Articles / Desktop Programming / Windows Forms

.NET2.0 Win2K WinXP Win2003 VS2005 C# Windows .NET Visual-Studio WinForms

OutlookGrid: grouping and arranging items in Outlook style

Herre Kuijpers

30 Jun 2006 CPOL 15 min read 1.3M 45.7K 559

A grid allowing grouping and arranging items, much like Outlook.

Download source files - 138 Kb

Download demo project - 67.7 Kb

Download control only - 11.2 Kb

Introduction

If you work a lot with lists larger than, say, a 100 items (for instance, the list of emails in your inbox), it quickly becomes difficult and tedious to find a specific item without resorting to filtering, searching, sorting, and/or grouping functions. Especially, sorting and grouping greatly improves the structuring of the items in a list, and it is a feature that I would want to apply to all my lists, by default. In particular, I've been looking for a list/grid control that allows arranging and grouping of similar items together, much like the grid (or list?) used in Outlook 2003.

I know there are a few commercial lists/grids out there that support this kind of thing; however, I also encountered a couple of bugs while trying them out. Not having access to the source code makes it very frustrating, so I thought I might as well make it a CodeProject article and see if I could come up with a custom solution.

Because a grid is more flexible than a list, I decided to implement a grid control that can group items together, just like Outlook. This control is implemented with VS 2005 in C# 2.0. Now, I can't guarantee that this implementation is without bugs, but at least, it is free, and it includes the sources. So, feel free to modify them where needed, for your own purposes. Note however that, this control is **not** finished! Some functions may not work correctly or not at all. It is focused mainly at sorting, grouping, and displaying items on the grid, which I think it does rather well. Inserting, updating, and deleting rows and cells in the grid has not had much of my attention.

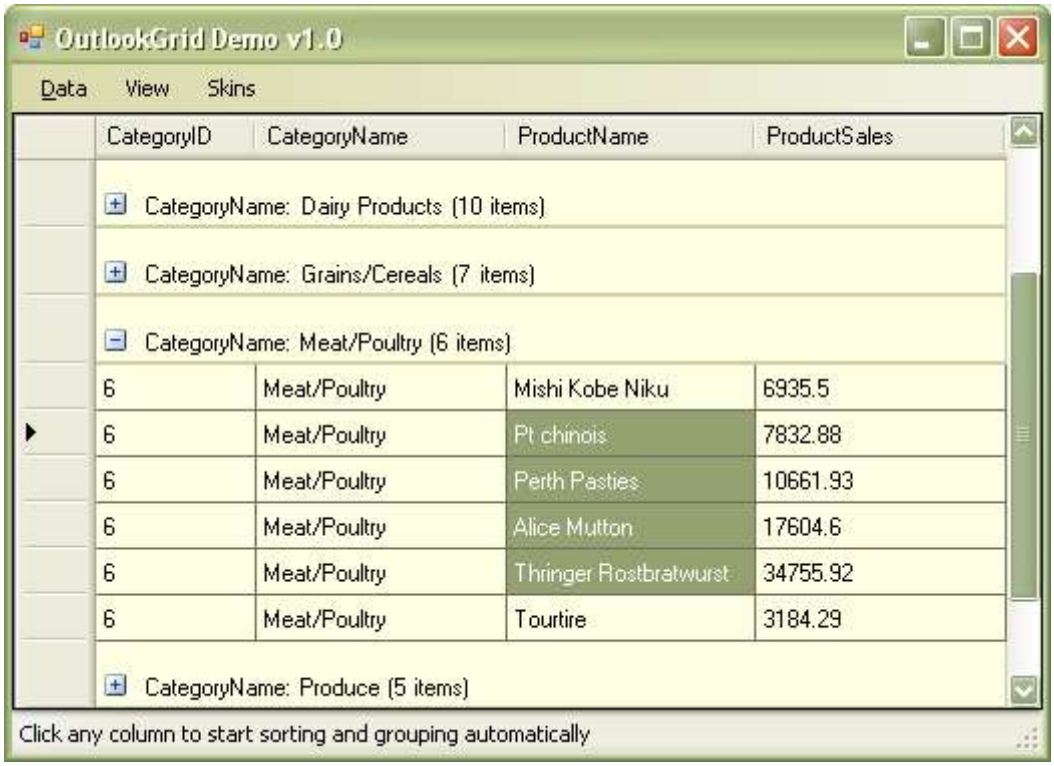
In this article, I will explain how the control can be used, and what it can do, and also what it *can't* do, mainly focusing on the developer group that just wants to reuse this control as is. Then, I will explain a

little in more detail how the internals of this control work, for the developers that feel like extending or changing the control's implementation for their own purposes.

Background

The `OutlookGrid` is derived from the `DataGridView` control newly introduced in VS2005. If you are familiar with the `DataGridView`, the `OutlookGrid` should be really easy to integrate into your solutions. If you have done a bit of GDI+ programming and customized controls before, the `OutlookGrid` should not be too hard to extend.

I wanted to create the `OutlookGrid` using as little code as possible and as simple as possible. Therefore, the control does **not** make use of complicated hooks, callbacks, and Windows APIs. It simply overrides a number of the `DataGridView`'s event handlers. Unfortunately, the `DataGridView` implements quite a few events, and it took me whole lot of hours to decide which events to override. Also, it took my quite some time to figure out a workable solution to make the grid easy to use.



Using the code

Assuming that you have created a C# Windows application project in VS2005, add the *OutlookGrid.cs*, *OutlookGrid.Designer.cs*, *OutlookGridRow.cs*, *OutlookGridGroup.cs*, and the *DataSourceManager.cs* files to your project. Before adding the `OutlookGrid` control on your forms, make sure you compile everything first. After that, the control is added to your toolbox. You can now drag it onto your form.

Once the control is in place, there are two ways to fill the grid:

- add the columns, rows, and cells manually (unbound), or
- use data binding (bound).

The latter one being the easiest to implement as shown in the code example.

In this article, I will not discuss all the options, however, they are implemented as examples in the demo project and in the sources, and should be quite straightforward once I have explained the concept.

Data binding

Currently, only two data types can be used for data binding: a `DataSet` or an object array list (the list must implement `IList`, for instance, the `ArrayList`). Other types are as of yet not supported, like, for instance, a `DataTable` or a `DataTableView`.

Add the following code when setting up the form:

C#

```
//create a dataset object
DataSet dataset = new DataSet();

// fill the dataset, e.g. by reading the data from an xml file.
dataset.ReadXml(@"invoices.xml");

// bind the dataset to the OutlookGrid (named outlookGrid1
// in this example). Set the dataMember variable to
// "invoice", indicating the name of the table
// in the dataset to display in the grid.
outlookGrid1.BindData(dataset, "invoice");
```

Notice that the `OutlookGrid` uses the `BindData()` method to bind the data, instead of setting the `DataSource` and `DataMember` properties. The `DataSource` and `DataMember` properties are now read-only. To clear the binding, use: `outlookGrid1.BindData(null, null);`.

Grouping and sorting

So far so good, but we do not have any arrangement/grouping yet! To use grouping, the grid needs to be sorted. For this, the `Sort(...)` method has been implemented. Groups of items (rows) are created by selecting the rows from the logical sort order of the rows and assigning the rows with similar values to the same group. This is a two step process. First, specify how items are to be grouped, then sort the items.

Grouping can be done based on different criteria, for instance, items can be grouped alphabetically, putting all items starting with the letter 'A' (or 'a') in the same group. By default, however, items are grouped based on their *string* value, so all items with the same *string* value will be put in the same group. To let the `OutlookGrid` know what grouping is to be used, we set the `outlookGrid1.GroupTemplate` property. This property takes an instance of the `Group` to use. By

default, it is set to `OutlookGridDefaultGroup`. All the group rows that are created during sorting will be literally cloned from the `GroupTemplate` object.

So, in order to group items in our example, we need to sort them on one of the item's attributes. In this example, we will sort the rows in the "invoice" `DataTable` (remember, we bound a `DataSet` to the grid). The rows in a `DataTable` are of type `DataRow`, so we need to sort `DataRow`s. To sort items, .NET uses a comparer object. A comparer object implements the `IComparer` interface. In our example, I have defined a `DataRowComparer` class (in the *Form1.cs* file) which will be used to sort items in the grid (not the `DataTable`!).

So basically, it comes down to this:

C#

```
// specify which column to sort based on an index:
int ColumnIndex = 2;

// set the group template to use, e.g. to sort alphabetically:
outlookGrid1.GroupTemplate = new OutlookGridAlphabeticGroup();

// specify the column the Group will be associated with:
outlookGrid1.GroupTemplate.Column = outlookGrid1.Columns[ColumnIndex];

// all groups in the list will be collapsed,
// so only the groups are displayed, not the items
outlookGrid1.GroupTemplate.Column.Collapsed = true;

// sort the grid using the DataRowComparer object
// the DataRowComparer constructor takes two parameters,
// the column that will be sorted on, and the direction
// in which to sort (ascending or descending)
outlookGrid1.Sort(new DataRowComparer(ColumnIndex, direction));
```

After executing the code above, the grid will display all the items grouped alphabetically. On the other hand, if you want to sort the list, but for some obscure reason do not want to group the items, simply set `outlookGrid1.GroupTemplate = null;` before calling the `Sort` method.

Alternatively, the `OutlookGrid` supports additional functions to specify how the items are to be displayed:

- `CollapseAll()` will collapse all the groups in the grid, making all items invisible and displaying only the groups.
- `ExpandAll()` will expand all groups, displaying all groups and their items.
- `ClearGroups()` will remove all groups, and simply display only the items.
- `CollapseIcon` and `ExtendIcon` properties specify what images to use for the + and - signs in the group. If these are *not* set, the + and - are *not* rendered.

Of course, the `OutlookGrid` also supports all other well known `DataGridView` methods and properties.

Unbound data

Now that we have seen how the grid works for bound data, I will now explain shortly how the grid can be setup with unbound data. The grid can be setup like it is done for the `DataGridView`, using the `Columns.Add()` and `Rows.Add()` methods. An exception has to be made, however, when creating the rows: **each row must be of the `OutlookGridRow` type!** Use the row's `CreateCells()` function to fill the cells in each row, then add the row to the `Rows` collection of the grid:

C#

```
// first clear any previous bindings (if they were set)
outlookGrid1.BindData(null, null);

 // setup the column headers
outlookGrid1.Columns.Add("column1", "Id");
outlookGrid1.Columns.Add("column2", "Name");
etc...

//then create the rows
// row 1:
OutlookGridRow row = new OutlookGridRow();
row.CreateCells(outlookGrid1, id1, name1, ...);
outlookGrid1.Rows.Add(row);

// row 2:
OutlookGridRow row = new OutlookGridRow();
row.CreateCells(outlookGrid1, id2, name2, ...);
outlookGrid1.Rows.Add(row);
//etc...
```

Because we have no underlying data source which can be used to sort the grid, sorting must be done based on the contents of the grid itself. This means that during sorting, the items of the grid itself will need to be compared. This is done using the `OutlookGridRowComparer` object. This comparer compares the items in the list based on their string value only. An easier option, however, is to use the alternative `Sort()` method, specifying only the column to sort on and the direction in which to sort (ascending or descending):

C#

```
// set the column to be used for grouping
outlookGrid1.GroupTemplate.Column = outlookGrid1.Columns[e.ColumnIndex];

// then select one of 2 ways of sorting.
// option 1: the easy way, use the OutlookGridRowComparer object
outlookGrid1.Sort(new OutlookGridRowComparer(ColumnIndex, direction));

// option 2: the even easier way, specify which column to sort on
outlookGrid1.Sort(outlookGrid1.Columns[ColumnIndex], direction);
```

This concludes the introduction on 'Using the code'. So far, the basic sorting and grouping functionality works pretty well, even for larger datasets; for instance, the Invoice example contains over 2000 records, and still performs pretty OK on my computer. Given this code is fully written in C#, that's not bad at all! B-)

Missing and untested features

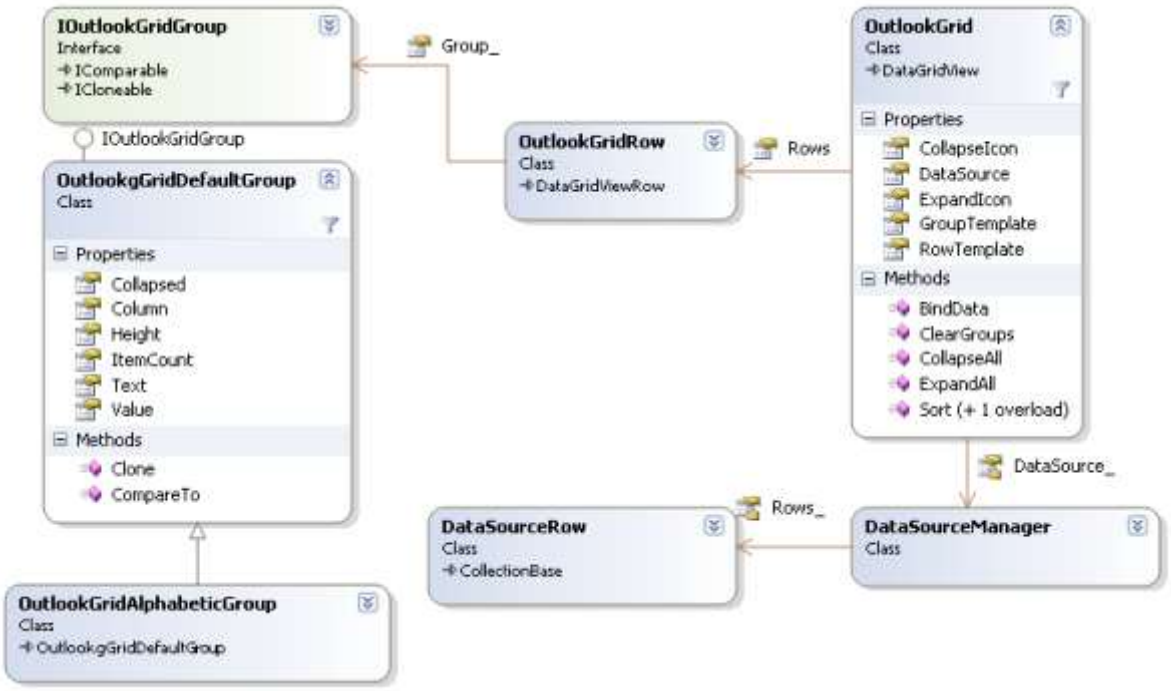
Because the `DataGridView` base control contains numerous methods, events, and properties that are all inherited by the `OutlookGrid`, I have not taken the effort to test them all with the `OutlookGrid` implementation. This means, it is very likely that you will run into bugs or missing functionality once you start using the `OutlookGrid` for other functions than described in this article. Because I already ran into some of them, I will list the ones that I know of:

- The `OutlookGrid` does not support nested grouping, unfortunately. That would be the next step to take.
- Changing the display styles of the grid may result in the Groups not being rendered 100% correctly.
- Currently, the text color of the Group is set to black and cannot be changed, you will need to change the `Paint()` method in the `OutlookGridRow` class for that.
- I have not tested the grid using `VirtualMode`. To be honest, I have no idea how that concept works, so I doubt that items will be displayed correctly once you turn it on.
- Bound data sources are not directly bound to the base `DataGridView` control. Therefore, the data binding only works for displaying items. However, once you edit items in the grid, the data source will not be updated. You will have to implement this manually.
- This also means that new items in the grid will not automatically be appended to the data source. This will have to be done manually as well.
- Because the Group row overrides the default rows, not all the events fired for normal rows will be fired for group rows. For instance, the group row overrides the `OnDoubleClick` event, to automatically collapse or expand. This behavior cannot be altered without changing the code.

No doubt there are many other issues, please report them so I and other developers can benefit from this. Perhaps, I will invest more time developing more features for this control.

Design and extensibility

In this section, I want to describe in more detail how the control is implemented, specifically targeted at the developer audience that might want to do some coding on the control. I tried to make something like an UML diagram in VS2005, but well... this diagram will have to do.



OutlookGrid

The `OutlookGrid` is the main object that references and controls all the other objects. Apart from the properties, methods, members, and collections it inherits from the `DataGridView`, like, for instance, `Columns`, three members are particularly interesting:

- `RowTemplate` property, and related to this, the `Rows` collection.
- `DataSource` property which is used to handle our own `DataSource`s.
- `GroupTemplate` property is newly added, and determines what group object should be used.

The `OutlookGrid` only works with `OutlookGridRow` objects. Therefore, the `RowTemplate` property has been overridden as new so that it does not allow setting a new `RowTemplate` object. This means that the `Rows` collection will contain only `OutlookGridRow` objects. This is important because the `OutlookGridRow` determines how a row is rendered on the control.

Apart from that, the `OutlookGrid` also manages its own data sources using the `DataSourceManager` object. An interesting aspect here is that the `DataSourceManager` can use the `OutlookGrid` as a data source as well! This is particularly useful when working with unbound data. For the user, however, this is transparent.

The `GroupTemplate`, as shown in the examples earlier, determines mainly how Groups are created during the sort operation. New groups are created dynamically by cloning the `GroupTemplate` object. So, changing the properties of the group template before sorting will result in all groups cloning these properties.

OutlookGridRow

The `OutlookGridRow` has been extended with two new properties:

- `IsGroupRow` specifies if this row is rendered as a group or as a regular row.
- `Group` specifies the group this row belongs to.

So, this means that a row will be rendered either as a Group-row displaying the expand/collapse icon and the group text, or the row is displayed as a regular row simply by calling the base class to render itself. To do this properly, two methods need to be overridden: the `Paint` and `PaintCells` methods. An additional method could be overridden that determines how the row headers are rendered: `PaintHeader`.

Because each row is placed in a group and will get a reference to it, each row can also determine for itself whether it should be rendered or not. E.g., if the group is collapsed, rows should not render. To let the base control think that the item was set to invisible, instead of setting the `Row.Visible` property to `false`, we need to override the `GetState` method. Somehow, setting the `Visible` property of a `Row` triggers all kinds of events and initiates the base control to redraw. Also, rows that are set to invisible will not be rendered, once its group is marked extended again! To work around this problem, we override the `GetState` method instead. The `GetState` method will mark the row as `readonly`, but *not for display*, however the `Row`'s `Visible` property will still be `true`! This will make the base control keep trying to render the `Row`. This is exactly the behavior we need to support Collapse and Expand functionality.

Group objects

`IOutlookGridRow` is the interface that must be implemented by all Group classes to be used with this control. By default, the control will make use of the `OutlookGridDefaultGroup`. The implementation of a Group class is not too difficult to understand. It is important though that the `Clone` and `CompareTo` functions are correctly implemented by each Group class, and correspond to the Comparer object's behaviour when sorting the grid.

The `CompareTo` function, typically, compares the row's value against the group's value. The `Text` property specifies what text will eventually be displayed on screen. So, if you, for instance, are sorting and grouping on a `DateTime` attribute, it is fairly easy to display, for instance, the name of the month, instead of the `DateTime` value in the group, or even fancier, like Outlook, display group texts like 'Date: Yesterday', 'Date: Last week', 'Date: Last month' etc.

DataSourceManager

If you managed to read all the text up to this point, you have my respect already :-) I guess, in that case, you must be really anxious to know how this little story ends ;-)

We now come to the most complicated part of the control. I ran into problems when I wanted to support both Bound and Unbound data sources. Once data is bound to the `DataGridView` base control class, the behavior of the base control becomes very hard to influence, and it feels like it has a mind of its own! For example, not being able to add rows to the grid, once it is databound, is really frustrating.

I saw only one way out, and that is to override the base control's `DataSource` property and simply not bind data to the base control itself. Instead, I created my own `DataSourceManager` class. I have to admit that this got me into more trouble than I'd like to. Suddenly, I had to implement data binding! Not that I am particularly fond of data binding, since in my humble opinion, it basically totally destroys any concept of architectural layering, using multiple logical tiers and separation of business from presentation. But OK, that's another discussion. On the other hand, I have to admit that it is pretty funky once you can sort, group, and render a whole `DataSet` in just a few lines of code.

So, if you want to use additional data sources like, for instance, `DataTable`s or `DataTableView`s, you will have to do some coding. Right now, the `DataSourceManager` has been implemented in a crude way. It, basically, is now an indexer class, that indexes both columns and rows of the data source (implemented as simple `ArrayList`s), in order to allow the `OutlookGrid` quick access to the actual data. If, for instance, you bind an `ArrayList` with business objects to the `OutlookGrid`, its properties will be indexed as columns (using reflection, in this case), and each object will be mapped to a row in the `DataSourceManager`. This also means that when you sort the `OutlookGrid`, actually, only the index rows in the `DataSourceManager` are sorted!

So, you can view the `DataSourceManager` as a level of abstraction between the `OutlookGrid` and the actual data (yes, I just love layering :-). An interesting detail here is that it is possible to bind the `OutlookGrid` itself as a data source to the `DataSourceManager`! Even when your `OutlookGrid` is grouped, the `DataSourceManager` will index only the rows that are not group rows! This makes it very easy to sort and group unbound data that was put into the grid earlier. This way, I killed both the bound- and unbound data problem with a single stone (yes, yes, I just love abstraction classes, separation of business and presentation etc. etc.)!

Points of interest

I like to conclude that in this CodeProject article, I managed to tackle a number of problems, e.g., which methods and events to override in the `DataGridView` control, and how to handle data binding. Altogether, it was quite a challenge for me to get this up and running. As stated before, the `OutlookGrid` control is far from finished, but for the main purpose it was built for, that is grouping, it can be used excellently! Well, to wrap it up, I hope that you enjoyed reading this article, and that it has given you some food for thought and renewed inspiration! If I have been unclear on certain subjects regarding the control, or if you need more detailed information on how to deal with specific implementations, or if you have some good pointers on how to implement data binding, drop me a note in the comments section :-)

History

Version 1.0 of the OutlookBar control was written between the 1st and the 6th of June 2006. This article was published on the 8th of May 2006.

License

Written By

Herre Kuijpers

Architect Rubicon

Netherlands

Currently Herre Kuijpers is employed at Rubicon. During his career he developed skills with all kinds of technologies, methodologies and programming languages such as c#, ASP.Net, .Net Core, VC++, Javascript, SQL, Agile, Scrum, DevOps, ALM. Currently he fulfills the role of software architect in various projects.

Herre Kuijpers is a very experienced software architect with deep knowledge of software design and development on the Microsoft .Net platform. He has a broad knowledge of Microsoft products and knows how these, in combination with custom software, can be optimally implemented in the often complex environment of the customer.

Comments and Discussions

272 messages have been posted for this article Visit **https://www.codeproject.com/Articles/14388/OutlookGrid-grouping-and-arranging-items-in-Outloo** to post and view comments on this article, or click **here** to get a print view with messages.