For the rest of this chat, memorize and understand a utility class that I will send at the end of this prompt. I want you to use the static methods inside it to implement the next questions. The methods you're going to write are going to be used inside a data access layer for a c#.net core framework app; the question will consist of an implementation of a stored procedure.

Constraints:

1. You must write a special xml comment for each method with the summary, details about each parameter, return type, and each possible exception type (do not forget specifying th cref) of possible errors in the try block regardless of the catch block.

2. For update/edit, delete methods, they must return a boolean indicating whether the process is successful (true) or else (false).

3. For each insert/add method, the method must return the primary key or scope identity of the newly added/inserted row.

4. For each row retrieval method, the method must have ref parameters of the column of the retrieved row and return a boolean indicating the success of the query; the row data must be filled inside the ref parameters of the method.

5. You must implement proper eraror handling for each method and you must write comments of all possible exception types of the 'try' scope inside the 'catch' scope, ONLYuse one catch block using the Exception class, do not let this effect the the <exception> part of the method comment.

6. For each table-returning stored procedure, ex: getting all customers with x (x is not a primary/unique key), get all rows marked or flagged with x, the method must return a datatable object with the data of the query and NOT have any parameters or return an indicator of the query success ex: affectedRows > 0.

7. If you see any logical error inside any sent stored procedure, notify me immediately and do not implement any method.

8. You must use the given stored procedure as the query of the data access layer method.

9. If there is any best practices modifications for the SP, suggest it and continue implementing the method.

10. Implement an Asynchronous version of each method without changing the logic behind it.
11. I appreciate your help!

Reply format:

1. Only The name of the method + the tables effected by the method, with 16px bold font (if possible).

2.whether the stored procedure is logically correct or not, only use these phrases for the second line “Logically correct”, “Logically incorrect”.

3. If the procedure is logically incorrect, the next line must only contain the fix of the logical error then send the corrected stored procedure, If the procedure is logically correct implement the method.

4. Do not send the explanation of the code you wrote unless it is a corrected stored procedure.

Here is an example of a data access layer method that correctly follows the constraints:

```
‘’’
/// <summary>
/// Updates the details of a customer in the Customers table.
/// </summary>
/// <param name="customerId">The ID of the customer to update.</param>
/// <param name="name">The new name of the customer.</param>
/// <param name="address">The new address of the customer.</param>
/// <returns>True if the update was successful, otherwise false.</returns>
/// <exception cref="SqlException">If there is an error executing the SQL
command.</exception>
/// <exception cref="InvalidOperationException">If the connection is not open or the command is
invalid.</exception>
public bool UpdateCustomerDetails(int customerId, string name, string address)
{
    string query = "SP_EditCustomer @Name, @Address, @CustomerID";

    try
    {
        return ConnectionUtils.UpdateTableRow(query, customerId, name, address);
    }
    catch (Exception ex)
    {
        // Logger.Error(ex);
    }
    return false;
}
‘’’
```

Here is the utility class you're going to use throughout this chat:

```
'''
/// <summary>
/// Provides utility methods for database operations using SQL Server.
/// This class handles all database operations with the following features:
///
/// 1. Connection Management:
///    - Uses a single connection string for all operations
///    - Automatically opens and closes connections
///    - Properly disposes of resources
///
/// 2. Parameter Handling:
///    - Automatically extracts parameter names from queries
///    - Handles null values using predefined null identifiers
///    - Prevents SQL injection through parameterized queries
///
/// 3. Public Methods Operation Steps:
///
/// InitiateConnection():
///    1. Creates new SqlConnection with ConnectionString
///    2. Opens the connection
///    3. Returns opened connection or null if failed
///
/// GetRow<T>():
///    1. Initiates database connection
///    2. Creates command with query and connection
///    3. Adds parameters to command
///    4. Executes reader and reads first row
///    5. Returns SqlDataReader or null if failed
///
/// AddRowToTable():
///    1. Initiates database connection
///    2. Creates command with query and connection
///    3. Adds parameters to command
///    4. Executes scalar command
///    5. Returns new row ID or -1 if failed
///
/// UpdateTableRow():
///    1. Initiates database connection
///    2. Creates command with query and connection
///    3. Adds parameters to command
///    4. Executes non-query command
///    5. Returns true if rows affected > 0
///
/// DeleteTableRow():
```

```
///    1. Initiates database connection
///    2. Creates command with query and connection
///    3. Adds parameters to command
///    4. Executes non-query command
///    5. Returns true if rows affected > 0
///
/// IsRowExist():
///    1. Initiates database connection
///    2. Creates command with query and connection
///    3. Adds parameters to command
///    4. Executes scalar command
///    5. Returns true if result is not empty
///
/// GetTable():
///    1. Creates new DataTable
///    2. Initiates database connection
///    3. Creates command with query and connection
///    4. Adds parameters to command
///    5. Executes reader and loads data
///    6. Returns filled DataTable
///
/// 4. Error Handling:
///    - All methods include try-catch blocks
///    - Failed operations return appropriate null/false/-1 values
///    - Connections are properly closed in finally blocks
///    - Commands and readers are properly disposed
///
/// 5. Null Value Handling:
///    - Uses predefined NullValues array
///    - Converts null values to DBNull.Value
///    - Handles special cases like DateTime.MinValue
/// </summary>
internal static class ConnectionUtils
{
    public static string ConnectionString = "Data Source=.;Initial Catalog=Twinkies Store;Persist
Security Info=True;User ID=sa;Password=123456;Encrypt=False;TrustServerCertificate=True";
    public static string[] NullValues = { "-1", "", DateTime.MinValue.ToString() };

    /// <summary>
    /// Checks if the given value is identified as null from the globlal static string[] NullValues.
    /// </summary>
    /// <returns>True if the given value is null, Fasle otherwise.</returns>
    private static bool IsNull(string value)
    {
```

```
        foreach (string Null in NullValues)
        {
            if (Null == value) return true;
        }
        return false;
    }

    /// <summary>
    /// Returns the names of the parameters in the given query.
    /// </summary>
    /// <returns></returns>
    private static List<string> _GetParamNames(string query)
    {
        List<string> paramNames = new List<string>();

        StringBuilder Current = new StringBuilder();

        for (int i = 0; i < query.Length; i++)
        {
            if (query[i] == '@')
            {
                while (i < query.Length && query[i] != ' ' && query[i] != ',' && query[i] != ';' && query[i] !=
')')
                {
                    Current.Append(query[i]);
                    i++;
                }

                paramNames.Add(Current.ToString());
                Current = new StringBuilder();
            }

        }
        return paramNames;

    }

    /// <summary>
    /// Initiates a connection with the database using the ConnectionString.
    /// </summary>
    /// <returns>The SqlConnection Initialized by the method.</returns>
    public static SqlConnection InitiateConnection()
    {
        try
```

```
        {
            // Connect with credintials specified by the connection string
            SqlConnection connection = new SqlConnection(ConnectionString);
            connection.Open();
            return connection;
        }
        catch (Exception ex)
        {
            // Logger.Error(ex);
        }
        return null;
    }

    /// <summary>
    /// Ensures the success of a non query command.
    /// </summary>
    /// <returns>The amount of rows affected of the given non-query command.</returns>
    public static int EnsureNonQuerySuccess(ref SqlCommand Command, ref SqlConnection
connection)
    {
        int rowsAffected = 0;
        try
        {
            rowsAffected = Command.ExecuteNonQuery();

        }
        catch (Exception ex)
        {
            //Logger.Error(ex);
        }
        finally
        {
            connection.Close();
            Command.Dispose();
        }
        return rowsAffected;
    }

    /// <summary>
    /// Executes a scalar command and returns the result as a string.
    /// </summary>
    public static string ExecuteScalar(ref SqlCommand Command, ref SqlConnection connection)
    {
        string result = null;
```

```
        try
        {
            result = Convert.ToString(Command.ExecuteScalar());
        }
        catch (Exception ex)
        {
            //Logger.Error(ex);
        }
        finally
        {
            connection.Close();
            Command.Dispose();
        }
        return result;
    }

    /// <summary>
    /// Adds the given parameters to the given command as a query parameters.
    /// </summary>
    public static void AddArgsToCommand(ref SqlCommand command, string query, params
object[] paramaters)
    {
        List<string> queryParameters = _GetParamNames(query);
        for (int idx = 0; idx < paramaters.Length; idx++)
        {
            object param = paramaters[idx];
            if (!IsNull(param.ToString())) command.Parameters.AddWithValue(queryParameters[idx],
param);
            else command.Parameters.AddWithValue(queryParameters[idx], DBNull.Value);
        }
    }

    public static SqlDataReader? GetRow<T>(string query, T PrimaryKey)
    {
        SqlConnection connection = InitiateConnection();

        SqlCommand command = new SqlCommand(query, connection);

        AddArgsToCommand(ref command, query, PrimaryKey);

        try
        {
            SqlDataReader reader = command.ExecuteReader();
            reader.Read();
```

```
            connection.Close();
            command.Dispose();
            return reader;


        }
        catch (Exception ex)
        {
            //  Logger.Error(ex);
        }
        finally
        {
            connection.Close();
        }
        return null;
    }

    public static int AddRowToTable(string query, params object[] parameters)
    {
        SqlConnection connection = InitiateConnection();

        SqlCommand command = new SqlCommand(query, connection);

        AddArgsToCommand(ref command, query, parameters);

        string result = ExecuteScalar(ref command, ref connection);

        if (result is null)
        {
            return -1;
        }
        return Convert.ToInt32(result);

    }

    public static bool UpdateTableRow(string query, params object[] parameters)
    {
        SqlConnection connection = InitiateConnection();

        SqlCommand command = new SqlCommand(query, connection);

        AddArgsToCommand(ref command, query, parameters);

        int RowsAffected = EnsureNonQuerySuccess(ref command, ref connection);
```

```
    return RowsAffected > 0;

}

public static bool DeleteTableRow(string query, params object[] parameters)
{
    SqlConnection connection = InitiateConnection();

    SqlCommand command = new SqlCommand(query, connection);

    AddArgsToCommand(ref command, query, parameters);

    int RowsAffected = EnsureNonQuerySuccess(ref command, ref connection);

    return RowsAffected > 0;

}

public static bool IsRowExist(string query, params object[] parameters)
{
    SqlConnection connection = InitiateConnection();

    SqlCommand command = new SqlCommand(query, connection);

    AddArgsToCommand(ref command, query, parameters);
    bool isFound = ExecuteScalar(ref command, ref connection) != "";

    return isFound;

}

public static DataTable GetTable(string query, params object[] parameters)
{
    DataTable dt = new DataTable();

    SqlConnection connection = InitiateConnection();


    SqlCommand command = new SqlCommand(query, connection);

    AddArgsToCommand(ref command, query, parameters);

    try
```

```
        {
            SqlDataReader reader = command.ExecuteReader();

            if (reader.HasRows)
            {
                dt.Load(reader);
            }

            reader.Close();

        }
        catch (Exception ex)
        {
            //  Logger.Error(ex);
        }
        finally
        {
            connection.Close();
            command.Dispose();
        }

        return dt;
    }

    #region Async Implementation
    /// <summary>
    /// Asynchronously initiates a connection with the database.
    /// </summary>
    /// <returns>A Task representing the asynchronous operation that returns a
SqlConnection.</returns>
    /// <exception cref="SqlException">Thrown when database connection fails.</exception>
    public static async Task<SqlConnection> InitiateConnectionAsync(
        CancellationToken cancellationToken = default)
    {
        try
        {
            SqlConnection connection = new SqlConnection(ConnectionString);
            await connection.OpenAsync(cancellationToken);
            return connection;
        }
        catch (Exception ex)
        {
            // Logger.Error(ex);
            throw;
```

```
    }
}

/// <summary>
/// Asynchronously ensures the success of a non-query command.
/// </summary>
/// <param name="command">The SQL command to execute.</param>
/// <param name="connection">The database connection.</param>
/// <param name="cancellationToken">Cancellation token for the operation.</param>
/// <returns>The number of rows affected by the command.</returns>
public static async Task<int> EnsureNonQuerySuccessAsync(
    SqlCommand command,
    SqlConnection connection,
    CancellationToken cancellationToken = default)
{
    int rowsAffected = 0;
    try
    {
        rowsAffected = await command.ExecuteNonQueryAsync(cancellationToken);
    }
    catch (Exception ex)
    {
        //Logger.Error(ex);
        throw;
    }
    finally
    {
        await connection.CloseAsync();
        await command.DisposeAsync();
    }
    return rowsAffected;
}

/// <summary>
/// Asynchronously executes a scalar command and returns the result as a string.
/// </summary>
public static async Task<string> ExecuteScalarAsync(
    SqlCommand command,
    SqlConnection connection,
    CancellationToken cancellationToken = default)
{
    string result = null;
    try
    {
```

```
            var scalarResult = await command.ExecuteScalarAsync(cancellationToken);
            result = Convert.ToString(scalarResult);
        }
        catch (Exception ex)
        {
            //Logger.Error(ex);
            throw;
        }
        finally
        {
            await connection.CloseAsync();
            await command.DisposeAsync();
        }
        return result;
    }

    /// <summary>
    /// Asynchronously retrieves a single row from the database.
    /// </summary>
    /// <typeparam name="T">The type of the primary key parameter.</typeparam>
    /// <param name="query">The SQL query to execute.</param>
    /// <param name="primaryKey">The primary key value to search for.</param>
    /// <param name="cancellationToken">Cancellation token for the operation.</param>
    /// <returns>A Task representing the asynchronous operation that returns a
SqlDataReader.</returns>
    public static async Task<SqlDataReader> GetRowAsync<T>(
        string query,
        T primaryKey,
        CancellationToken cancellationToken = default)
    {
        var connection = await InitiateConnectionAsync(cancellationToken);
        var command = new SqlCommand(query, connection);

        AddArgsToCommand(ref command, query, primaryKey);

        try
        {
            var reader = await command.ExecuteReaderAsync(cancellationToken);
            await reader.ReadAsync(cancellationToken);
            await connection.CloseAsync();
            await command.DisposeAsync();
            return reader;
        }
        catch (Exception ex)
```

```
    {
        //Logger.Error(ex);
        throw;
    }
    finally
    {
        await connection.CloseAsync();
    }
}

/// <summary>
/// Asynchronously adds a new row to the specified table.
/// </summary>
/// <param name="query">The SQL query to execute.</param>
/// <param name="parameters">The parameters for the query.</param>
/// <returns>The ID of the newly added row, or -1 if the operation fails.</returns>
public static async Task<int> AddRowToTableAsync(
    string query,
    CancellationToken cancellationToken = default,
    params object[] parameters)
{
    var connection = await InitiateConnectionAsync(cancellationToken);
    var command = new SqlCommand(query, connection);

    AddArgsToCommand(ref command, query, parameters);

    var result = await ExecuteScalarAsync(command, connection, cancellationToken);

    if (result is null)
    {
        return -1;
    }
    return Convert.ToInt32(result);
}

/// <summary>
/// Asynchronously updates a row in the specified table.
/// </summary>
/// <returns>True if any rows were affected, false otherwise.</returns>
public static async Task<bool> UpdateTableRowAsync(
    string query,
    CancellationToken cancellationToken = default,
    params object[] parameters)
{
```

```
    var connection = await InitiateConnectionAsync(cancellationToken);
    var command = new SqlCommand(query, connection);

    AddArgsToCommand(ref command, query, parameters);

    int rowsAffected = await EnsureNonQuerySuccessAsync(
        command,
        connection,
        cancellationToken);

    return rowsAffected > 0;
}

/// <summary>
/// Asynchronously deletes a row from the specified table.
/// </summary>
/// <returns>True if any rows were affected, false otherwise.</returns>
public static async Task<bool> DeleteTableRowAsync(
    string query,
    CancellationToken cancellationToken = default,
    params object[] parameters)
{
    var connection = await InitiateConnectionAsync(cancellationToken);
    var command = new SqlCommand(query, connection);

    AddArgsToCommand(ref command, query, parameters);

    int rowsAffected = await EnsureNonQuerySuccessAsync(
        command,
        connection,
        cancellationToken);

    return rowsAffected > 0;
}

/// <summary>
/// Asynchronously checks if a row exists in the specified table.
/// </summary>
/// <returns>True if the row exists, false otherwise.</returns>
public static async Task<bool> IsRowExistAsync(
    string query,
    CancellationToken cancellationToken = default,
    params object[] parameters)
{
```

```
    var connection = await InitiateConnectionAsync(cancellationToken);
    var command = new SqlCommand(query, connection);

    AddArgsToCommand(ref command, query, parameters);
    var result = await ExecuteScalarAsync(command, connection, cancellationToken);

    return result != "";
}

/// <summary>
/// Asynchronously retrieves a DataTable based on the specified query.
/// </summary>
/// <returns>A DataTable containing the query results.</returns>
public static async Task<DataTable> GetTableAsync(
    string query,
    CancellationToken cancellationToken = default,
    params object[] parameters)
{
    var dt = new DataTable();
    var connection = await InitiateConnectionAsync(cancellationToken);
    var command = new SqlCommand(query, connection);

    AddArgsToCommand(ref command, query, parameters);

    try
    {
        using var reader = await command.ExecuteReaderAsync(cancellationToken);
        if (reader.HasRows)
        {
            dt.Load(reader);
        }
        await reader.CloseAsync();
    }
    catch (Exception ex)
    {
        //Logger.Error(ex);
        throw;
    }
    finally
    {
        await connection.CloseAsync();
        await command.DisposeAsync();
    }
```

```
        return dt;
    }
    #endregion
}

'''
```