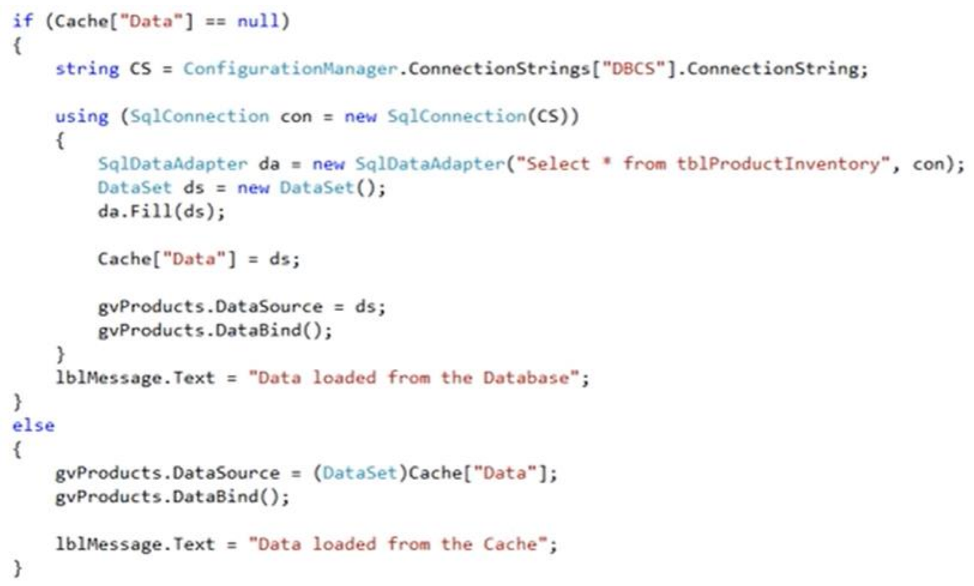
In memory representation of the Database



```
string CS = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
using (SqlConnection con = new SqlConnection(CS))
{
    SqlDataAdapter da = new SqlDataAdapter("spGetDate", con);
    da.SelectCommand.CommandType = CommandType.StoredProcedure;

    DataSet ds = new DataSet();
    da.Fill(ds);

    ds.Tables[0].TableName = "Products";
    ds.Tables[1].TableName = "Categories";

    GridView1.DataSource = ds.Tables["Products"];
    GridView1.DataBind();

    GridView2.DataSource = ds.Tables["Categories"];
    GridView2.DataBind();
```

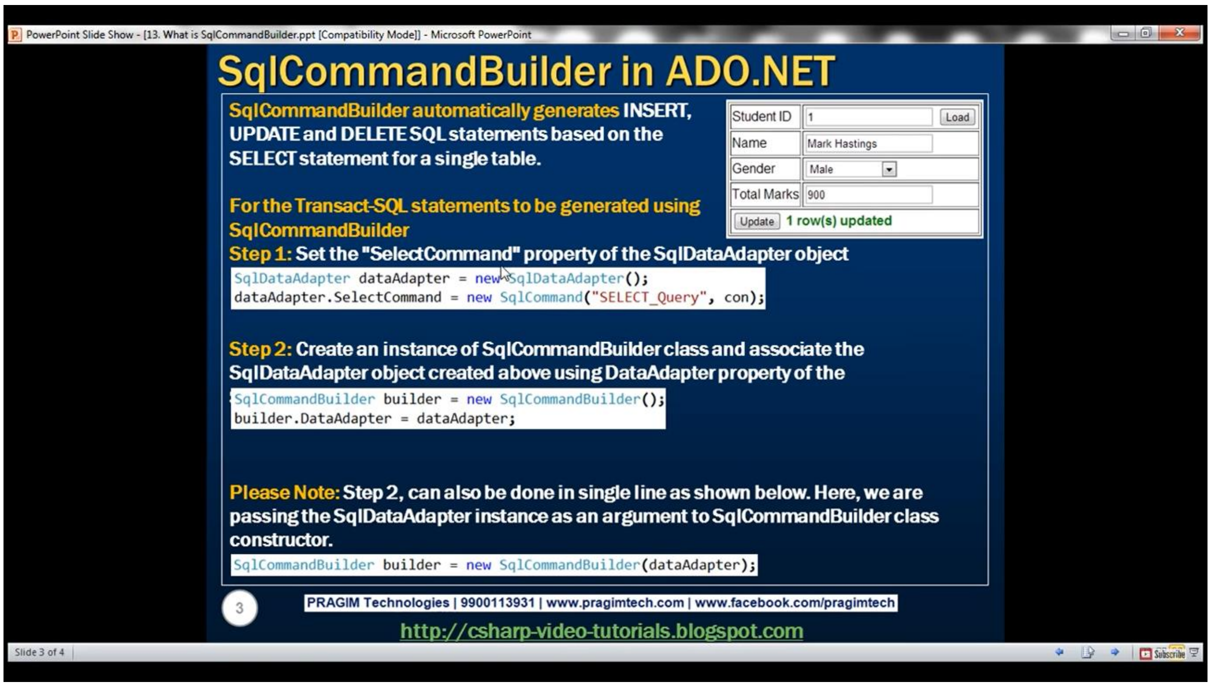
```
            string cs =
        ConfigurationManager.ConnectionStrings["windowsAuthentication"].ConnectionStrin
        g;
            using (SqlConnection con= new SqlConnection(cs))
            {
                SqlDataAdapter sda = new SqlDataAdapter();
                sda.SelectCommand = new SqlCommand("Select * from [User]; Select *
from [Person]", con);
                sda.SelectCommand.CommandType = System.Data.CommandType.Text;

                DataSet ds = new DataSet();
                sda.Fill(ds);

                ds.Tables[0].TableName = "Users";
                ds.Tables[1].TableName = "Persons";

                GridView1.DataSource = ds.Tables["Users"];
                GridView1.DataBind();
                GridView2.DataSource = ds.Tables["Persons"];
                GridView2.DataBind();
            }
```

Cache Data Using Data Set



```
if (Cache["Data"] == null)
{
    string CS = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;

    using (SqlConnection con = new SqlConnection(CS))
    {
        SqlDataAdapter da = new SqlDataAdapter("Select * from tblProductInventory", con);
        DataSet ds = new DataSet();
        da.Fill(ds);

        Cache["Data"] = ds;

        gvProducts.DataSource = ds;
        gvProducts.DataBind();
    }
    lblMessage.Text = "Data loaded from the Database";
}
else
{
    gvProducts.DataSource = (DataSet)Cache["Data"];
    gvProducts.DataBind();

    lblMessage.Text = "Data loaded from the Cache";
}
```

SQL Command Builder


PowerPoint Slide Show - [13. What is SqlCommandBuilder.ppt [Compatibility Mode]] - Microsoft PowerPoint

SqlCommandBuilder in ADO.NET

SqlCommandBuilder automatically generates INSERT, UPDATE and DELETE SQL statements based on the SELECT statement for a single table.

Student ID	1	Load
Name	Mark Hastings	
Gender	Male	
Total Marks	900	
Update	1 row(s) updated	

For the Transact-SQL statements to be generated using SqlCommandBuilder

Step 1: Set the "SelectCommand" property of the SqlDataAdapter object

```
SqlDataAdapter dataAdapter = new SqlDataAdapter();
dataAdapter.SelectCommand = new SqlCommand("SELECT_Query", con);
```

Step 2: Create an instance of SqlCommandBuilder class and associate the SqlDataAdapter object created above using DataAdapter property of the

```
SqlCommandBuilder builder = new SqlCommandBuilder();
builder.DataAdapter = dataAdapter;
```

Please Note: Step 2, can also be done in single line as shown below. Here, we are passing the SqlDataAdapter instance as an argument to SqlCommandBuilder class constructor.

```
SqlCommandBuilder builder = new SqlCommandBuilder(dataAdapter);
```

3

PRAGIM Technologies | 9900113931 | www.pragimtech.com | www.facebook.com/pragimtech

http://csharp-video-tutorials.blogspot.com

Slide 3 of 4

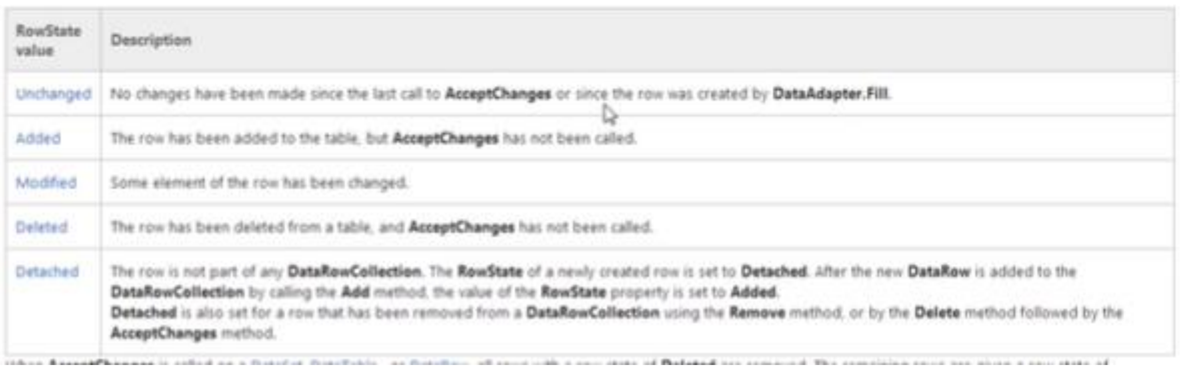
```
sda = new SqlDataAdapter("SELECT * from [Professors]", con);
```

SqlCommandBuilder.getDeleteCommand

INSERT INTO [Professors] ([ID], [Name], [Age], [Department]) VALUES (@p1, @p2, @p3, @p4)

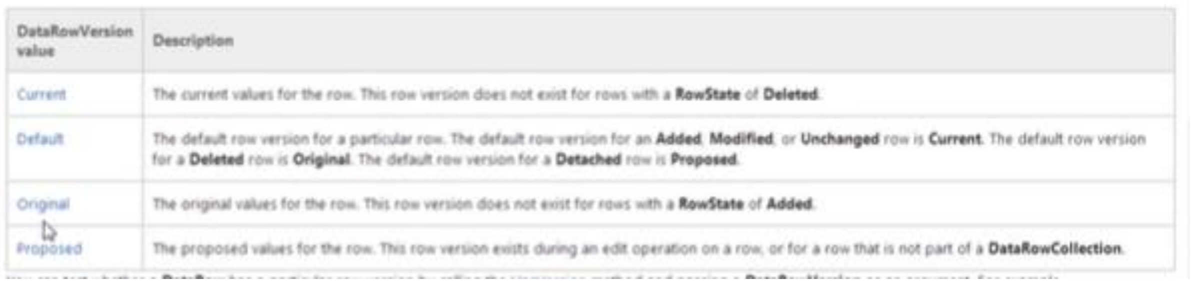
DELETE FROM [Professors] WHERE (([ID] = @p1) AND ((@p2 = 1 AND [Name] IS NULL) OR ([Name] = @p3)) AND ((@p4 = 1 AND [Age] IS NULL) OR ([Age] = @p5)) AND ((@p6 = 1 AND [Department] IS NULL) OR ([Department] = @p7)))

```
UPDATE [Professors] SET [ID] = @p1, [Name] = @p2,
[Age] = @p3, [Department] = @p4 WHERE (([ID] =
@p5) AND ((@p6 = 1 AND [Name] IS NULL) OR ([Name]
= @p7)) AND ((@p8 = 1 AND [Age] IS NULL) OR ([Age] =
@p9)) AND ((@p10 = 1 AND [Department] IS NULL) OR
([Department] = @p11)))
```



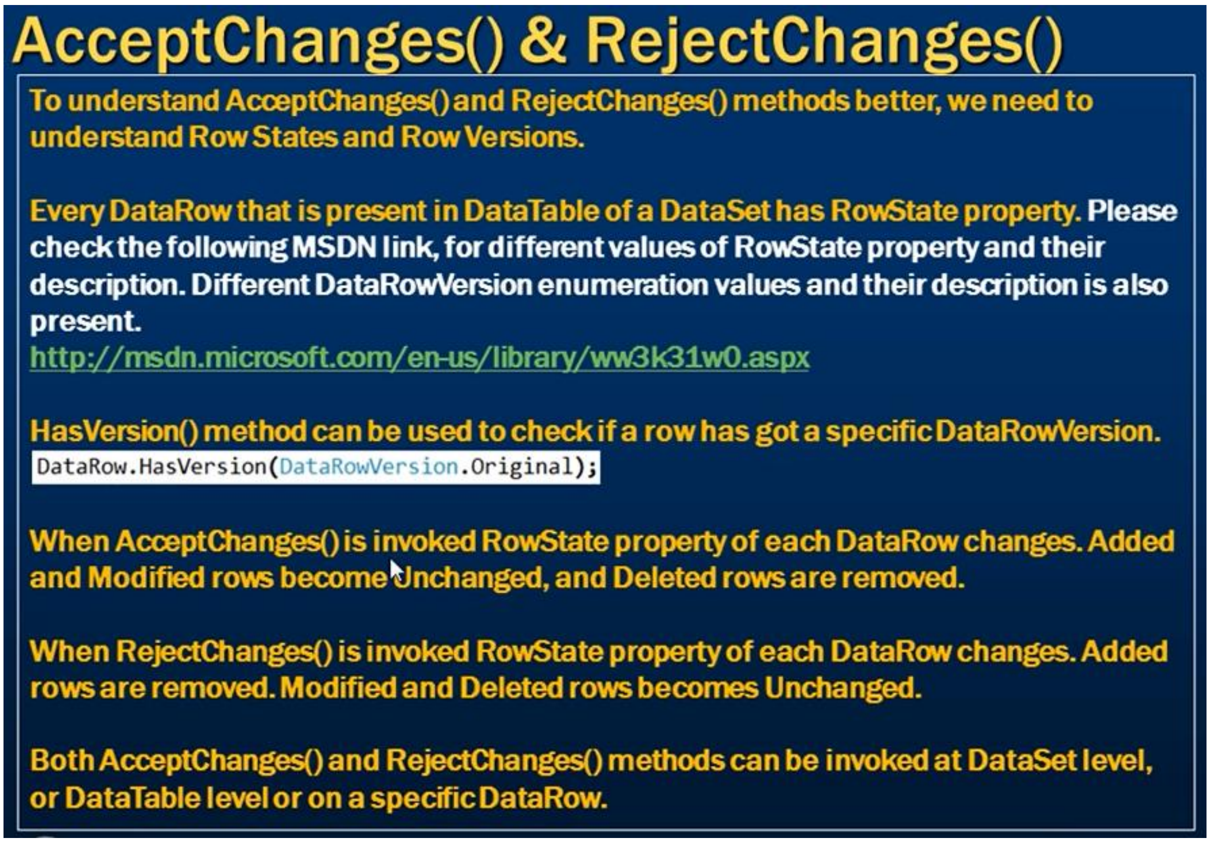
RowState value	Description
Unchanged	No changes have been made since the last call to **AcceptChanges** or since the row was created by **DataAdapter.Fill**.
Added	The row has been added to the table, but **AcceptChanges** has not been called.
Modified	Some element of the row has been changed.
Deleted	The row has been deleted from a table, and **AcceptChanges** has not been called.
Detached	The row is not part of any **DataRowCollection**. The **RowState** of a newly created row is set to **Detached**. After the new **DataRow** is added to the **DataRowCollection** by calling the **Add** method, the value of the **RowState** property is set to **Added**. **Detached** is also set for a row that has been removed from a **DataRowCollection** using the **Remove** method, or by the **Delete** method followed by the **AcceptChanges** method.

[illegible]



DataRowVersion value	Description
Current	The current values for the row. This row version does not exist for rows with a **RowState** of **Deleted**.
Default	The default row version for a particular row. The default row version for an **Added**, **Modified**, or **Unchanged** row is **Current**. The default row version for a **Deleted** row is **Original**. The default row version for a **Detached** row is **Proposed**.
Original	The original values for the row. This row version does not exist for rows with a **RowState** of **Added**.
Proposed	The proposed values for the row. This row version exists during an edit operation on a row, or for a row that is not part of a **DataRowCollection**.

[illegible]



AcceptChanges() & RejectChanges()

To understand AcceptChanges() and RejectChanges() methods better, we need to understand Row States and Row Versions.

Every DataRow that is present in DataTable of a DataSet has RowState property. Please check the following MSDN link, for different values of RowState property and their description. Different DataRowVersion enumeration values and their description is also present.
http://msdn.microsoft.com/en-us/library/ww3k31w0.aspx

HasVersion() method can be used to check if a row has got a specific DataRowVersion.

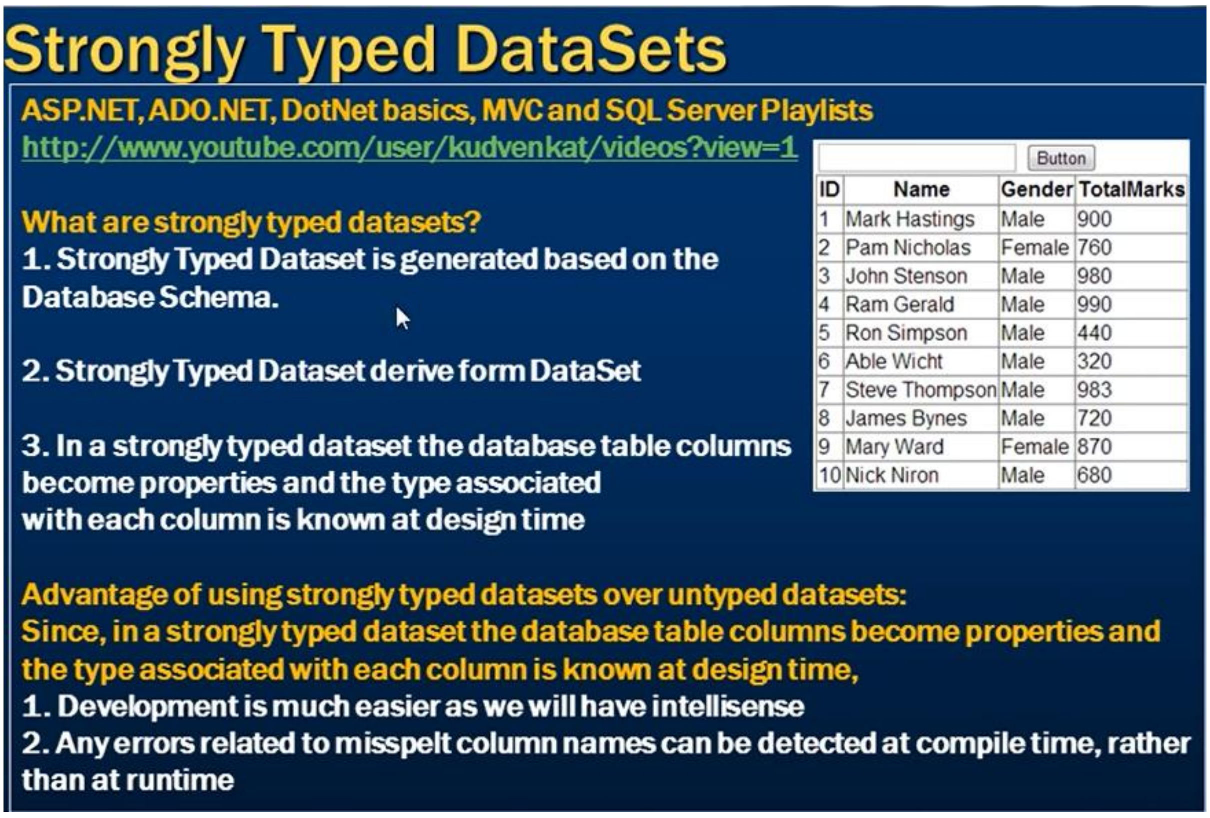
```
DataRow.HasVersion(DataRowVersion.Original);
```

When AcceptChanges() is invoked RowState property of each DataRow changes. Added and Modified rows become Unchanged, and Deleted rows are removed.

When RejectChanges() is invoked RowState property of each DataRow changes. Added rows are removed. Modified and Deleted rows becomes Unchanged.

Both AcceptChanges() and RejectChanges() methods can be invoked at DataSet level, or DataTable level or on a specific DataRow.

Strongly typed Data Set



Strongly Typed DataSets

ASP.NET, ADO.NET, DotNet basics, MVC and SQL Server Playlists
http://www.youtube.com/user/kudvenkat/videos?view=1

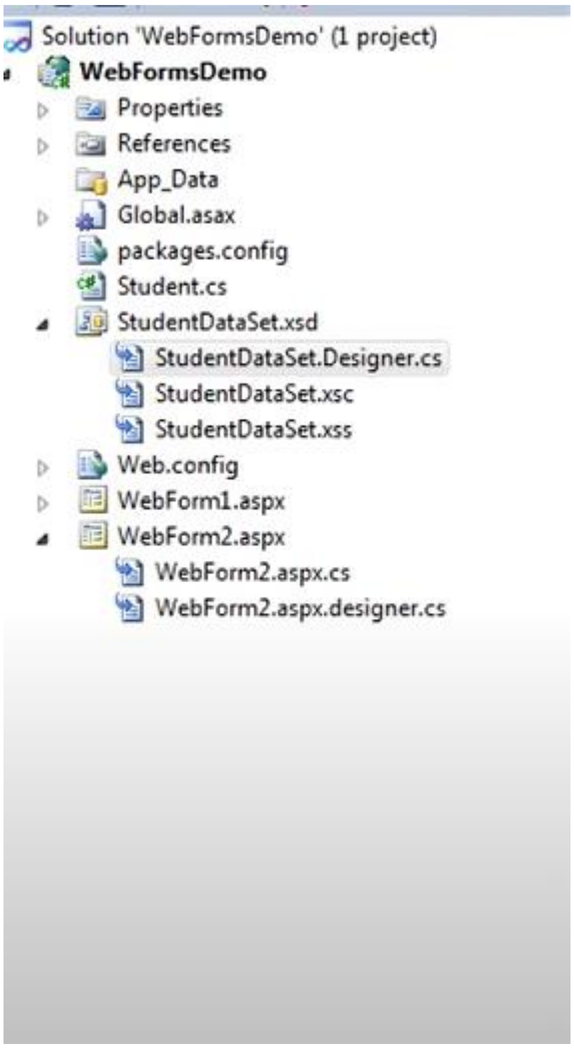
ID	Name	Gender	TotalMarks
1	Mark Hastings	Male	900
2	Pam Nicholas	Female	760
3	John Stenson	Male	980
4	Ram Gerald	Male	990
5	Ron Simpson	Male	440
6	Able Wicht	Male	320
7	Steve Thompson	Male	983
8	James Bynes	Male	720
9	Mary Ward	Female	870
10	Nick Niron	Male	680

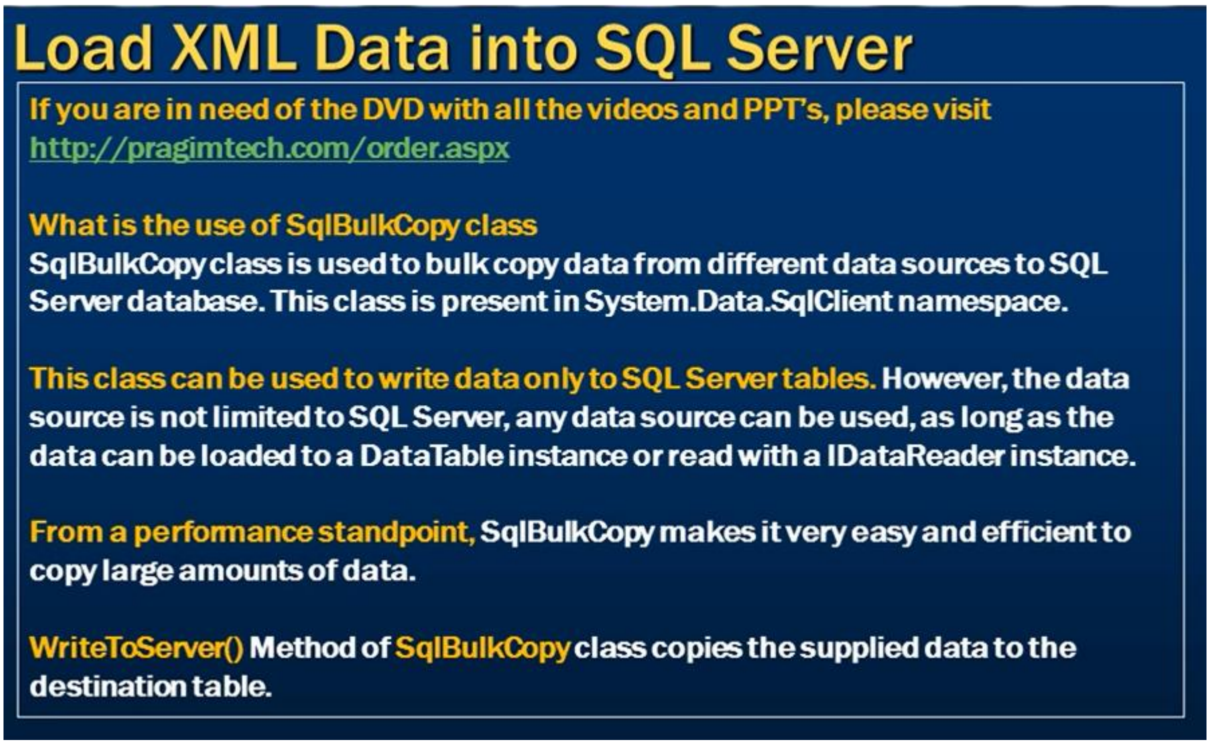
What are strongly typed datasets?
1. Strongly Typed Dataset is generated based on the Database Schema.

2. Strongly Typed Dataset derive form DataSet

3. In a strongly typed dataset the database table columns become properties and the type associated with each column is known at design time

Advantage of using strongly typed datasets over untyped datasets:
Since, in a strongly typed dataset the database table columns become properties and the type associated with each column is known at design time,
1. Development is much easier as we will have intellisense
2. Any errors related to misspelt column names can be detected at compile time, rather than at runtime





Load XML Data into SQL Server

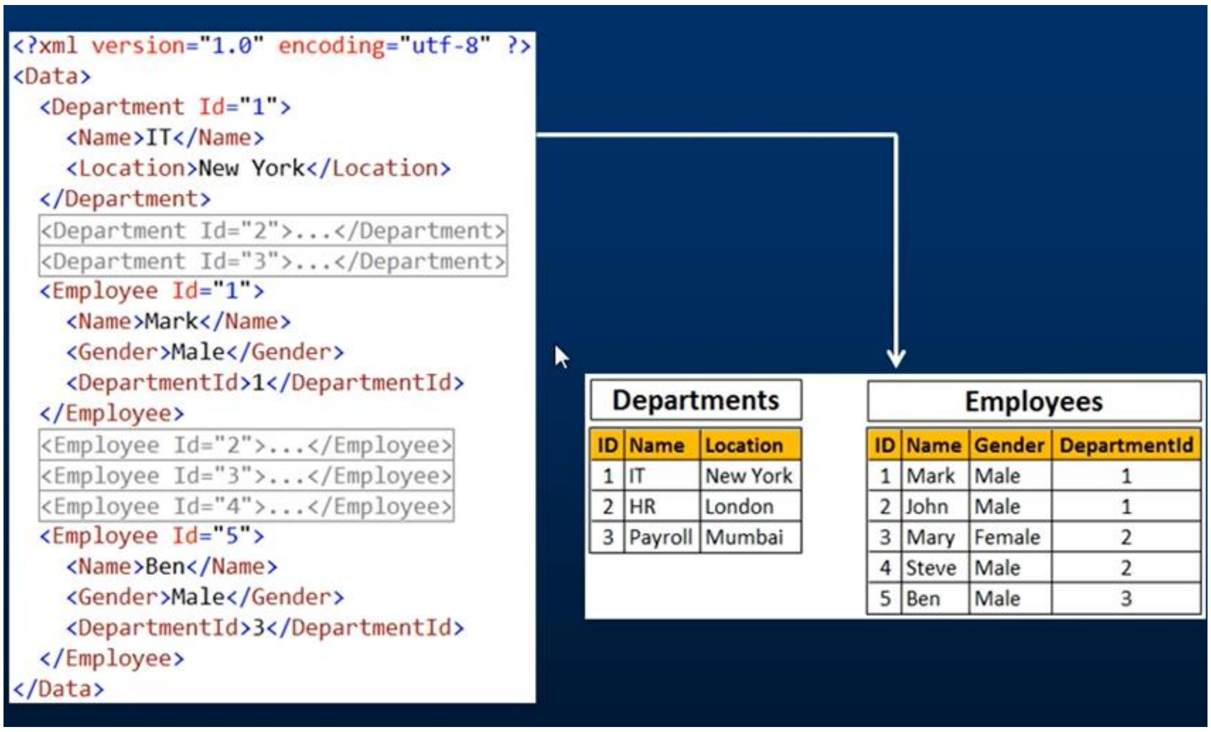
If you are in need of the DVD with all the videos and PPT's, please visit
http://pragimtech.com/order.aspx

What is the use of SqlBulkCopy class
SqlBulkCopy class is used to bulk copy data from different data sources to SQL Server database. This class is present in System.Data.SqlClient namespace.

This class can be used to write data only to SQL Server tables. However, the data source is not limited to SQL Server, any data source can be used, as long as the data can be loaded to a DataTable instance or read with a IDataReader instance.

From a performance standpoint, SqlBulkCopy makes it very easy and efficient to copy large amounts of data.

WriteToServer() Method of SqlBulkCopy class copies the supplied data to the destination table.



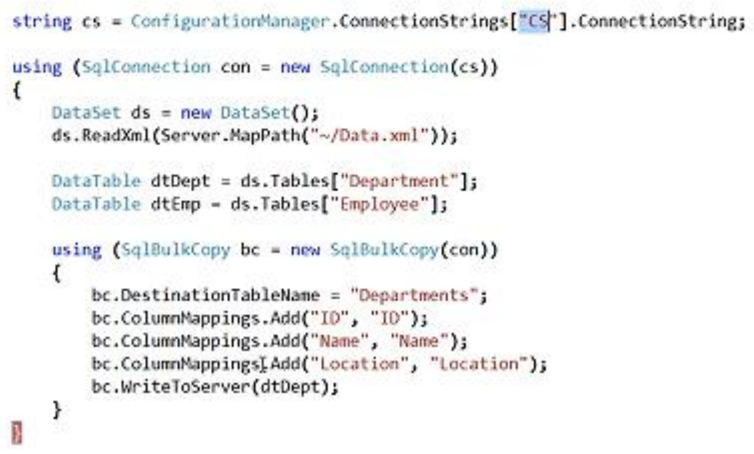
```xml
<?xml version="1.0" encoding="utf-8" ?>
<Data>
  <Department Id="1">
    <Name>IT</Name>
    <Location>New York</Location>
  </Department>
  <Department Id="2">...</Department>
  <Department Id="3">...</Department>
  <Employee Id="1">
    <Name>Mark</Name>
    <Gender>Male</Gender>
    <DepartmentId>1</DepartmentId>
  </Employee>
  <Employee Id="2">...</Employee>
  <Employee Id="3">...</Employee>
  <Employee Id="4">...</Employee>
  <Employee Id="5">
    <Name>Ben</Name>
    <Gender>Male</Gender>
    <DepartmentId>3</DepartmentId>
  </Employee>
</Data>
```

Departments

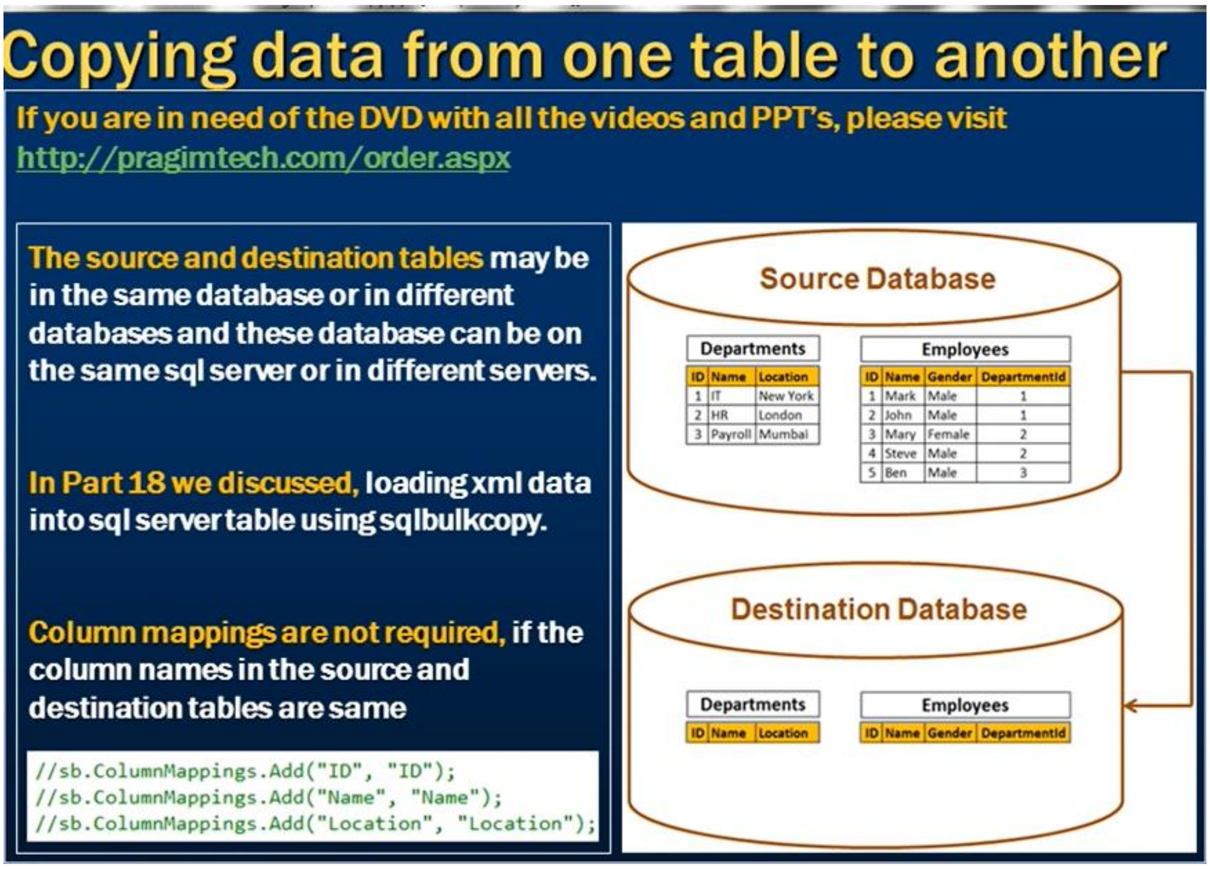
ID	Name	Location
1	IT	New York
2	HR	London
3	Payroll	Mumbai

Employees

ID	Name	Gender	DepartmentId
1	Mark	Male	1
2	John	Male	1
3	Mary	Female	2
4	Steve	Male	2
5	Ben	Male	3



```csharp
string cs = ConfigurationManager.ConnectionStrings["CS"].ConnectionString;

using (SqlConnection con = new SqlConnection(cs))
{
    DataSet ds = new DataSet();
    ds.ReadXml(Server.MapPath("~/Data.xml"));

    DataTable dtDept = ds.Tables["Department"];
    DataTable dtEmp = ds.Tables["Employee"];

    using (SqlBulkCopy bc = new SqlBulkCopy(con))
    {
        bc.DestinationTableName = "Departments";
        bc.ColumnMappings.Add("ID", "ID");
        bc.ColumnMappings.Add("Name", "Name");
        bc.ColumnMappings.Add("Location", "Location");
        bc.WriteToServer(dtDept);
    }
}
```



Copying data from one table to another

If you are in need of the DVD with all the videos and PPT's, please visit
http://pragimtech.com/order.aspx

The source and destination tables may be in the same database or in different databases and these database can be on the same sql server or in different servers.

In Part 18 we discussed, loading xml data into sql server table using sqlbulkcopy.

Column mappings are not required, if the column names in the source and destination tables are same

```
//sb.ColumnMappings.Add("ID", "ID");
//sb.ColumnMappings.Add("Name", "Name");
//sb.ColumnMappings.Add("Location", "Location");
```

Source Database

Departments

ID	Name	Location
1	IT	New York
2	HR	London
3	Payroll	Mumbai

Employees

ID	Name	Gender	DepartmentId
1	Mark	Male	1
2	John	Male	1
3	Mary	Female	2
4	Steve	Male	2
5	Ben	Male	3

Destination Database

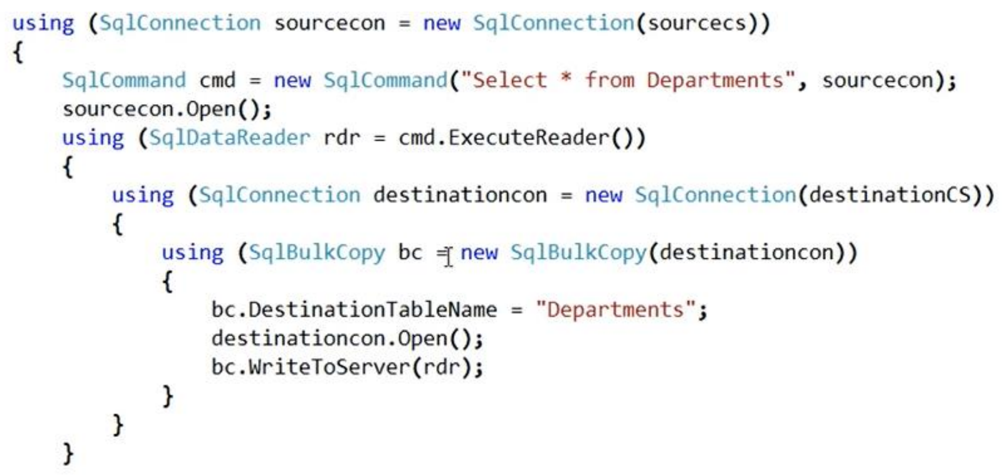
Departments

ID	Name	Location

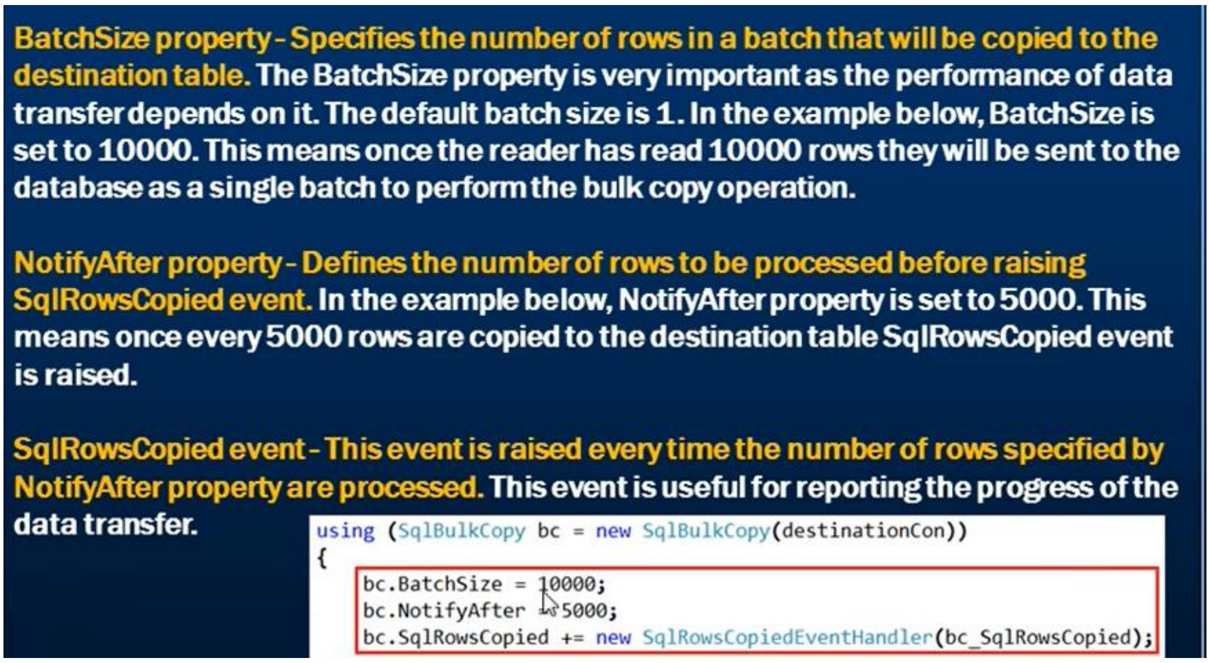
Employees

ID	Name	Gender	DepartmentId

SqlDataReader are better than DataTable in case of reading data as DataReader are move-forward cursors



```
using (SqlConnection sourcecon = new SqlConnection(sourcecs))
{
    SqlCommand cmd = new SqlCommand("Select * from Departments", sourcecon);
    sourcecon.Open();
    using (SqlDataReader rdr = cmd.ExecuteReader())
    {
        using (SqlConnection destinationcon = new SqlConnection(destinationCS))
        {
            using (SqlBulkCopy bc = new SqlBulkCopy(destinationcon))
            {
                bc.DestinationTableName = "Departments";
                destinationcon.Open();
                bc.WriteToServer(rdr);
            }
        }
    }
```

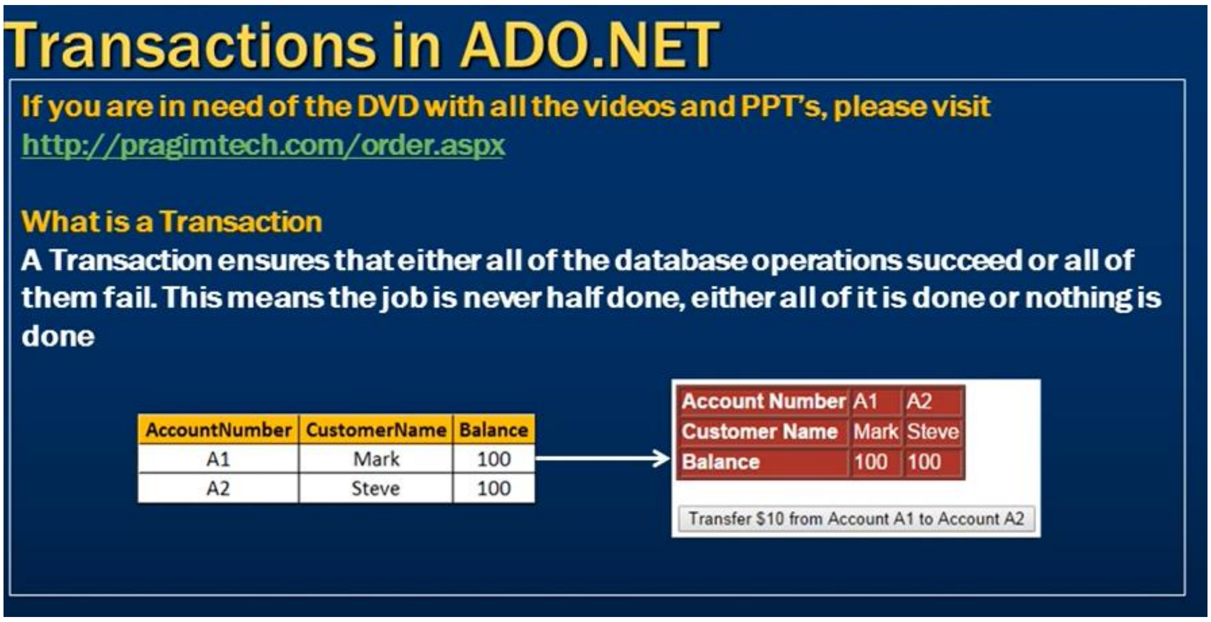


BatchSize property - Specifies the number of rows in a batch that will be copied to the destination table. The BatchSize property is very important as the performance of data transfer depends on it. The default batch size is 1. In the example below, BatchSize is set to 10000. This means once the reader has read 10000 rows they will be sent to the database as a single batch to perform the bulk copy operation.

NotifyAfter property - Defines the number of rows to be processed before raising SqlRowsCopied event. In the example below, NotifyAfter property is set to 5000. This means once every 5000 rows are copied to the destination table SqlRowsCopied event is raised.

SqlRowsCopied event - This event is raised every time the number of rows specified by NotifyAfter property are processed. This event is useful for reporting the progress of the data transfer.

```
using (SqlBulkCopy bc = new SqlBulkCopy(destinationCon))
{
    bc.BatchSize = 10000;
    bc.NotifyAfter = 5000;
    bc.SqlRowsCopied += new SqlRowsCopiedEventHandler(bc_SqlRowsCopied);
```



Transactions in ADO.NET

If you are in need of the DVD with all the videos and PPT's, please visit http://pragimtech.com/order.aspx

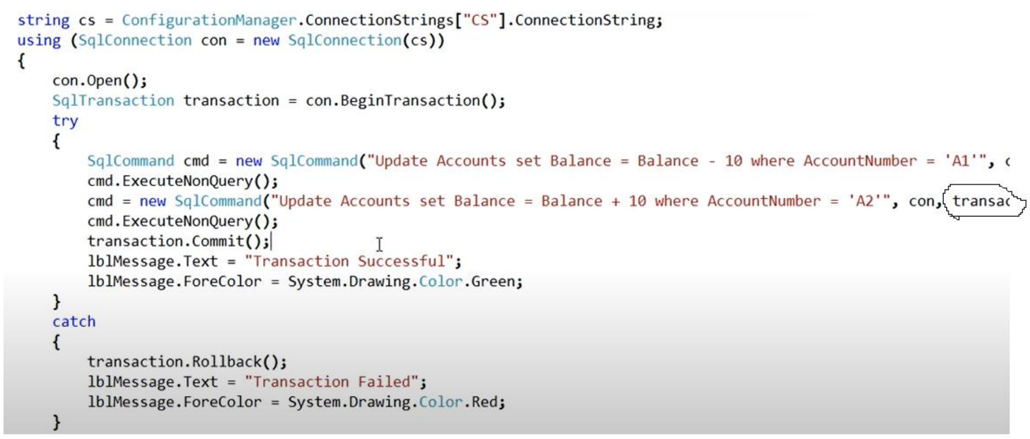
What is a Transaction

A Transaction ensures that either all of the database operations succeed or all of them fail. This means the job is never half done, either all of it is done or nothing is done

AccountNumber	CustomerName	Balance
A1	Mark	100
A2	Steve	100

Account Number	A1	A2
Customer Name	Mark	Steve
Balance	100	100

Transfer $10 from Account A1 to Account A2



```
string cs = ConfigurationManager.ConnectionStrings["CS"].ConnectionString;
using (SqlConnection con = new SqlConnection(cs))
{
    con.Open();
    SqlTransaction transaction = con.BeginTransaction();
    try
    {
        SqlCommand cmd = new SqlCommand("Update Accounts set Balance = Balance - 10 where AccountNumber = 'A1'", c
        cmd.ExecuteNonQuery();
        cmd = new SqlCommand("Update Accounts set Balance = Balance + 10 where AccountNumber = 'A2'", con, transac
        cmd.ExecuteNonQuery();
        transaction.Commit();
        lblMessage.Text = "Transaction Successful";
        lblMessage.ForeColor = System.Drawing.Color.Green;
    }
    catch
    {
        transaction.Rollback();
        lblMessage.Text = "Transaction Failed";
        lblMessage.ForeColor = System.Drawing.Color.Red;
    }
```