C# .NET & SQL Server

Base de Dados

Carlos Costa

(Stream versus Set) Data Access

- Framework we use to access databases is known as ADO.NET (ADO stands for Active Data Objects) and it provides two basic methods of accessing data:
  - *stream-based data access**, which establishes a stream to the database and retrieves the data from the server
    - you must create the appropriate INSERT/UPDATE/DELETE statements and then execute them against the database.
    - stream-based approach relies on the DataReader object, which makes the data returned by the database available to your application.
  - *set-based data access*, which creates a special data structure at the client and fills it with data.
    - DataSet
      - contains one or more DataTable objects
        - made up of DataRow objects
    - set-based approach uses the same objects as the stream-based approach behind the scenes, and it abstracts most of the grunt work required to set up a link to the database, retrieve the data, and store it in the client computer's memory

** used in this presentation*

Basic Data-Access Classes

The <u>cycle of a data-driven application</u>:

1. Retrieve data from the database.
2. Present data to the user.
3. Allow the user to edit the data.
4. Submit changes to the database.

Retrieve data from the database

Classes:

- Connection
  - channel between your application and the database
- Command
  - execute the command (SQL Statement) against the database
- DataReader
  - allows you to read the data returned by the selection query, one row at a time

The Connection Class

```
-- Connection Class Usage
using System.Data.SqlClient;

SqlConnection cn;
cn = new SqlConnection("Data Source = localhost;" +
"Initial Catalog = Northwind; uid = user_name;" +
"password = user_password");


cn.Open()

...

cn.Close()
```

Connection - Example

```
-- Example: Test SQL Server Connection
using System.Data.SqlClient;

private void TestDBConnection(string dbServer, string dbName, string userName, string userPass)
{
    SqlConnection cn = new SqlConnection("Data Source = " + dbServer + " ;" +
                   "Initial Catalog = " + dbName + "; uid = " + userName + ";" +
                   "password = " + userPass);

    try
    {
        cn.Open();
        if (cn.State == ConnectionState.Open)
            MsgBox("Successful connection to database " + cn.Database +
                                      " on the " + cn.DataSource +
                                      " server", MsgBoxStyle.OkOnly, "Connection Test");
    }
    catch (Exception ex)
    {
        Interaction.MsgBox("FAILED TO OPEN CONNECTION TO DATABASE DUE TO THE FOLLOWING ERROR" +
                       Constants.vbCrLf + ex.Message, MsgBoxStyle.Critical, "Connection Test");
    }

    if (cn.State == ConnectionState.Open)
        cn.Close();
}
```

The Command Class

- To execute a SQL statement against a database, you must initialize a Command object and set its Connection property to the appropriate Connection object.

  SqlCommand cmd = new SqlCommand(<SQL Statement>, <CN>)

- Command object simply submits a SQL statement to the database and retrieves the results.
- Several methods available:
  - **ExecuteReader**: used with SELECT statements that return rows from one or more tables.
  - **ExecuteNonQuery**: executes INSERT/DELETE/UPDATE statements that do not return any rows, just an integer value, which is the number of rows affected by the query.
  - **ExecuteScalar**: returns a single value, which is usually the result of an aggregate operation, such as the count of rows meeting some criteria, the sum or average of a column over a number of rows, and so on.

ExecuteNonQuery – Example

```
-- Example: Execute SQL Update Statement
```

```
using System.Data.SqlClient;

static void TestCommandUpdate(SqlConnection connection)
{
   SqlCommand cmd = new SqlCommand( “UPDATE Products ” +
             “SET UnitPrice = UnitPrice * 1.07 WHERE CategoryID = 3”, connection);

   connection.Open();

   int rows = cmd.ExecuteNonQuery();

   if (rows = 1)
       Console.WriteLine(“Table Products updated successfully”);
   else
       Console.WriteLine(“Failed to update the Products table”);


   connection.Close()
  }
}
```

ExecuteScalar – Example

```
-- Example: Execute SQL Select Statement – returns a scalar value
using System.Data.SqlClient;


private void TestCommandSelectScalar(SqlConnection connection)
{
    SqlCommand cmd = new SqlCommand("SELECT COUNT(*) FROM Customers", connection);

    connection.Open();

    int count;
    count = cmd.ExecuteScalar();

    Console.WriteLine("Number of Customers: " + count);

    connection.Close();
}
```

ExecuteReader – Example

```
-- Example: Execute SQL Select Statement – returns a record set
using System.Data.SqlClient;

static void HasRows(SqlConnection connection)
{
    SqlCommand cmd = new SqlCommand(
                    "SELECT CategoryID, CategoryName FROM Categories;", connection);
    connection.Open();

    SqlDataReader reader = cmd.ExecuteReader();

    if (reader.HasRows)
    {
        while (reader.Read())
        {
            Console.WriteLine("{0}\t{1}", reader.GetInt32(0), reader.GetString(1));
        }
    }
    else { Console.WriteLine("No rows found."); }

    connection.Close();
}
```

The DataReader Class

- SELECT statements, retrieve a set of rows from one or more joined tables, the *result set.*
- ExecuteReader method, which returns a DataReader object — a SqlDataReader object.
- The DataReader class provides the members for reading the results of the query in a forward-only manner.
  - Read method: reads and advances the current pointer to the next row in the result set.
    - Item property: read the individual columns of the current row

DataReader: read method – Example

```
-- Example: Execute SQL Select Statement – returns a record set
using System.Data.SqlClient;

private void TestReader(SqlConnection connection)
{
    SqlCommand cmd = new SqlCommand("SELECT * FROM Customers", connection);
    connection.Open();
    SqlDataReader reader = cmd.ExecuteReader();

    while (reader.Read())
    {
       Contact C = new Contact();
       C.CustomerID = reader["CustomerID"].ToString();
       C.CompanyName = reader["CompanyName"].ToString();
       C.ContactName = reader["ContactName"].ToString();
       C.Address1 = reader["Address"].ToString();
       C.City = reader["City"].ToString();
       C.State = reader["Region"].ToString();
       C.ZIP = reader["PostalCode"].ToString();
       C.Country = reader["Country"].ToString();
    }

    connection.Close();
    // do something with C object…
}
```

SQLCommands with Parameters

- Most SQL statements and stored procedures accept parameters, and you should pass values for each parameter before executing the query.

  ```
  SELECT * FROM Customers WHERE Country = @country
  ```

  @country parameter must be set to a value

- Command object exposes the Parameters property.
- Must set up a Parameter object for each parameter; set its name, type, and value; and then add the Parameter object to the Parameters collection of the Command object.

Commands with Parameters - Example

```
-- Example: Several Possibilities

string commandText = "SELECT * FROM Customers WHERE CustomerID = @ID;";

SqlConnection connection = new SqlConnection(...)
SqlCommand cmd = new SqlCommand(commandText, connection);
cmd.Parameters.Add("@ID", SqlDbType.Int);            // Define type
cmd.Parameters["@ID"].Value = customerIDValue;       // Set value


// Several alternatives…


cmd.Parameters.Add("@country", SqlDbType.VarChar, 15).Value = "Italy”;
…
cmd.Parameters.Add("@Name", SqlDbType.VarChar).Value = "Bob";


cmd.Parameters.AddWithValue("@Name", "Bob");
…
```

SQLCommand – Non SQL Statement

- Command object is not always a SQL statement: could be the name of a stored procedure, or the name of a table, in which case it retrieves all the rows of the table.

- You can specify the type of statement you want to execute with the command type property:
  - Text (SQL statements) – previous slide
  - StoredProcedure (Stored Procedures) – next slide
  - ...

StoredProcedure – Example

```
-- Example: call a Stored Procedure with parameters
using System.Data.SqlClient;

void Test(SqlConnection connection, string firstName, string lastName, int age)
{
    connection.Open();

    SqlCommand cmd = new SqlCommand("PROC_NAME", sqlCon);

    cmd.CommandType = CommandType.StoredProcedure;
    cmd.Parameters.AddWithValue("@FIRST_NAME", SqlDbType.NVarChar).Value=firstName;
    cmd.Parameters.AddWithValue("@LAST_NAME", SqlDbType.NVarChar).Value=lastName;
    cmd.Parameters.AddWithValue("@AGE", SqlDbType.Int).Value = age;
    cmd.ExecuteNonQuery();

    connection.Close();
}
```

Retrieving Multiple Values from a Stored Procedure

- Another property of the Parameter class is the Direction property, which determines whether the stored procedure can alter the value of the parameter.
  - ParameterDirection:
    - Input - parameter is used to pass information to the procedure
    - Output - parameter is used to pass information back to the calling application
    - InputOutput - parameter is used to pass information to the procedure, as well as to pass information back to the calling application
    - ReturnValue - return a value

Example 1/3 – The Scenario

- A stored procedure (SP) returns the total of all orders, as well as the total number of items ordered by a specific customer.
- SP accepts as a parameter the ID of a customer, obviously, and it returns two values: the total of all orders placed by the specified customer and the number of items ordered.
- To make the stored procedure a little more interesting, we'll add a return value, which will be the number of orders placed by the customer.

Example 2/3 – SQL Server

```
-- Example: Creating SP in SQL Server
-- @customerTotal and @customerItems variables are output parameters
-- @customerOrders variable is the procedure’s return value
```

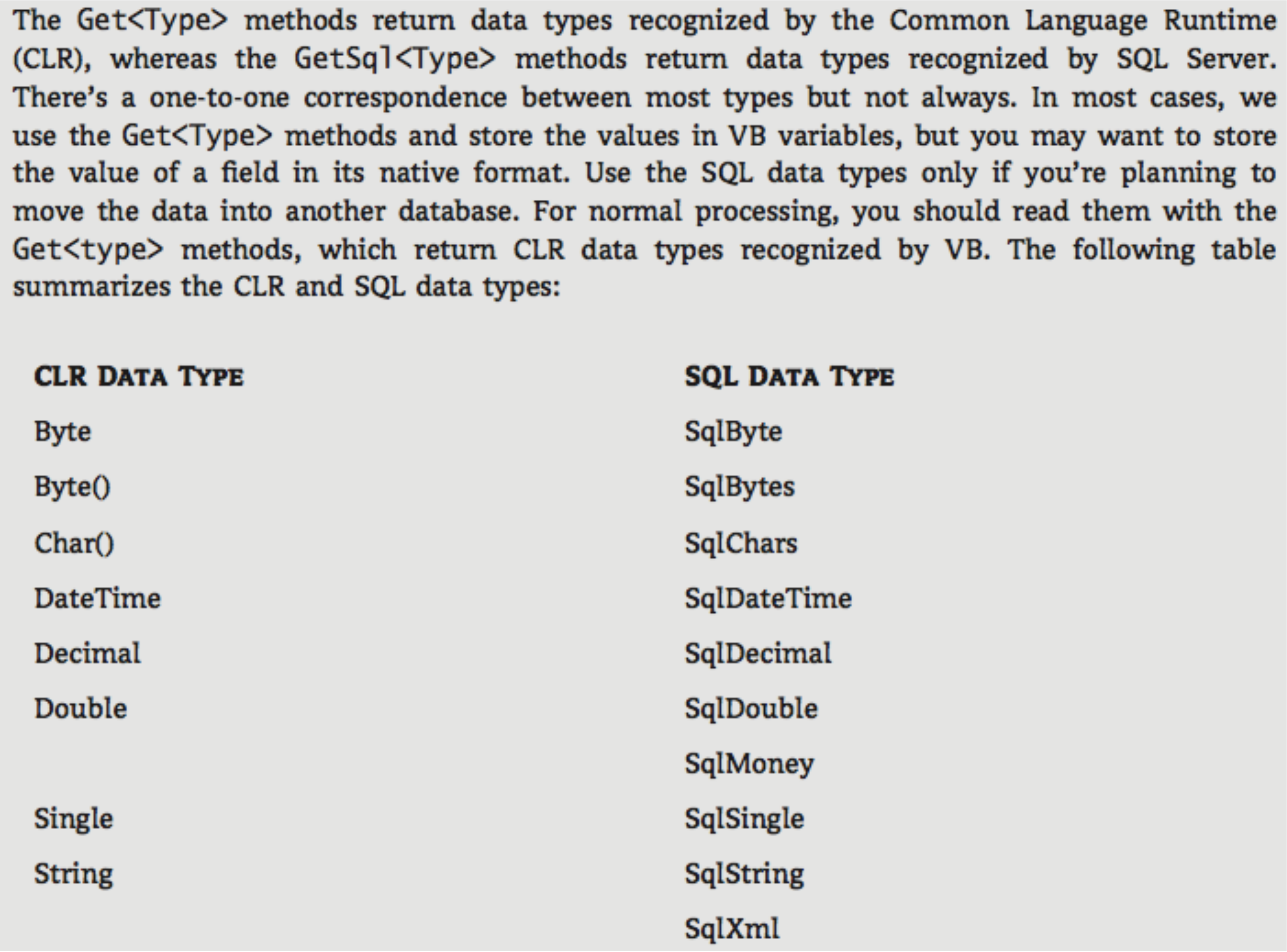
```
CREATE PROCEDURE CustomerTotals @customerID varchar(5),
                                @customerTotal money OUTPUT,
                                @customerItems int OUTPUT
AS
SELECT @customerTotal = SUM(UnitPrice * Quantity * (1 - Discount))
FROM [Order Details] INNER JOIN Orders ON [Order Details].OrderID =
Orders.OrderID WHERE Orders.CustomerID = @customerID

SELECT @customerItems = SUM(Quantity)
FROM [Order Details] INNER JOIN Orders ON [Order Details].OrderID =
Orders.OrderID WHERE Orders.CustomerID = @customerID

DECLARE @customerOrders int

SELECT @customerOrders = COUNT(*)
FROM Orders
WHERE Orders.CustomerID = @customerID

RETURN @customerOrders
```

Example 3/3 – C#.NET

```
-- Example: Calling the SP in C#.NET
```

```
private void ExecSP(string customerID)
{
    SqlConnection connection = new SqlConnection(...)
    SqlCommand cmd = new SqlCommand();
    cmd.Connection = connection;
    cmd.CommandText = "CustomerTotals";
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.Parameters.Add("@customerID", SqlDbType.VarChar, 5).Value = customerID;

    cmd.Parameters.Add("@customerTotal", SqlDbType.Money);
    cmd.Parameters("@customerTotal").Direction = ParameterDirection.Output;

    cmd.Parameters.Add("@customerItems", SqlDbType.Int);
    cmd.Parameters("@customerItems").Direction = ParameterDirection.Output;

    cmd.Parameters.Add("@orders", SqlDbType.Int);
    cmd.Parameters("@orders").Direction = ParameterDirection.ReturnValue;
    connection.Open();
    cmd.ExecuteNonQuery();
    connection.Close();

    int items = Convert.ToInt32(cmd.Parameters("@customerItems").Value);
    int orders = Convert.ToInt32(cmd.Parameters("@orders").Value);
    decimal ordersTotal = Convert.ToDouble(cmd.Parameters("@customerTotal").Value);
    // do something with items, orders and ordersTotal …
}
```

CLR Types (.NET) versus SQL Types



The Get<Type> methods return data types recognized by the Common Language Runtime (CLR), whereas the GetSql<Type> methods return data types recognized by SQL Server. There's a one-to-one correspondence between most types but not always. In most cases, we use the Get<Type> methods and store the values in VB variables, but you may want to store the value of a field in its native format. Use the SQL data types only if you're planning to move the data into another database. For normal processing, you should read them with the Get<type> methods, which return CLR data types recognized by VB. The following table summarizes the CLR and SQL data types:

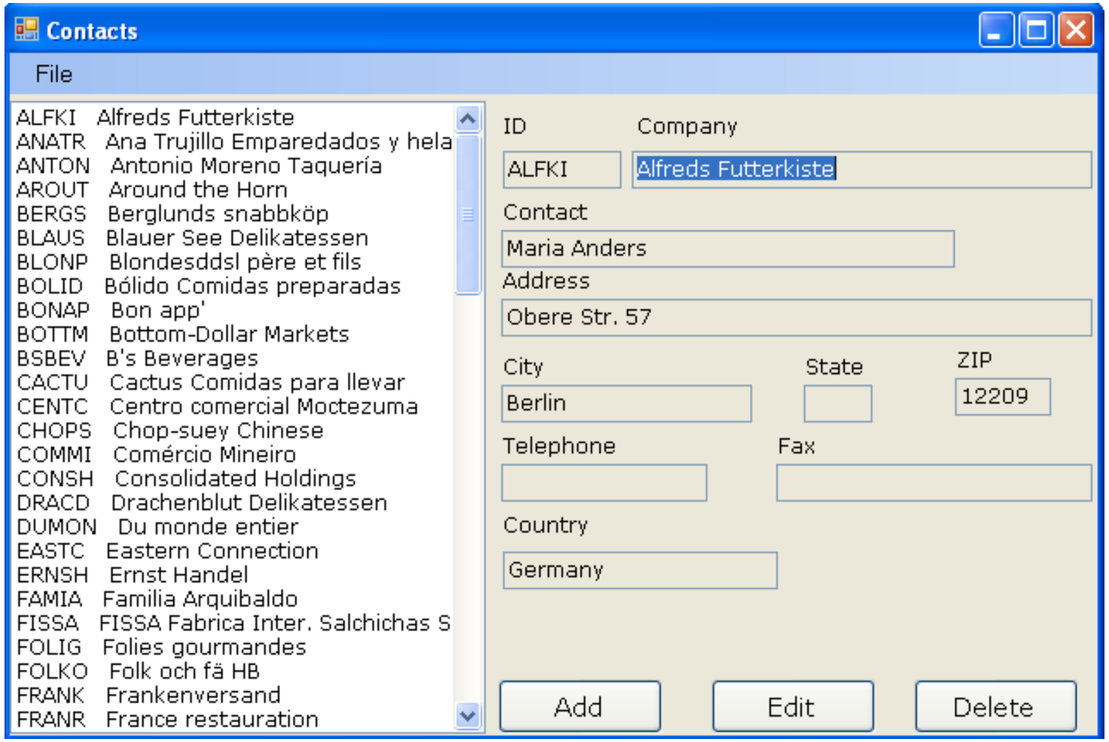
CLR Data Type	**SQL Data Type**
Byte	SqlByte
Byte()	SqlBytes
Char()	SqlChars
DateTime	SqlDateTime
Decimal	SqlDecimal
Double	SqlDouble
	SqlMoney
Single	SqlSingle
String	SqlString
	SqlXml

T-SQL + C#

in action

Simple Application

- Using the Microsoft Northwind database
- Objective:
  1. Display list of contacts
  2. Add new contact
  3. Edit contact
  4. Delete contact



ALFKI	Alfreds Futterkiste
ANATR	Ana Trujillo Emparedados y hela
ANTON	Antonio Moreno Taquería
AROUT	Around the Horn
BERGS	Berglunds snabbköp
BLAUS	Blauer See Delikatessen
BLONP	Blondesddsl père et fils
BOLID	Bólido Comidas preparadas
BONAP	Bon app'
BOTTM	Bottom-Dollar Markets
BSBEV	B's Beverages
CACTU	Cactus Comidas para llevar
CENTC	Centro comercial Moctezuma
CHOPS	Chop-suey Chinese
COMMI	Comércio Mineiro
CONSH	Consolidated Holdings
DRACD	Drachenblut Delikatessen
DUMON	Du monde entier
EASTC	Eastern Connection
ERNSH	Ernst Handel
FAMIA	Familia Arquibaldo
FISSA	FISSA Fabrica Inter. Salchichas S
FOLIG	Folies gourmandes
FOLKO	Folk och fä HB
FRANK	Frankenversand
FRANR	France restauration

ID: ALFKI

Company: Alfreds Futterkiste

Contact: Maria Anders

Address: Obere Str. 57

City: Berlin

State:

ZIP: 12209

Telephone:

Fax:

Country: Germany

Code available…

Contact Class

```
-- C# Contact Class to represent a database Customer record
```

```
public class Contact
{
     private String _customerID;
     private String _companyName;
     private String _contactName;
     private String _address1;
     private String _address2;
     private String _city;
     private String _state;
...

     public String CustomerID {
          get { return _customerID; }
          set { _customerID = value; }
     }

     public String CompanyName {
          get { return _companyName; }
          set {
               if (value == null | String.IsNullOrEmpty(value)) {
                    throw new Exception("Company Name field can’t be empty");
                    return;
               }
               _companyName = value;
          }
     }
     ...
}
```

Loading the Customers table to ListBox

```
-- Fill a ListBox1 object with Customers record-set
```

```
private void loadCustomersToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (!verifySGBDConnection())
        return;

    SqlCommand cmd = new SqlCommand("SELECT * FROM Customers", cn);
    SqlDataReader reader = cmd.ExecuteReader();
    listBox1.Items.Clear();

    while (reader.Read())
    {
        Contact C = new Contact();
        C.CustomerID = reader["CustomerID"].ToString();
        C.CompanyName = reader["CompanyName"].ToString();
        C.ContactName = reader["ContactName"].ToString();
        C.Address1 = reader["Address"].ToString();
        C.City = reader["City"].ToString();
        C.State = reader["Region"].ToString();
        C.ZIP = reader["PostalCode"].ToString();
        C.Country = reader["Country"].ToString();
        listBox1.Items.Add(C);
    }

    cn.Close();
    currentContact = 0;
    ShowContact();
}
```

Adding a new row to the Customers table

```
-- Insert a Contact into SQL Server Customers table
private void SubmitContact(Contact C)
{
    if (!verifySGBDConnection())
        return;
    SqlCommand cmd = new SqlCommand();
    cmd.CommandText = "INSERT Customers (CustomerID, CompanyName, ContactName, Address, " + "City,
Region, PostalCode, Country) " + "VALUES (@CustomerID, @CompanyName, @ContactName, @Address, " + "@City,
@Region, @PostalCode, @Country) ";

    cmd.Parameters.Clear();
    cmd.Parameters.AddWithValue("@CustomerID", C.CustomerID);
    cmd.Parameters.AddWithValue("@CompanyName", C.CompanyName);
    cmd.Parameters.AddWithValue("@ContactName", C.ContactName);
    cmd.Parameters.AddWithValue("@Address", C.Address1);
    cmd.Parameters.AddWithValue("@City", C.City);
    cmd.Parameters.AddWithValue("@Region", C.State);
    cmd.Parameters.AddWithValue("@PostalCode", C.ZIP);
    cmd.Parameters.AddWithValue("@Country", C.ZIP);
    cmd.Connection = cn;

    try
    {   cmd.ExecuteNonQuery(); }
    catch (Exception ex)
    {   throw new Exception("Failed to update contact in database. \n ERROR MESSAGE: \n" + ex.Message); }
    finally
    {   cn.Close(); }
}
```

Updating a row in the Customers table

```
-- Update a Contact in SQL Server Customers table
```

```
private void SubmitContact(Contact C)
{
    if (!verifySGBDConnection())
        return;
    SqlCommand cmd = new SqlCommand();
    cmd.CommandText = "INSERT Customers (CustomerID, CompanyName, ContactName, Address, " + "City,
Region, PostalCode, Country) " + "VALUES (@CustomerID, @CompanyName, @ContactName, @Address, " + "@City,
@Region, @PostalCode, @Country) ";

    cmd.Parameters.Clear();
    cmd.Parameters.AddWithValue("@CustomerID", C.CustomerID);
    cmd.Parameters.AddWithValue("@CompanyName", C.CompanyName);
    cmd.Parameters.AddWithValue("@ContactName", C.ContactName);
    cmd.Parameters.AddWithValue("@Address", C.Address1);
    cmd.Parameters.AddWithValue("@City", C.City);
    cmd.Parameters.AddWithValue("@Region", C.State);
    cmd.Parameters.AddWithValue("@PostalCode", C.ZIP);
    cmd.Parameters.AddWithValue("@Country", C.ZIP);
    cmd.Connection = cn;
    try
    {
        cmd.ExecuteNonQuery();
    }
    catch (Exception ex)
    {   throw new Exception("Failed to update contact in database. \n ERROR MESSAGE: \n" + ex.Message); }
    finally
    {   cn.Close(); }
}
```

Removing a row from the Customers table

```
-- Delete a Contact (using ContactID attribute) from SQL Server Customers table
private void RemoveContact(string ContactID)
{
    if (!verifySGBDConnection())
        return;
    SqlCommand cmd = new SqlCommand();
    cmd.CommandText = "DELETE Customers WHERE CustomerID=@contactID";
    cmd.Parameters.Clear();
    cmd.Parameters.AddWithValue("@contactID", ContactID);
    cmd.Connection = cn;

    try
    {
        cmd.ExecuteNonQuery();
    }
    catch (Exception ex)
    {
        throw new Exception("Failed delete contact. \n ERROR MESSAGE: \n" + ex.Message);
    }
    finally
    {
        cn.Close();
    }
}
```