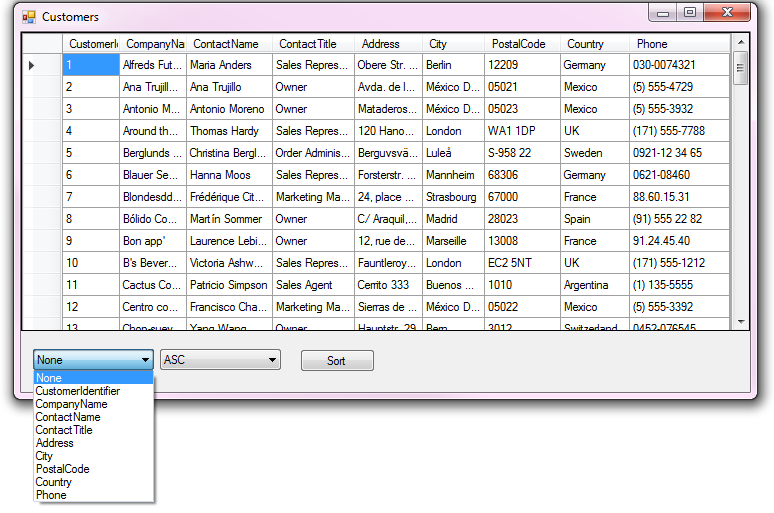
**Introduction**

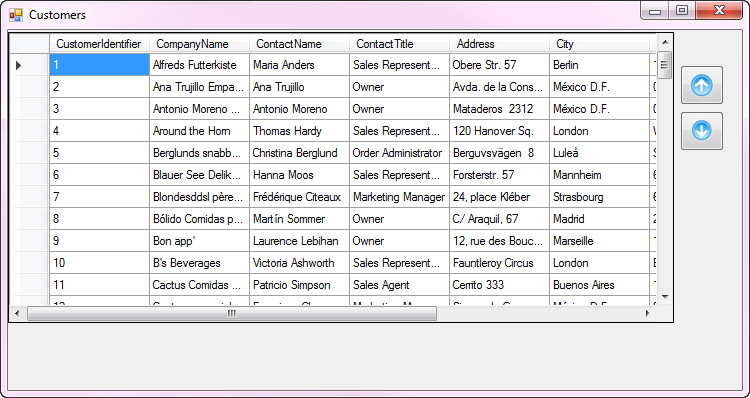
Typical business applications which interact with data usually present data in the user interface un-sorted or sorted by a fixed order or user-defined order (simple example in figure 1) while there is another possibility, provide the user with an option to sort data in a non-orthodox sort order such as business wants the capability to order by row e.g. move row one in a DataGridView, a ListBox, a CheckedListBox or a ListView up or down and remember the position when reopening their application.

This article provides method to provide the ability to move rows up/down and remember positions when working with a SQL-Server database. These methods will also work with other databases by following the logic found in the class responsible for reading and updating backend data. This article would not be complete without mentioning implementing drag and drop functionality. Although for some this may be appealing in reality using drag and drop is harder for a user to master, no different than users mastering drag and down in Windows Explorer. If you have done drag and drop then there is code within the form included for moving rows up/down for a DataGridView coupled together with methods to move rows up/down using language extension methods which is the focus of this article.

**Figure 1**



**Figure 2**



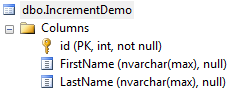
**Fundamentals for working with tables**

The data source for controls presenting data must be unsorted as with sorting enabled when moving rows up or down the underlying data source sort will take over and force the sort leaving the positioning appear as it didn’t work and yes it didn’t work properly.

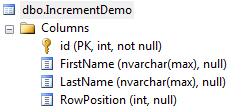
Each database table will require a int field which will hold the row position used when displaying data in a control (DataGridView, ListBox, CheckedListBox or ListView). This field is hidden when presenting data in the user interface. Method provided do not change the row positions but instead house this field and when saving the row position will reorder them.

**Instrumenting a database table** (sub topic)

For demonstration purposes the following table would start with the following columns



Add RowPosition field.



Updating the RowPosition field. In SQL-Server Management Studio create a new query or in Visual Studio using Server Explorer create a new query. Using the table above for this to work a variable is needed to store a integer value which will be used in a UPDATE statement to increment each row. Below there is a UPDATE then a SELECT to verify the UPDATE was successful.

DECLARE @MaxSurrogateKey INT = -1;

UPDATE IncrementDemo

SET @MaxSurrogateKey = RowPosition = @MaxSurrogateKey + 1

WHERE RowPosition IS NULL;

SELECT id ,

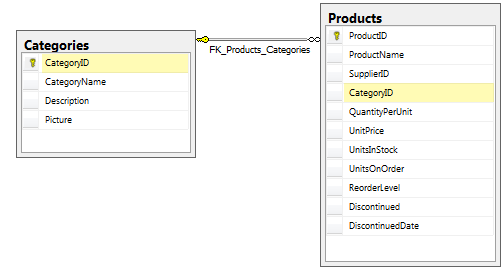
FirstName ,

LastName ,

RowPosition

FROM dbo.IncrementDemo;

The above works fine when a table does not have a foreign key to another table, to handle this situation even though this can be done in SQL a better method would be to write code. Example product table has a category field of type integer which is a key to a category table.



This means each category must be handled since in a schema like this products are shown by category, so each product needs to be updated by category. There is a working example in the code provided which runs from a console application. Step 1 is to obtain all category keys from the category table then pass them off to a method one by one to update each product in the product table.

Console app:

class Program

{

static void Main(string[] args)

{

var ops = new RowPositionWorker();

var catIdentifiers = ops.CategoryIdentifiers();

foreach (var id in catIdentifiers)

{

ops.ProductTable(id);

Console.WriteLine(id);

}

Console.ReadLine();

}

}

Backend code to update the product table

using System;

using System.Collections.Generic;

using System.Data;

using System.Data.SqlClient;

using DataConnections;

using SqlUtilities.ConcreteClasses;

namespace SqlUtilities

{

public class RowPositionWorker : BaseSqlServerConnection

{

/// <summary>

/// You want to add the capability to perform up/down operations in

/// a windows desktop app where prior to this there was no field

/// to store the row position.

///

/// First add a new int field named RowPosition then mimic the

/// code below in ProductTable method. In the case of Product table

/// there is a relationship to the Categories table so we must

/// increment on each category, not the entire table as we want

/// to show one category at a time to the user.

/// </summary>

public RowPositionWorker()

{

DefaultCatalog = "OrderingRows";

}

/// <summary>

/// Pass in default catelog

/// </summary>

/// <param name="pDefaultCatalog"></param>

public RowPositionWorker(string pDefaultCatalog)

{

DefaultCatalog = pDefaultCatalog;

}

public List<int> CategoryIdentifiers()

{

var idList = new List<int>();

var selectStatement = "SELECT DISTINCT CategoryID FROM dbo.Categories ";

using (SqlConnection cn = new SqlConnection() {ConnectionString = ConnectionString})

{

using (SqlCommand cmd = new SqlCommand() {Connection = cn})

{

cmd.CommandText = selectStatement;

cn.Open();

var reader = cmd.ExecuteReader();

while (reader.Read())

{

idList.Add(reader.GetInt32(0));

}

}

}

return idList;

}

/// <summary>

/// This method is for setting RowPosition for a specific category which

/// is needed after adding this field to the product table.

///

/// This assumes product table started prior to implementing rows up/down

/// and that the postion field has been added to the table.

/// </summary>

/// <param name="pCategoryIdentifier"></param>

/// <returns></returns>

public bool ProductTable(int pCategoryIdentifier)

{

mHasException = false;

var selectStatement = "SELECT ProductID,CategoryID,RowPosition " +

"FROM dbo.Products WHERE CategoryID = @CategoryID " +

"ORDER BY CategoryID, ProductName";

var updateStatement = "UPDATE dbo.Products " +

"SET RowPosition = @RowPosition " +

"WHERE ProductID = @ProductId";

var productList = new List<Product>();

using (SqlConnection cn = new SqlConnection() {ConnectionString = ConnectionString})

{

using (SqlCommand cmd = new SqlCommand() {Connection = cn})

{

cmd.CommandText = selectStatement;

cmd.Parameters.AddWithValue("@CategoryID", pCategoryIdentifier);

try

{

cn.Open();

var reader = cmd.ExecuteReader();

var counter = 0;

while (reader.Read())

{

productList.Add(new Product()

{

ProductID = reader.GetInt32(0),

CategoryID = pCategoryIdentifier,

RowPosition = counter

});

counter += 1;

}

reader.Close();

cmd.Parameters.Clear();

cmd.CommandText = updateStatement;

cmd.Parameters.Add(new SqlParameter()

{

ParameterName = "@ProductId",

SqlDbType = SqlDbType.Int

});

cmd.Parameters.Add(new SqlParameter()

{

ParameterName = "@RowPosition",

SqlDbType = SqlDbType.Int

});

for (int index = 0; index < productList.Count; index++)

{

cmd.Parameters["@ProductId"].Value = productList[index].ProductID;

cmd.Parameters["@RowPosition"].Value = productList[index].RowPosition;

cmd.ExecuteNonQuery();

}

}

catch (Exception ex)

{

mHasException = true;

mLastException = ex;

return false;

}

}

}

return true;

}

}

}

The following code snippet provides the logic to reorder the row position field. A DataTable is passed in with a field designated to hold the row position.

public static void ReorderPositionMarker(this DataTable sender)

{

if (sender.Columns.Contains("RowPosition"))

{

int indexer = 0;

for (int index = 0; index < sender.Rows.Count - 1; index++)

{

sender.Rows[index].SetField("RowPosition", indexer);

indexer += 1;

}

}

}

Example table

Product with the following fields; ProductId, ProductName, CategoryId and RowPostion. RowPosition is the field for positioning. We can overload the method as follows to provide a different field name.

public static void ReorderPositionMarker(this DataTable sender, string pFieldName)

{

if (sender.Columns.Contains(pFieldName))

{

int indexer = 0;

for (int index = 0; index < sender.Rows.Count - 1; index++)

{

sender.Rows[index].SetField(pFieldName, indexer);

indexer += 1;

}

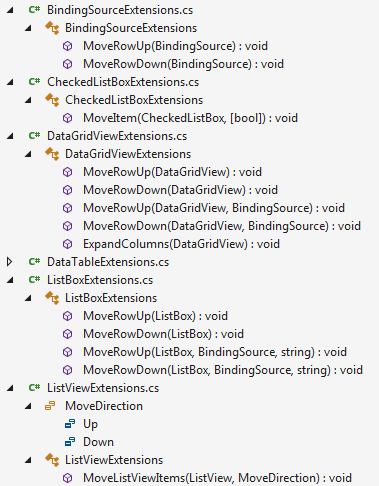
}

}

Both above language extension methods can be used prior to updating the backend database or as in the included code samples, the above logic is built into the update method without using extension methods.

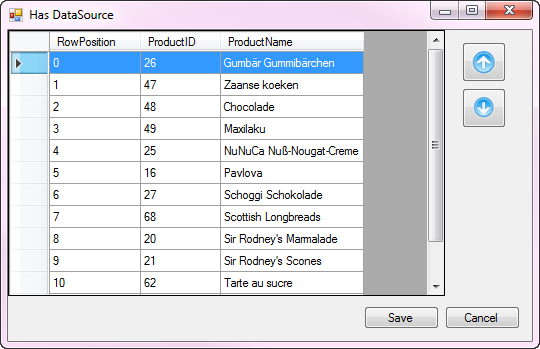
**User interface level/extension methods**

Implementation, examples found on the web are written directly in the form which requires moving rows up/down which means the only way to reuse them is to copy and paste which is not wise and can lead to problems down the road. The implementation in this article are in a class project where each control type has their own class.



There are similarities for each control yet distinct differences which is the reason for breaking each control type into their own classes. For instance, for a ListView we need to work with ListViewItems while a DataGridView there is a DataGridViewRowCollection, a CheckedListBox there is a need to deal with checked items etc.

**DataGridView loaded by its DataSource.**



The two buttons (included in the code samples are two buttons setup with images as shown above) on the right when clicked will call an extension method to move the current row along with setting a form level variable to indicate at least one row has been moved up or down. Note that RowPosition, the field to remember the row position and the primary key are shown for demonstration purposes only. The Save button and cancel button have no click event, instead their DialogResult is set to Ok and Cancel which in turn will close the form.

/// <summary>

/// Move current row up by one, mark \_hasChanges to true

/// to indicate we have at least one change.

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void upButton1\_Click(object sender, EventArgs e)

{

\_bsData.MoveRowUp();

\_hasChanges = true;

}

/// <summary>

/// Move current row down by one, mark \_hasChanges to true

/// to indicate we have at least one change.

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void downButton1\_Click(object sender, EventArgs e)

{

\_bsData.MoveRowDown();

\_hasChanges = true;

}

If \_hasChanges is true then on form closing the following sends the underlying DataTable to a update method in a backend data class.

/// <summary>

/// Check if there have been changes for row move up or down,

/// if there are changes save back to the database table.

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void DataGridViewRememberForm\_FormClosing(object sender, FormClosingEventArgs e)

{

((DataTable) \_bsData.DataSource).ReorderPositionMarker();

if (DialogResult == DialogResult.OK)

{

if (\_hasChanges)

{

\_ops.UpdateProductTable((DataTable)\_bsData.DataSource);

}

}

}

Before updating the database table, the row position field is set in a for/next followed by updating the database table.

public bool UpdateProductTable(DataTable pDataTable)

{

mHasException = false;

var selectStatement = $"UPDATE dbo.Products SET {KeyPositionFieldName} =" +

$" @{KeyPositionFieldName} WHERE ProductID = @ProductId";

using (var cn = new SqlConnection() { ConnectionString = ConnectionString })

{

using (var cmd = new SqlCommand() { Connection = cn })

{

cmd.CommandText = selectStatement;

cmd.Parameters.Add(new SqlParameter()

{

ParameterName = $"@{KeyPositionFieldName}",

SqlDbType = SqlDbType.Int

});

cmd.Parameters.Add(new SqlParameter()

{

ParameterName = "@ProductId",

SqlDbType = SqlDbType.Int

});

try

{

cn.Open();

// used to give new row postion

int newPosition = 0;

for (var rowIndex = 0; rowIndex < pDataTable.Rows.Count; rowIndex++)

{

// set new row position

cmd.Parameters[$"@{KeyPositionFieldName}"].Value = newPosition;

cmd.Parameters["@ProductId"].Value =

pDataTable.Rows[rowIndex].Field<int>("ProductId");

cmd.ExecuteNonQuery();

newPosition += 1;

}

}

catch (Exception e)

{

mHasException = true;

mLastException = e;

}

}

}

return !mHasException;

}

**DataGridView loaded by the row collection rather than its data source**.

The same applies as in the above example, two buttons are required which invoke the extension methods to move rows up/down but in this case there is no row position field.

Code to implement

private void DataGridViewNoRememberForm\_Shown(object sender, EventArgs e)

{

dataGridView1.AutoGenerateColumns = false;

var dt = \_ops.LoadProductsByCategory(true);

foreach (DataRow row in dt.Rows)

{

dataGridView1.Rows.Add(row.Field<int>("Id"), row.Field<string>("Name"));

}

dataGridView1.ExpandColumns();

}

private void upButton1\_Click(object sender, EventArgs e)

{

dataGridView1.MoveRowUp();

}

private void downButton1\_Click(object sender, EventArgs e)

{

dataGridView1.MoveRowDown();

}

Validating the extensions worked by using the code below which writes out the row index and value to the IDE Output window

private void cmdValidate\_Click(object sender, EventArgs e)

{

dataGridView1.Rows.OfType<DataGridViewRow>()

.Select(row => new

{

row.Index,

Value = Convert.ToString(row.Cells["nameColumn"].Value)

})

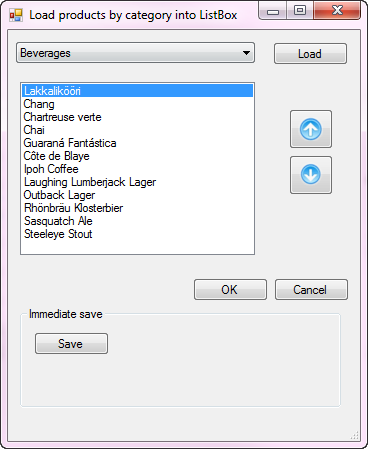
.ToList()

.ForEach(item => Console.WriteLine($"{item.Index,-4} {item.Value}"));

}

**ListBox data source set**.

As with the DataGridView with its DataSource set the same can be done with a ListBox. For demonstration purposes the following screen permits selecting a category to display products. One difference is the “immediate save” button which performs the update as if we had closed the window and the OK and Cancel button have no backing code, they will set DialogResult and close the window.



Just as done with a DataGridView the move up and down are consistent.

private void upButton1\_Click(object sender, EventArgs e)

{

if (lstProducts.SelectedIndex >= 0)

{

lstProducts.MoveRowUp(\_bsData, \_ops.KeyPositionFieldName);

\_hasChanges = true;

}

}

private void downButton1\_Click(object sender, EventArgs e)

{

if (lstProducts.SelectedIndex >= 0)

{

lstProducts.MoveRowDown(\_bsData, \_ops.KeyPositionFieldName);

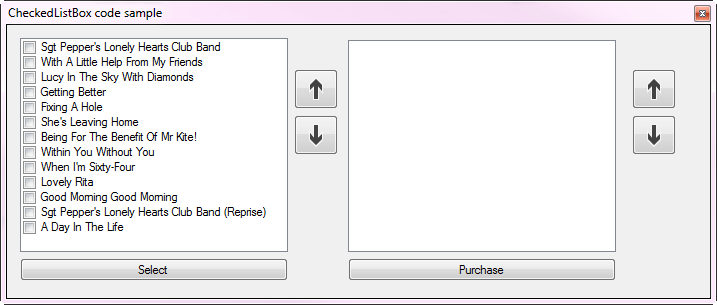
\_hasChanges = true;

}

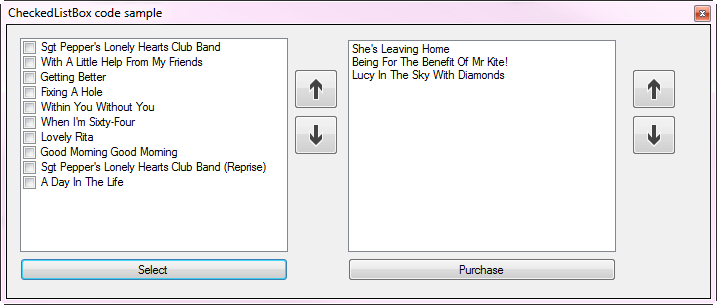
}

**CheckedListBox demonstration**.

It’s unlikely that items in a CheckedListBox need to be remembered (but is possible and ready to go) so the focus will be on a fictitious ordering system to purchase songs from an artist.



Using the left most up/down buttons to reorder songs, press select to move to the ListBox on the right where the rightmost buttons permit reordering the items.



Focusing on the CheckedListBox code, same exact pattern as with the DataGridView and ListBox

private void CheckedListBoxForm\_Shown(object sender, EventArgs e)

{

var ops = new Operations();

var songList = ops.LoadAlbumSongs();

foreach (Song song in songList)

{

checkedListBox1.Items.Add(song);

}

}

private void upButton1\_Click(object sender, EventArgs e)

{

checkedListBox1.MoveItem();

}

private void downButton1\_Click(object sender, EventArgs e)

{

checkedListBox1.MoveItem(false);

}

For the ListBox, same pattern

/// <summary>

/// Move current ListBox item (Song) up one

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void upButton2\_Click(object sender, EventArgs e)

{

listBox1.MoveRowUp();

}

/// <summary>

/// Move current ListBox item (Song) down one

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

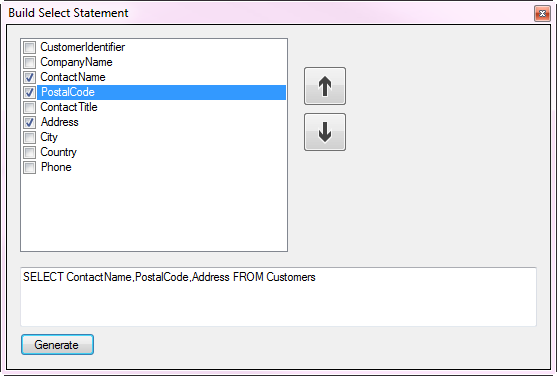
private void downButton2\_Click(object sender, EventArgs e)

{

listBox1.MoveRowDown();

}

Another example for working with a CheckedListBox, provide functionality to start a user interface to build dynamic SQL SELECT statements. The move up/down is no different than the other examples.



**ListView demonstration**.

Performing move up/down requires a little more work as when pressing the up or down button focus is removed from the control and the current row is greyed. Rather than work with a Windows API after the normal move has been performed the item is set as selected then the Focus method is invoked which keeps normal focus on the item currently being moved.

Methods for moving items up/down, same as the other controls

private void upButton1\_Click(object sender, EventArgs e)

{

listView1.MoveListViewItems(MoveDirection.Up);

}

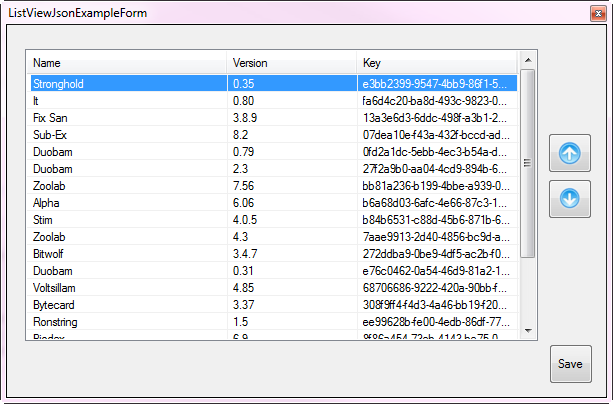
private void downButton1\_Click(object sender, EventArgs e)

{

listView1.MoveListViewItems(MoveDirection.Down);

}

There are two samples for the ListView, one basic to demonstrate moving up/down while the second example loads data from a JSON file, permits moving items up/down then provides the ability to save the changes back to the original file. Note that each time the project runs the JSON file is reset so if you want to inspect the changes open the JSON file in the debug folder prior to rebuilding the solution.



**Requirements**

* SQL-Server, Express edition or better
* Visual Studio 2017 Community Edition or higher version

**Preparing the code samples**.

* Run the script in DataBackEnd class, script.sql either from Visual Studio or within SQL-Server Management Studio.
* Build the solution. If there are build errors that point to Newtonsoft then right click on Solution Explorer and select restore Nu Get packages and then build the solution.

**Special note**.

Although the code to move items up/down is fairly simple your best option prior to implementing these extension methods into your project is to take time to read through the code.

**Implementing into your solution.**

Copy the project CommonLanguageExtensionsLibrary to your solution is the only requirement. Optionally copy WindowsFormsControls project to your solution to use the buttons used in the demos or use standard buttons.

**Conclusion**.

Using code in CommonLanguageExtensionsLibrary provides easy methods to provide methods to move items up/down in your projects without the need to down a good deal of code directly into your forms.

See also:

[User Experience Guidelines](https://msdn.microsoft.com/en-us/library/mt162310.aspx?f=255&MSPPError=-2147217396) <https://msdn.microsoft.com/en-us/library/mt162310.aspx?f=255&MSPPError=-2147217396>

[Extension Methods (C# Programming Guide)](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods) https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods