DOT NET PROGRAMMING

CHAPTER – 5: Data access with ADO.NET

SESSION – 37 : The Two Faces Of ADO.NET,

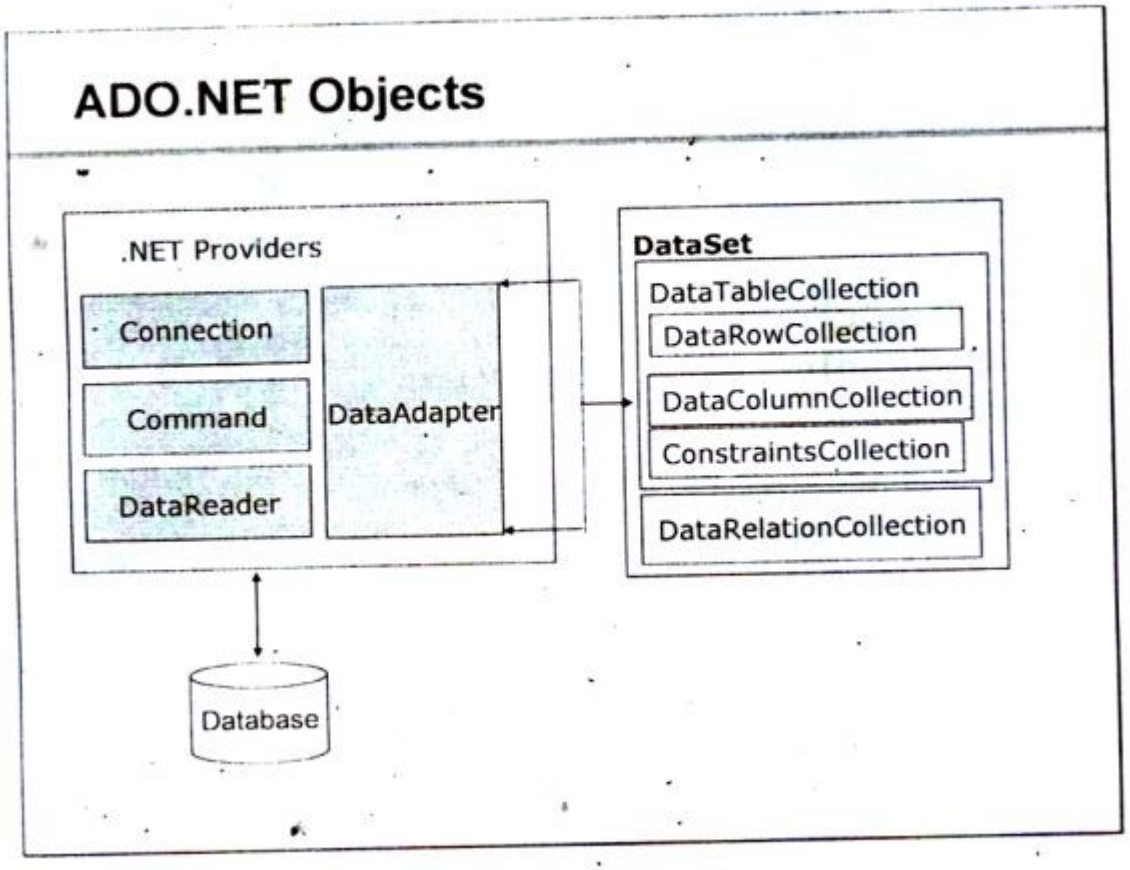
Definition: ADO is a rich set of classes, interfaces, structures, and enumerated types that manage data access from various types of data stores.

- Enterprise applications handle a large amount of data. This data is primarily stored in relational databases, like Oracle, SQL Server, Access, and so on. These databases use Structured Query Language (SQL) for retrieval of data.
- To access enterprise data from a .NET application, an interface was needed. This interface acts as a bridge between an RDBMS system and a .Net application. ADO.NET is such an interface that is created to connect .NET applications to RDBMS systems.
- In the .NET framework, Microsoft introduced a new version of Active X Data Objects (ADO) called ADO.NET. Any .NET application, either Windows-based or web-based, can interact with the database using a rich set of classes of the ADO.NET library. Data can be accessed from any database using connected or disconnected architecture.
- There were many data access technologies available prior to ADO.NET, primarily the following:
- Open Database Connectivity (ODBC)
- Data Access Objects (DAO)
- Remote Data Objects (RDO)
- Active X Data Objects (ADO)
-
- ADO is a simple component-based object-oriented interface to access data whether relational or non-relational databases. It is a successor of DAO and RDO.
-
- ADO reduces the number of objects. Their properties, methods, and events.
- ADO is built on COM; specifically Activex
- ADO supports universal data access using Object Linking and Embedding for DataBases (OLEDB). This means that there are no restrictions on the type of data that can be accessed.

ADO.NET provides mainly the following two types of architectures:

1. Connected Architecture
2. Disconnected Architecture

ADO.NET Architecture



Connected Architecture

1. In the connected architecture, connection with a data source is kept open constantly for data access as well as data manipulation operations.
2. The ADO.NET Connected architecture considers mainly three types of objects.
    1. SqlConnection con;
    2. SqlCommand cmd;
    3. SqlDataReader dr;

Disconnected Architecture

1. Disconnected is the main feature of the .NET framework. ADO.NET contains various classes that support this architecture. The .NET application does not always stay connected with the database. The classes are designed in a way that they automatically open and close the connection. The data is stored client-side and is updated in the database whenever required.
2. The ADO.NET Disconnected architecture considers primarily the following types of objects:

    1. DataSet ds;
    2. SqlDataAdapter da;
    3. SqlConnection con;
    4. SqlCommandBuilder bldr;

Introduction : The .NET platform defines a number of types that allow us to interact with local and remote data stores. These namespaces are known as ADO.NET.

The System.Data namespace types specifically DataColumn, DataRow, and DataTable. These classes allow us to define and manipulate a local in memory table of data. ADO.NET, the DataSet is an in-memory representation of a collection of interrelated tables. In this chapter, we will learn how to programmatically model table relationships, establish custom views based on a given DataTable, and submit queries against your in-memory DataSet. How to obtain a populated DataSet from a Database Management System (DBMS) such as MS SQL Server, Oracle, or MS Access. During the process, we will examine the role of .NET data providers and come to understand the use of ADO.NET data adapters, command objects, and command builders.

In contrast to the intrinsically disconnected world of DataSets and data adapters, this chapter also examines the connected layer of ADO.NET and the related data reader types. As you will see, the data reader is ideal when you simply wish to obtain a result set from a data store for display purposes. We wrap things up with an overview of various database-centric wizards of Visual Studio .NET, and come to see how these integrated tools can be used to lessen the amount of ADO.NET code you would otherwise need to write by hand.

The Need for ADO.NET : The very first thing we must understand when approaching ADO.NET is that it is not simply the latest and greatest version of classic ADO. While it is true that there is some symmetry between the two systems (e.g., each has the concept of connection and command objects), some familiar types (e.g., the Recordset) no longer exist. Furthermore, there are a number of new ADO.NET types that have no direct equivalent under classic ADO (such as the data adapter). In a nutshell, *"ADO.NET is a brand new database access technology focused on facilitating the development of disconnected (and connected) systems using the .NET platform."*

Unlike classic ADO, which was primarily designed for tightly coupled client/server systems, ADO.NET greatly extends the notion of the primitive ADO disconnected Recordset with a new creature named the DataSet. This type represents a local copy of any number of related tables. Using the DataSet, the client is able to manipulate and update its contents while disconnected from the data source and submit the modified data back for processing using a related data adapter.

Another major difference between classic ADO and ADO.NET is that ADO.NET has full support for XML data representation. In fact, the data obtained from a data store is internally represented, and transmitted, as XML. Given that XML is often transported between layers using standard HTTP, ADO.NET is not limited by firewall constraints.

As we might be aware, classic ADO makes use of the COM marshaling protocol to move data between tiers. While this was appropriate in some situations, COM marshaling poses a number of limitations. For example, most firewalls are configured to reject COM RPC packets, which made moving data between machines tricky.

Perhaps the most fundamental difference between classic ADO and ADO.NET is that ADO.NET is a managed library of code and therefore plays by all the same rules as any managed library. The types that comprise ADO.NET use the CLR memory management protocol, adhere to the same programming model, and work with many languages. Therefore, the types (and their members) are accessed in the same manner, regardless of which .NET-aware language you use.

The Two Faces of ADO.NET : The ADO.NET libraries can be used in two conceptually unique manners: connected or disconnected. When we are making use of the connected layer, we will make use of a .NET data reader. Data readers provide a way to pull records from a data store using a forward-only, read-only approach. As a given data reader pulls over records based on our SQL query, we are directly connected to the data store and stay that way until we explicitly close the connection. In addition to simply reading data via a data reader, the connected layer of ADO.NET allows us to insert, update, or remove records using a related command object.

The disconnected layer, on the other hand, allows us to obtain a set of DataTable types (typically contained within a DataSet) that serves as a local client-side copy of information. When we obtain a DataSet using a data adapter type, the connection is automatically terminated immediately after the fill request (as you would guess, this approach helps quickly free up connects for other callers). At this point, the client application is able to manipulate the DataSet's contents without incurring any network traffic. If the client wishes to push the changes back to the data store, the data adapter

(in conjunction with a set of SQL queries) is used once again to update the data source, at which point the connection is again closed immediately.

In some respects, this approach may remind us of the classic ADO disconnected Recordset. The key difference is that a disconnected Recordset represents a single set of record data, whereas ADO.NET DataSets can model a collection of related tables. In fact, it is technically possible to obtain a client-side DataSet that represents all of the tables found within the remote database. However, as we would expect, a DataSet will more commonly contain a reasonable subset of information.

Understanding the ADO.NET Namespaces : .

NET version 1.1 ships with five data providers out of the box, each of which is logically represented by a specific .NET namespace. In addition, ADO.NET defines some common

namespaces that are used by all data provider implementations. Below table gives a quick rundown of each data-centric .NET namespace.

Table 17-1: ADO.NET Namespaces

ADO.NET Namespace	Meaning in Life
System.Data	This core namespace defines types that represent tables, rows, columns, constraints, and DataSets. This namespace does not define types to connect to a data source. Rather, it defines the types that represent the data itself.
System.Data.Common	This namespace contains types shared between data providers. Many of these types function as base classes to the concrete types defined by a given data provider.
System.Data.OleDb	This namespace defines the types that allow you to connect to an OLE DB-compliant data source. Typically you will use this namespace only if you need to communicate with a data store that does not have a custom data provider.
System.Data.Odbc	This namespace defines the types that constitute the ODBC data provider.
System.Data.OracleClient	This namespace defines the types that constitute the Oracle data provider.
System.Data.SqlClient	This namespace defines the types that constitute the SQL data provider.
System.Data.SqlServerCe	This namespace defines the types that constitute the SQL CE data provider.
System.Data.SqlTypes	Represents native data types used by Microsoft SQL Server. Although you are always free to use the corresponding CLR data types, the SqlTypes are optimized to work with SQL Server.

The System.Data, System.Data.Common, System.Data.OleDb, System.Data.SqlClient, System.Data.Odbc, and System.Data.SqlTypes namespaces are all contained within the System.Data.dll assembly. However, the types of the System.Data.OracleClient namespaces are contained within a separate assembly named System.Data.OracleClient.dll, while the SqlCe types are placed within System.Data.Sqlservice.dll. Thus, like any .NET endeavor, be sure to set the correct external references (and C# "using" statements) for our current project.

The Types of System.Data : Of all the ADO.NET namespaces, System.Data is the lowest common denominator. We simply cannot build ADO.NET applications without specifying this namespace in our data access applications. This namespace contains types that are shared among all ADO.NET data providers, regardless of the underlying data store. In a nutshell, *System.Data contains types that represent the data we obtain*

from a data store, but not the types that make the literal connection. In addition to a number of database-centric exceptions (NoNullAllowedException, RowNotInTableException, MissingPrimaryKeyException, and the like), these types are little more than OO representations of common database primitives (tables, rows, columns, constraints, and so on). Below table lists some of the core types, grouped by related functionality.

Table 17-2: Key Members of the System.Data Namespace

System.Data Type	Meaning in Life
DataColumnCollection DataColumn	DataColumnCollection is used to represent all of the columns used by a given DataTable.DataColumn represents a specific column in a DataTable.
ConstraintCollection Constraint	The ConstraintCollection represents all constraints (foreign key constrains, unique constraints) assigned to a given DataTable. Constraint represents an OO wrapper around a single constraint assigned to one or more DataColumns.
DataRowCollection DataRow	These types represent a collection of rows for a DataTable (DataRowCollection) and a specific row of data in a DataTable (DataRow).
DataRowView DataView	DataRowView allows you to carve out a predefined "view" from an existing row. The DataView type represents a customized view of a DataTable that can be used for sorting, filtering, searching, editing, and navigation.
DataSet	Represents an in-memory cache of data that may consist of multiple related DataTables.
ForeignKeyConstraint UniqueConstraint	ForeignKeyConstraint represents an action restriction enforced on a set of columns in a primary key/foreign key relationship. The UniqueConstraint type represents a restriction on a set of columns in which all values must be unique.
DataRelationCollection DataRelation	This collection represents all relationships (e.g., DataRelation types) between the tables in a DataSet.
DataTableCollection DataTable	The DataTableCollection type represents all of tables (e.g., DataTable types) for a particular DataSet.

Selecting a Data Provider :

.NET 1.1 ships with five data providers. The first of these is OleDb data provider, which is composed of types defined in System.Data.OleDb namespace. The OleDb provider allows us to access data located in any data store that supports the classic OLE DB protocol. Thus, like with classic ADO, we may use the ADO.NET data provider to access SQL Server, Oracle, or MS Access databases. Because the types in the System.Data.OleDb namespace must communicate with unmanaged code (e.g., the OLE DB providers), we need to be aware that a number of .NET to COM translations occur behind the scenes, which can affect performance. By and large, this namespace is useful when we are attempting to communicate with a data source that does not have a specific data provider assembly. fic data provider assembly.

The SQL data provider offers direct access to MS SQL Server data stores, and only SQL Server data stores (version 7.0 and greater). The System.Data.SqlClient namespace contains the types used by the SQL provider and provides the same functionality as the OleDb provider. In fact, for the most part, both namespaces have similarly named items. The key difference is that the SQL provider does not use the OLE DB or classic ADO protocols and thus offers numerous performance benefits.

If we are interested in making use of the System.Data.Oracle, System.Data.Odbc, or System.Data.SqlServerCe namespaces, I will assume you will check out the details as you see fit. However, as you would hope, once you are comfortable with one data provider, you can easily manipulate other providers. Recall that while the exact names of the types will differ between namespaces (for example, OleDbConnection vs. SqlConnection vs. OdbcConnection), semantically related types can be treated in a polymorphic manner given the IDbCommand, IDbConnection, IDbDataAdapter, and IDataReader interfaces et al.

To begin, we'll examine how to connect to a data source using the OleDb data provider, therefore don't forget to specify the proper using directives (recall this data provider is contained within the System.Data.dll assembly) :

```
// Using the OleDb data provider.
using System.Data;
using System.Data.OleDb;
```

Once we have checked out how to interact with a data store using the OleDb data provider, we will see how to make use of the types within the System.Data.SqlClient namespace.

The Types of the System.Data.OleDb Namespace :

Below Table provides a walkthrough of the core types of the System.Data.OleDb namespace.

Table 17-12: Key Types of the System.Data.OleDb Namespace

System.Data.OleDb Type	Meaning in Life
OleDbCommand	Represents a SQL query command to be made to a data source.
OleDbConnection	Represents an open connection to a data source.
OleDbDataAdapter	Represents a set of data commands and a database connection used to fill and update the contents of a DataSet.
OleDbDataReader	Provides a way of reading a forward-only stream of data records from a data source.
OleDbErrorCollection OleDbError OleDbException	OleDbErrorCollection maintains a collection of warnings or errors returned by the data source, each of which is represented by an OleDbError type. When an error is encountered, an exception of type OleDbException is thrown.
OleDbParameterCollection OleDbParameter	Much like classic ADO, the OleDbParameterCollection collection holds onto the parameters sent to a stored procedure in the database. Each parameter is of type OleDbParameter.

Understanding the ADO.NET Namespaces : .

NET version 1.1 ships with five data providers out of the box, each of which is logically represented by a specific .NET namespace. In addition, ADO.NET defines some common namespaces that are used by all data provider implementations. Below table gives a quick rundown of each data-centric .NET namespace.

Table 17-1: ADO.NET Namespaces

ADO.NET Namespace	Meaning in Life
System.Data	This core namespace defines types that represent tables, rows, columns, constraints, and DataSets. This namespace does not define types to connect to a data source. Rather, it defines the types that represent the data itself.
System.Data.Common	This namespace contains types shared between data providers. Many of these types function as base classes to the concrete types defined by a given data provider.
System.Data.OleDb	This namespace defines the types that allow you to connect to an OLE DB-compliant data source. Typically you will use this namespace only if you need to communicate with a data store that does not have a custom data provider.
System.Data.Odbc	This namespace defines the types that constitute the ODBC data provider.
System.Data.OracleClient	This namespace defines the types that constitute the Oracle data provider.
System.Data.SqlClient	This namespace defines the types that constitute the SQL data provider.
System.Data.SqlServerCe	This namespace defines the types that constitute the SQL CE data provider.
System.Data.SqlTypes	Represents native data types used by Microsoft SQL Server. Although you are always free to use the corresponding CLR data types, the SqlTypes are optimized to work with SQL Server.

Selecting a Data Provider :

.NET 1.1 ships with five data providers. The first of these is OleDb data provider, which is composed of types defined in System.Data.OleDb namespace. The OleDb provider allows us to access data located in any data store that supports the classic OLE DB protocol. Thus, like with classic ADO, we may use the ADO.NET data provider to access SQL Server, Oracle, or MS Access databases. Because the types in the System.Data.OleDb namespace must communicate with unmanaged code (e.g., the OLE DB providers), we need to be aware that a number of .NET to COM translations occur behind the scenes, which can affect performance. By and large, this namespace is useful when we are attempting to communicate with a data source that does not have a specific data provider assembly. fic data provider assembly.

The SQL data provider offers direct access to MS SQL Server data stores, and only SQL Server data stores (version 7.0 and greater).

The System.Data.SqlClient namespace contains the types used by the SQL provider and provides the same functionality as the OleDb provider. In fact, for the most part, both namespaces have similarly named items. The key difference is that the SQL provider does not use the OLE DB or classic ADO protocols and thus offers numerous performance benefits.

If we are interested in making use of the System.Data.Oracle, System.Data.Odbc, or System.Data.SqlServerCe namespaces, I will assume you will

check out the details as you see fit. However, as you would hope, once you are comfortable with one data provider, you can easily manipulate other providers. Recall that while the exact names of the types will differ between namespaces (for example, OleDbConnection vs. SqlConnection vs. OdbcConnection), semantically related types can be treated in a polymorphic manner given the IDbCommand, IDbConnection, IDbDataAdapter, and IDataReader interfaces et al.

To begin, we'll examine how to connect to a data source using the OleDb data provider, therefore don't forget to specify the proper using directives (recall this data provider is contained within the System.Data.dll assembly) :

```
// Using the OleDb data provider.
using System.Data;
using System.Data.OleDb;
```

Once we have checked out how to interact with a data store using the OleDb data provider, we will see how to make use of the types within the System.Data.SqlClient namespace.

The Types of the System.Data.OleDb Namespace :

Below Table provides a walkthrough of the core types of the System.Data.OleDb namespace.

Table 17-12: Key Types of the System.Data.OleDb Namespace

System.Data.OleDb Type	Meaning in Life
OleDbCommand	Represents a SQL query command to be made to a data source.
OleDbConnection	Represents an open connection to a data source.
OleDbDataAdapter	Represents a set of data commands and a database connection used to fill and update the contents of a DataSet.
OleDbDataReader	Provides a way of reading a forward-only stream of data records from a data source.
OleDbErrorCollection OleDbError OleDbException	OleDbErrorCollection maintains a collection of warnings or errors returned by the data source, each of which is represented by an OleDbError type. When an error is encountered, an exception of type OleDbException is thrown.
OleDbParameterCollection OleDbParameter	Much like classic ADO, the OleDbParameterCollection collection holds onto the parameters sent to a stored procedure in the database. Each parameter is of type OleDbParameter.

The Role of ADO.NET Data Providers :

Rather than providing a single set of objects to communicate to a variety of data stores, ADO.NET makes use of multiple data providers.Simply put, a data provider is a set of types (within some .NET assembly) that understand how to communicate with a specific data source. Although the names of these types will differ among data providers, each provider will have (at minimum) a set of class types thatimplement some key interfaces defined in the System.Data namespace, specifically IDbCommand, IDbDataAdapter,

IDbConnection, and IDataReader.

As we would guess, these interfaces define the behaviors a managed provider must support to provide connected and disconnected access to the underlying data store (as shown in Figure).

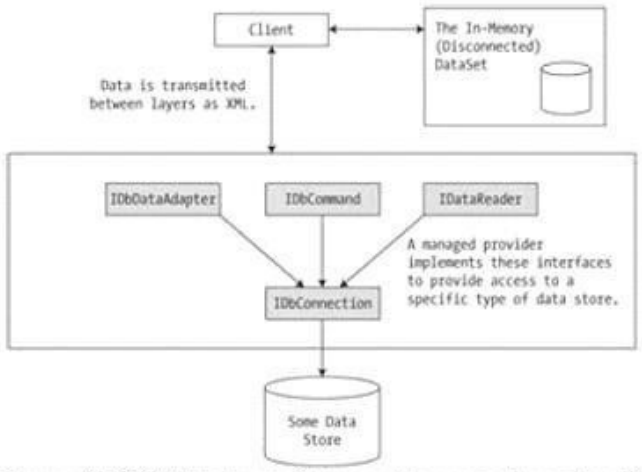


Figure : ADO.NET data providers provide access to a given DBMS.

The Role of the IDbConnection and IDbTransaction Interfaces : First we have the IDbConnection type, which is implemented by a data provider's connection object. This interface defines a set of members used to connect to (and disconnect from) a specific data store, as well as allowing us to obtain the data provider's transactional object, which implements the System.Data.IDbTransaction interface:

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

As we can see, the overloaded BeginTransaction() method provides access to an IDbTransaction-compatible type. Using the members defined by this interface, we are able to programmatically interact with a transactional session and the underlying data store:

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

The Role of the IDbCommand, IDbDataParameter, and IDataParameter Interfaces :

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 (note that each parameter object implements the IDbDataParameter type). In addition, command objects also provide access to the data provider's data reader type via the overloaded ExecuteReader() method:

```
public interface System.Data.IDbCommand : IDispos
        able { string CommandText { get; set; }
```

```
int CommandTimeout { get; set; } Comm
andType CommandType { get; set; } IDb
Connection Connection { get; set; } IData
ParameterCollection Parameters { get; } I
DbTransaction Transaction { get; set; }
UpdateRowSource UpdatedRowSource { get;
set; } void Cancel();
System.Data.IDbDataParameter CreateParamet
er(); int ExecuteNonQuery();
System.Data.IDataReader ExecuteReader();
System.Data.IDataReader ExecuteReader(System.Data.CommandBehavior behavi
or);
Object ExecuteScalar();
        void Prepare();
}
```

Notice that the Parameters property returns a strongly typed collection th at implements IDataParameterCollection. This interface provides access to a se t of IDbDataParametercompliant data types (e.g., parameter objects):

```
public interface System.Data.IDbDataParameter : System.Data.IDataPara
        meter { byte Precision { get; set; }
        byte Scale { get; se
        t; } int Size { get; se
        t; }
}
```

IDbDataParameter extends the IDataParameter interface to obtain the following additional behaviors :

```
public interface System.Data.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;
        }
}
```

The functionality of the IDbDataParameter and IDataParameter interfaces allow us to represent parameters within a SQL query (as well as stored procedures) via specific ADO.NET parameter objects, rather than hard-coded strings.

The Role of the IDbDataAdapter and IDataAdapter Interfaces :Recall that data adapters are used to push and pull DataSets to and from a given data store. Given this, IDbDataAdapter interface defines a set of properties that are used to maintain the SQL statements for the related SELECT, INSERT, UPDATE, and DELETE operations:

```
public interface System.Data.IDbDataAdapter : System.Data.IDataAd
        apter {

        IDbCommand DeleteCommand { get; set; }

        IDbCommand InsertCommand { get; se
        t; } IDbCommand SelectCommand { ge
        t; set;}

        IDbCommand UpdateCommand { get; set; }
}
```

In addition to these four properties, an ADO.NET data adapter also picks up the behavior defined in the base interface, IDataAdapter. This interface defines the key function of a data adapter type: the ability to push and pull DataSets between the caller and underlying data store

using the Fill() and Update() methods. Also, the IDataAdapter interface allows you to map database column names to a more human-readable display name via the TableMappings property:

```
public interface System.Data.IDataAdapter
{
        MissingMappingAction MissingMappingAction { get; set; }
        MissingSchemaAction MissingSchemaAction { get; set; }
         ITableMappingCollection TableMappings { get; }
        int Fill(System.Data.DataSet dataSet);

        System.Data.DataTable[] FillSchema(System.Data.DataSet data
        Set, System.Data.SchemaType schemaType); System.Data.IDat
        aParameter[] GetFillParameters(); int Update(System.Data.Data
        Set dataSet);
}
```

The Role of the IDataReader and IDataRecord Interfaces : The IDataReader interface, represents the common behaviors supported by a given data reader type. When we obtain an IDataReader-compatible type from an ADO.NET data provider, we are able to iterate over the result set using a forward-only, read-only manner.

```
public interface System.Data.IDataReader : IDispo
sable, System.Data.IDataRecord

{

        int Depth { get; } b
        ool IsClosed { get;
        }

        int RecordsAffected { ge
        t; } void Close();
        System.Data.DataTable GetSchemaTa
        ble(); bool NextResult();

        bool Read();

}
```

IDataReader extends IDataRecord, which defines an additional set of members that allow us to extract out 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 System.Data.IDataRecord
{

        int FieldCount { get; }

        object this[ string name ] { ge
        t; } object this[ int i ] { get; }

        bool GetBoolean(int
        i); byte GetByte(int
        i);

        long GetBytes(int i, long fieldOffset, byte[] buffe
        r, int bufferoffset, int length);

        char GetChar(int i);

        long GetChars(int i, long fieldoffset, char[] buffer,
        int bufferoffset, int length); System.Data.IDataRe
        ader GetData(int i); string GetDataTypeName(int
        i); DateTime GetDateTime(int i);

}
```

ADO.NET

ADO.NET is a set of classes (a framework) to interact with data sources such as databases and XML files. ADO is the acronym for ActiveX Data Objects. It allows us to connect to underlying data or databases. It has classes and methods to retrieve and manipulate data.

The following are a few of the .NET applications that use ADO.NET to connect to a database, execute commands and retrieve data from the database.

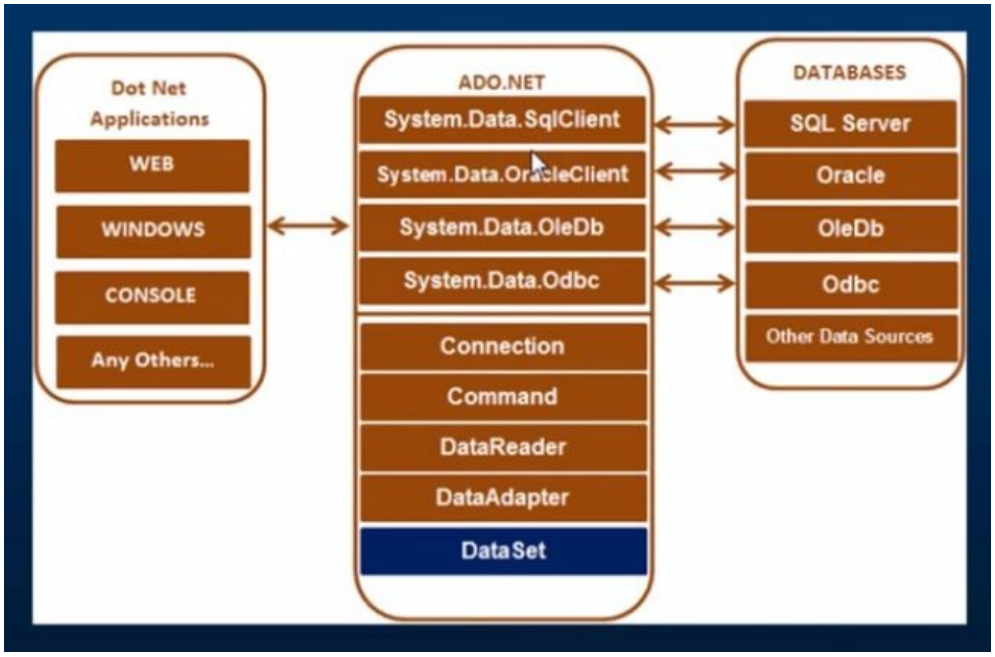
- ASP.NET Web Applications
- Console Applications
- Windows Applications.

Various Connection Architectures

There are the following two types of connection architectures:

1. **Connected architecture:** the application remains connected with the database throughout the processing.
2. **Disconnected architecture:** the application automatically connects/disconnects during the processing. The application uses temporary data on the application side called a DataSet.

Understanding ADO.NET and it's class library



In this diagram, we can see that there are various types of applications (Web Application, Console Application, Windows Application and so on) that use ADO.NET to connect to databases (SQL Server, Oracle, OleDb, ODBC, XML files and so on).

Important Classes in ADO.NET

We can also observe various classes in the preceding diagram. They are:

1. Connection Class
2. Command Class
3. DataReader Class
4. DataAdaptor Class
5. DataSet.Class

1. Connection Class

In ADO.NET, we use these connection classes to connect to the database. These connection classes also manage transactions and connection pooling. To learn more about connection classes, start here: Connection in ADO.NET.

Connection Object

- One of the first ADO.NET objects is the connection object, that allows you to establish a connection to a data source.
- The connection objects have the methods for opening and closing connections, for beginning a transaction of data.
- The .Net Framework provides two types of connection classes:
  The sqlconnection object, that is designed specially to connect to Microsoft SQL Server and the OleDbConnection object, that is designed to provide connection to a wide range of databases, such as Microsoft Access and Oracle.
- A connection is required to interact with the database. A Connection object helps to identify the database server name, user name and password to connect to the database. The Connection object is used by commands on the database.
- A Connection object has the responsibility of establishing a connection with the data store.
- How to use the Sqlconnection object:
  - Instantiate the SqlConnection class.
  - Open connection.
  - Pass the connection to ADO.NET objects.
  - Perform the database operations with ADO.NET object.
  - Close the connection.

Connection String

No.	Connection String Parameter Name	Description

1	Data Source	Identify the server. Could be a local machine, machine domain name, or IP Address.
2	Initial Catalog	Data base name.
3	Integrated Security	Set to SSIP to make a connection with the user's window log in.
4	User ID	Name of user configured in SQL Server.
5	Password	Password matching SQL Server User ID

The connection string is different for each of the various data providers available in .NET 2.0. There are different connection strings for the various types of data sources. You can find a list of all the available providers for creating a connection in a table:

No	Provider	Description
1	System.Data.SqlClient	Provides data for Microsoft SQL Server
2	System.Data.OleDb	Provides data access for data sources exposed using OLE DB
3	System.Data.Odbc	Provides data access for data source exposed using ODBC.
4	System.Data.OracleClient	Provides data access for Oracle.

2. Command Class

The Command class provides methods for storing and executing SQL statements and Stored Procedures. The following are the various commands that are executed by the Command Class.

- **ExecuteReader:** Returns data to the client as rows. This would typically be an SQL select statement or a Stored Procedure that contains one or more select statements. This method returns a DataReader object that can be used to fill a DataTable object or used directly for printing reports and so forth.
- **ExecuteNonQuery:** Executes a command that changes the data in the database, such as an update, delete, or insert statement, or a Stored Procedure that contains one or more of these statements. This method returns an integer that is the number of rows affected by the query.
- **ExecuteScalar:** This method only returns a single value. This kind of query returns a count of rows or a calculated value.
- **ExecuteXMLReader:** (SqlClient classes only) Obtains data from an SQL Server 2000 database using an XML stream. Returns an XML Reader object.

3. DataReader Class

The DataReader is used to retrieve data. It is used in conjunction with the Command class to execute an SQL Select statement and then access the returned rows. Learn more here: Data Reader in C#.

4. DataAdapter Class

The DataAdapter is used to connect DataSets to databases. The DataAdapter is most useful when using data-bound controls in Windows Forms, but it can

also be used to provide an easy way to manage the connection between your application and the underlying database tables, views and Stored Procedures. Learn more here: Data Adapter in ADO.NET.

5. DataSet Class

The DataSet is the heart of ADO.NET. The DataSet is essentially a collection of DataTable objects. In turn each object contains a collection of DataColumn and DataRow objects. The DataSet also contains a Relations collection that can be used to define relations among Data Table Objects.

ADO.NET :

The connected layer provides functionality for running SQL syntax against a connected database. The statements can be select, insert, update or delete as well as execute schema statements and the ability to run stored procedures and functions.

The connected layer requires the database connection to remain open while the database transactions are being performed.

Command Objects

The Command object represents a SQL statement to be run; select, insert, update, delete, execute schema, execute stored procedure etc.

The DbCommand is configured to execute an sql statement or stored procedure via the CommandText property.

The CommandType defines the type of sql statement to be executed which is stored in the CommandText property.

CommandType	Description
StoredProcedure	CommandText contains the name of a StoredProcedure or UserFunction.
TableDirect	CommandText contains a table name. All rows and columns will be returned from the table.
Text	CommandText defines an SQL statement to be executed.

The CommandType will default to Text however it is good practice to explicitly set it.

```
1 var connectionDetails =
```

```
ConfigurationManager.ConnectionStrings ["MyDatabase"];

var providerName = connectionDetails.ProviderName;
var connectionString = connectionDetails.ConnectionString;

var dbFactory = DbProviderFactories.GetFactory (providerName);

using(var cn = dbFactory.CreateConnection())
{
 cn.ConnectionString = connectionString;
 cn.Open();

 var cmd = cn.CreateCommand();

 cmd.CommandText = "Select * from MyTable";
 cmd.CommandType = CommandType.Text;
 cmd.Connection = cn;
}
```

The command should be passed the connection via the Connection property.

The command object will not touch the database until the either the ExecuteReader, ExecuteNonQuery or equivalent has been called upon.

DataReader

The DataReader class allows a readonly iterative view of the data returned from a configured command object.

A DataReader is instantiated by calling ExecuteReader upon the command object.

The DataReader exposes a forward only iterator via the Read method which returns false when the collection has been completely iterated.

The DataReader represents both the collection of records and also the individual records; access to the data is made by calling methods upon the DataReader.

DataReader implements IDisposable and should be used within a using scope.

Data can be accessed by field name and also ordinal position via the index method[] which returns the data cast into object. Alternatively methods exist to get the data for any .NET primitive type.

```
using (var dr = cmd.ExecuteReader ()) {
while (dr.Read ()) {
var val1 = (string)dr ["FieldName"];
var val2 = (int)dr [0];

// Get the field name or its ordinal position
var name = dr.GetName (1);

var pos = dr.GetOrdinal ("FieldName");

// Strongly Types Data
var val3 = dr.GetInt32 (1);
var val4 = dr.GetDecimal (2);
var val5 = dr.GetDouble (3);
var val6 = dr.GetString (4);

var isNull = dr.IsDBNull (5);

var fdType = dr.GetProviderSpecificFieldType (6);
}
}
```

The IsDBNull method can be used to determine if a value contains a null. A .NET type representation of the contained field can be retrieved with the GetProviderSpecificFieldType method.

Multiple Results

If the select statement of a command has multiple results the DataReader.NextResult() method exposes a forward only iterator through each result group:

```
string strSQL = "Select * From MyTable;Select * from MyOtherTable";

NextResult increments to the next result group, just like Read it returns false on ce the collection has been exhausted:

using (var dr = cmd.ExecuteReader ()) {
while (dr.NextResult ()) {
while (dr.Read ()) {
}
}
}
```

DataSet

DataReader can be used to fill a DataSet. We talk more about DataSet within the disconnected layer in the next chapter.

```
var dtable = new DataTable();

using(var dr = cmd.ExecuteReader())
{
dtable.Load(dr);
}
```

ExecuteNonQuery

ExecuteNonQuery allows execution of an insert, update or delete statement as well as executing a schema statement.

The method returns an integral representing the number of affected rows.

```
using (var cn = dbFactory.CreateConnection ()) {
cn.ConnectionString = connectionString;
cn.Open ();

var cmd = cn.CreateCommand ();

cmd.CommandText = @"Insert Into MyTable (FieldA) Values ('Hello')";
cmd.CommandType = CommandType.Text;
cmd.Connection = cn;

var count = cmd.ExecuteNonQuery ();
}
```

Command Parameters

Command parameters allow configuring stored procedure parameters as well as parameterized sql statements which can help protect against SQL injection attacks.

It is strongly advised that any information collected from a user should be sent to the database as parameter regardless if it is to be persisted or used within a predicate.

Parameters can be added with the Command.Parameters.AddWithValue or by instantiating a DbParameter class.

The example below shows the former.

```
using (var cn = dbFactory.CreateConnection ()) {
cn.ConnectionString = connectionString;
cn.Open ();

var cmd = cn.CreateCommand ();

cmd.CommandText = @"Insert Into MyTable (FieldA) Values (@Hello)";
cmd.CommandType = CommandType.Text;

var param = cmd.CreateParameter ();

param.ParameterName = "@Hello";
param.DbType = DbType.String;
param.Value = "Value";
param.Direction = ParameterDirection.Input;
cmd.Parameters.Add (param);

cmd.Connection = cn;

var count = cmd.ExecuteNonQuery ();
```

```
}
```

The DbCommand has a DbType property which allows setting the type of the parameter, it is vendor agnostic.

SqlCommand and MySqlCommand also provide SqlDbType and MySqlDbType which can be used to set the type field from a vendor specific enum. Setting the DbType will maintain the vendor specific column and vice versa.

Connection Object and Connection string

Connection Object

1. One of the first ADO.NET objects is the connection object, that allows you to establish a connection to a data source.
2. The connection objects have the methods for opening and closing connections, for beginning a transaction of data.
3. The .Net Framework provides two types of connection classes:
   The sqlconnection object, that is designed specially to connect to Microsoft SQL Server and the OleDbConnection object, that is designed to provide connection to a wide range of databases, such as Microsoft Access and Oracle.
4. A connection is required to interact with the database. A Connection object helps to identify the database server name, user name and password to connect to the database. The Connection object is used by commands on the database.
5. A Connection object has the responsibility of establishing a connection with the data store.
6. How to use the Sqlconnection object:
   - Instantiate the SqlConnection class.
   - Open connection.
   - Pass the connection to ADO.NET objects.
   - Perform the database operations with ADO.NET object.
   - Close the connection.

Connection String

No.	Connection String Parameter Name	Description
1	Data Source	Identify the server. Could be a local machine, machine domain name, or IP Address.
2	Initial Catalog	Data base name.
3	Integrated Security	Set to SSIP to make a connection with the user's window log in.
4	User ID	Name of user configured in SQL Server.
5	Password	Password matching SQL Server User ID

The connection string is different for each of the various data providers available in .NET 2.0. There are different connection strings for the various types of data sources. You can find a list of all the available providers for creating a connection in a table:

No	Provider	Description
1	System.Data.SqlClient	Provides data for Microsoft SQL Server

2	System.Data.OleDb	Provides data access for data sources exposed using OLE DB
3	System.Data.Odbc	Provides data access for data source exposed using ODBC.
4	System.Data.OracleClient	Provides data access for Oracle.

Executing a Stored Procedure

A stored procedure is executed by configuring a DbCommand against the name of the stored procedure along with any required parameters followed by calling ExecuteScalar or ExecuteReader.

ExecuteScalar is used to return a single value. If multiple result sets, rows and columns are returned it will return the first column from the first row in the first result set.

Parameters can either be input or output.

```
using(var cn = dbFactory.CreateConnection())
{
 cn.ConnectionString = connectionString;
 cn.Open();

 var cmd = cn.CreateCommand();
 cmd.CommandText = "spGetFoo";
 cmd.Connection = cn;

 cmd.CommandType = CommandType.StoredProcedure;

// Input param.
 var paramOne = cmd.CreateParameter();
  paramOne.ParameterName = "@inParam";
  paramOne.DbType = DbType.Int32;
  paramOne.Value = 1;
  paramOne.Direction = ParameterDirection.Input;
  cmd.Parameters.Add(paramOne);

// Output param.
 var paramTwo = cmd.CreateParameter();
  paramTwo.ParameterName = "@outParam";
  paramTwo.DbType = DbType.String;
  paramTwo.Size = 10;
  paramTwo.Direction = ParameterDirection.Output;
  cmd.Parameters.Add(paramTwo);

// Execute the stored proc.
var count = cmd.ExecuteScalar();

// Return output param.
var outParam = (int)cmd.Parameters["@outParam"].Value;

// This can be made on the parameter directly
 var outParam2 = (int)paramTwo.Value;
 }
```

Member name	Description
Input	The parameter is an input parameter (default).
InputOutput	The parameter is capable of both input and output.
Output	The parameter is an output parameter and has be suffixed with the out keyword in the parameter list of a stored procedure, built in function or user defined function.
ReturnValue	The parameter is a return value, scalar or similar but not a return set. This is determined by the return keyword in a stored procedure, built in function or user defined function.

If the stored procedure returns a set and not a single value the ExecuteReader can be used to iterate over the result:

```
using (var cn = dbFactory.CreateConnection ()) {
cn.ConnectionString = connectionString;
cn.Open ();

var cmd = cn.CreateCommand ();
cmd.CommandText = "spGetFoo";
cmd.Connection = cn;

cmd.CommandType = CommandType.StoredProcedure;

using (var dr = cmd.ExecuteReader ()) {
while (dr.NextResult ()) {
while (dr.Read ()) {
}
}
}
}
```

Database Transactions

When the writable transaction of a database involves more than one writable actions, it is essential that all the actions are wrapped up in a unit of work called a database transaction.ACID

The desired characteristics of a transaction are defined by the term ACID:

Member name	Description
Atomicity	All changes within a unit of work complete or none complete; they are atomic.
Consistency	The state of the data is consistent and valid. If upon completing all of the changes the data is considered invalid, all the changes are undone to return the data to the original state.
Isolation	All changes within a unit of work occur in isolation from other readable and writable transactions. No one can see any changes until all changes have been completed in full.
Durability	Once all the changes within a unit of work have completed, all the changes will be persisted.

No one can see any changes until all changes have been completed in full.
Durability
Once all the changes within a unit of work have completed, all the changes will be persisted.

Syntax

A transaction is started through the connection object and attached to any command which should be run in the same transaction.

Commit should be called upon the transaction upon a successful completion while Rollback should be called if an error occured. This can be achieved by a try catch statement:

```
using (var cn = dbFactory.CreateConnection ()) {
cn.ConnectionString = connectionString;
cn.Open ();

var cmdOne = cn.CreateCommand ();
cmdOne.CommandText = "spUpdateFoo";
cmdOne.Connection = cn;
cmdOne.CommandType = CommandType.StoredProcedure;

var cmdTwo = cn.CreateCommand ();
cmdTwo.CommandText = "spUpdateMoo";
cmdTwo.Connection = cn;
cmdTwo.CommandType = CommandType.StoredProcedure;

var tran = cn.BeginTransaction ();

try {

cmdOne.Transaction = tran;
cmdTwo.Transaction = tran;

cmdOne.ExecuteNonQuery ();
cmdTwo.ExecuteNonQuery ();

tran.Commit ();
} catch (Exception ex)
    {
     tran.Rollback ();
```

```
}
}
```

The DbTransaction class implements IDisposable and can therefore be used within a using statement rather than a try catch. The Dispose method will be called implicitly upon leaving the scope of the using statement; this will cause any uncommitted changes to be rolled back while allowing any committed changes to remain as persisted. This is the preferred syntax for writing transactions in ADO.NET:

```
var connectionDetails =
ConfigurationManager.ConnectionStrings ["MyDatabase"];

var providerName = connectionDetails.ProviderName;
var connectionString = connectionDetails.ConnectionString;

var dbFactory = DbProviderFactories.GetFactory (providerName);

using (var cn = dbFactory.CreateConnection ()) {
cn.ConnectionString = connectionString;
cn.Open ();

var cmdOne = cn.CreateCommand ();
cmdOne.CommandText = "spUpdateFoo";
cmdOne.Connection = cn;
cmdOne.CommandType = CommandType.StoredProcedure;

var cmdTwo = cn.CreateCommand ();
cmdTwo.CommandText = "spUpdateMoo";
cmdTwo.Connection = cn;
cmdTwo.CommandType = CommandType.StoredProcedure;

using (var tran = cn.BeginTransaction ()) {

cmdOne.Transaction = tran;
cmdTwo.Transaction = tran;

cmdOne.ExecuteNonQuery ();
cmdTwo.ExecuteNonQuery ();

tran.Commit ();
{
}
}
}
```

Working with the Connected Layer of ADO.NET :

The first step to take when working with the OleDb data provider is to establish a session with the data source using the OleDbConnection type (which, as we recall, implements the IDbConnection interface). Much like the classic ADO Connection object, OleDbConnection types are provided with a formatted connection string, containing a number of name/value pairs separated by semicolons. This information is used to identify the name of the machine we wish to connect to, required security settings, the name of the database on that machine, and, most

importantly, the name of the OLE DB provider. (See online help for a full description of each name/value pair.)

The connection string may be set using the OleDbConnection.ConnectionString property or including it as a constructor argument. Assume we wish to connect to the Cars database on our local machine using the SQL OLE DB provider. The following logic does the trick:

```
// Build a connection string.
 OleDbConnection cn = new OleDbConnection();
 cn.ConnectionString = "Provider=SQLOLEDB.1;" + "User ID=sa;
Pwd=;
Initial Catalog=Cars;" + "Data Source=(local);";
```

As we can conclude from the preceding code comments, the Initial Catalog name refers to the database we are attempting to establish a session with (Pubs, Northwind, Cars, and so on). The Data Source name identifies the name of the machine that maintains the database (for simplicity, I have assumed no specific password is required for local system administrators). The final point of interest is the Provider segment, which specifies the name of the OLE DB provider that will be used to access the data store. Below Table describes some possible values.

Table 17-13: Core OLE DB Providers

Provider Segment Value	Meaning in Life
Microsoft.JET.OLEDB.4.0	You want to use the Jet OLE DB provider to connect to an Access database.
MSDAORA	You want to use the OLE DB provider for Oracle.
SQLOLEDB	You want to use the OLE DB provider for MS SQL Server. Once you have configured the connection string, the next step is to open a session with the data source, do some work, and release your connection to the data source, as shown in the code snippet following this table.

```
// Build a connection string.
OleDbConnection cn = new OleDbConnection
(); cn.ConnectionString = "Provider=SQLOLE
DB.1;" + "User ID=sa;Pwd=;Initial Catalog=Ca
rs;" + "Data Source=(local);";

cn.Open();

 ----

cn.Close();
```

In addition to the ConnectionString, Open(), and Close() members, the OleDbConnection class provides a number of members that let us configure attritional

settings regarding our connection, such as timeout settings and transactional information. Below table gives a partial rundown.

Table 17-14: Members of the OleDbConnection Type

OleDbConnection Member	Meaning in Life
BeginTransaction() CommitTransaction() RollbackTransaction()	Used to programmatically commit, abort, or roll back a transaction.
Close()	Closes the connection to the data source.
ConnectionString	Gets or sets the string used to open a session with a data store.
ConnectionTimeout	This read-only property returns the amount of time to wait while establishing a connection before terminating and generating an error (the default value is 15 seconds). If you wish to change this default, specify a "Connect Timeout" segment in the connection string (e.g., Connect Timeout=30).
Database	Gets the name of the database maintained by the connection object.
DataSource	Gets the location of the database maintained by the connection object.
Open()	Opens a database connection with the current property settings.
GetOleDbSchemaTable()	Obtains schema information from the data source.
Provider	Gets the name of the provider maintained by the connection object.
State	Gets the current state of the connection, represented by the ConnectionState enumeration.

As we can see, the properties of the OleDbConnection type are typically read-only in nature, and are only useful when we wish to obtain the characteristics of a connection at runtime. When we wish to override default settings, we must alter the construction string itself. For example, consider the following code, which changes the default connection timeout setting from 15 seconds to 30 seconds (via the Connect Timeout segment of the connection string) :

```
OleDbConnection cn = new OleDbConnection();
cn.ConnectionString = "Provider=SQLOLEDB.1;" + "User ID=sa;Pwd=;Ini
tial Catalog=Cars;" + "Data Source=(local);
cn.Connection Timeout=30";
cn.Open();
Console.WriteLine("***** Info about your connection *****"); Console.W
riteLine("Database location: {0}", cn.DataSource); Console.WriteLine
("Database name: {0}", cn.Database); Console.WriteLine("Provider:
{0}", cn.Provider);
Console.WriteLine("Timeout: {0}", cn.ConnectionTimeout); Console.Writ
eLine("Connection state: {0}",  cn.State.ToString());
cn.Close();
Console.WriteLine("Connection state: {0}", cn.State.ToString());
```

Notice that this connection is explicitly opened and closed each time before making a call to the State property. As mentioned in the previous table, this property may take any value of the ConnectionState enumeration:

```
public enum System.Data.ConnectionState
{
        Broken, Closed, Co
        nnecting, Executing,
        Fetching, Open

}
```

Connecting to an Access Database : Much like classic ADO, the process of connecting to an Access database using ADO.NET requires little more than retrofitting our construction string. First, set the Provider segment to the JET engine, rather than SQLOLEDB. Beyond this adjustment, set the data source segment to point to the path of your *.mdb file, as shown here:

```
// Be sure to update the data source segment if necessary!
OleDbConnection cn = new OleDbConnection();
 cn.ConnectionString = "Provider=Microsoft.JET.OLEDB.4.0;" + @"data source = C:\car
s.mdb";
cn.Open();
```

Once the connection has been made, we can read and manipulate the contents of our data table.

Command Object

- A Command object executes SQL statements on the database. These SQL statements can be SELECT, INSERT, UPDATE, or DELETE. It uses a connection object to perform these actions on the database.
- A Connection object specifies the type of interaction to perform with the database, like SELECT, INSERT, UPDATE, or DELETE.
- A Command object is used to perform various types of operations, like SELECT, INSERT, UPDATE, or DELETE on the database.
- **SELECT**
    1. cmd =new SqlCommand("select * from Employee", con);
- **INSERT**

```
1. cmd = new SqlCommand("INSERT INTO Employee(Emp_ID,
2. Emp_Name)VALUES ('" + aa + "','" + bb + "')", con);
```

- **UPDATE**

```
1. SqlCommand cmd =new SqlCommand("UPDATE Employee SET
2. Emp_ID ='" + aa + "', Emp_Name ='" + bb + "' WHERE
3. Emp_ID = '" + aa + "'", con);
```

- **DELETE**

```
1. cmd =new SqlCommand("DELETE FROM Employee where
2. Emp_ID='" + aa + "'", con);
```

- A Command object exposes several execute methods like:

1. **ExecuteScaler()**
   Executes the query, and returns the first column of the first row in the result set returned by the query. Extra columns or rows are ignored.
2. **ExecuteReader()**
   Display all columns and all rows in the client-side environment.
   In other words, we can say that they display datatables client-side.
3. **ExecuteNonQuery()**
   Something is done by the database but nothing is returned by the database.

3. Data Reader Object

A DataReader object is used to obtain the results of a SELECT statement from a command object. For performance reasons, the data returned from a data reader is a forward-only stream of data. This means that the data can be accessed from the stream in a sequential manner. This is good for speed, but if data needs to be manipulated then a dataset is a better object to work with.

Example

```
1. dr = cmd.ExecuteReader();
2. DataTable dt = new DataTable();
3. dt.Load(dr);
```

- It is used in Connected architecture.
- Provide better performance.
- DataReader Object has Read-only access.
- DataReader Object Supports a single table based on a single SQL query of one database.
- While DataReader Object is Bind to a single control.
- DataReader Object has Faster access to data.
- DataReader Object Must be manually coded.
- we can't create a relation in the data reader.
- whereas Data reader doesn't support.
- The data reader communicates with the command object.
- DataReader can not modify data.

4. Data Adapter Object

- A Data Adapter represents a set of data commands and a database connection to fill the dataset and update a SQL Server database.
- A Data Adapter contains a set of data commands and a database connection to fill the dataset and update a SQL Server database. Data Adapters form the bridge between a data source and a dataset.
- Data Adapters are designed depending on the specific data source. The following table shows the Data Adapter classes with their data source.

Provider-Specific Data Adapter classes	Data Source
SqlDataAdapter	SQL Server
OledbDataAdapter	OLE DB provider
OdbcDataAdapter	ODBC driver
OracleDataAdapter	Oracle

Provider-Specific Data Adapter classes

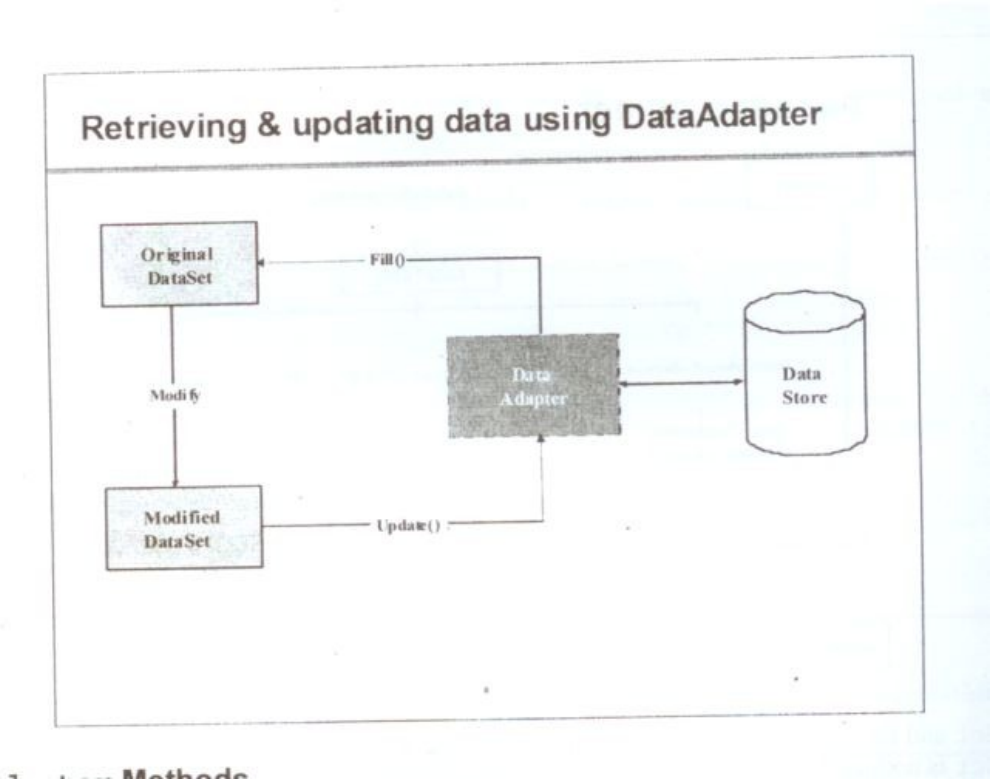
Data Source

- A Data Adapter object accesses data in a disconnected mode. Its object contains a reference to a connection object.
- It is designed in a way that implicitly opens and closes the connection whenever required.
- It maintains the data in a DataSet object. The user can read the data if required from the dataset and write back the changes in a single batch to the database. Additionally, the Data Adapter contains a command object reference for SELECT, INSERT, UPDATE, and DELETE operations on the data objects and a data source.
- A Data Adapter supports mainly the following two methods:

    - **Fill ()**
      The Fill method populates a dataset or a data table object with data from the database. It retrieves rows from the data source using the SELECT statement specified by an associated select command property.
      The Fill method leaves the connection in the same state as it encountered it before populating the data. If subsequent calls to the method for refreshing the data are required then the primary key information should be present.
    - **Update ()**
      The Update method commits the changes back to the database. It also analyzes the RowState of each record in the DataSet and calls the appriopriate INSERT, UPDATE, and DELETE statements.
      A Data Adapter object is formed between a disconnected ADO.NET object and a data source.

Example

```
1. SqlDataAdapter da=new SqlDataAdapter("Select * from Employee", con);
2. da.Fill(ds,"Emp");
3. bldr =new SqlCommandBuilder(da);
4. dataGridView1.DataSource = ds.Tables["Emp"];
```



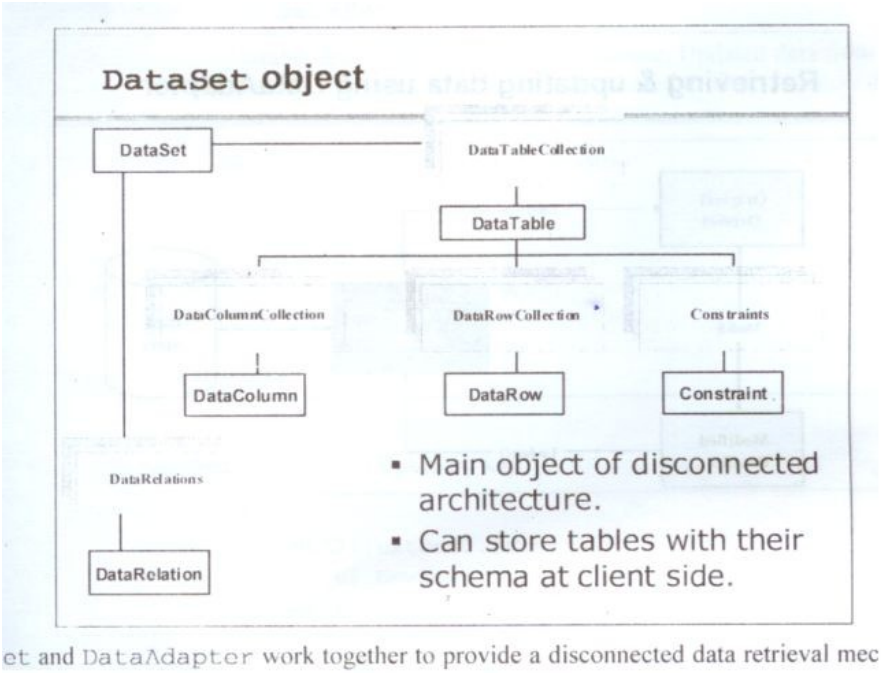
5. DataSet Object

- In the disconnected scenario, the data retrieved from the database is stored in a local buffer called DataSet. It is explicitly designed to access data from any data source. This class is defined in the System.Data namespace.
- A Data Set object is an in-memory representation of the data. It is specially designed to manage data in memory and to support disconnected operations on data.
- A Data Set is a collection of DataTable and DataRelations. Each DataTable is a collection of DataColumn, DataRows, and Constraints.

- A DataTable, DataColumn, and DataRows could be created as follows.

Example

```
1. DataTable dt = new DataTable();
2. DataColumn col =new DataColumn();
3. Dt.columns.Add(col2);
4. DataRow row = dt.newRow();
```



et and DataAdapter work together to provide a disconnected data retrieval mec

- It is used in a disconnected architecture.
- Provides lower performance.
- A DataSet object has read/write access.
- A DataSet object supports multiple tables from various databases.
- A DataSet object is bound to multiple controls.
- A DataSet object has slower access to data.
- A DataSet object is supported by Visual Studio tools.
- We can create relations in a dataset.
- A Dataset supports integration with XML.
- A DataSet communicates with the Data Adapter only.
- A DataSet can modify data.

6. Command Builder Object

- Automatically generates insert, update, delete queries using the SelectCommand property of a DataAdapter.
- A Command Builder Object is used to build commands for data modification from objects based on a single table query. CommandBuilders are designed depending on the specific data source. The following table shows the CommandBuilder classes with their data source.

Provider-Specific Data Adapter classes	Data Source
SqlDataAdapter	SQL Server

OledbDataAdapter	OLE DB provider
OdbcDataAdapter	ODBC driver
OracleDataAdapter	Oracle

Example

```
1. da = new SqlDataAdapter("Select * from Employee", con);
2. ds = new DataSet();
3. da.MissingSchemaAction = MissingSchemaAction.AddWithKey;
4. da.Fill(ds, "Emp");
5. bldr = new SqlCommandBuilder(da);
6. dataGridView1.DataSource = ds.Tables["Emp"];
```

Differences Between DataReader and DataSet

No	Data Reader	DataSet
1	Used in a connected architecture	used in a disconnected architecture
2	Provides better performance	Provides lower performance
3	DataReader object has read-only access	A DataSet object has read/write access
4	DataReader object supports a single table based on a single SQL query of one database	A DataSet object supports multiple tables from various databases
5	A DataReader object is bound to a single control	A DataSet object is bound to multiple controls
6	A DataReader object has faster access to data	A DataSet object has slower access to data
7	A DataReader object must be manually coded	A DataSet object is supported by Visual Studio tools
8	We can't create a relation in a data reader	We can create relations in a dataset
9	Whereas a DataReader doesn't support data reader communicates with the command object.	A Dataset supports integration with XML Dataset communicates with the Data Adapter only
10	DataReader cannot modify data	A DataSet can modify data

DataView Object

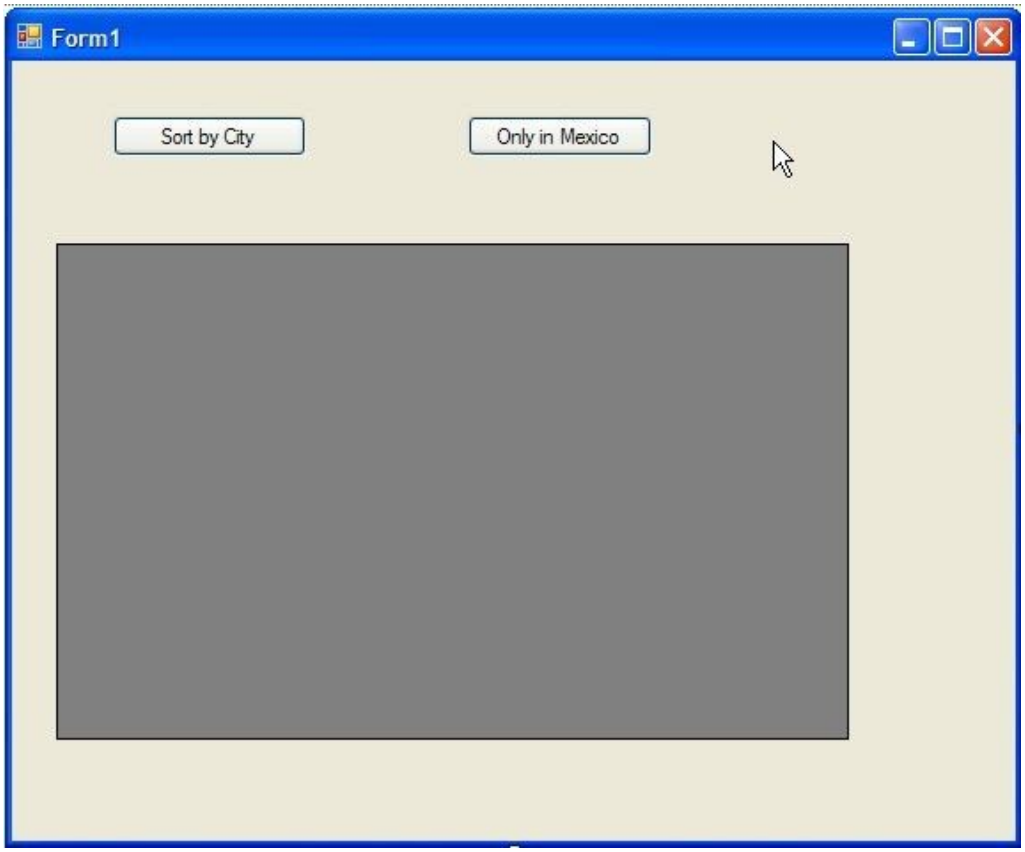
A DataView is the same as a read-only mini-dataset.

- You typically load only a subset into a DataView.
- A DataView provides a dynamic view of data. It provides a datarow using the DataView.

Example

Add two buttons and a DataGridView control. Change the text of the first button to sort by city and that of button2 to only select records in Mexico and add the code in form.cs.

Form Design



Coding Part

```
1. SqlConnection con;
2. SqlCommand cmd;
3. SqlDataReader dr;
4. public DataTable GetTable()
5. {
6.     con = new SqlConnection("Data Source=.\\sqlexpress;Initial
   Catalog=information;Integrated Security=True;Pooling=False");
7.     con.Open();
8.     cmd = new SqlCommand("select * from Customers", con);
9.     dr = cmd.ExecuteReader();
10.    DataTable dt = new DataTable();
11.    dt.Load(dr);
12.    dataGridView1.DataSource = dt;
13.    con.Close();
14.    return dt;
15. }
```

Form1_Load

```
1. dataGridView1.DataSource = GetTable().DefaultView;
```

ShortByCity_Click

```
1. DataView dv = new DataView(GetTable());
2. dv.Sort = "City ASC";
3. dataGridView1.DataSource = dv;
```

OnlyInMexico_Click

```
1. DataView dv = new DataView(GetTable());
2. dv.RowFilter = "Country = 'Mexico'";
3. dataGridView1.DataSource = dv;
```

- At the click of the sort by city button, the data already in the DataGridView control is sotred dy city.
- On clicking the second button, only the records in Mexico are displayed in the DataGridView control. The output after clicking the only in Mexico button is as the Mexico button.

Inserting, Updating, and Deleting Records Using OleDbCommand :

The ExecuteReader() method allows us to examine the results of a SQL SELECT statement using a forward-only, read-only flow of information. However, when we wish to submit SQL commands that result in the modification of a given table, we make use of the OleDbCommand.ExecuteNonQuery() method. This single method will perform inserts, updates, and deletes based on the format of our command text. Be very aware that when we are making use of the OleDbCommand.ExecuteNonQuery() method, we are operating within the connected layer of ADO.NET, meaning this method has nothing to do with obtaining populated DataSet types via a data adapter.

To illustrate modifying a data source via ExecuteNonQuery(), assume we wish to insert a new record into the Inventory table of the Cars database. Once we have configured our connection type, the remainder of our task is as simple as authoring the correct SQL:

```
class UpdateWithCommandObj
{
static void Main(string[] args)
{
// Open a connection to Cars
db. OleDbConnection cn = new OleDbConnection();
```

```
cn.ConnectionString = "Provider=SQLOLEDB.1;" + "User ID=sa;Pwd=;Initial C
atalog=Cars;" + "Data Source=(local);
Connect Timeout=30";
cn.Open();
```

```
// SQL INSERT statement.
string sql = "INSERT INTO Inventory" + "(CarID, Make, Color, PetName) VAL
UES" + "('777', 'Honda', 'Silver', 'NoiseMaker')";

// Insert the record.
OleDbCommand cmd = new OleDbCommand(sql, cn);
 try
{
        cmd.ExecuteNonQuery();
} catch(Exception ex)
{
   Console.WriteLine(ex.Message); }
   cn.Close();
}
}
```

Updating or removing a record :

```
// UPDATE existing record.
sql = "UPDATE Inventory SET Make = 'Hummer' WHERE CarID = '777'";
 cmd.CommandText = sql;
try
{
   cmd.ExecuteNonQuery();
}
catch(Exception ex)
{
   Console.WriteLine(ex.Message);
}
```

// DELETE a record.

```
sql = "Delete from Inventory where CarID = '777'";
cmd.CommandText = sql;
try
{
   cmd.ExecuteNonQuery();
}
 catch(Exception ex)
{
   Console.WriteLine(ex.Message);
}
```

Although you do not bother to obtain the value returned from the ExecuteNonQuery() method, do understand

that this member returns a System.Int32 that represents the number of affected rec ords:

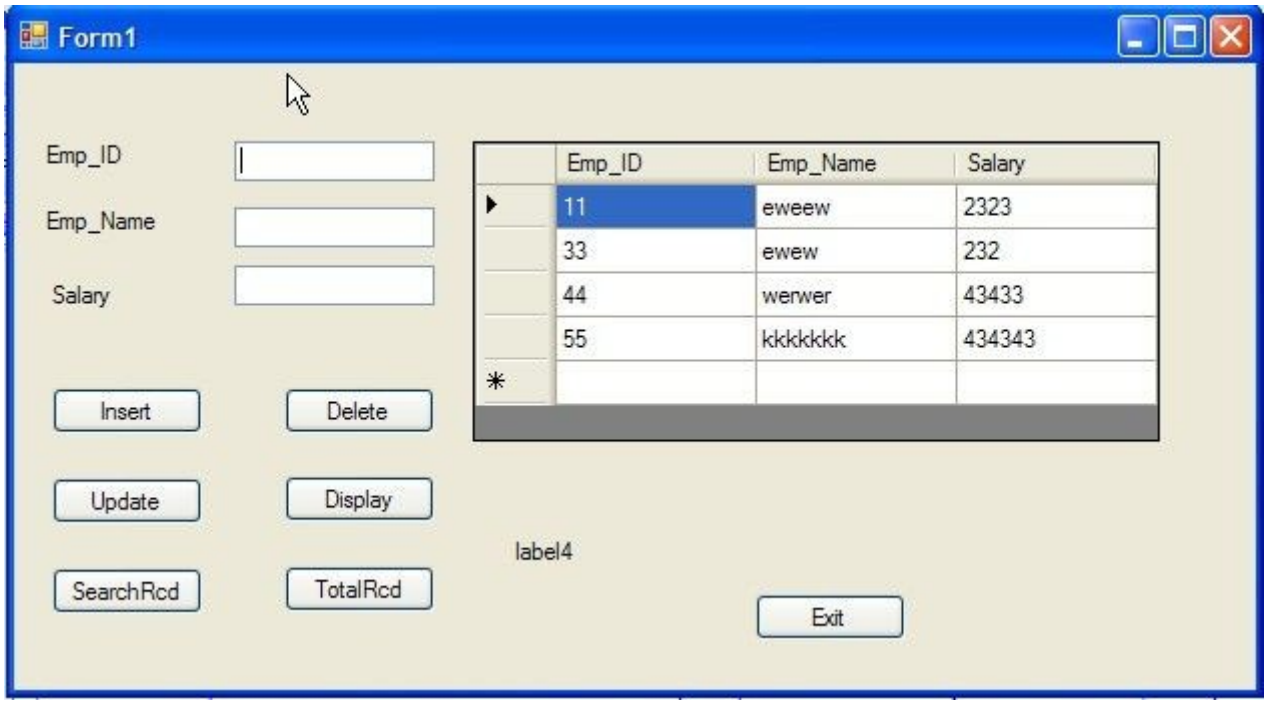
```
try
{
Console.WriteLine("Number of rows effected: {0}", cmd.ExecuteNonQuery());
}
```

Working With System.Data.SqlClient and System.Data.Oledb

Design a Simple Winform for accepting the details of an Employee. Using the connected architecture of ADO.NET, perform the following operations:

- Insert record.
- Search record.
- Update record.
- Delete record.

Form Design



Coding Part

Step 1: add namespace using System.Data.Oledb; for access Database.

Step 2: Create connection object.

1. OleDbConnection con;
2. OleDbCommand cmd;

3. OleDbDataReader dr;

Step 3: Form1_Load

```
1. con = new OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\\Documents and Settings\\Admin\\Desktop\\Ado Connected Demo\\db1.mdb");
2. private void display()
3. con.Open();
4. cmd = new OleDbCommand("select * from Employee",con);
5. dr = cmd.ExecuteReader();
6. DataTable dt = new DataTable();
7. dt.Load(dr);
8. dataGridView1.DataSource = dt;
9. con.Close();
```

Display_Click

```
1. display();
```

Insert_Click

```
1. con.Open();
2. int aa = Convert.ToInt32(textBox1.Text);
3. string bb = textBox2.Text;
4. int cc = Convert.ToInt32(textBox3.Text);
5. cmd = new OleDbCommand("INSERT INTO Employee(Emp_ID, Emp_Name,Salary) VALUES ('" + aa + "','" + bb + "','" + cc + "')", con);
6. cmd.ExecuteNonQuery();
7. MessageBox.Show("one record inserted:");
8. con.Close();
9. display();
```

Delete_Click

```
1. con.Open();
2. int aa = Convert.ToInt32(textBox1.Text);
3. cmd = new OleDbCommand("DELETE FROM Employee where Emp_ID='" + aa + "'", con);
4. cmd.ExecuteNonQuery();
5. MessageBox.Show("one record Delete:");
6. con.Close();
7. display();
```

update_Click

```
1. con.Open();
2. int aa = Convert.ToInt32(textBox1.Text);
3. string bb = textBox2.Text;
4. int cc = Convert.ToInt32(textBox3.Text);
5. string abc = "UPDATE Employee SET Emp_ID ='" + aa + "', Emp_Name ='" + bb +
   "', Salary ='" + cc + "' WHERE Emp_ID = '" + aa + "'";
6. OleDbCommand cmd = new OleDbCommand(abc, con);
7. cmd.ExecuteNonQuery();
8. MessageBox.Show("one record updated:");
9. con.Close();
10. display();
```

Find_Click

```
1. con.Open();
2. int aa = Convert.ToInt32(textBox1.Text);
3. string abc = "SELECT Emp_ID,Emp_Name,Salary FROM Employee where Emp_ID
   ='" + aa + "'";
4. cmd = new OleDbCommand(abc, con);
5. MessageBox.Show("one record search:");
6. dr = cmd.ExecuteReader();
7. DataTable dt = new DataTable();
8. dt.Load(dr);
9. dataGridView1.DataSource = dt;
10. con.Close();
```

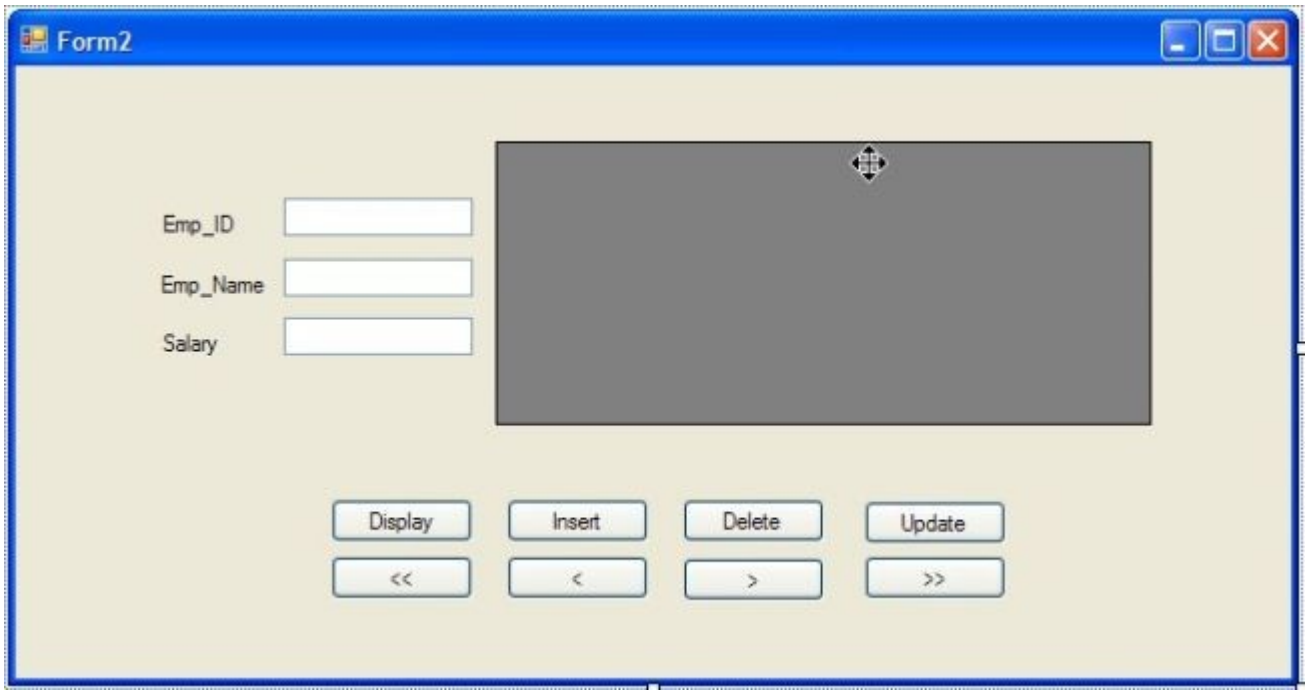
Exit_Click

```
Application.Exit();
```

Program: Design a simple Winform for accepting the details of an Employee. Using the disc onnected architecture of ADO.NET, perform the following operations.

- Insert record.
- Display record.
- Update record.
- Delete record

Form Design



Coding Part

Step 1: add namespace using System.Data.SqlClient;for SQL dataBase.

Step 2: Create connection object.

```
1. DataSet ds;
2. SqlDataAdapter da;
3. SqlConnection con;
4. SqlCommandBuilder bldr;
```

Step 3: Form1_Load

```
1. con = new SqlConnection("Data Source=.\\sqlexpress;Initial Catalog=information;Int
   egratedSecurity=True;Pooling=False");
2. private void display()
3. da = new SqlDataAdapter("Select * from Employee", con);
4. ds = new DataSet();
5. da.MissingSchemaAction = MissingSchemaAction.AddWithKey;
6. da.Fill(ds, "Emp");
7. bldr = new SqlCommandBuilder(da);
8. dataGridView1.DataSource = ds.Tables["Emp"];
```

Display_Click

```
1. display();
```

Insert_Click

```
1. DataRow drnew = ds.Tables["Emp"].NewRow();
2. drnew[0] = textBox1.Text;
3. drnew[1] = textBox2.Text;
4. drnew[2] = textBox3.Text;
5. ds.Tables["Emp"].Rows.Add(drnew);
6. da.Update(ds, "Emp");
7. MessageBox.Show("Record added");
8. dataGridView1.DataSource = ds.Tables["Emp"];
```

Delete_Click

```
1. DataRow row = ds.Tables["Emp"].Rows.Find(Convert.ToInt32(textBox1.Text));
2. row.Delete();
3. da.Update(ds, "Emp");
4. MessageBox.Show("Record Deleted");
5. dataGridView1.DataSource = ds.Tables["Emp"];
```

Update_Click

```
1. DataRow dr = ds.Tables[0].Rows.Find(textBox1.Text);
2. dr["Emp_Name"] = textBox2.Text;
3. dr["Salary"] = textBox3.Text;
4. da.Update(ds, "Emp");
5. MessageBox.Show("updated..");
6. dataGridView1.DataSource = ds.Tables[0];
```

FirstRcd_Click

```
1. this.BindingContext[ds.Tables[0]].Position = 0;
```

LastRcd_Click

```
1. this.BindingContext[ds.Tables[0]].Position = ds.Tables[0].Rows.Count - 1;
```

NextRcd_Click

```
1. if (this.BindingContext[ds.Tables[0]].Position > 0)
2. {
3.   this.BindingContext[ds.Tables[0]].Position -= 1;
4. }
```

PreviousRcd_Click

```
1. if (this.BindingContext[ds.Tables[0]].Position < ds.Tables[0].Rows.Count - 1)
2. {
3.   this.BindingContext[ds.Tables[0]].Position += 1;
4. }
```

This example shows how to insert ,update, delete and select data in Access File(CRUD Operations).

-
- Then, we can use System.Data API.
- The database in the sample file contains a table that has a name “TABLE1”.

TABLE1:

ID	NAME	SURNAME
1	Jack	Sparrow

Select Command Example:

```
string mdfFile = @"csharpexamples.mdb";

using (OleDbConnection connection = new OleDbConnection(string.Format("Provider
=Microsoft.Jet.OLEDB.4.0;Data Source={0}", mdfFile)))
{
      using (OleDbCommand selectCommand = new OleDbCommand("SELECT TOP
10 * FROM TABLE1", connection))
      {
            connection.Open();

            DataTable table = new DataTable();
            OleDbDataAdapter adapter = new OleDbDataAdapter();
            adapter.SelectCommand = selectCommand;
            adapter.Fill(table);

            foreach (DataRow row in table.Rows)
            {
                  object nameValue = row["NAME"];
                  object surnameValue = row["SURNAME"];
            }
      }
}
```

Insert Command Example:

```
string mdfFile = @"csharpexamples.mdb";

using (OleDbConnection connection = new OleDbConnection(string.Format("Provider
=Microsoft.Jet.OLEDB.4.0;Data Source={0}", mdfFile)))
{
    using (OleDbCommand insertCommand = new OleDbCommand("INSERT INTO
TABLE1 ([NAME],[SURNAME]) VALUES (?,?)", connection))
    {
        connection.Open();

        insertCommand.Parameters.AddWithValue("@NAME", "Brad");
        insertCommand.Parameters.AddWithValue("@SURNAME", "Pitt");

        insertCommand.ExecuteNonQuery();
    }
}
```

Update Command Example:

```
string mdfFile = @"csharpexamples.mdb";

using (OleDbConnection connection = new OleDbConnection(string.Format("Provider
=Microsoft.Jet.OLEDB.4.0;Data Source={0}", mdfFile)))
{
    using (OleDbCommand updateCommand = new OleDbCommand("UPDATE TA
BLE1 SET [NAME] = ?, [SURNAME] = ? WHERE [ID] = ?", connection))
    {
        connection.Open();

        updateCommand.Parameters.AddWithValue("@NAME", "Brad2");
        updateCommand.Parameters.AddWithValue("@SURNAME", "Pitt2");
        updateCommand.Parameters.AddWithValue("@ID", 2);

        updateCommand.ExecuteNonQuery();
    }
}
```

Delete Command Example:

```
string mdfFile = @"csharpexamples.mdb";

using (OleDbConnection connection = new OleDbConnection(string.Format("Provider
=Microsoft.Jet.OLEDB.4.0;Data Source={0}", mdfFile)))
{
        using (OleDbCommand deleteCommand = new OleDbCommand("DELETE FRO
M TABLE1 WHERE [ID] = ?", connection))
        {
                connection.Open();

                deleteCommand.Parameters.AddWithValue("@ID", 2);

                deleteCommand.ExecuteNonQuery();
        }
}
```