CHAPTER 8

C#

DATABASE PROGRAMMING

READING ASSIGNMENT

Contents

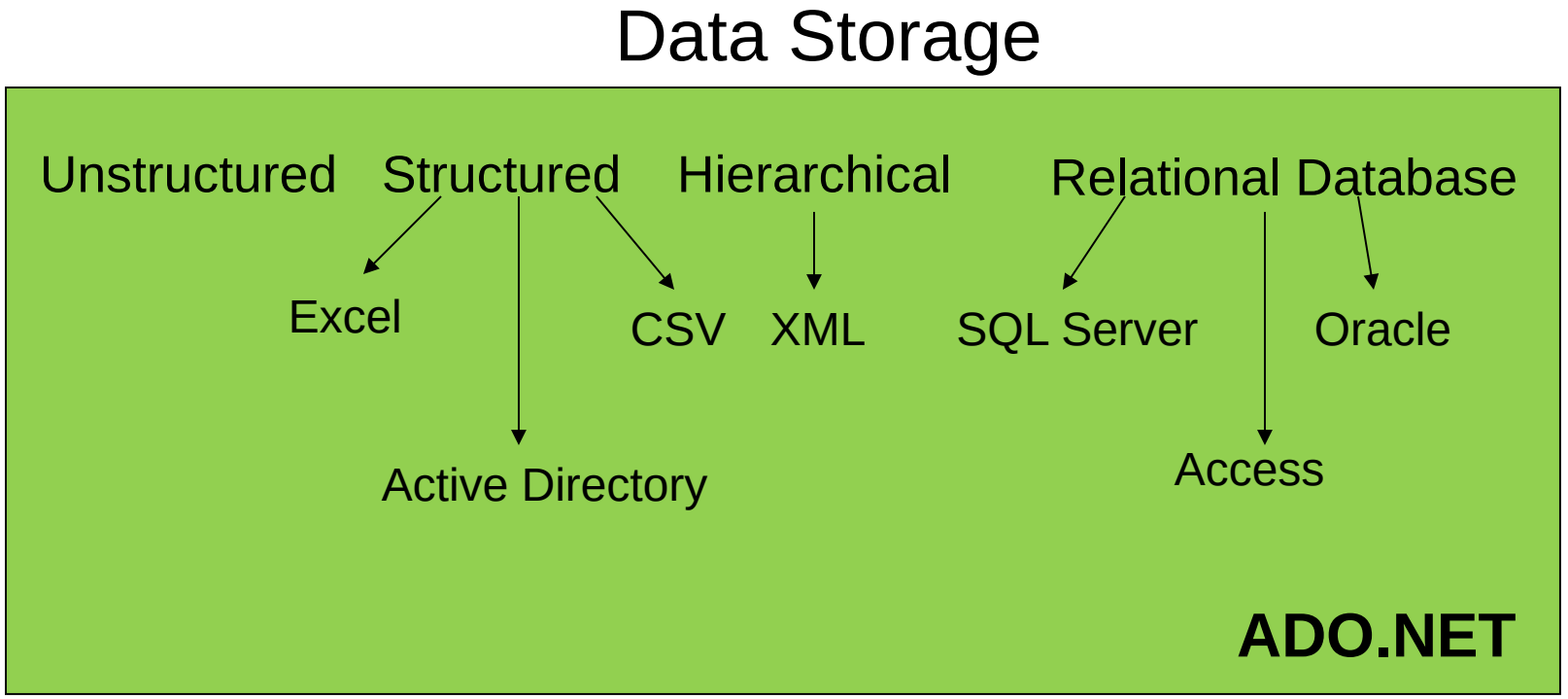
1. Introduction to ADO.NET
2. Database Namespace and Classes
3. Connecting to a Database
4. Architecture of ADO.NET
5. Managing database connection
6. Execute a SQL Command or Stored Procedure

Introduction to ADO.NET

- C# and .Net can work with a majority of databases, the most common being Microsoft SQL Server and Oracle.
  - But with every database, the logic behind working with all of them is mostly the same.
- In the Microsoft .NET Framework, access to a wide variety of data sources is enabled through a group of classes collectively named Microsoft ADO.NET.

- **ADO.NET** is a data access technology from Microsoft .Net Framework , which **provides communication** between *relational* and *non-relational* systems through a common set of components .
  - **ADO.NET** consist of a set of Objects that expose data access services to the .NET environment.
- ADO.NET is designed to be easy to use, and Visual Studio provides several wizards and other features that you can use to generate **ADO.NET** data access code.

- **Data Storage**: method of storing specific items that together constitute a unit of information.



Database Access

- Three steps:
  1. Open connection to database
  2. Execute SQL to retrieve records / update DB
  3. Close connection

Database Namespace

- The namespaces in the following table expose the classes and interfaces used

Namespace	Brief Description
System.Data	All generic data access classes
System.Data.Common	Classes shared (or overridden) by individual data providers
System.Data.EntityClient	Entity Framework classes
System.Data.Linq.SqlClient	LINQ to SQL provider classes
System.Data.Odbc	ODBC provider classes
System.Data.OleDb	OLE DB provider classes
System.Data.ProviderBase	New base classes and connection factory classes
System.Data.Sql	New generic interfaces and classes for SQL Server data access

- **System.Data.SqlClient**
- This assembly (namespace) of .NET Framework contains all of the classes required to connect to the databases,

Object	Description
`DataSet`	Represents data retrieved from a data source
`SqlCommand`	Executes a command, such as a SQL command, against a SQL Server database
`SqlConnection`	Provides access to a SQL Server database
`SqlDataAdapter`	Controls the interaction of a `DataSet` object with a SQL Server database
`SqlDataReader`	Returns read-only, forward-only data from a SQL Server database
`SqlError`	Contains error information returned from SQL Server
`SqlException`	Represents the exception that is thrown when an error or warning is returned from SQL Server

Connecting to an SQL Server Database

- **SqlConnection** class methods

Method	Description
BeginTransaction()	Begins a transaction
ChangeDatabase()	Changes the currently opened database
Close()	Closes a data source connection
CreateCommand()	Creates and returns a Command object associated with the SqlConnection object
GetSchema()	Returns schema information from the data source
Open()	Opens a data source connection
ClearPool()	Empties the SqlConnection object pool for the specified connection
ClearAllPool()	Empties all SqlConnection object pools

- **SqlConnection** class properties

Property	Description
ConnectionString	The string used to open a SQL Server database
ConnectionTimeout	The time to wait before abandoning a SQL Server database connection attempt
Database	The name of the current SQL Server database to use after a connection has been established
DataSource	The name of the SQL Server instance
ServerVersion	The SQL Server version to which the database is connected
State	A string indicating the current status of the SQL Server database connection

Connecting to a Database

- Connection to a database requires a connection string.
  - This string has the information about the **server** you're going to connect, the **database** you will require and the **credentials** that you can use to connect.
  - ach database has its own properties, its own server, name and type of login information, using which you can connect to the database to read/write the data from it.
- The connection string may include **attributes** such as the name of the **driver**, **server** and **database**, as well as security information such as **user name** and **password**.

- You configure a connection object using a **connection string**.
  - A connection string is a set of **semicolon-separated** name/value pairs.
- You can supply a connection string either as a **constructor argument** or by setting a connection object's **ConnectionString** property *before opening the connection*.
- Each connection class implementation requires that you provide different information in the connection string.

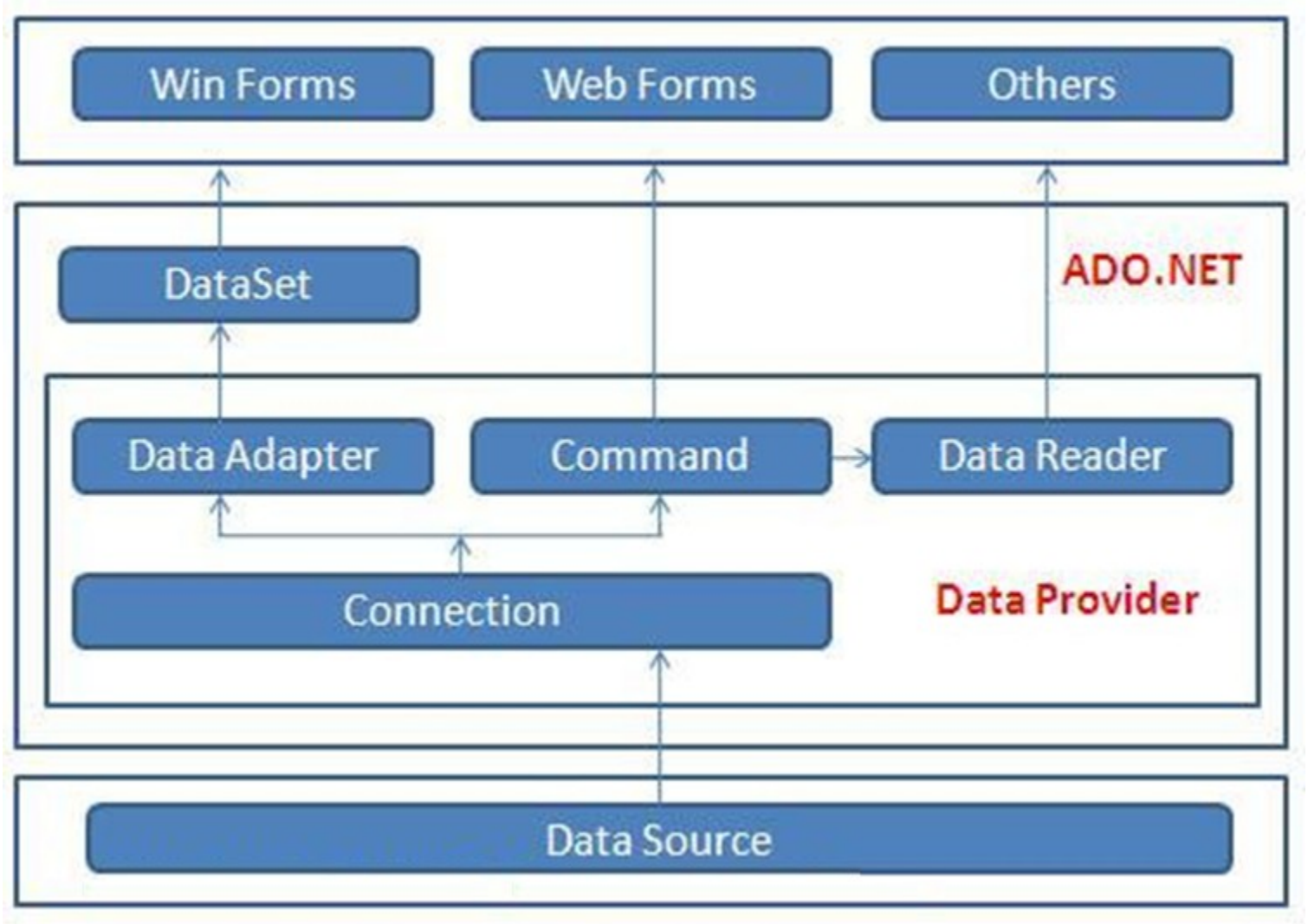
- Connected Environment (Scenario)
  1. Open connection
  2. Execute command
  3. Process rows in reader
  4. Close reader
  5. Close connection

- Working with data directly via open connection
  - **Advantages**
    - Simple security realization
    - Work with real data
    - Simple organization of distributed work
  - **Drawbacks**
    - Continual connection
    - Not available via Internet

- Disconnected Environment (Scenario)
  1. Open connection
  2. Fill the DataSet
  3. Close connection
  4. Process the DataSet
  5. Open connection
  6. Update the data source
  7. Close connection

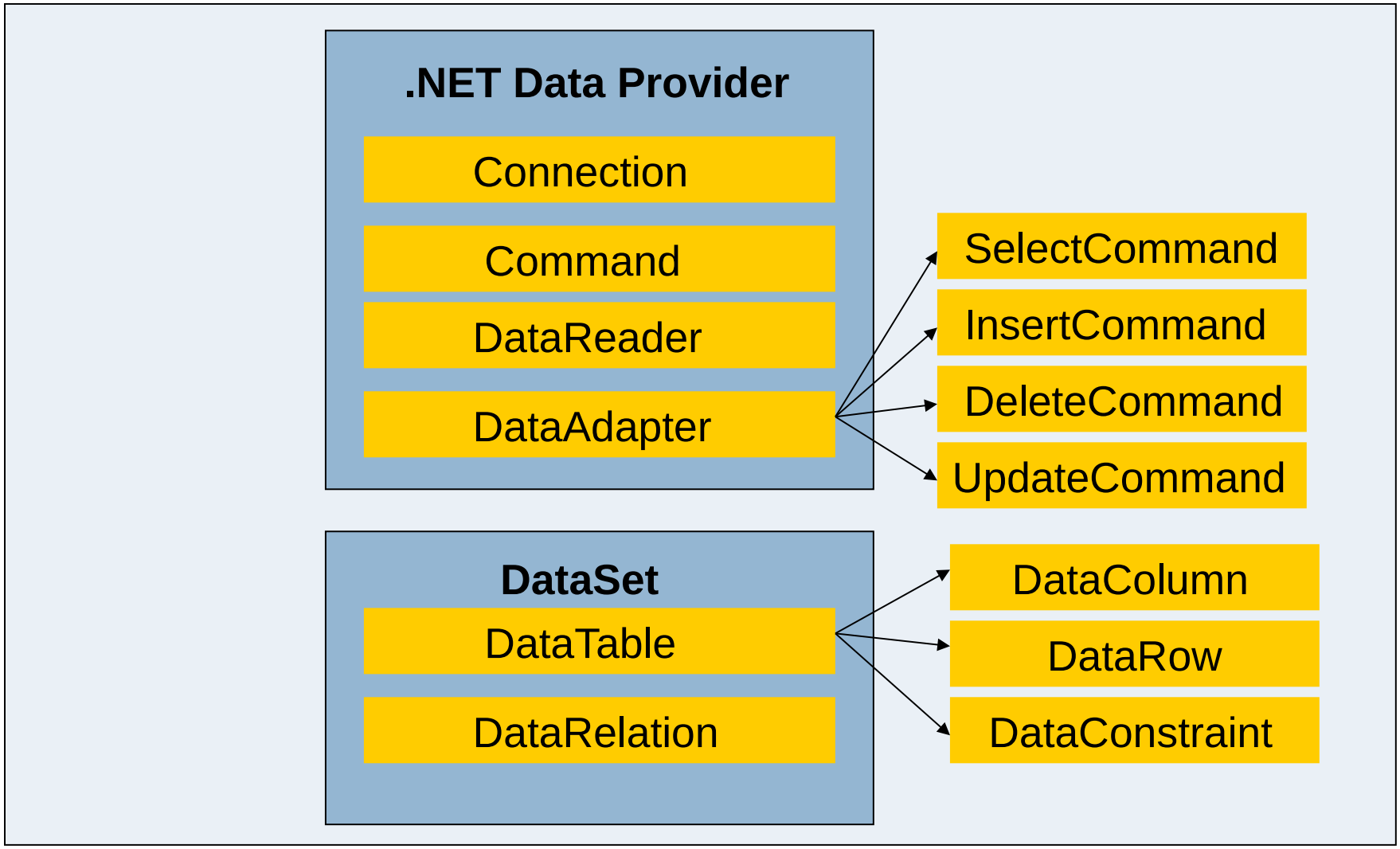
- Storage of data local copy from repository
- Possibility to update the main data source
  - **Advantages**
    - Economy of server resources
    - Does not require continual connection
  - **Drawbacks**
    - Demands conflict resolution while data update
    - Data is not always up to date

Architecture of ADO.NET



- The two key components of ADO.NET are **Data Providers** and **DataSet** .
  - The .Net Framework includes mainly three Data Providers for ADO.NET. They are the Microsoft **SQL Server Data Provider** , **OLEDB Data Provider** and **ODBC Data Provider** + **Oracle .NET Data Provider**
    - SQL Server uses the SqlConnection object ,
    - OLEDB uses the OleDbConnection Object and
    - ODBC uses OdbcConnection Object respectively.

- **DataSet** provides a disconnected representation of result sets from the Data Source, and it is completely independent from the Data Source.
  - DataSet provides much greater flexibility when dealing with related Result Sets.
  - **DataSet** consists of a collection of **DataTable** objects that you can relate to each other with **DataRelation** objects.
    - The DataTable contains a collection of **DataRow** and **DataCoulumn** Object which contains Data.
    - The DataAdapter Object provides a bridge between the DataSet and the Data Source.


.NET Data Provider
Connection
Command
DataReader
DataAdapter
SelectCommand
InsertCommand
DeleteCommand
UpdateCommand
DataSet
DataTable
DataRelation
DataColumn
DataRow
DataConstraint

Managing database connection (Creating and Closing)

- **Creating a connection**

1. Creating connection object using constructor and arguments
   - Example: (using integrated Windows security)

```
sqlConnection conn=new SqlConnection(new
SqlConnection("Data Source=.;Initial
Catalog=university;Integrated Security=True");
```

   - ConnectionString Parameters
     - Provider
     - Data Source
     - Initial Catalog
     - Integrated Security
     - UserID/Password

2. Creating connection object using constructor with no argument

```
SqlConnection conn=new SqlConnection();
Conn.ConnectionString="UserID=sa;password=;
Data Source=MyServer;Initial Catalog=Northwind;";
```

Closing connection

- When you’re finished with a connection, you should always call its **Close** method to free the underlying database connection and system resources.
- Eg: **conn.Close()**
- Alternatively, **using statement** makes very clean and efficient way of using connection objects in your code
  - When using statement is used, the method Close() of connection object is automatically called to close the opened connection and to release any resource.

Example


```
String output = "";
using (SqlConnection conn=new SqlConnection())
{
    conn.Open();
    conn.ConnectionString = "Data Source=.;Initial    Catalog=university;Integrated Security=True";
    if(conn.State==ConnectionState.Open)
    {
        output += conn.State;
        output+="Database Source = " + con.DataSource;
        output+="Database = " + con.Database;
        output+="Server Version = " + con.ServerVersion;
        output+=" Workstation Id = " + con.WorkstationId;
        output+=" Timeout = " + con.ConnectionTimeout;
        output+=" Packet Size = " + con.PacketSize;

        MessageBox.Show(output);
    }
}
```

Execute a SQL Command or Stored Procedure

- SqlCommand object
  - A SqlCommand object allows you to specify **what type of interaction** you want to perform with a database. For example, you can do select, insert, modify, and delete commands on rows of data in a database table.
  - **Properties**
    - Connection
    - CommandType
    - CommandText
    - Parameters
  - **Methods**
    - ExecuteNonQuery
    - ExecuteReader
    - ExecuteScalar

- **SqlCommand.CommandText** Property
  - Gets or sets the SQL statement, table name or stored procedure to execute at the data source.
  - Eg:
    - SqlCommand command = new SqlCommand();
      command.CommandText = "SELECT * FROM Categories ORDER BY CategoryID";
- **SqlCommand.CommandType** Property
  - Gets or sets a value indicating how the **CommandText** property is to be interpreted.
  - Possible values are:
    - CommandType.Text
    - CommandType.StoredProcedure

- **SqlCommand.CommandTimeout** Property
  - The time in seconds to wait for the command to execute. The default is 30 seconds.
  - Gets or sets the wait time before terminating the attempt to execute a command and generating an error.

Creating SqlCommand object

- **Constructors**
  1. SqlCommand()
  2. SqlCommand(cmdText)
  3. SqlCommand(cmdText, connection)
  - **Example**

```
SqlCommand command = new SqlCommand();
command.CommandText = "SELECT * FROM
  Categories";
command.CommandTimeout = 15;
command.CommandType = CommandType.Text;
```

  - **Remarks**
    - When the CommandType property is set to StoredProcedure, the CommandTextproperty should be set to **the name of the stored procedure**.

How to add parameters to commands?

- Use Parameters property of Command Object
  - **command.Parameters.Add**("@i", SqlDbType.Int);
  - **command.Parameters**["@i"].**Value** = idno;
  - command.Parameters.Add("@f", SqlDbType.Text);
  - command.Parameters["@f"].Value =fn;

  Or

  - **command.Parameters.AddWithValue**("@i", idno);

- SqlCommand cmd = new SqlCommand("select * from Customers where city = **@City**", conn);

```
//define parameters used in command object
SqlParameter param = new SqlParameter();
param.ParameterName = "@City";
param.Value = inputCity;
 // add new parameter to command object cmd.Parameters.Add(param);
```

Or

```
command.Parameters.Add(new SqlParameter("@i", idno));
// command.Parameters.Add(new SqlParameter("@f", fn));
```

Executing commands

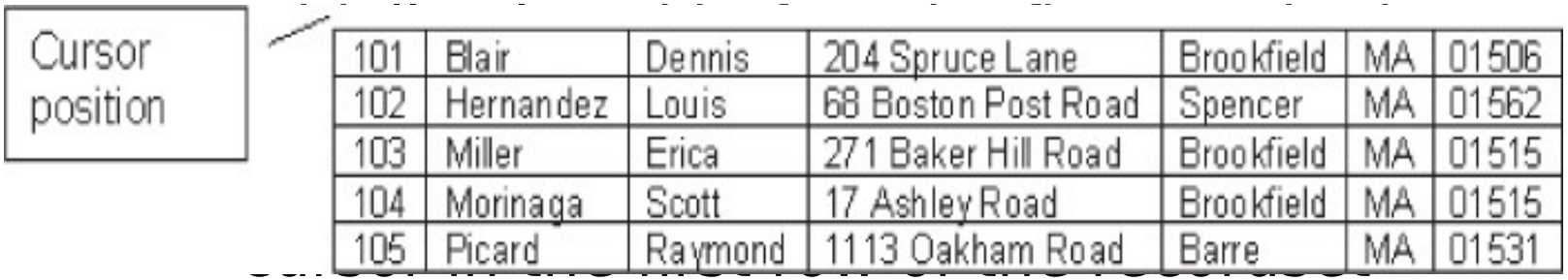
- The SqlCommand class provides the following methods for executing commands against the SQL Server database:
  - **ExecuteReader():** Executes the command and returns a typed IDataReader
  - **ExecuteNonQuery():** Executes the command but does not return any output
  - **ExecuteScalar():** Executes the command and returns the value from the first column of the first row of any result set

- **ExecuteReader()** mthod
  - Executes commands that return rows.
  - Sends the CommandText to the Connection and builds a SqlDataReader.
  - ExecuteReader() - Returns DataReader
  - **Example**
    - SqlCommand command = new SqlCommand(queryString, connection);
    - SqlDataReader reader = command.ExecuteReader();

Retrieving Records with the SqlDataReader Class

- **DataReader** object: used to retrieve read-only, forward-only data from a data source
- **Forward-only**: the program can only move forward sequentially through the records in the returned data from the first to the last
- Use a **DataReader** object when you want to read data but not add, delete, or modify records
- **SqlDataReader** class: used to retrieve data from SQL Server

- **ExecuteReader()** method of the SqlCommand class: creates a SqlDataReader object
  - Must assign the SqlDataReader object to a variable
- **Read()** method of the SqlDataReader class: advances the SqlDataReader object to the next record
- **Cursor**: your position within the recordset



- Use the Read() method to determine **if a next record is available**
  - Returns **true** if there is another row in the recordset
- Field names in a database table are assigned as variables in a SqlDataReader object collection
  - Content of each variable changes when the cursor position moves to a new row
- Use the **Close()** method of the SqlDataReader class to close it when you are finished working with it
  - SqlDataReader has exclusive access to the connection object
  - You cannot access any other commands until the SqlDataReader object is closed.

Example


```
using (SqlConnection conn = new SqlConnection())
{
    conn.ConnectionString = "Data Source=localhost;
    Initial Catalog=University;Integrated Security=True";
    SqlCommand command = new SqlCommand();
    command.CommandText = "Select * from student where idNo=@idno";
    command.CommandType = CommandType.Text;
    command.Connection = conn;
    //command.Parameters.AddWithValue("@idno",textBox1.Text);
    command.Parameters.Add(new SqlParameter("@idno", textBox1.Text));
    conn.Open();
    SqlDataReader dr = command.ExecuteReader();
    while (dr.Read())
    {
        listBox1.Items.Add(dr.GetInt32(0) + dr.GetString(1));
        //Or  you can access using column name
        listBox1.Items.Add(dr["idNo"+ dr["fname"]);
    }
}
```

Display Select Query result using dataDGridView control


```
using (SqlConnection conn = new SqlConnection())
{
    conn.ConnectionString = "Data Source=localhost;Initial Catalog=University;
     Integrated Security=True;Pooling=False";
    SqlCommand command = new SqlCommand();

    command.CommandText = "Select * from student";
    command.CommandType = CommandType.Text;
    command.Connection = conn;
    conn.Open();
    SqlDataReader dr = command.ExecuteReader();
    DataTable dt = new DataTable();

    dt.Load(dr);
    //or
    //dt.Load(command.ExecuteReader());
    dataGridView1.DataSource = dt;
}
```

- **Executing action queries**
  - Use action queries when you need to modify a DB
    - Updates
    - Inserts
    - Deletes
- Execute action queries via **ExecuteNonQuery** method…
  - Basic Idea
    - **Open**, **execute**, and **close**:

ExecuteNonQuery

- To execute operations where database is changed
- Example: insert, update, delete, create database, create table, etc.
- Insert, update, delete: **Returns number of rows** affected by the operation and **returns -1** for others
- Example

```
SqlCommand cmd = new SqlCommand
   ("insert into titles (title_id, title, type, pubdate)"+
    "values ('CS150','C++ Programming'," +
    " 'computer science', 'May 2006'), conn);
cmd.ExecuteNonQuery();
```

Insert Query

```
using (SqlConnection conn = new SqlConnection())
{
    conn.ConnectionString = "Data Source=localhost;
     Initial Catalog=University;Integrated Security=True;Pooling=False";
    SqlCommand command = new SqlCommand();

    command.CommandText = "insert into student values (@idno,@fn)";
    command.CommandType = CommandType.Text;
    command.Connection = conn;
    command.Parameters.Add(new SqlParameter("@idno", textBox1.Text));
    command.Parameters.AddWithValue("@fn",textBox2.Text);
    conn.Open();
    int rows=command.ExecuteNonQuery();
    MessageBox.Show(rows + "  are affected");
}
```

Delete Query

```
using (SqlConnection conn = new SqlConnection())
{
    conn.ConnectionString = "Data Source=localhost;
      Initial Catalog=University;Integrated Security=True;Pooling=False";
    SqlCommand command = new SqlCommand();

    SqlCommand command = new SqlCommand("delete from student
      where idNo=@idno", conn);

    command.Parameters.Add(new SqlParameter("@idno", textBox1.Text));
    conn.Open();
    int rows = command.ExecuteNonQuery();
    MessageBox.Show(rows + "  are affected");
}
```

Update Query

```
using (SqlConnection conn = new SqlConnection())
{
    conn.ConnectionString = "Data Source=localhost;
      Initial Catalog=University;Integrated Security=True;Pooling=False";

    SqlCommand command = new SqlCommand("update student set
      fName=@fn where idNo=@idno", conn);

    command.Parameters.Add(new SqlParameter("@idno", textBox1.Text));
    command.Parameters.AddWithValue("@fn", textBox2.Text);
    conn.Open();
    int rows = command.ExecuteNonQuery();
    MessageBox.Show(rows + "  are affected");
}
```

ExecuteScalar

- Returns the 1st row of the 1st column in the result
- Used for commands such as: count, avg

```
try{
    conn.Open();
    SqlCommand cmd = new SqlCommand
    ("select max (advance) from title", conn);
    decimal amount = (decimal) cmd.ExecuteScalar ();
    Console.WriteLine ("ExecuteScalar returned (0:c)", amount);
}
Catch (SqlException ex}
{
    Console.Writeln (ex.Message);
}
finally {
    conn.Close();
}
```

- Example

```
DataBaseConnection dbCon = new DataBaseConnection();
SqlConnection conn = dbCon.getConnection();
SqlCommand command = new SqlCommand("select count(*) from student", conn);

dbCon.OpenConnection(conn);
int totalRecords =(int) command.ExecuteScalar();
MessageBox.Show("There are " + totalRecords + " records in the table");
dbCon.CloseConnection(conn);
```

- Displaying column names of returned record set
- Using **FieldCount** property and

```
SqlDataReader dr = command.ExecuteReader();
string output1 = "";
for (int i = 0; i < dr.FieldCount; i++)
    output1 += dr.GetName(i) + " ";
MessageBox.Show(output1);
```

Questions?

Thank You