ComponentOne

FlexGrid for WPF and Silverlight

ComponentOne, a division of GrapeCity
201 South Highland Avenue, Third Floor
Pittsburgh, PA 15206 USA

Website: http://www.componentone.com
Sales: sales@componentone.com
Telephone: 1.800.858.2739 or 1.412.681.4343 (Pittsburgh, PA USA Office)

Trademarks

Warranty

ComponentOne warrants that the media on which the software is delivered is free from defects in material and workmanship, assuming normal use, for a period of 90 days from the date of purchase. If a defect occurs during this time, you may return the defective media to ComponentOne, along with a dated proof of purchase, and ComponentOne will replace it at no charge. After 90 days, you can obtain a replacement for the defective media by sending it and a check for $25 (to cover postage and handling) to ComponentOne.

Except for the express warranty of the original media on which the software is delivered is set forth here, ComponentOne makes no other warranties, express or implied. Every attempt has been made to ensure that the information contained in this manual is correct as of the time it was written. ComponentOne is not responsible for any errors or omissions. ComponentOne’s liability is limited to the amount you paid for the product. ComponentOne is not liable for any special, consequential, or other damages for any reason.

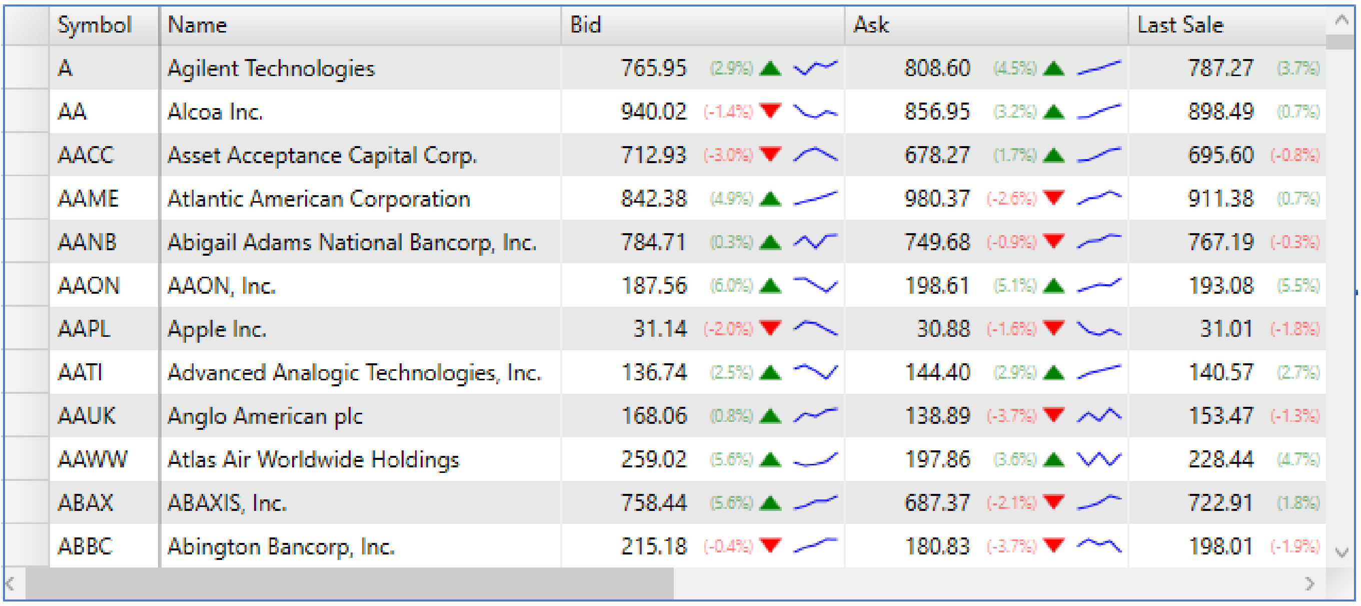
Copying and Distribution

While you are welcome to make backup copies of the software for your own use and protection, you are not permitted to make copies for the use of anyone else. We put a lot of time and effort into creating this product, and we appreciate your support in seeing that it is used by licensed users only.

1 Table of Contents

FlexGrid for WPF and Silverlight Overview

FlexGrid for WPF and Silverlight is a lightweight data grid control designed on a flexible object model. Based on the popular WinForms version, **FlexGrid** offers many unique features such as unbound mode, flexible cell merging, and multi-cell row and column headers that make it a proven solution for data management and tabulation.



Symbol	Name	Bid	Ask	Last Sale
A	Agilent Technologies	765.95 (2.9%) ▲	808.60 (4.5%) ▲	787.27 (3.7%)
AA	Alcoa Inc.	940.02 (-1.4%) ▼	856.95 (3.2%) ▲	898.49 (0.7%)
AACC	Asset Acceptance Capital Corp.	712.93 (-3.0%) ▼	678.27 (1.7%) ▲	695.60 (-0.8%)
AAME	Atlantic American Corporation	842.38 (4.9%) ▲	980.37 (-2.6%) ▼	911.38 (0.7%)
AANB	Abigail Adams National Bancorp, Inc.	784.71 (0.3%) ▲	749.68 (-0.9%) ▼	767.19 (-0.3%)
AAON	AAON, Inc.	187.56 (6.0%) ▲	198.61 (5.1%) ▲	193.08 (5.5%)
AAPL	Apple Inc.	31.14 (-2.0%) ▼	30.88 (-1.6%) ▼	31.01 (-1.8%)
AATI	Advanced Analogic Technologies, Inc.	136.74 (2.5%) ▲	144.40 (2.9%) ▲	140.57 (2.7%)
AAUK	Anglo American plc	168.06 (0.8%) ▲	138.89 (-3.7%) ▼	153.47 (-1.3%)
AAWW	Atlas Air Worldwide Holdings	259.02 (5.6%) ▲	197.86 (3.6%) ▲	228.44 (4.7%)
ABAX	ABAXIS, Inc.	758.44 (5.6%) ▲	687.37 (-2.1%) ▼	722.91 (1.8%)
ABBC	Abington Bancorp, Inc.	215.18 (-0.4%) ▼	180.83 (-3.7%) ▼	198.01 (-1.9%)

Help with WPF and Silverlight Edition

Getting Started

For information on installing **ComponentOne Studio WPF and Silverlight Edition**, licensing, technical support, namespaces and creating a project with the controls, please visit Getting Started with WPF Edition or Getting Started with Silverlight Edition.

Key Features

FlexGrid offers several advanced data visualization features that are beyond simple grids. These features are listed below:

- **Flexible data binding**
  FlexGrid can be used in bound mode, where it displays data from a data source, or in unbound mode, where the grid itself manages the data.

- **Advanced grid features**
  FlexGrid supports advanced grid features including cell merging, data filtering, sorting, editing, aggregation etc. You can merge contiguous, like-valued cells to make the data span across multiple cell; calculate totals averages, and other statistics for ranges of cells; and apply filters to each column on the grid.

- **Hierarchical styles**
  FlexGrid summarizes data in hierarchical style like a tree. Each record can be expanded or collapsed to expose details in child grids.

- **Integrated printing support**
  FlexGrid supports integrated printing wherein it has control over paper orientation, margins, and footer text. With a rich object model, the control provides varied printing events to handle page breaks, add repeating header rows, or add custom elements to each page. You can also show a dialog to let users select and set up the printer.

- **Advanced grouping and filtering feature**
  FlexGrid supports grouping as a UI feature through a separate control, FlexGridGroupPanel, available as a separate assembly. Similarly, the control comes with FlexGridFilter component, which provides ad-hoc filtering and is shipped separately to achieve lower footprint.

- **Custom cells**
  FlexGrid supports significant customization in the grid through custom cells. The control provides CellFactory class and built-in CellTemplate and CellEditingTemplate to customize visual elements.

- **Row details**
  FlexGrid provides the flexibility to show row details in a data template, which can be used to display text, images as well as data bound controls.

Feature Comparison

This section provides you with comparison matrices to compare features offered by FlexGrid across different platforms, and features of FlexGrid for WPF and Silverlight with those of other grid controls.

Comparing WPF Grids
    Comparison of various grids available in WPF and Silverlight edition.
Comparing FlexGrids
    Comparison of object model of C1FlexGrid class with FlexGrid in other edition.

Comparing WPF Grids

Explore all of the features offered by various WPF grids including C1FlexGrid, C1DataGrid, and MS DataGrid. You can download the matrix in PDF.

Data Binding

Features	C1FlexGrid	C1DataGrid	MS DataGrid
Bound mode	✓	✓	✓
Unbound mode	✓		

Layout and Appearance

Features	C1FlexGrid	C1DataGrid	MS DataGrid
Themes	17 Themes	17 Themes	
ClearStyle		✓	

Presentation

Features	C1FlexGrid	C1DataGrid	MS DataGrid
Autogenerate Columns	✓	✓	✓
Text Column	✓	✓	✓
CheckBox Column	✓	✓	✓
ComboBox Column	✓	✓	✓
Hyperlink Column		✓	✓
DateTime Column		✓	
Numeric Column		✓	
Image Column		✓	
Frozen Columns	✓	✓	✓
Frozen Rows	✓	✓	
Custom Columns	✓	✓	✓
Custom Rows		✓	

Custom Cells (Cell Factory)	✓		
Add New Row	✓	✓	✓
Merged Cells	✓	✓	
Row Details	✓	✓	✓
Hierarchical View	✓	with custom code	

Sorting

Features	C1FlexGrid	C1DataGrid	MS DataGrid
ICollectionView	✓	✓	✓

Filtering

Features	C1FlexGrid	C1DataGrid	MS DataGrid
ICollectionView	✓	✓	✓
Excel-like Filtering	✓	✓	
Filter Row	with custom code	✓	
Custom Filters		✓	
Full-text Search		✓	

Grouping

Features	C1FlexGrid	C1DataGrid	MS DataGrid
ICollectionView	✓	✓	✓
Drag and Drop Grouping	✓	✓	
Subtotals	✓	✓	

Editing

Features	C1FlexGrid	C1DataGrid	MS DataGrid
In-cell Editing	✓	✓	✓
Validation	✓	✓	✓
IDataErrorInfo	✓	✓	✓
IEditableObject		✓	✓
ICustomTypeDescriptor	✓	✓	✓
Data Annotations	✓	✓	

Printing

Feature	C1FlexGrid	C1DataGrid	MS DataGrid
Printing	✓	✓	

Export

Features	C1FlexGrid	C1DataGrid	MS DataGrid
Excel	✓	✓	
Text	✓		
HTML	✓		

Ux

Features	C1FlexGrid	C1DataGrid	MS DataGrid
Keyboard Navigation	✓	✓	✓
RTL Support	✓	✓	✓
Touch Support	✓	✓	✓
Clipboard Support	✓	✓	✓
Multiple Selection Modes	✓	✓	✓

Localization

Features	C1FlexGrid	C1DataGrid	MS DataGrid
.NET Localization Support	✓	✓	✓
Included Translations	25 Languages	25 Languages	✓
Regional Settings (Number, date, currency)	✓	✓	✓

Performance

Features	C1FlexGrid	C1DataGrid	MS DataGrid
Deferred Scrolling	✓	✓	✓
UI Virtualization	25 Languages	25 Languages	✓
Server-side Data Virtualization with C1DataSource	✓	✓	

Other

Features	C1FlexGrid	C1DataGrid	MS DataGrid
Design-time Support	✓	✓	✓
WPF/Silverlight Compatibility	✓	✓	✓
UI Automation	✓	✓	✓
Assembly Size	301 KB	776 KB	part of PresentationFramework.dll

FlexGrid API Comparison

Explore the object model of C1FlexGrid class in WinForms, and WPF and Silverlight editions.

Elements

Properties

FlexGrid for WinForms	FlexGrid for WPF and Silverlight	Comments
BottomRow	BottomRow	
Cols	Columns	
ExtendLastCol		Set the width of the last column to "*".
LeftCol	LeftColumn	
Rows		
RightCol		
TopRow		

Methods

FlexGrid for WinForms	FlexGrid for WPF and Silverlight	Comments
FindRow(...)		Not supported (Can be easily done through code)

Events

FlexGrid for WinForms	FlexGrid for WPF and Silverlight	Comments
EnterCell	SelectionChanged	
LeaveCell	SelectionChanging	
RowValidated	OnRowEditEnded	
RowValidating	OnRowEditEnding	

Core Features

Properties

FlexGrid for WinForms	FlexGrid for WPF and Silverlight	Comments
AllowAddNew		Supported at data source level, no built-in UI.
AllowDelete		Not supported.
AllowEditing	IsReadOnly	Renamed for consistency across WPF and Silverlight controls.
AllowFiltering		Supported at data source level, no built-in UI.
AllowFreezing		See Rows.Frozen and Columns.Frozen properties.
AllowMerging	AllowMerging	
AllowSorting	AllowSorting	
AutoSearch		Not supported.
AutoSearchDelay		Not supported.
AutoGenerateColumns	AutoGenerateColumns	
DataSource	ItemsSource	
Enabled	IsEnabled	
EditOptions		See CellFactory property, PrepareCellForEdit event.
SelectionMode	SelectionMode	
Subtotal	Column.GroupAggregate	
SubtotalPosition		Not supported.

Methods

FlexGrid for WinForms	FlexGrid for WPF and Silverlight	Comments
BeginUpdate		
EndUpdate		

FinishEditing (bool cancel)		
LoadExcel (string)		
SaveExcel (string)		
Select(int row, int col, bool scrollIntoView)		
Sort (order, int col1, int col2)		
StartEditing (row, col)		

Events		
FlexGrid for WinForms	**FlexGrid for WPF and Silverlight**	**Comments**
AfterSort		
AfterSubtotal		Not supported
BeforeSubtotal		Not supported
StartEdit	BeginningEdit	
SetUpEditor	PrepareCellForEdit	

Layout and Appearance

Properties		
FlexGrid for WinForms	**FlexGrid for WPF and Silverlight**	**Comments**
BackColor	Background	
ClipboardCopyMode	ClipboardCopyMode	
DrawMode		See CellFactory property.
HighLight		Not supported.
NewRowWatermark		Not supported.
ShowButtons		Not supported.
ShowCellLabels		Not supported.
ShowCursor		Not supported.
ShowErrors		Not supported.
ShowSort	ShowSort	
FocusRect		Not supported.

User Interaction

Properties		
FlexGrid for WinForms	**FlexGrid for WPF and Silverlight**	**Comments**
AllowDragging	AllowDragging	

AllowResizing	AllowResizing	
AutoResize		See AutoSizeRows and AutoSizeColumns methods.
ScrollBars	HorizontalScrollbarVisibility VerticalScrollbarVisibility	Not supported.
ScrollOptions		Not supported.
ScrollPosition	ScrollPosition	

Events		
FlexGrid for WinForms	**FlexGrid for WPF and Silverlight**	**Comments**
AfterScroll	ScrollPositionChanged	
BeforeScroll	ScrollPositionChanging	
AfterSelChange	SelectionChanged	
BeforeSelChange	SelectionChanging	
SelChange	SelectionChanged	

Keyboard Interaction

Properties		
FlexGrid for WinForms	**FlexGrid for WPF and Silverlight**	**Comments**
AutoClipboard		See ClipboardCopyMode and ClipboardPasteMode
ClipboardCopyMode	ClipboardCopyMode	
KeyActionEnter	KeyActionEnter	
KeyActionTab	KeyActionTab	
ScrollOptions		Not supported.
ScrollPosition	ScrollPosition	

Hierarchical Features

Properties		
FlexGrid for WinForms	**FlexGrid for WPF and Silverlight**	**Comments**
TreeIndent	Tree.Indent	
Tree.Show(level)	CollapseGroupsToLevel(level)	

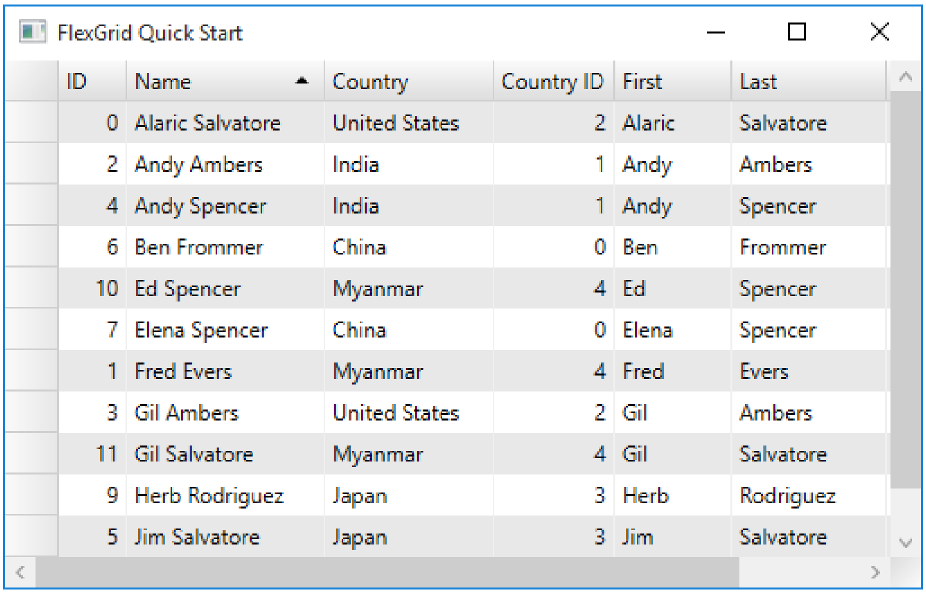
Quick Start

This quick start familiarizes you with adding a FlexGrid control and populating it with data. You begin with creating a WPF application in Visual Studio, adding the FlexGrid control, and binding it with data.

To create a simple WPF application for adding and displaying data in FlexGrid, complete the following steps:

1. **Add FlexGrid control to WPF application**
2. **Add data to display in FlexGrid**
3. **Bind FlexGrid with data**

The following image shows a FlexGrid populated with a sample data of customers.



ID	Name	Country	Country ID	First	Last
0	Alaric Salvatore	United States	2	Alaric	Salvatore
2	Andy Ambers	India	1	Andy	Ambers
4	Andy Spencer	India	1	Andy	Spencer
6	Ben Frommer	China	0	Ben	Frommer
10	Ed Spencer	Myanmar	4	Ed	Spencer
7	Elena Spencer	China	0	Elena	Spencer
1	Fred Evers	Myanmar	4	Fred	Evers
3	Gil Ambers	United States	2	Gil	Ambers
11	Gil Salvatore	Myanmar	4	Gil	Salvatore
9	Herb Rodriguez	Japan	3	Herb	Rodriguez
5	Jim Salvatore	Japan	3	Jim	Salvatore

Add FlexGrid control to WPF application

You can add FlexGrid control to WPF application through XAML, and through code.

Through XAML

1. Create a WPF project in Visual Studio.
2. Drag the FlexGrid control onto the XAML designer, that is MainWindow.xaml.
   The C1.WPF.FlexGrid.dll gets added to the references folder of your project.
3. Set the name of the control as 'grid' by editing the XAML code as illustrated in the following code example.

WPF

```
<Window x:Class="FilterRow.MainWindow"
        ...
        xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml"
        ... >
    <Grid x:Name="LayoutRoot">
        <c1:C1FlexGrid Name='grid' Grid.Row="1"/>
    </Grid>
</Window>
```

Silverlight

```
<UserControl x:Class="MainTestApplication.MainPage"
             ...
             xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml"
             ... >
  <Grid x:Name="LayoutRoot">
```

```
    <c1:C1FlexGrid Name='grid' Grid.Row="1"/>
  </Grid>
</UserControl>
```

Through code

1. Create a WPF project in Visual Studio.
2. Add the C1.WPF.FlexGrid .dll file to the **References** folder of the project.
3. Switch to the code view, that is MainWindow.xaml.cs file, and add the following using statement.

WPF

```
using C1.WPF.FlexGrid;
```

Silverlight

```
using C1.Silverlight.FlexGrid;
```

4. Add the following code in the MainWindow class constructor to add a FlexGrid control.

WPF

```
var grid = new C1FlexGrid();
Parent.Children.Add(grid);
```

Silverlight

```
var grid = new C1.Silverlight.FlexGrid.C1FlexGrid();
LayoutRoot.Children.Add(grid);
```

Back to Top

Add data to display in FlexGrid

1. Switch to the code view, that is MainWindow.xaml.cs.
2. Create a class, **Customer**, and use the following code to add data to be displayed in the FlexGrid.

C# copyCode

```
public class Customer
{
    //fields
    int _id, _countryID;
    string _first, _last;
    double _weight;

    //data generators
    static Random _rnd = new Random();
    static string[] _firstNames =
"Andy|Ben|Charlie|Dan|Ed|Fred|Gil|Herb|Jim|Elena|Stefan|Alaric|Gina".Split('|');
    static string[] _lastNames =
"Ambers|Bishop|Cole|Danson|Evers|Frommer|Salvatore|Spencer|Saltzman|Rodriguez".Split('|');
    static string[] _countries = "China|India|United States|Japan|Myanmar".Split('|');

    public Customer()
        : this(_rnd.Next())
    {
    }
    public Customer(int id)
    {
        ID = id;
        First = GetString(_firstNames);
        Last = GetString(_lastNames);
        CountryID = _rnd.Next() % _countries.Length;
        Weight = 50 + _rnd.NextDouble() * 50;
    }
    //Object model
    public int ID
    {
        get { return _id; }
        set
        {
            if (value != _id)
            {
                _id = value;
                RaisePropertyChanged("ID");
            }
        }
    }
    public string Name
    {
        get { return string.Format("{0} {1}", First, Last); }
    }
    public string Country
    {
        get { return _countries[_countryID]; }
    }
    public int CountryID
    {
        get { return _countryID; }
        set
        {
            if (value != _countryID && value > -1 && value < _countries.Length)
            {
                _countryID = value;
                RaisePropertyChanged(null);
            }
        }
    }
    public string First
    {
        get { return _first; }
        set
        {
            if (value != _first)
            {
                _first = value;
                RaisePropertyChanged(null);
            }
        }
    }
    public string Last
    {
        get { return _last; }
        set
        {
            if (value != _last)
```

```
            {
                _last = value;
                RaisePropertyChanged(null);
            }
        }
    }
    public double Weight
    {
        get { return _weight; }
        set
        {
            if (value != _weight)
            {
                _weight = value;
                RaisePropertyChanged("Weight");
            }
        }
    }
    // ** utilities
    static string GetString(string[] arr)
    {
        return arr[_rnd.Next(arr.Length)];
    }
    static string GetName()
    {
        return string.Format("{0} {1}", GetString(_firstNames), GetString(_lastNames));
    }
    // ** static list provider
    public static ObservableCollection<Customer> GetCustomerList(int count)
    {
        var list = new ObservableCollection<Customer>();
        for (int i = 0; i < count; i++)
        {
            list.Add(new Customer(i));
        }
        return list;
    }

   // this interface allows bounds controls to react to changes in the data objects.
    void RaisePropertyChanged(string propertyName)
    {
        OnPropertyChanged(new PropertyChangedEventArgs(propertyName));
    }
    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChanged(PropertyChangedEventArgs e)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, e);
    }
}
```

Bind FlexGrid with data

1. Set the ItemsSource property to populate the grid with random data coming from the Customer class.

C# copyCode

```
//Binding the grid with data
grid.ItemsSource = Customer.GetCustomerList(12);
```

Back to Top

Object Model Summary

FlexGrid comes with a rich object model, providing various classes, objects, collections, and associated methods and properties. The following table lists some of these objects and their major properties.

C1FlexGrid
Properties: AllowAddNew, AllowDragging, AllowMerging, AllowResizing, AllowSorting, AutoComplete, AutoGenerateColumns, CellFactory, Cells, Columns, FrozenRows, ItemsSource, Rows, SelectedIndex, SelectionMode, ShowErrors, ShowMarquee, TreeIndent **Methods:** AutoSizeColumn, AutoSizeColumns, AutoSizeRow, AutoSizeRows, GetAggregate, Copy, HitTest
Column
Property: AllowSorting, AutoGenerated, Binding, CellEditingTemplate, CellTemplate, Format, GroupAggregate, Header
CellStyle
Properties: Background, CornerRadius, FontSize, HorizontalAlignment, Tag, TextWrapping, VerticalAlignment
C1FlexGridFilter
Properties: Editor, FilterDefinition, NullValueString, Owner, UseCollectionView **Methods:** GetColumnFilter, LoadFilterDefinition, SaveFilterDefinition, ShowFilterEditor
C1FlexGridGroupPanel
Properties: DragMarkerColor, FlexGrid, HideGroupedColumns, MaxGroups, Watermark, WatermarkText **Methods:** OnApplyTemplate, OnPropertyGroupCreated
Row
Properties: ActualHeight, Bottom, DataItem, Grid, GridPanel, Index, Selected, Top **Methods:** GetDataFormatted, GetDataRaw, GetErrors
RowCol
Properties: AllowDragging, AllowMerging, AllowResizing, CellStyle, CellStyle, GridPanel, Foreground, HeaderTextWrapping

Features

Features section comprises all the features available in the FlexGrid control.

Columns
Learn how to configure auto complete and data-mapped columns.
Unbound FlexGrid
Learn how to create an unbound grid using the FlexGrid control.
Populate FlexGrid
Learn how to populate data in FlexGrid.
Selection
Learn how implement selection in FlexGrid.
Cell Merging
Learn how to implement cell merging in FlexGrid.
Custom Editors
Learn how to create custom editors in FlexGrid.
Data Grouping
Learn how to group data in FlexGrid.
Data Filtering
Learn how to filter data in FlexGrid.
Data Aggregation
Learn how to aggregate data in FlexGrid.
Row Details
Learn how to display row details in an empty data template in FlexGrid.
Custom Cells
Learn how to create custom cells in FlexGrid.
Print Support
Learn about various printing options available in FlexGrid.
Layout and Appearance
Learn about the layout and appearance related features in FlexGrid.

Columns

FlexGrid generates columns based on the data being displayed in the grid. When displaying data through collection, FlexGrid generates separate column for each public property. In addition, you can explicitly specify the columns in XAML.

Autocomplete and mapped columns
Learn how to implement autocomplete and mapped columns in FlexGrid.
Data-mapped columns
Learn how to implement data-mapped columns.

Autocomplete and Mapped Columns

Auto-complete and mapped columns are implemented with a built-in class called ColumnValueConverter. This class deals with three common binding scenarios:

Auto-complete exclusive mode (ListBox-style editing)

Columns that can only take on a few specific values. For example, you have a "Country" column of type string and a list of country names. Users should select a country from the list, and not be allowed to enter any countries not on the list.

You can handle this scenario with two lines of code:

C#

```
var c = _flexEdit.Columns["Country"];
c.ValueConverter = new ColumnValueConverter(GetCountryNames(), true);
```

The first parameter in the ColumnValueConverter constructor provides the list of valid values. The second parameter determines whether users should be allowed to enter values that are not on the list (in this example they are not).

Auto-complete non-exclusive mode (ComboBox-style editing)

Columns that have a few common values, but may take on other values as well. For example, you have a "Country" column of type string and want to provide a list of common country names that users can select easily. But in this case users should also be allowed to type values that are not on the list.

You can also handle this scenario with two lines of code:

C#

```
var c = _flexEdit.Columns["Country"];
c.ValueConverter = new ColumnValueConverter(GetCountryNames(), false);
```

As before, the first parameter in the ColumnValueConverter constructor provides the list of valid values. The second parameter in this case determines that the list is not exclusive, so users are allowed to enter values that are not on the list.

Auto-complete using a key-value dictionary

Columns that contain keys instead of actual values. For example, the column may contain an integer that represents a country ID, but users should see and edit the corresponding country name instead. The code below shows how you can handle this scenario:

C#

```
// build key-value dictionary
var dct = new Dictionary<int, string>();
foreach (var country in GetCountryNames())
{
  dct[dct.Count] = country;
}

// get column
var c = _flexEdit.Columns["CountryID"];

// create and assign converter with value dictionary
c.ValueConverter = new ColumnValueConverter(dct);

// align column to the left
c.HorizontalAlignment = HorizontalAlignment.Left;
```

Data-Mapped columns

Data-mapped columns contain keys instead of actual values. For example, the column may contain an integer that represents a country ID, but users should see and edit the corresponding country name instead.

This scenario requires a little more than two lines of code:

C#

```csharp
// build key-value dictionary
var dct = new Dictionary<int, string>();
foreach (var country in GetCountryNames())
{
  dct[dct.Count] = country;
}

// assign dictionary to column
var c = _flexEdit.Columns["CountryID"];
c.ValueConverter = new ColumnValueConverter(dct);
c.HorizontalAlignment = HorizontalAlignment.Left;
```

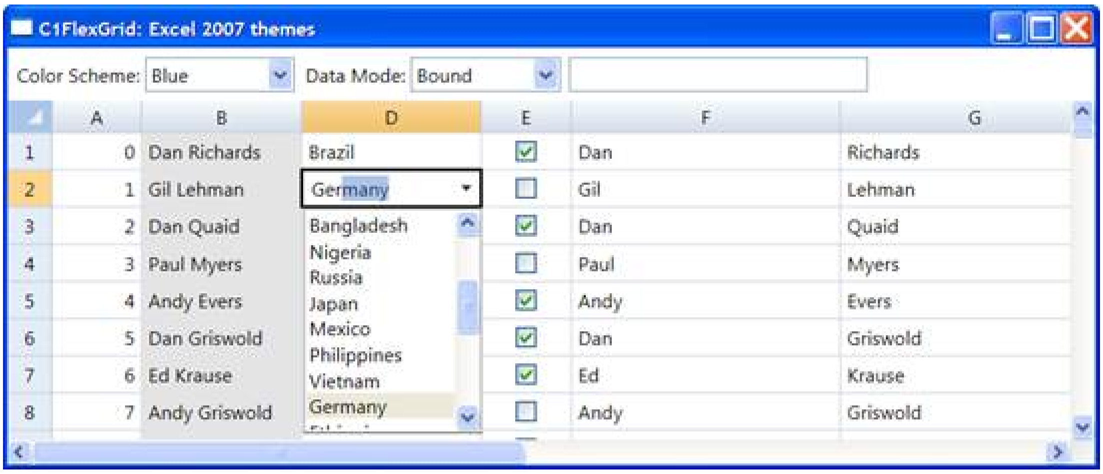
The code starts by building a dictionary that maps country ID values (integers) to country names (strings).

It then uses the dictionary to build a ColumnValueConverter and assigns the converter to the column's ValueConverter property as in the previous examples.

The user will be able to select any countries present in the dictionary, and will not be able to enter any unmapped values.

Finally, the code sets the column alignment to left. Since the column actually contains integer values, it is aligned to the right by default. But since we are now displaying names, left alignment is a better choice here.

The image below shows the appearance of the editor while selecting a value from a list. Notice how the editor supports smart auto-completion, so as the user types “Ger,” the dropdown automatically selects the only valid option “Germany” (and not “Guatemala,” then “Eritrea,” then “Romania”).



Color Scheme: Blue
Data Mode: Bound

	A	B	D	E	F	G
1	0	Dan Richards	Brazil	☑	Dan	Richards
2	1	Gil Lehman	Germany	☐	Gil	Lehman
3	2	Dan Quaid		☑	Dan	Quaid
4	3	Paul Myers		☐	Paul	Myers
5	4	Andy Evers		☑	Andy	Evers
6	5	Dan Griswold		☑	Dan	Griswold
7	6	Ed Krause		☑	Ed	Krause
8	7	Andy Griswold		☐	Andy	Griswold

Data Binding

FlexGrid supports bound and unbound grids to display data. In bound mode, you can bind the FlexGrid to a data source to populate data in the grid, while in unbound mode, you can manually populate the grid with data.

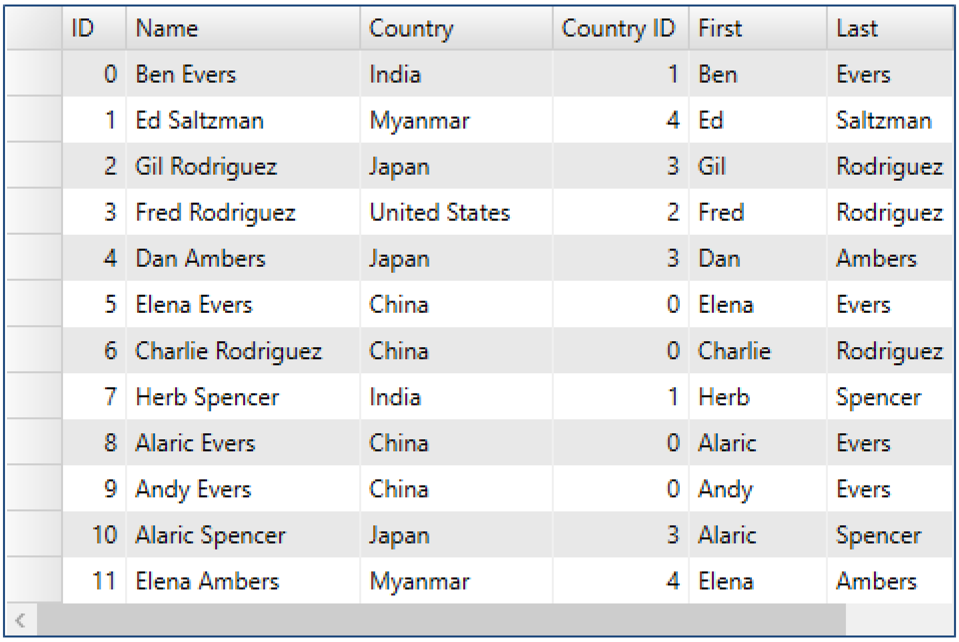
Bound FlexGrid
    Learn how to create a bound grid.

Unbound FlexGrid
    Learn how to create an unbound grid.

Bound FlexGrid

Like other data visualization controls, FlexGrid supports bound mode and can be populated with data from various data sources. The C1FlexGrid class provides the ItemsSource property to populate data in the grid. In WPF, the ItemsSource property binds FlexGrid to IEnumerable interface, or to ICollectionView interface. In Silverlight, FlexGrid provides PagedCollectionView class that implements the **ICollectionView** interface. The **PagedCollectionView** constructor takes an **IEnumerable** object as a parameter and automatically provides all **ICollectionView** services.

The following image shows a bound FlexGrid.



ID	Name	Country	Country ID	First	Last
0	Ben Evers	India	1	Ben	Evers
1	Ed Saltzman	Myanmar	4	Ed	Saltzman
2	Gil Rodriguez	Japan	3	Gil	Rodriguez
3	Fred Rodriguez	United States	2	Fred	Rodriguez
4	Dan Ambers	Japan	3	Dan	Ambers
5	Elena Evers	China	0	Elena	Evers
6	Charlie Rodriguez	China	0	Charlie	Rodriguez
7	Herb Spencer	India	1	Herb	Spencer
8	Alaric Evers	China	0	Alaric	Evers
9	Andy Evers	China	0	Andy	Evers
10	Alaric Spencer	Japan	3	Alaric	Spencer
11	Elena Ambers	Myanmar	4	Elena	Ambers

The following code examples illustrate how to bind a list of customer objects to FlexGrid control. The code below causes the grid to scan the data source and automatically generate columns for each public property of the items in the data source. You can also customize automatically created columns using code, or disable automatic column generation altogether and create custom columns through code or XAML.

WPF

```
grid.ItemsSource = Customer.GetCustomerList(12);
```

Silverlight

```
List<Customer> list = GetCustomerList();
PagedCollectionView view = new PagedCollectionView(list);
grid.ItemsSource = view;
```

FlexGrid can also be bound directly to a list (customer list). However, binding to an **ICollectionView** is usually better as it retains a lot of the data configuration for the application, which can be shared across controls. If many controls are bound to the same **ICollectionView** object, they all show the same view. Selecting an item in one control automatically updates the selection on all other bound controls. Filtering, grouping, or sorting are also shared by all controls bound to the same view.

For example, the following code disables automatic column generation and specifies columns in XAML instead.

C#

```
<!-- create columns in a C1FlexGrid -->
<grid:C1FlexGrid x:Name="_flexiTunes"
     AutoGenerateColumns="False" >
   <grid:C1FlexGrid.Columns>
     <grid:Column Binding="{Binding Name}" Header="Title"
         AllowDragging="False" Width="300"/>
     <grid:Column Binding="{Binding Duration}"
         HorizontalAlignment="Right" />
     <grid:Column Binding="{Binding Size}"
         HorizontalAlignment="Right" />
     <grid:Column Binding="{Binding Rating}" Width="200"
         HorizontalAlignment ="Center" />
   </grid:C1FlexGrid.Columns>
</grid:C1FlexGrid>
```

Note that the binding can be used as an indexer into the grid's **Columns** collection. For example, to set the width of the "Rating" column to 300 pixels, use the following code.

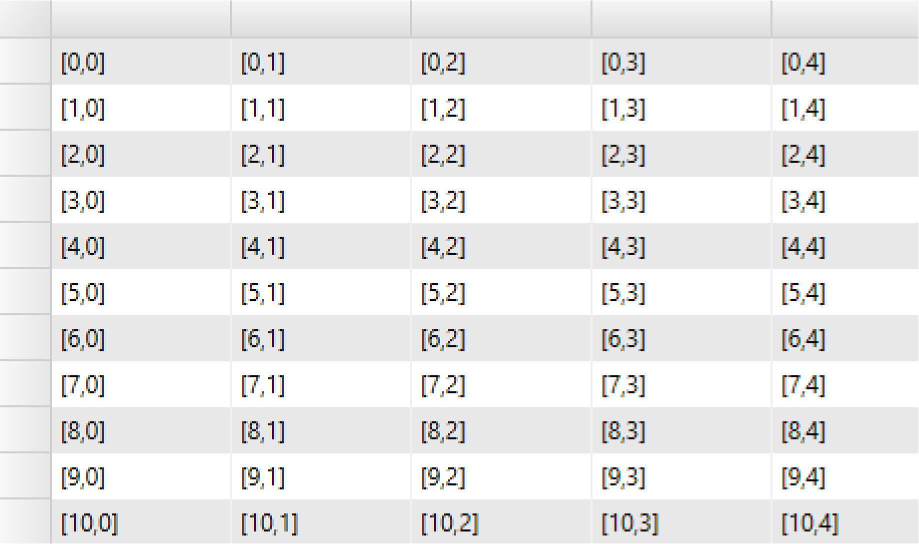
C#

```
_flexiTunes.Columns["Rating"].Width = new GridLength(300);
```

Unbound FlexGrid

The FlexGrid control is designed to work with various data sources and collections such as ObservableCollection and ICollectionView to leverage its full capabilities. However, the control is not restricted to data sources and can be used in unbound mode.

The image given below shows an unbound grid populated with cell index notation.



[0,0]	[0,1]	[0,2]	[0,3]	[0,4]
[1,0]	[1,1]	[1,2]	[1,3]	[1,4]
[2,0]	[2,1]	[2,2]	[2,3]	[2,4]
[3,0]	[3,1]	[3,2]	[3,3]	[3,4]
[4,0]	[4,1]	[4,2]	[4,3]	[4,4]
[5,0]	[5,1]	[5,2]	[5,3]	[5,4]
[6,0]	[6,1]	[6,2]	[6,3]	[6,4]
[7,0]	[7,1]	[7,2]	[7,3]	[7,4]
[8,0]	[8,1]	[8,2]	[8,3]	[8,4]
[9,0]	[9,1]	[9,2]	[9,3]	[9,4]
[10,0]	[10,1]	[10,2]	[10,3]	[10,4]

To create an unbound grid, add rows and columns to the grid using the Add method. The following code illustrates adding rows and columns in a grid and populating it with an indexing notation that specifies a cell by corresponding row and column index.

C#

```
 for (int i = 0; i < 20; i++)
```

```
   {
      grid.Columns.Add(new Column());
   }
for (int i = 0; i < 500; i++)
   {
      grid.Rows.Add(new Row());
    }

// populate the unbound grid with some index
 for (int r = 0; r < grid.Rows.Count; r++)
   {
        for (int c = 0; c < grid.Columns.Count; c++)
           {
               grid[r, c] = string.Format("[{0},{1}]", r, c);
            }
    }
```

The indexing notation displayed in the grid specifies a cell by row and column index. The cells can also be specified by row index and column name, or by row index and column name. The indexing notation works in bound and unbound modes. In bound mode, the data is retrieved or applied to the items in the data source. In unbound mode, the data is stored internally by the grid. The new indexing notation displayed in the grid contains no items in the 0th row. This notation makes indexing easier as the indices match the index of data items and the column count matches the number of displayed properties. The only drawback of this notation is that a new method is required to access the content of fixed cells. The new method consists of additional properties called RowHeaders and ColumnHeaders that return an object of type GridPanel, which can be seen as a sub-grid having its own set of rows and columns. For example, use the following code to customize row headers.

C#

```
// get grid's row headers
 GridPanel rowHeader = grid.RowHeaders;

// add a new fixed column to the grid
rowHeader.Columns.Add(new Column());

// set the width and content of the row headers
for (int c = 0; c < rowHeader.Columns.Count; c++)
  {
     for (int r = 0; r < rowHeader.Rows.Count; r++)
         {
              // set content for this cell
              rowHeader[r, c] = string.Format("hdr {0},{1}", r, c);
          }
   }
```

The GridPanel class exposes Rows and Columns collections just as the main grid does, and supports the same indexing notation. The row and column headers can be customized and populated using the same object model and techniques you use while working with the content area of the grid (the scrollable part).

Selection

Most grid controls allow users to select parts of the data using the mouse and the keyboard. FlexGrid supports following selection modes through the SelectionMode property:

- **Cell**: To select a single cell.
- **CellRange**: To select a cell range (block of adjacent cells).
- **Row**: To select an entire row.
- **RowRange**: To select a set of contiguous rows.
- **ListBox**: To select an arbitrary set of rows (not necessarily contiguous).

The default **SelectionMode** is **CellRange**, which provides an Excel-like selection behavior. The row-based options are also useful in scenarios where it makes sense to select whole data items instead of individual cells. Regardless of the selection mode, FlexGrid exposes the current selection with the Selection property. This property gets or sets the current selection as a **CellRange** object.

Monitor Selection in FlexGrid

Whenever the selection changes, either as a result of user actions or code, the grid fires the SelectionChanged event, which allows you to react to the new selection.

For example, the code below monitors the selection and sends information to the console when the selection changes:

C#

```
private void grid_SelectionChanged(object sender, CellRangeEventArgs e)
        {
            CellRange sel = grid.Selection;
            Console.WriteLine("selection: {0}, {1} - {2}, {3}",
                sel.Row, sel.Column, sel.Row2, sel.Column2);
            Console.WriteLine("selection content: {0}",
            GetClipString(grid, sel));
        }

        static string GetClipString(C1FlexGrid grid, CellRange sel)
        {
            var sb = new System.Text.StringBuilder();
            for (int r = sel.TopRow; r <= sel.BottomRow; r++)
            {
                for (int c = sel.LeftColumn; c <= sel.RightColumn; c++)
                {
                    sb.AppendFormat("{0}\t", grid[r, c].ToString());
                }
                sb.AppendLine();
            }
            return sb.ToString();
        }
```

Whenever the selection changes, the code lists the coordinates of the CellRange that represents the current selection. It also outputs the content of the selected range using a GetClipString method that loops through the selected items and retrieves the content of each cell in the selection using the grid's indexer described earlier in this document. Notice that the for loops in the GetClipString method use the CellRange's TopRow, BottomRow, LeftColumn, and RightColumn properties instead of the Row, Row2, Column, and Column2 properties. This is necessary because Row may be greater or smaller than Row2, depending on how the user performs the selection (dragging the mouse up or down while selecting).

You can easily extract a lot of useful information from the Selection using the GetDataItems method, which returns a collection of data items associated with a cell range. Once you have this collection, you can use LINQ to extract and summarize information about the selected items. For example, consider this alternate implementation of the SelectionChanged event for a grid bound to a collection of Customer objects.

C#

```
void grid_SelectionChanged(object sender, CellRangeEventArgs e)
{
  // get customers in the selected range
  var customers =
    grid.Rows.GetDataItems(grid.Selection).OfType<Customer>();

  // use LINQ to extract information from the selected customers
  _lblSelState.Text = string.Format(
    "{0} items selected, {1} active, total weight: {2:n2}",
    customers.Count(),
    (from c in customers where c.Active select c).Count(),
    (from c in customers select c.Weight).Sum());
}
```

The above code uses the **OfType** operator to cast the selected data items to type **Customer**. Once that is done, the code uses LINQ to get a total count, a count of "active" customers, and the total weight of the customers in the selection. LINQ is the perfect tool for this type of job. It is flexible, expressive, compact, and efficient.

Select Cells and Objects

The Selection property is read-write, so you can select cell ranges using code. You can also perform selections using the Select method. The Select method allows you to select cells or ranges, and optionally scroll the new selection into view so the user can see it. The following code example illustrates selecting the first cell in FlexGrid and ensures the selection is visible to the user.

C#

```
// select row zero, column zero, make sure the cell is visible
grid.Select(0, 0, true);
```

The selection methods are based on row and column indices. But these methods can be used to make selections based on cell content. For example, the code below selects the first row that contains a given string in the grid's "Name" column.

C#

```
bool SelectName(string name)
{
  // find row that contains the given string in the "Name" column
  int col = _flexGroup.Columns["Name"].Index;
  int row = FindRow(grid, name, grid.Selection.Row, col, true);
  if (row > -1)
  {
    grid.Select(row, col);
    return true;
  }
  // not found...
  return false;
}
```

The code uses the FindRow helper method defined below:

C#

```
// look for a row that contains some text in a specific column
int FindRow(C1FlexGrid grid, string text,
            int startRow, int col, bool wrap)
{
  int count = grid.Rows.Count;
  for (int off = 0; off <= count; off++)
  {
    // reached the bottom and not wrapping? quit now
    if (!wrap && startRow + off >= count)
    {
      break;
    }

    // get text from row
    int row = (startRow + off) % count;
    var content = grid[row, col];

    // found? return row index
    if (content != null &&
        content.ToString().IndexOf(text,
        StringComparison.OrdinalIgnoreCase) > -1)
    {
      return row;
    }
  }

  // not found...
  return -1;
}
```

The FindRow method searches the specified column for a string, starting at a given row and optionally wrapping the search to start from the top if the string is not found. It is flexible enough to be used in many scenarios. Another common selection scenario is the case where you want to select a specific object in the data source. Your first impulse might be to find the index of the object in the source collection using the **PagedCollectionView.IndexOf** method, then use the index to select the row. The problem with that approach is that it works only if the data is not grouped. If the data is grouped, the group rows also count, so the indices of items in the data source do not match the row indices on the grid.

The easy way to solve this problem is to enumerate the rows and compare each row's DataItem property to the item you are looking for. The code below shows how this is done.

C#

```
var customer = GetSomeCustomer;

#if false // ** don't use this, won't work with grouped data
```

```
  int index = view.IndexOf(customer);
  if (index > -1)
  {
    grid.Select(index, 0);
  }

#else // this is the safe way to look for objects in the grid

  for (int row = 0; row <= grid.Rows.Count; row++)
  {
    if (row.DataItem == customer)
    {
      grid.Select(row, 0);
      break;
    }
  }
#endif
```

Customize Selection Display

FlexGrid includes two features that allow you to customize the way in which the selection is highlighted for the user.

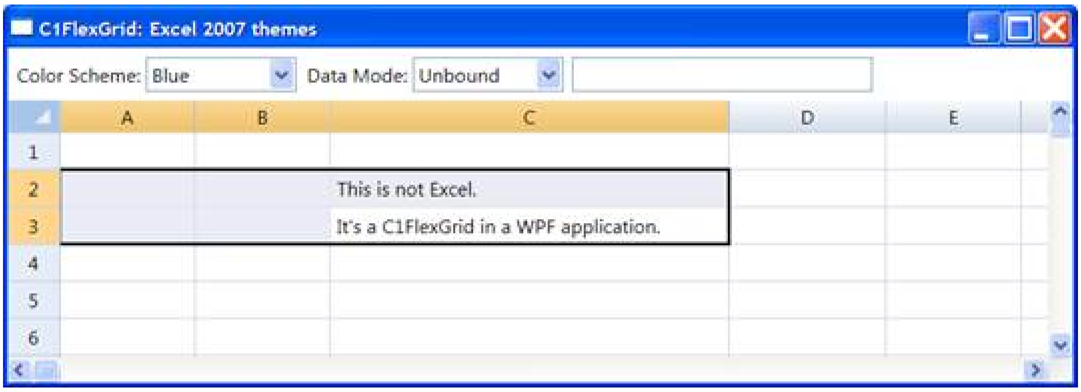
Excel-Style Marquee

If you set the ShowMarquee property to true, the grid automatically draws a rectangle around the selection, making it extremely easy to see. By default, the marquee is a two-pixel thick black rectangle, but you can customize it using the Marquee property.

Selected Cell Headers

If you assign custom brush objects to the grid's ColumnHeaderSelectedBackground and RowHeaderSelectedBackground properties, the grid highlights headers that correspond to the selected cells, making it easy for users to see which rows and columns contain the selection.

Together, these properties let you implement grids that have the familiar Excel look and feel. The image below shows an example:



C1FlexGrid: Excel 2007 themes

Color Scheme: Blue

Data Mode: Unbound

This is not Excel.

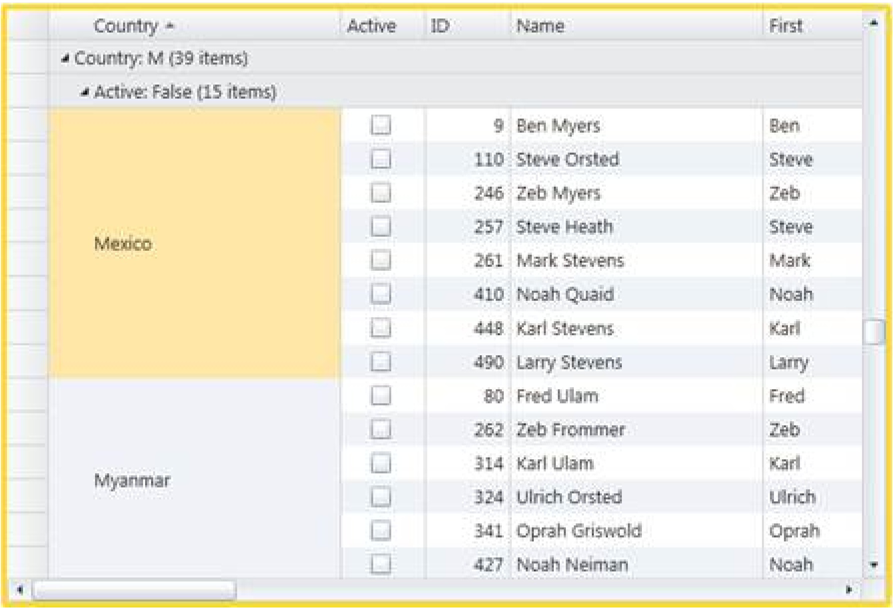
It's a C1FlexGrid in a WPF application.

FlexGrid designer provides a context menu with options to select Excel-like color schemes (Blue, Silver, Black). In addition, you can easily copy the XAML generated by the designer into reusable style resources.

Cell Merging

FlexGrid supports cell merging to allow data to span across multiple rows and columns. The cell merging capability can be used to enhance the appearance of data. Cell merging can be enabled by setting the AllowMerging property in code. To merge columns, you need to set the AllowMerging property for each column that you want to merge to **true**. Similarly, to merge rows, set the AllowMerging property to **true** for each row to be merged.

The following image shows merged columns in a FlexGrid.



Country: M (39 items)

Active: False (15 items)

Country	Active	ID	Name	First
Mexico	☐	9	Ben Myers	Ben
	☐	110	Steve Orsted	Steve
	☐	246	Zeb Myers	Zeb
	☐	257	Steve Heath	Steve
	☐	261	Mark Stevens	Mark
	☐	410	Noah Quaid	Noah
	☐	448	Karl Stevens	Karl
	☐	490	Larry Stevens	Larry
Myanmar	☐	80	Fred Ulam	Fred
	☐	262	Zeb Frommer	Zeb
	☐	314	Karl Ulam	Karl
	☐	324	Ulrich Orsted	Ulrich
	☐	341	Oprah Griswold	Oprah
	☐	427	Noah Neiman	Noah

The code below illustrates merging columns (cells) containing the same country.

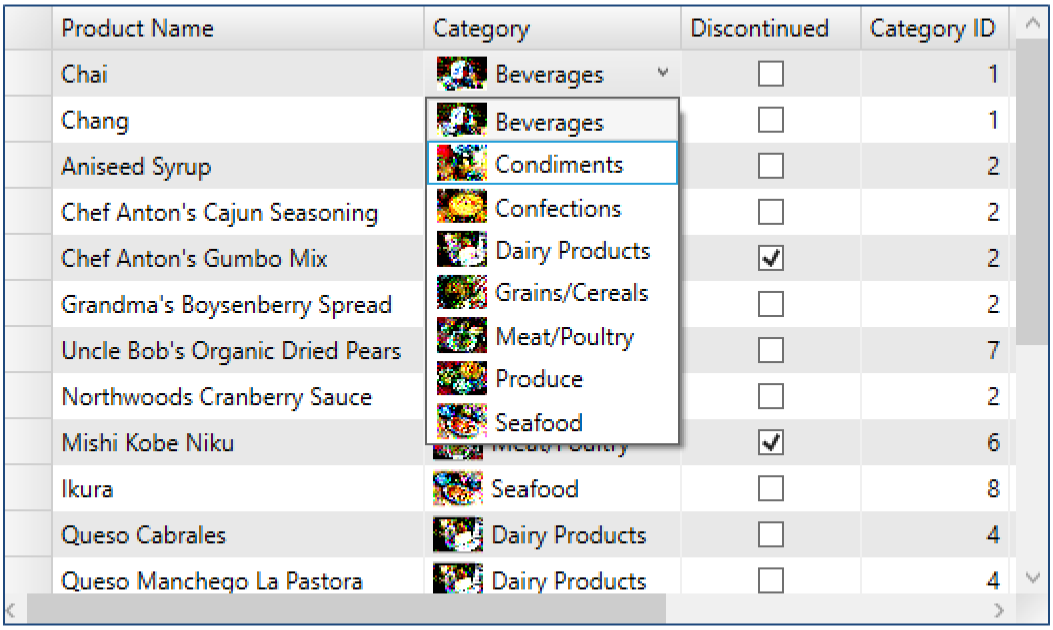
C#

```
// enable merging in the scrollable area
grid.AllowMerging = AllowMerging.Cells;

// on columns "Country" and "FirstName"
grid.Columns["Country"].AllowMerging = true;
grid.Columns["FirstName"].AllowMerging = true;
```

Custom Editors

The FlexGrid control comes with two built-in editors including checkbox and combo box. The checkbox is supported to represent Boolean values, while the combo box lets you type text and select values from auto complete list. The combo box is implemented through the C1FlexComboBox class that extends a regular text box with auto complete list.

The following image shows checkbox and combo box used as editors in FlexGrid.



Product Name	Category	Discontinued	Category ID
Chai	Beverages	☐	1
Chang		☐	1
Aniseed Syrup		☐	2
Chef Anton's Cajun Seasoning		☐	2
Chef Anton's Gumbo Mix		☑	2
Grandma's Boysenberry Spread		☐	2
Uncle Bob's Organic Dried Pears		☐	7
Northwoods Cranberry Sauce		☐	2
Mishi Kobe Niku		☑	6
Ikura	Seafood	☐	8
Queso Cabrales	Dairy Products	☐	4
Queso Manchego La Pastora	Dairy Products	☐	4

However, FlexGrid is not limited to built-in editors as you can create and use your own editors. To create custom editors, you can follow the approaches mentioned below:

- Implement a custom CellFactory class and override the CreateCellEditor method to create and bind your editor to the underlying data value.
- Use XAML to specify a CellEditingTemplate for the columns that need the custom editors.

Whether you are using built-in or custom editors, you can use PrepareCellForEdit event to configure the editor before it is activated. For example, the code below changes the editor to show selections as yellow on blue:

C#

```
// hook up event handler
_grid.PrepareCellForEdit += _grid_PrepareCellForEdit;

// customize editor by changing the appearance of the selection
void _grid_PrepareCellForEdit(object sender, CellEditEventArgs e)
{
  var b = e.Editor as Border;
  var tb = b.Child as TextBox;
  tb.SelectionBackground = new SolidColorBrush(Colors.Blue);
  tb.SelectionForeground = new SolidColorBrush(Colors.Yellow);
}
```

Data Grouping

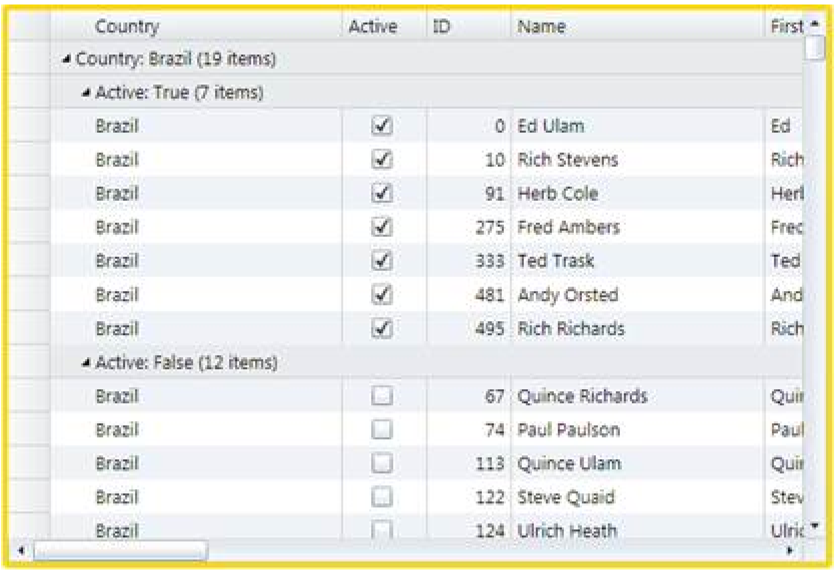
FlexGrid supports data grouping through ICollectionView, as well as separately through UI, which is shipped under a separate assembly, C1.WPF.FlexGridGroupPanel. FlexGridGroupPanel is available as a separate control to keep the footprint of FlexGrid control. This section discusses grouping data in FlexGrid using two approaches.

Group Data using ICollectionView
    Learn how to group data using ICollectionView.
Group Data using FlexGridGroupPanel
    Learn how to create an unbound grid using the FlexGrid control.

Group Data using ICollectionView

FlexGrid supports grouping through **ICollectionView** interface. You can create hierarchical views in FlexGrid by defining each level of grouping through the PropertyGroupDescription class. Using the PropertyGroupDescription object, you can select the property to group data, and implement ValueConverter to determine how to use property value while grouping. You can also disable grouping at the grid level by setting the grid's GroupRowPosition property to **None**.

The following image shows data grouped by country and their active state. Users can click the icons on group headers to collapse or expand the groups, as they would do with a TreeView control.



Country	Active	ID	Name	First
◢ Country: Brazil (19 items)				
◢ Active: True (7 items)				
Brazil	☑	0	Ed Ulam	Ed
Brazil	☑	10	Rich Stevens	Rich
Brazil	☑	91	Herb Cole	Herl
Brazil	☑	275	Fred Ambers	Frec
Brazil	☑	333	Ted Trask	Ted
Brazil	☑	481	Andy Orsted	And
Brazil	☑	495	Rich Richards	Rich
◢ Active: False (12 items)				
Brazil	☐	67	Quince Richards	Quir
Brazil	☐	74	Paul Paulson	Paul
Brazil	☐	113	Quince Ulam	Quir
Brazil	☐	122	Steve Quaid	Stev
Brazil	☐	124	Ulrich Heath	Ulric

The following code example illustrates data grouping by country and active state through ICollectionView.

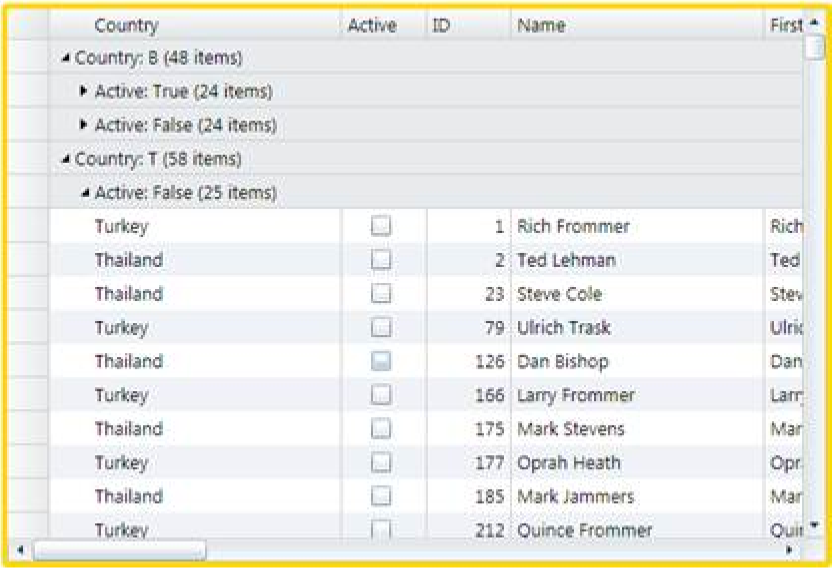
XAML

```
List<Customer> list = GetCustomerList();
PagedCollectionView view = new PagedCollectionView(list);
using (view.DeferRefresh())
{
  view.GroupDescriptions.Clear();
  view.GroupDescriptions.Add(new PropertyGroupDescription("Country"));
  view.GroupDescriptions.Add(new PropertyGroupDescription("Active"));
}
grid.ItemsSource = view;
```

> **Note:** The statement *using (view.DeferRefresh())* is optional. It improves performance by suspending notifications from the data source until all the groups are set.

To group data by country initials

Besides grouping data by country column, you can implement grouping in FlexGrid as per various end-use requirement. For example, consider a custom scenario where data should be grouped by country initials in place of country column itself. The following image shows data grouped in FlexGrid through the first letter of the country name instead of complete country name.



Country	Active	ID	Name	First
◢ Country: B (48 items)				
▸ Active: True (24 items)				
▸ Active: False (24 items)				
◢ Country: T (58 items)				
◢ Active: False (25 items)				
Turkey	☐	1	Rich Frommer	Rich
Thailand	☐	2	Ted Lehman	Ted
Thailand	☐	23	Steve Cole	Stev
Turkey	☐	79	Ulrich Trask	Ulric
Thailand	☐	126	Dan Bishop	Dan
Turkey	☐	166	Larry Frommer	Larr
Thailand	☐	175	Mark Stevens	Mar
Turkey	☐	177	Oprah Heath	Opr
Thailand	☐	185	Mark Jammers	Mar
Turkey	☐	212	Quince Frommer	Quir

To achieve this scenario through code, you need to create a custom class, **CountryInitialConverter**, that implements IValueConverter interface to return the first letter of the country name using the letter for grouping instead of the complete country name. In addition, you need to set the converter in MainWindow.xaml.cs as illustrated in the code examples given below.

CountryInitialConverter

```
class CountryInitialConverter : IValueConverter
{
  public object Convert(object value, Type targetType,
    object parameter,
    System.Globalization.CultureInfo culture)
  {
    return ((string)value)[0].ToString().ToUpper();
  }
  public object ConvertBack(object value, Type targetType,
    object parameter,
    System.Globalization.CultureInfo culture)
  {
    throw new NotImplementedException();
  }
}
```

MainWindow.xaml.cs

```
List<Customer> list = GetCustomerList();
PagedCollectionView view = new PagedCollectionView(list);
using (view.DeferRefresh())
{
  view.GroupDescriptions.Clear();
  view.GroupDescriptions.Add(new PropertyGroupDescription("Country"));
  view.GroupDescriptions.Add(new PropertyGroupDescription("Active"));
  var gd = view.GroupDescriptions[0] as PropertyGroupDescription;
  gd.Converter = new CountryInitialConverter();
}
grid.ItemsSource = view;
```

Notice that the group rows display information about the groups they represent (the property and value being grouped on, and the item count). You can customize this information by creating a new **IValueConverter** class and assign it to the grid's GroupHeaderConverter property. For example, the default group header converter is implemented in the following code.

C#

```
// class used to format group captions for display
public class GroupHeaderConverter : IValueConverter
{
  public object Convert(object value,
    Type targetType, object parameter,
    System.Globalization.CultureInfo culture)
  {
    var gr = parameter as GroupRow;
    var group = gr.Group;
    if (group != null && gr != null && targetType == typeof(string))
    {
      var desc = gr.Grid.View.GroupDescriptions[gr.Level] as
                 PropertyGroupDescription;
      return desc != null
        ? string.Format("{0}: {1} ({2:n0} items)",
                 desc.PropertyName, group.Name, group.ItemCount)
        : string.Format("{0} ({1:n0} items)",
                 group.Name, group.ItemCount);
    }
    return value;
  }
  public object ConvertBack(object value, Type targetType,
    object parameter,
    System.Globalization.CultureInfo culture)
  {
    return value;
  }
}
```

Group Data using FlexGridGroupPanel

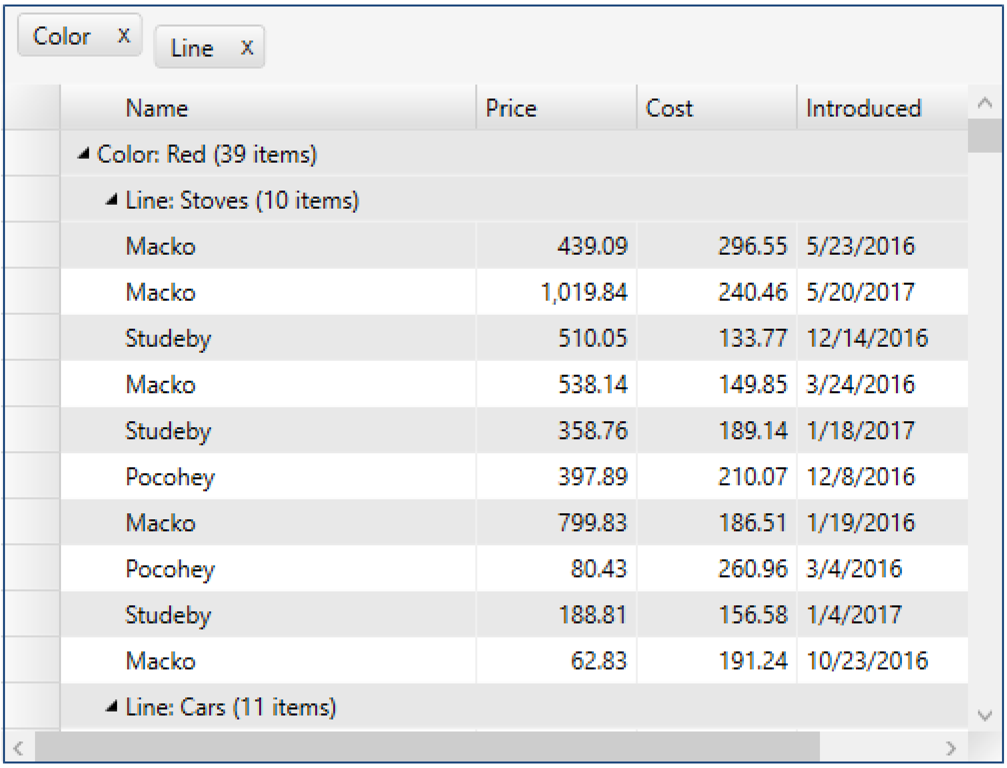
FlexGrid allows grouping of data through **FlexGridGroupPanel**, a separate control designed to organize data in groups. FlexGridGroupPanel is shipped along with FlexGrid through C1.WPF.FlexGridGroupPanel extender assembly. FlexGridGroupPanel manages groups using the **ICollectionView** interface of the collection used as the grid's data source. All changes made to the groups are visible to other controls bound to the same ICollectionView.

> **Note:** Grouping feature cannot be implemented in unbound grids.

To group data in FlexGrid, FlexGridGroupPanel can be added above FlexGrid to add an empty panel where columns can be dragged. Once a group is created, the corresponding column gets hidden. However, this behavior can be disabled by setting the HideGroupedColumns property to **false**. The group markers can be dragged within the grouping area to re-arrange the groups, or dragged into the grid to remove the group and restore column. The group markers also have close buttons ("x") to remove the group.

FlexGridGroupPanel picks up the required attributes from the FlexGrid to which it is attached. If you change the background, foreground, or font of the column headers on the grid, the FlexGridGroupPanel automatically uses those elements to render group markers that complement the column headers.

The following image show data grouped by **Color** and **Line** in FlexGrid.



Name	Price	Cost	Introduced
◢ Color: Red (39 items)			
◢ Line: Stoves (10 items)			
Macko	439.09	296.55	5/23/2016
Macko	1,019.84	240.46	5/20/2017
Studeby	510.05	133.77	12/14/2016
Macko	538.14	149.85	3/24/2016
Studeby	358.76	189.14	1/18/2017
Pocohey	397.89	210.07	12/8/2016
Macko	799.83	186.51	1/19/2016
Pocohey	80.43	260.96	3/4/2016
Studeby	188.81	156.58	1/4/2017
Macko	62.83	191.24	10/23/2016
◢ Line: Cars (11 items)			

The color of the drag markers can be changed by setting the DragMarkerColor property. You can click the group markers to sort the data in ascending or descending order. To remove the applied sort, press the control key and click the group headers to be removed.

To enable grouping in FlexGrid through FlexGridGroupPanel

1. Drag the FlexGridGroupPanel control above a FlexGrid control in XAML view.
2. Set the FlexGrid property of FlexGridGroupPanel to reference the FlexGrid object as illustrated in the following code.

XAML

```
<c1:C1FlexGridGroupPanel
    Background="WhiteSmoke"
    FlexGrid="{Binding ElementName=grid}"/>
  <c1:C1FlexGrid x:Name="grid" Grid.Row="1" />
```

To create custom group converters using FlexGridGroupPanel

By default, the FlexGridGroupPanel lets you group data by columns, which may or may not have .

On creating a new group, FlexGridGroupPanel fires a PropertyGroupCreated event that allows the application to customize the new groups. The new groups can be customized to create custom group converter based on some pre-defined constraints. For example, custom group converters can be created to categorize an arbitrary **Amount** field into high, medium or low depending upon some condition defined by user in code, or group a date time field.

The following code example

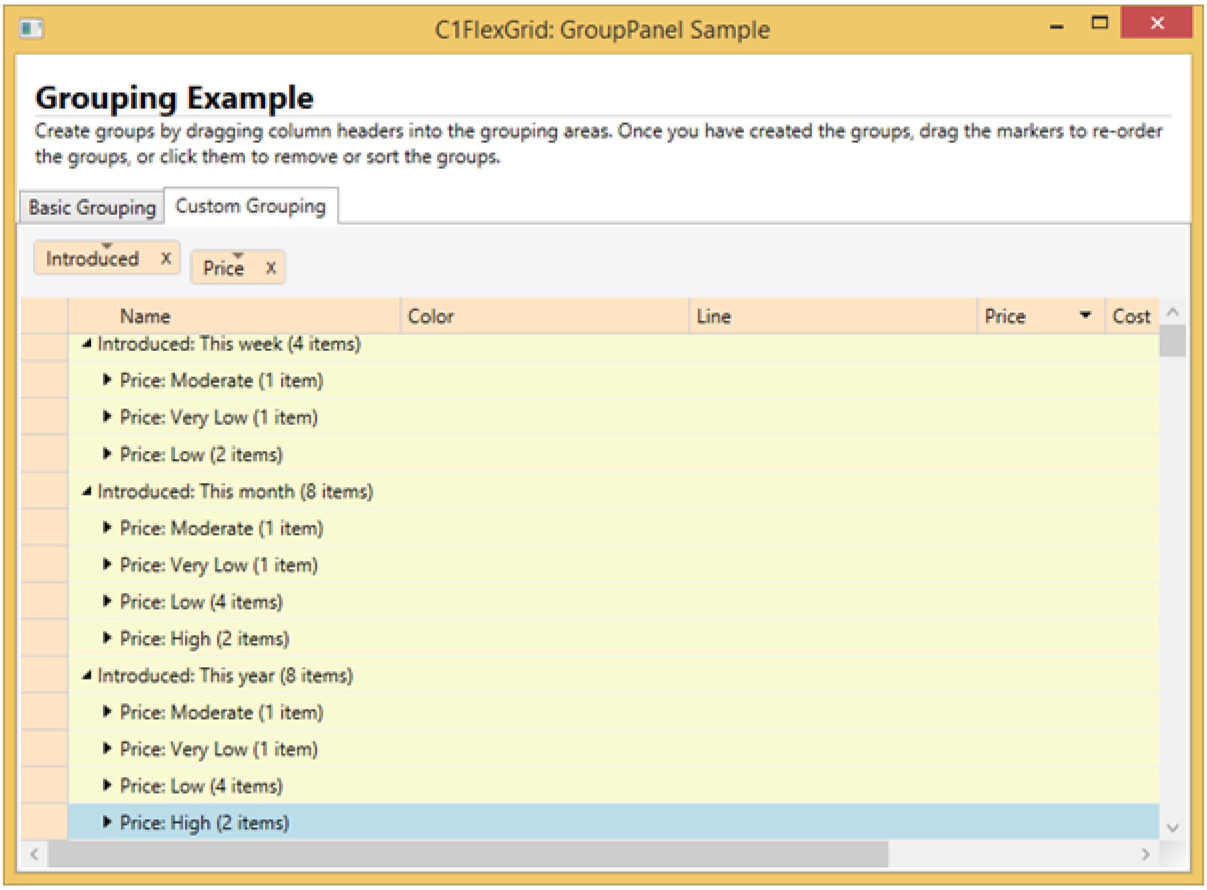
C#

```
/// <summary>
/// Customize group descriptors created by the C1FlexGridGroupPanel.
/// </summary>
```

```
void _groupPanel_PropertyGroupCreated(object sender, PropertyGroupCreatedEventArgs e)
{
  var pgd = e.PropertyGroupDescription;
  switch (pgd.PropertyName)
  {
    case "Introduced":
      pgd.Converter = new DateTimeGroupConverter();
      break;
    case "Price":
      pgd.Converter = new AmountGroupConverter(1000);
      break;
    case "Cost":
      pgd.Converter = new AmountGroupConverter(300);
      break;
  }
}
```

This code handles the **PropertyGroupCreated** event and assigns custom converters to different columns in the data source. In this case, the **DateTimeGroupConverter** and **AmountGroupConverter** classes are simple converters used to group **DateTime** and **double** values into ranges.

The image below shows the effect achieved with custom grouping:



Grouping Example

Create groups by dragging column headers into the grouping areas. Once you have created the groups, drag the markers to re-order the groups, or click them to remove or sort the groups.

Introduced: This week (4 items)
  Price: Moderate (1 item)
  Price: Very Low (1 item)
  Price: Low (2 items)
Introduced: This month (8 items)
  Price: Moderate (1 item)
  Price: Very Low (1 item)
  Price: Low (4 items)
  Price: High (2 items)
Introduced: This year (8 items)
  Price: Moderate (1 item)
  Price: Very Low (1 item)
  Price: Low (4 items)
  Price: High (2 items)

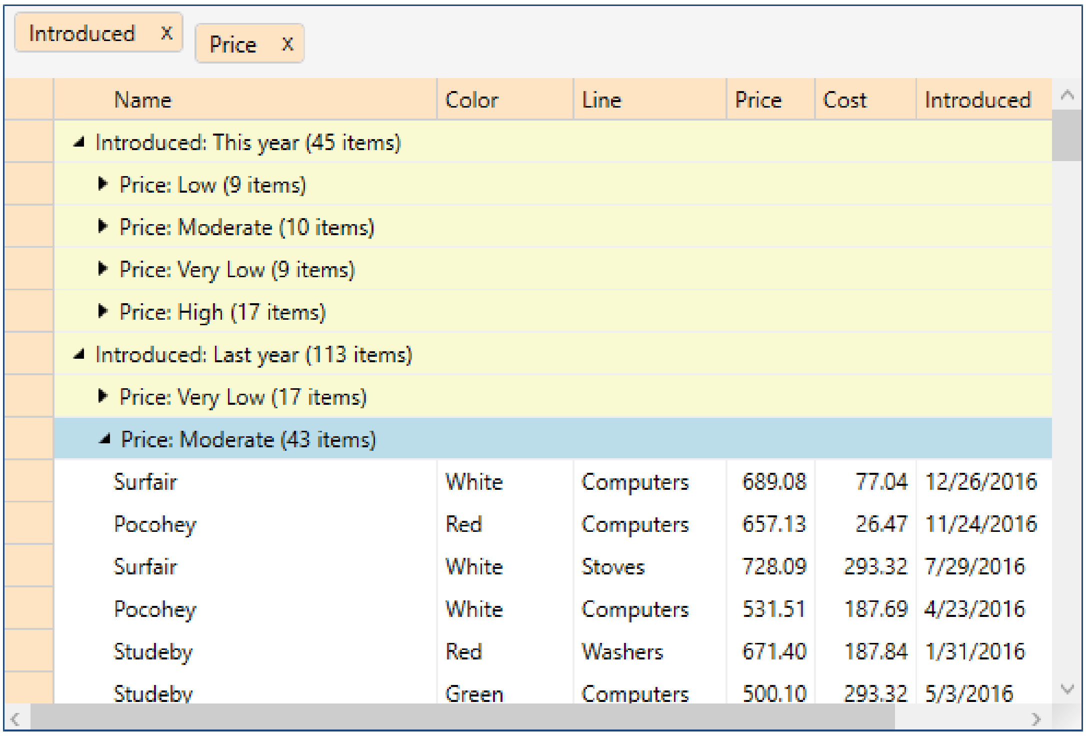
Notice that items may appear in multiple groups. For example, the DateTimeGroupConverter groups dates into This week, This month, and This year. Items in the "This week" group are also included in the "This year" group.

This is a feature of the **ICollectionView** interface, not particular to the **FlexGrid** or **FlexGridGroupPanel** controls.

Custom Group Converters

Besides providing support to group data on explicit columns, FlexGrid also supports creating custom group converters to group data based on some categories. For example, grouping data by some arbitrary order date or price point, or both can be vague. However, if the column price points can be further categorized into sub groups such as high, moderate, low, etc., then data organization becomes more meaningful. The FlexGridGroupPanel control enables users in realizing such grouping requirements by creating sub-groups or defining range.

The image given below illustrates custom data grouping on the basis of price range.



Name	Color	Line	Price	Cost	Introduced
◢ Introduced: This year (45 items)					
▶ Price: Low (9 items)					
▶ Price: Moderate (10 items)					
▶ Price: Very Low (9 items)					
▶ Price: High (17 items)					
◢ Introduced: Last year (113 items)					
▶ Price: Very Low (17 items)					
◢ Price: Moderate (43 items)					
Surfair	White	Computers	689.08	77.04	12/26/2016
Pocohey	Red	Computers	657.13	26.47	11/24/2016
Surfair	White	Stoves	728.09	293.32	7/29/2016
Pocohey	White	Computers	531.51	187.69	4/23/2016
Studeby	Red	Washers	671.40	187.84	1/31/2016
Studeby	Green	Computers	500.10	293.32	5/3/2016

To create custom group converters in code

1. Create a WPF application in Visual Studio.
2. Drag FlexGrid control onto the XAML designer.
3. Add a FlexGridGroupPanel above the FlexGrid control using the following XAML code.

XAML copyCode

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition />
    </Grid.RowDefinitions>
          <c1:C1FlexGridGroupPanel
             x:Name="_groupPanelCustomGrouping" Grid.Row="0" Background="WhiteSmoke"
             WatermarkText="Drag column headers here to create groups."
             MaxGroups="8"
             HideGroupedColumns="False"
             DragMarkerColor="#FF5C54"
             FlexGrid="{Binding ElementName=_flexCustomGrouping}"
             PropertyGroupCreated="_groupPanelCustomGrouping_PropertyGroupCreated"/>
          <c1:C1FlexGrid
             x:Name="_flexCustomGrouping" Grid.Row="1"
             TopLeftCellBackground="Bisque"
             ColumnHeaderBackground="Bisque"
             RowHeaderBackground="Bisque"
             GroupRowBackground="LightGoldenrodYellow"
             RowBackground="White"
             AlternatingRowBackground="White"/>
</Grid>
```

4. Add a new class to your project, Products.cs, to display data in the grid.

C# copyCode

```
public class Product : BaseObject
{
    string _name, _color, _line;
    double _price, _cost;
    DateTime _date;
    bool _discontinued;

    static Random _rnd = new Random();
    static string[] _names = "Macko|Surfair|Pocohey|Studeby".Split('|');
    static string[] _lines = "Computers|Washers|Stoves|Cars".Split('|');
    static string[] _colors = "Red|Green|Blue|White".Split('|');

    public Product()
    {
        Name = PickOne(_names);
        Line = PickOne(_lines);
        Color = PickOne(_colors);
        Price = 30 + _rnd.NextDouble() * 1000;
        Cost = 3 + _rnd.NextDouble() * 300;
        Discontinued = _rnd.NextDouble() < .2;
        Introduced = DateTime.Today.AddDays(_rnd.Next(-600, 0));
    }
    string PickOne(string[] options)
    {
        return options[_rnd.Next() % options.Length];
    }

    //[Display(Name = "Name")]
    public string Name
    {
        get { return _name; }
        set { SetProperty("Name", ref _name, value); }
    }

    //[Display(Name = "Color")]
    public string Color
    {
        get { return _color; }
        set { SetProperty("Color", ref _color, value); }
    }

    //[Display(Name = "Line")]
    public string Line
    {
        get { return _line; }
        set { SetProperty("Line", ref _line, value); }
    }

    //[Display(Name = "Price")]
    public double Price
    {
        get { return _price; }
        set { SetProperty("Price", ref _price, value); }
    }

    //[Display(Name = "Cost")]
    public double Cost
    {
        get { return _cost; }
        set { SetProperty("Cost", ref _cost, value); }
    }

    //[Display(Name = "Introduced")]
    public DateTime Introduced
    {
        get { return _date; }
        set { SetProperty("Introduced", ref _date, value); }
    }

    //[Display(Name = "Discontinued")]
    public bool Discontinued
    {
        get { return _discontinued; }
        set { SetProperty("Discontinued", ref _discontinued, value); }
    }
}

public class BaseObject : INotifyPropertyChanged, IEditableObject
{
    protected bool SetProperty<T>(string propName, ref T field, T value)
    {
        if (EqualityComparer<T>.Default.Equals(field, value)) return false;
        field = value;
        OnPropertyChanged(propName);
        return true;
    }

    public event PropertyChangedEventHandler PropertyChanged;
    void OnPropertyChanged(string propName)
    {
        OnPropertyChanged(new PropertyChangedEventArgs(propName));
    }
    protected virtual void OnPropertyChanged(PropertyChangedEventArgs e)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, e);
    }

    object _clone = null;
    public void BeginEdit()
    {
        _clone = this.MemberwiseClone();
    }
    public void CancelEdit()
    {
        foreach (var p in this.GetType().GetProperties())
        {
            var value = p.GetValue(_clone, null);
            p.SetValue(this, value, null);
        }
    }
    public void EndEdit()
    {
        _clone = null;
    }
}
```

5. Add two classes, AmountGroupConverter.cs and DateTimeGroupConverter.cs, in your project. These classes are simple converters to group double values of the **Price** and **Cost** column, and DateTime values of **Introduced** and **Discontinued** columns into well-defined ranges.

AmountGroupConverter

```
public class AmountGroupConverter : IValueConverter
{
    double _maxValue;
    public AmountGroupConverter(double maxValue)
    {
        _maxValue = maxValue;
    }
    public object Convert(object value, Type targetType,
        object parameter, System.Globalization.CultureInfo culture)
    {
        var pct = (double)value / _maxValue;
        if (pct < .25) return "Very Low";
        if (pct < .50) return "Low";
        if (pct < .75) return "Moderate";
        return "High";
    }
    public object ConvertBack(object value, Type targetType,
        object parameter, System.Globalization.CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}
```

DateTimeGroupConverter

```
public class DateTimeGroupConverter : IValueConverter
{
    public object Convert(object value, Type targetType,
        object parameter, System.Globalization.CultureInfo culture)
    {
        var list = new List<string>();
        var date = (DateTime)value;
        var today = DateTime.Today;
        if (today.Subtract(date).TotalDays <= 7) list.Add("This week");
        if (date.Year == today.Year && date.Month == today.Month) list.Add("This month");
        if (date.Year == today.Year)
        {
            list.Add("This year");
        }
```

```
            else if (date.Year == today.Year - 1)
            {
                list.Add("Last year");
            }
            else
            {
                list.Add("Before last year");
            }
            return list;
        }
        public object ConvertBack(object value, Type targetType,
            object parameter, System.Globalization.CultureInfo culture)
        {
            throw new NotImplementedException();
        }
    }
```

6. Switch to the MainWindow.cs file and add the following code to add data in the FlexGrid.

C# copyCode

```
// create a data source
var list = new List<Product>();
for (int i = 0; i < 200; i++)
{
    list.Add(new Product());
}
// assign the data source to grid
var cvs = new CollectionViewSource() { Source = list };
_flexCustomGrouping.ItemsSource = cvs.View;
```

7. Add the following code in the handler of the PropertyGroupCreated event to assign custom converters to different columns in the data source.

C# copyCode

```
private void _groupPanelCustomGrouping_PropertyGroupCreated
    (object sender, C1.WPF.FlexGrid.PropertyGroupCreatedEventArgs e)
{
    var pgd = e.PropertyGroupDescription;
    switch (pgd.PropertyName)
    {
        case "Introduced":
            pgd.Converter = new DateTimeGroupConverter();
            break;
        case "Discontinued":
            pgd.Converter = new DateTimeGroupConverter();
            break;
        case "Price":
            pgd.Converter = new AmountGroupConverter(1000);
            break;
        case "Cost":
            pgd.Converter = new AmountGroupConverter(300);
            break;
    }
}
```

Working with FlexGridGroupPanel

The FlexGridGroupPanel is a custom control that comprises a grid element with text block and a stack panel. The text block displays a watermark message and the stack panel shows available groups in the source **ICollectionView**. The groups are represented by GroupMarker element that can be clicked to sort or close the groups, or dragged to re-arrange the grouping order. The control comes with the **DragDropManager** utility class, which handles four drag actions listed below:

- Dragging GroupMarkers within the grouping area to re-arrange the groups.
- Dragging GroupMarkers into the grid to remove the group and restore columns at specific positions.
- Dragging ColumnHeader elements from the grid to the grouping area to create new groups.
- Dragging ColumnHeader elements within the grid to re-arrange the grid columns.

The first two drag actions are initiated by the **GroupMarker** class, which detects mouse drag actions and calls the **DoDragDrop** method with marker as a parameter. The last two actions are initiated in response to the FlexGrid's DraggingColumn event. When the **DoDragDrop** method is called, the DragDropManager displays a transparent element over the whole page, captures the mouse, and raises the Dragging event so that the caller can update the drop location. When the user releases the mouse, the **DragDropManager** raises the **Dropped** event so that the caller can finish the drag-drop action.

> **Note:** The grouping feature is implemented through the C1.WPF.FlexGridGroupPanel extender assembly, which is shipped separately to:
>
> - Minimize the footprint of FlexGrid and keep it compact and fast.
> - Allow customization in grouping UI so that users can create their own custom versions.
> - Showcase the flexibility and extensibility of the FlexGrid control.

Data Filtering

Data filtering is a common requirement in grid controls. In FlexGrid, you can implement filtering through ICollectionView or through **FlexGridFilter**, a separate control that is shipped with FlexGrid under a separate assembly. This section discusses data filtering in FlexGrid using these approaches.

Filter Data using ICollectionView
    Learn how to filter data using ICollectionView.
Filter Data using FlexGridFilter
    Learn how to filter data using FlexGridGroupPanel.

Filter Data using ICollectionView

The ICollectionView interface supports data filtering in FlexGrid through the **Filter** property. The Filter property specifies a method to call for each item in the collection. If the method returns true, the item is included in the view. If the method returns false, the item is filtered out of view. (This type of method is called a *predicate*).

Let us take a user control named SearchBox that consists of a TextBox control where the user types a value to search, and a timer that provides a small delay to allow users to type the values to search for without re-applying the filter after each character. Here we use ICollectionView filter to implement Search box.

When the user stops typing, the timer elapses and applies the filter using this code:

C#

```
_view.Filter = null;
_view.Filter = (object item) =>
{
    // get search text
    var srch = _txtSearch.Text;

    // no text? show all items
    if (string.IsNullOrEmpty(srch))
        {
            return true;
        }

// show items that contain the text in any of the specified properties
    foreach (PropertyInfo pi in _propertyInfo)
    {
       var value = pi.GetValue(item, null) as string;
       if (value != null && value.IndexOf(srch, StringComparison.OrdinalIgnoreCase) >
-1)
         {
            return true;
         }
    }

  // exclude this item...
  return false;
};
```

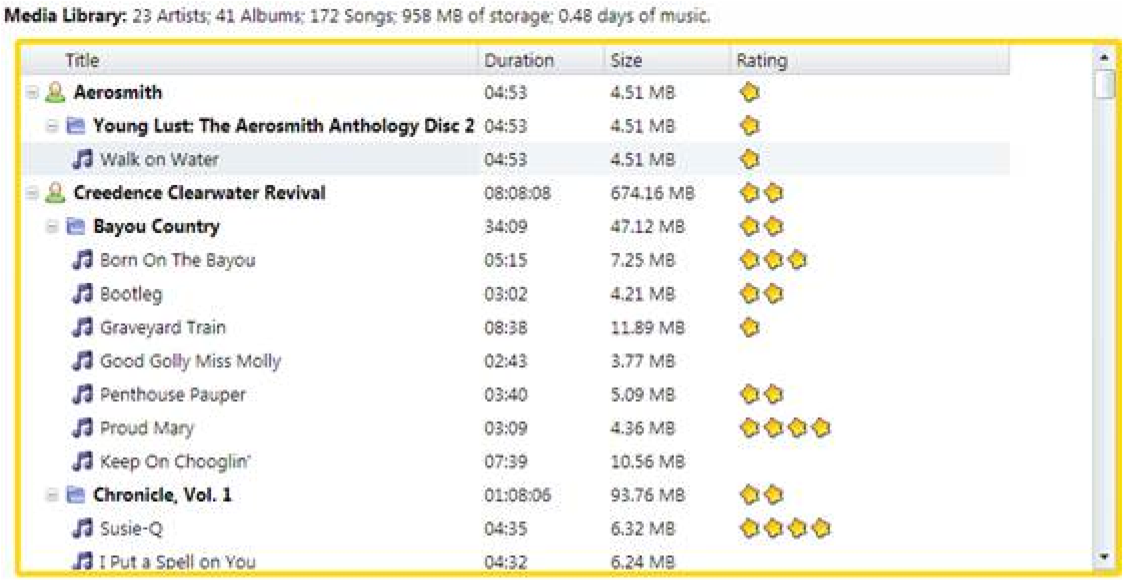
Note how the code sets the value of the **Filter** property using a lambda function. We could have provided a separate method, but this notation is often more convenient because it is concise and allows us to use local variables if we need them.

The lambda function takes an item as a parameter, gets the value of the specified properties for the object, and returns true if any of the object's properties contain the string being searched for.

For example, if the objects are of type "Song" and the properties specified are "Title," "Album," and "Artist," then the function returns true if the string being searched for is found in the song's title, album, or artist. This is a powerful and easy-to-use search mechanism similar to the one used in Apple's iTunes application.

As soon as the filter is applied, the grid (and any other controls bound to the **ICollectionView** object) reflect the result of the filter by showing only the items selected by the filter.

Note that filtering and grouping work together. The image below (from the **MainTestApplication** sample) shows a very large song list with a filter applied to it:



Media Library: 23 Artists; 41 Albums; 172 Songs; 958 MB of storage; 0.48 days of music.

Title	Duration	Size	Rating
Aerosmith	04:53	4.51 MB	★
Young Lust: The Aerosmith Anthology Disc 2	04:53	4.51 MB	★
Walk on Water	04:53	4.51 MB	★
Creedence Clearwater Revival	08:08:08	674.16 MB	★★
Bayou Country	34:09	47.12 MB	★★
Born On The Bayou	05:15	7.25 MB	★★★
Bootleg	03:02	4.21 MB	★★
Graveyard Train	08:38	11.89 MB	★
Good Golly Miss Molly	02:43	3.77 MB	
Penthouse Pauper	03:40	5.09 MB	★★
Proud Mary	03:09	4.36 MB	★★★★
Keep On Chooglin'	07:39	10.56 MB	
Chronicle, Vol. 1	01:08:06	93.76 MB	★★
Susie-Q	04:35	6.32 MB	★★★★
I Put a Spell on You	04:32	6.24 MB	

The image shows the filter set to the word "Water." The filter looks for matches in all fields (song, album, artist), so all "Creedence Clearwater Revival" songs are automatically included.

Notice the status label above the grid. It automatically updates whenever the list changes, so when the filter is applied the status updates to reflect the new filter. The routine that updates the status uses LINQ to calculate the number of artists, albums, and songs selected, as well as the total storage and play time. The song status update routine is implemented as follows:

C#

```
// update song status
 void UpdateSongStatus()
 {
   var view = _flexiTunes.ItemsSource as ICollectionView;
   var songs = view.OfType<Song>();
   _txtSongs.Text = string.Format(
      "{0:n0} Artists; {1:n0} Albums; {2:n0} Songs; " +
      "{3:n0} MB of storage; {4:n2} days of music.",
     (from s in songs select s.Artist).Distinct().Count(),
     (from s in songs select s.Album).Distinct().Count(),
     (from s in songs select s.Name).Count(),
     (double)(from s in songs select s.Size/1024.0/1024.0).Sum(),
     (double)(from s in songs select s.Duration/3600000.0/24.0).Sum());
 }
```

This routine is not directly related to the grid, but is listed here because it shows how you can leverage the power of LINQ to summarize status information that is often necessary when showing grids bound to large data sources.

The LINQ statement above uses the **Distinct** and **Count** commands to calculate the number of artists, albums, and songs currently exposed by the data source. It also uses the **Sum** command to calculate the total storage and play time for the current selection.

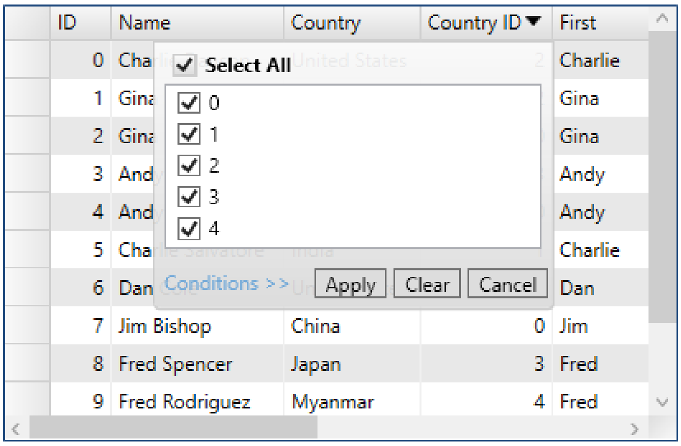
The Filter predicate of ICollectionView cannot filter the child rows. To filter a child row, the type of child rows must be ICollectionView. You can filter both child rows and parent grid. However, you need to make sure that the parent and child both must be of ICollectionView type and their binding names should also be different. Note that filtering child rows using ICollectionView does not support sorting as of now.

Filter Data using FlexGridFilter

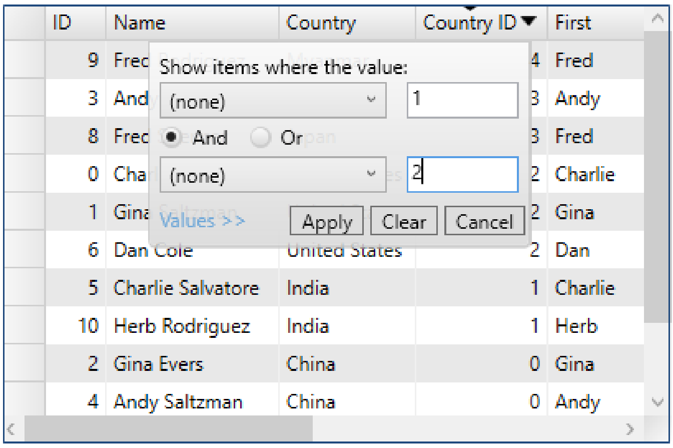
FlexGrid provides Excel-like filtering feature to filter data through drop down icons in column headers. This functionality is available in FlexGrid through a separate control called FlexGridFilter, which is implemented through an extender assembly, C1.WPF.FlexGridFilter. Once the FlexGridFilter control is added, the grid displays a drop-down icon on hovering the column headers. The drop-down icon shows an editor that allows users to specify filters on columns. Users may choose between the two types of filters:

- **Value filter**: This filter lets you filter specific values in the column.
- **Condition filter**: This filter lets you specify conditions composed of an operator (greater than, less than, etc.) and a parameter. The conditions can be combined using an **AND** or an **OR** operator.

The images below show the filters displayed on clicking the drop-down icon.



Value Filter



Conditional Filter

To enable filtering through FlexGridFilter

You can enable filtering in FlexGrid by manually adding the C1.WPF.FlexGridFilter assembly to your project, creating an object of the C1FlexGridFilter class, and attaching this object to an existing FlexGrid object as illustrated in the following code.

XAML

```
<c1:C1FlexGrid Name="grid" >
  <!-- add filtering support to the control: -->
  <c1:C1FlexGridFilterService.FlexGridFilter>
    <c1:C1FlexGridFilter />
  </c1:C1FlexGridFilterService.FlexGridFilter>
</c1:C1FlexGrid>
```

C#

```
// create a FlexGrid
var grid = new C1FlexGrid();
//enable filtering through FlexGridFilter
var gridFilter = new C1FlexGridFilter(grid);
```

To select filter mode

The filter operates in two modes depending on the value of the UseCollectionView property. If the UseCollectionView property is set to **false**, rows that do not satisfy the filter are hidden (the filter sets their Visible property to **false**). In this mode, the filter has no effect on the row count. You can use this mode in bound and unbound grids.

If the filter's UseCollectionView property is set to **true**, the filter gets applied to the data source. In this mode, changes to the filter affect the number of items exposed by the data source to the grid and to any other controls bound to the same data source. This filtering mode can only be used in bound grids.

XAML

```
<c1:C1FlexGrid Name="grid" >
  <!-- add filtering support to the control: -->
  <c1:C1FlexGridFilterService.FlexGridFilter>
    <c1:C1FlexGridFilter UseCollectionView=”True”/>
  </c1:C1FlexGridFilterService.FlexGridFilter>
</c1:C1FlexGrid>
```

C#

```
// create C1FlexGrid
var grid = new C1FlexGrid();

// enable filtering on the grid
var gridFilter = new C1FlexGridFilter(grid);

// filter at the data source level
gridFilter.UseCollectionView = true;
```

To customize filter type for each column

By default, filters are enabled for every column. Columns that contain Boolean or enumerated data get a value filter, and columns that contain other data types get **value** and **condition** You can use the FilterType property to change this behavior and specify the type of filter to enable for each column.

Specifying the filter type is important in scenarios where columns have a large number of unique values or when columns contain bindings that do not work with the filters. For example, columns containing images cannot be filtered with value or condition filters. In this case, you would disable the filter by setting the FilterType property to **None**.

A grid containing several thousand items may have a unique ID column, which adds too many items to the value filter, making it slow and not very useful. In this case, disable the value filter by setting the FilterType property to **Condition**.

The code below shows how to accomplish this:

C#

```
// create C1FlexGrid
var grid= new C1FlexGrid();

// enable filtering on the grid
var gridFilter = new C1FlexGridFilter(grid);

// disable filtering on the Image column
var columnFilter = gridFilter.GetColumnFilter(flex.Columns[“Image”]);
columnFilter.FilterType = FilterType.None;

// disable value filtering on the ID column
columnFilter = gridFilter.GetColumnFilter(flex.Columns[“ID”]);
columnFilter.FilterType = FilterType.Condition;
```

To specify filter type in code

In most cases, users set the filters. But the ColumnFilter class exposes a full object model that enables developers to customize filter conditions through code. For example, the code below applies a filter to the second column. The filter causes the grid to show items where the value in the second column contains the letter Z:

C#

```
// create C1FlexGrid
var grid = new C1FlexGrid();

// enable filtering on the grid
var gridFilter = new C1FlexGridFilter(grid);

// get filter for the first column
var columnFilter = gridFilter.GetColumnFilter(grid.Columns[0]);

// create filter condition (Contains ‘Z’)
var condition = columnFilter.ConditionFilter.Condition1;
condition.Operator = ConditionOperator.Contains;
condition.Parameter = "Z";

// apply the filter
gridFilter.Apply();
```

To persist filters

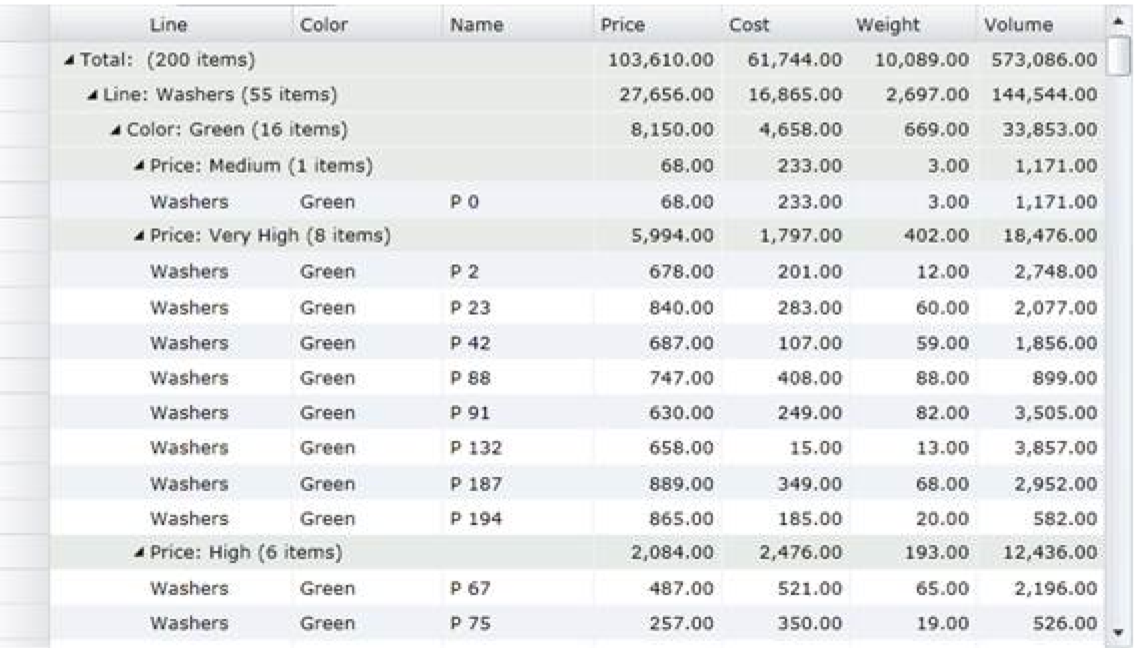
The C1FlexGridFilter class contains a FilterDefinition property that gets or sets the current filter state as an XML string. You can use this string to persist the filter state when the user quits the application, so that it can be restored later. You may also save several filter definitions, and allow the user to select and then customize these pre-set filters. You can also save and restore filter definitions to streams using the SaveFilterDefinition and LoadFilterDefinition methods.

> Note: The C1.WPF.FlexGridFilter extender assembly is shipped separately with FlexGrid due to following reasons:
>
> - To minimize the footprint of FlexGrid assembly.
> - To provide complete flexibility in choosing extensions.
> - To extend the functionality of the C1FlexGrid class through custom codes.

Data Aggregation

FlexGrid lets you compute and display aggregate value for each group created after grouping data. The control shows groups in a collapsible outline format and automatically displays the number of items in each group. However, you can go one step further and display aggregate values for every grouped column. This feature enables users in drawing out more insights from random data. For instance, a company's sales data grouped by country or product category can be more useful if the aggregate sales can be indicated against each country and product category in the grid itself. The Column class provides the GroupAggregate property that can be set to **Aggregate** to automatically calculate and display aggregates. The aggregates calculated by setting the GroupAggregate property automatically recalculate the aggregate values when the data changes.

The following image shows a FlexGrid with aggregate values displayed in the columns.



Line	Color	Name	Price	Cost	Weight	Volume
◢ Total: (200 items)			103,610.00	61,744.00	10,089.00	573,086.00
◢ Line: Washers (55 items)			27,656.00	16,865.00	2,697.00	144,544.00
◢ Color: Green (16 items)			8,150.00	4,658.00	669.00	33,853.00
◢ Price: Medium (1 items)			68.00	233.00	3.00	1,171.00
Washers	Green	P 0	68.00	233.00	3.00	1,171.00
◢ Price: Very High (8 items)			5,994.00	1,797.00	402.00	18,476.00
Washers	Green	P 2	678.00	201.00	12.00	2,748.00
Washers	Green	P 23	840.00	283.00	60.00	2,077.00
Washers	Green	P 42	687.00	107.00	59.00	1,856.00
Washers	Green	P 88	747.00	408.00	88.00	899.00
Washers	Green	P 91	630.00	249.00	82.00	3,505.00
Washers	Green	P 132	658.00	15.00	13.00	3,857.00
Washers	Green	P 187	889.00	349.00	68.00	2,952.00
Washers	Green	P 194	865.00	185.00	20.00	582.00
◢ Price: High (6 items)			2,084.00	2,476.00	193.00	12,436.00
Washers	Green	P 67	487.00	521.00	65.00	2,196.00
Washers	Green	P 75	257.00	350.00	19.00	526.00

To set aggregate through code

You can display the aggregate value for each group in FlexGrid by setting the grid's AreGroupHeadersFrozen property to **false** and then setting the GroupAggregate property on each column to one of the supported aggregate values. Some of the supported aggregate values are **Sum**, **Average**, **Count**, **Minimum**, **Maximum**, etc.

> **Note:** Since the aggregates appear in the group header rows, it is necessary to make them visible by setting the AreGroupHeadersFrozen property to **false**.

The following code illustrates displaying totals for the Price, Cost, Weight and Volume columns through XAML.

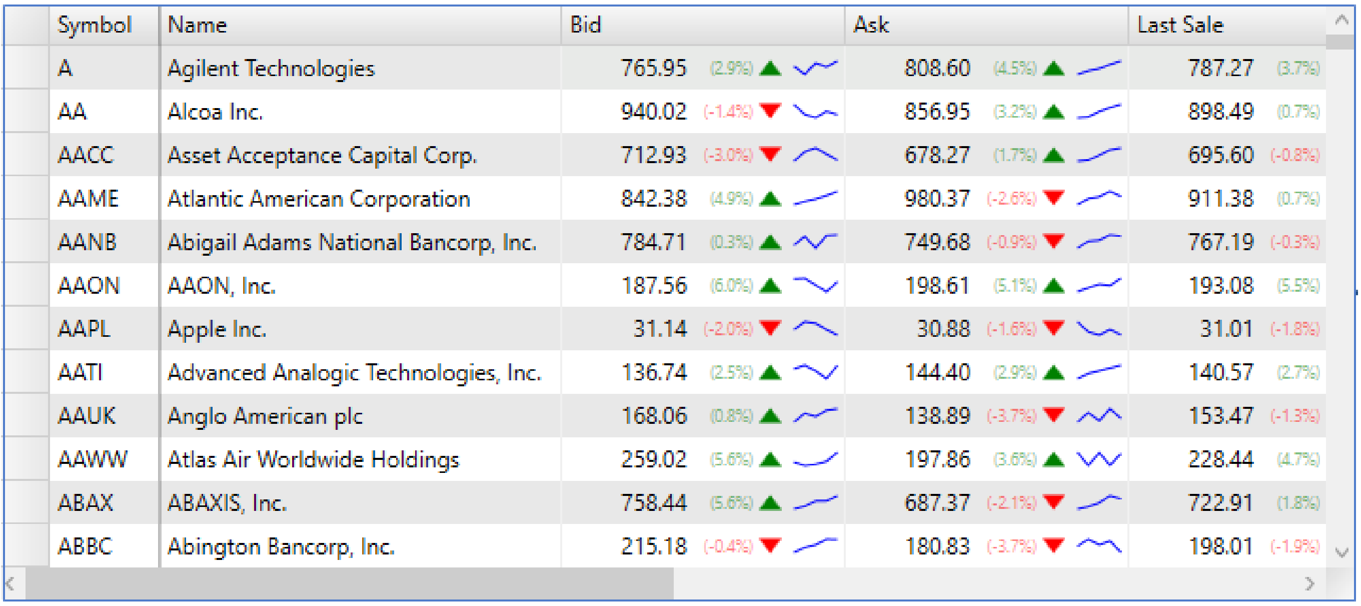
XAML

```
<fg:C1FlexGrid x:Name="_flex" AutoGenerateColumns="False"
  AreRowGroupHeadersFrozen="False">
  <fg:C1FlexGrid.Columns>
    <fg:Column Header="Line" Binding="{Binding Line}" />
    <fg:Column Header="Color" Binding="{Binding Color}" />
    <fg:Column Header="Name" Binding="{Binding Name}" />
    <fg:Column Header="Price" Binding="{Binding Price}"
      Format="n2" HorizontalAlignment="Right" Width="*"
      GroupAggregate="Sum"/>
    <fg:Column Header="Cost" Binding="{Binding Cost}"
      Format="n2" HorizontalAlignment="Right" Width="*"
      GroupAggregate="Sum"/>
    <fg:Column Header="Weight" Binding="{Binding Weight}"
      Format="n2" HorizontalAlignment="Right" Width="*"
      GroupAggregate="Sum"/>
    <fg:Column Header="Volume" Binding="{Binding Volume}"
      Format="n2" HorizontalAlignment="Right" Width="*"
      GroupAggregate="Sum"/>
  </fg:C1FlexGrid.Columns>
</fg:C1FlexGrid>
```

Custom Cells in code: CellFactory class

FlexGrid comes with the CellFactory class to create every cell that appears on the grid. You can create custom cells by creating a class in your project that implements the ICellFactory interface and assign it to the CellFactory property of FlexGrid. Custom ICellFactory classes can be highly specialized and application-specific, or can be very generic and reusable. In general, custom ICellFactory classes are simpler than custom columns since they deal directly with cells. Implementing custom **ICellFactory** classes is fairly easy because you can inherit from the default CellFactory class included with the C1FlexGrid class. The default CellFactory class was designed to be extensible, so you can let it handle all the details of cell creation and customize only what you need.

The following image shows custom cells created through CellFactory in FlexGrid.



Symbol	Name	Bid	Ask	Last Sale
A	Agilent Technologies	765.95 (2.9%) ▲	808.60 (4.5%) ▲	787.27 (3.7%)
AA	Alcoa Inc.	940.02 (-1.4%) ▼	856.95 (3.2%) ▲	898.49 (0.7%)
AACC	Asset Acceptance Capital Corp.	712.93 (-3.0%) ▼	678.27 (1.7%) ▲	695.60 (-0.8%)
AAME	Atlantic American Corporation	842.38 (4.9%) ▲	980.37 (-2.6%) ▼	911.38 (0.7%)
AANB	Abigail Adams National Bancorp, Inc.	784.71 (0.3%) ▲	749.68 (-0.9%) ▼	767.19 (-0.3%)
AAON	AAON, Inc.	187.56 (6.0%) ▲	198.61 (5.1%) ▲	193.08 (5.5%)
AAPL	Apple Inc.	31.14 (-2.0%) ▼	30.88 (-1.6%) ▼	31.01 (-1.8%)
AATI	Advanced Analogic Technologies, Inc.	136.74 (2.5%) ▲	144.40 (2.9%) ▲	140.57 (2.7%)
AAUK	Anglo American plc	168.06 (0.8%) ▲	138.89 (-3.7%) ▼	153.47 (-1.3%)
AAWW	Atlas Air Worldwide Holdings	259.02 (5.6%) ▲	197.86 (3.6%) ▲	228.44 (4.7%)
ABAX	ABAXIS, Inc.	758.44 (5.6%) ▲	687.37 (-2.1%) ▼	722.91 (1.8%)
ABBC	Abington Bancorp, Inc.	215.18 (-0.4%) ▼	180.83 (-3.7%) ▼	198.01 (-1.9%)

When using custom cells, it is important to understand that grid cells are transient. Cells are constantly created and destroyed as the user scrolls, sorts, or selects ranges on the grid. This process is known as virtualization and is quite common in WPF and Silverlight applications. Without virtualization, a grid would typically have to create several thousand visual elements at the same time, which would impact its performance.

The following code shows **ICellFactory** interface.

C#

```
public interface ICellFactory
{
  FrameworkElement CreateCell(
    C1FlexGrid grid,
    CellType cellType,
    CellRange range);

  FrameworkElement CreateCellEditor(
    C1FlexGrid grid,
    CellType cellType,
    CellRange range)

  void DisposeCell(
    C1FlexGrid grid,
    CellType cellType,
    FrameworkElement cell);
}
```

The first method, CreateCell, creates **FrameworkElement** objects to represent cells. The parameters include the grid that owns the cells, the type of cell to create, and the CellRange to represent. The CellType parameter specifies whether the cell being created is a regular data cell, a row or column header, or the fixed cells at the top left and bottom right of the grid. The CreateCellEditor method is analogous to the first but creates a cell in edit mode. The last method, DisposeCell, is called after the cell is removed from the grid. It gives the caller a chance to dispose of any resources associated with the cell object.

Custom Cells using XAML Templates

If you prefer to create custom cells in XAML instead of writing code, you can do that as well. The Column class has CellTemplate and CellEditingTemplate properties that you can use to specify the visual elements responsible for showing and editing cells in the column.

For example, the XAML code below defines custom visual elements used to show and edit values in a column. Cells in that column are shown as green, bold, center-aligned text, and edited using a textbox that has an edit icon next to it:

XAML

```
<c1:C1FlexGrid x:Name="_fgTemplated">
  <c1:C1FlexGrid.Columns>
    <!-- add a templated column -->
    <c1:Column ColumnName="_colTemplated" Header="Template" Width="200">
      <!-- template for cells in display mode -->
      <c1:Column.CellTemplate>
        <DataTemplate>
          <TextBlock Text="{Binding Name}"
           Foreground="Green" FontWeight="Bold"
           VerticalAlignment="Center"/>
        </DataTemplate>
      </c1:Column.CellTemplate>
      <!-- template for cells in edit mode -->
      <c1:Column.CellEditingTemplate>
        <DataTemplate>
          <Grid>
            <Grid.ColumnDefinitions>
              <ColumnDefinition Width="Auto" />
              <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>
            <Image Source="edit_icon.png" Grid.Column="0" />
            <TextBox Text="{Binding Name, Mode=TwoWay}" Grid.Column="1" />
          </Grid>
        </DataTemplate>
      </c1:Column.CellEditingTemplate>
    </c1:Column>
  </c1:C1FlexGrid.Columns>
</c1:C1FlexGrid>
```

Print Support

FlexGrid supports basic as well as advanced printing capabilities. You can specify scale, margins, number of pages while printing to ensure that the data renders correctly with intact style, fonts and gradients.

Basic printing
Learn how to implement basic printing in FlexGrid.
Advanced printing
Learn how to implement advanced printing in FlexGrid.

Basic Printing

The **Print** method is the easy way to print a **C1FlexGrid**.

The **Print** method allows you to specify the document name, page margins, scaling, and a maximum number of pages to print. The output is a faithful rendering of the grid, including all style elements, fonts, gradients, images, etc. Row and column headers are included on every page.

> **Note**: The C1.Silverlight.FlexGrid.5 library provides additional overloads for the **Print** method to support **printerFallback** and **useDefaultPrinter** capabilities in Silverlight 5. These overloads are only available in the Silverlight 5 version of the control.

Advanced Printing

If you want more control over the printing process, use the GetPageImages method to automatically break up the grid into images that can be rendered onto individual pages. Each image is a 100% accurate representation of a portion of the grid, including styles, custom elements, repeating row and column headers on every page, and so on.

The **GetPageImages** method also allows callers to scale the images so the entire grid renders in actual size, scales to fit onto a single page, or scales to the width of a single page.

Once you have obtained the page images, you can use the WPF or Silverlight printing support to render them into documents with complete flexibility. For example, you can create documents that contain several grids, charts, and other types of content. You can also customize headers and footers, add letterheads, and so on.

The printing frameworks in WPF and Silverlight are different. The following sections demonstrate how an application can render the **FlexGrid** using the **GetPageImages** onto a print document in either platform.

Printing a C1FlexGrid in WPF

Printing documents in WPF requires a slightly different sequence of steps than in Silverlight:

1. Create a **PrintDialog** object.
2. If the dialog box's **ShowDialog** method returns true, then:
3. Create a **Paginator** object that will provide the document content.
4. Call the dialog's **Print** method.

The code below shows a sample implementation of this mechanism.

C#

```
// print the grid
void _btnPrint_Click(object sender, RoutedEventArgs e)
{
  var pd = new PrintDialog();
  if (pd.ShowDialog().Value)
  {
    // get margins, scale mode
    var margin = 96.0;
    var scaleMode =;

    // get page size
    var pageSize = new Size(pd.PrintableAreaWidth,
                            pd.PrintableAreaHeight);

    // create paginator
    var paginator = new FlexPaginator(
      _flex, ScaleMode.PageWidth,
      pageSize,
      new Thickness(margin), 100);

    // print the document
    pd.PrintDocument(paginator, "C1FlexGrid printing example");
  }
}
```

The **FlexPaginator** class provides the page images and is conceptually similar to the **PrintPage** event handler used in Silverlight. Implement it as follows:

C#

```
/// <summary>
/// DocumentPaginator class used to render C1FlexGrid controls.
/// </summary>
public class FlexPaginator : DocumentPaginator
{
  Thickness _margin;
  Size _pageSize;
  ScaleMode _scaleMode;
  List<FrameworkElement> _pages;

  public FlexPaginator(C1FlexGrid flex,
    ScaleMode scaleMode,
    Size pageSize,
    Thickness margin, int maxPages)
  {
    // save parameters
    _margin = margin;
    _scaleMode = scaleMode;
    _pageSize = pageSize;

    // adjust page size for margins before building grid images
    pageSize.Width -= (margin.Left + margin.Right);
    pageSize.Height -= (margin.Top + margin.Bottom);

    // get grid images for each page
    _pages = flex.GetPageImages(scaleMode, pageSize, maxPages);
  }
```

The constructor creates the page images. They are later rendered onto pages when the printing framework invokes the paginator's **GetPage** method:

C#

```
public override DocumentPage GetPage(int pageNumber)
{
  // create page element
  var pageTemplate = new PageTemplate();

  // set margins
  pageTemplate.SetPageMargin(_margin);

  // set content
  pageTemplate.PageContent.Child = _pages[pageNumber];
  pageTemplate.PageContent.Stretch =
    _scaleMode == ScaleMode.ActualSize
      ? System.Windows.Media.Stretch.None
      : System.Windows.Media.Stretch.Uniform;

  // set footer text
  pageTemplate.FooterRight.Text = string.Format("Page {0} of {1}",
      pageNumber + 1, _pages.Count);

  // arrange the elements on the page
  pageTemplate.Arrange(
      new Rect(0, 0, _pageSize.Width, _pageSize.Height));

  // return new document page
  return new DocumentPage(pageTemplate);
}
```

As in the Silverlight example, a helper **PageTemplate** class is used to hold the grid images and to provide the margins, headers, and footers.

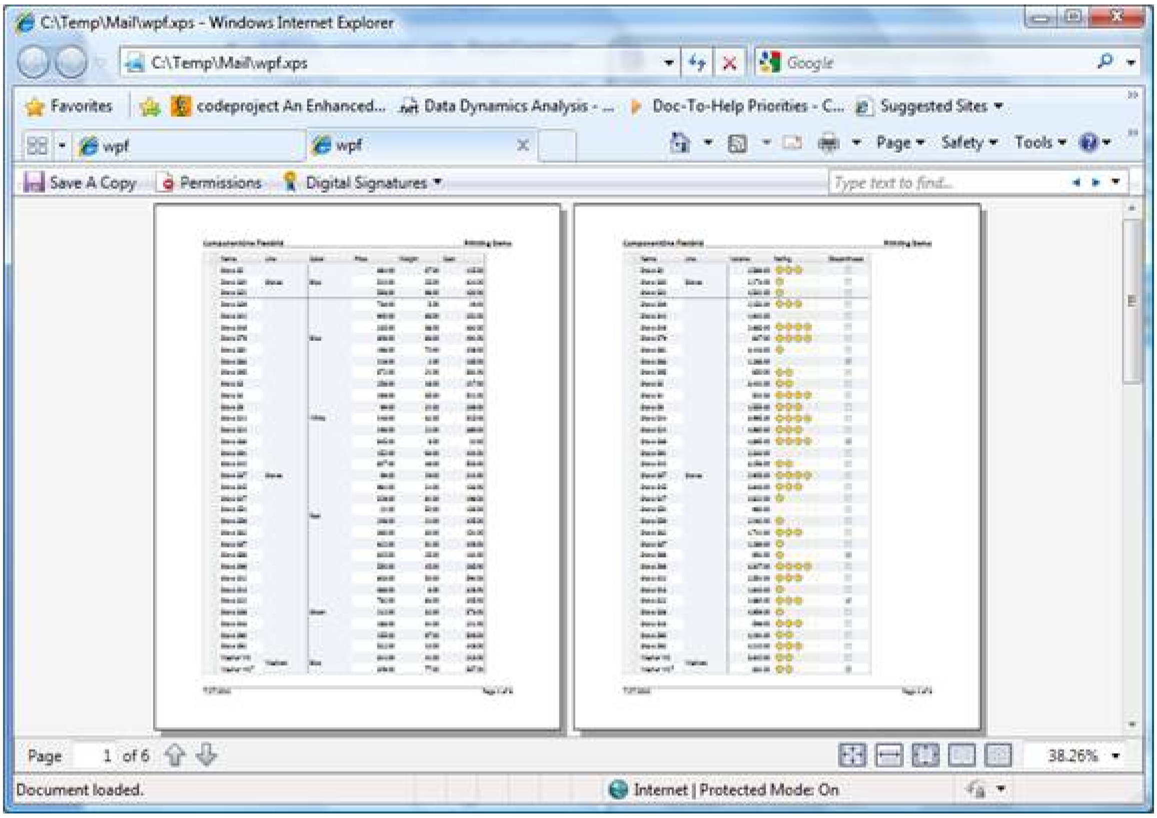
The remaining paginator methods have trivial implementations:

C#

```
public override int PageCount
{
  get { return _pages.Count; }
}
public override IDocumentPaginatorSource Source
{
  get { return null; }
}
public override Size PageSize
```

```
  {
    get { return _pageSize; }
    set { throw new NotImplementedException(); }
  }
  public override bool IsPageCountValid
  {
    get { return true; }
  }
}
```

The image below shows the document created when the grid is rendered into an XPS file. The image is very accurate, including a custom rating cell used in the sample. Row and column headers are automatically included in every page, as well as a simple page header and the standard “Page n of m” page footers.



Printing a C1FlexGrid in Silverlight

Follow these steps to print documents in Silverlight:

1. Create a **PrintDocument** object.
2. Connect handlers to the **BeginPrint**, **PrintPage**, and **EndPrint** events.
3. Call the document’s **Print** method.

The **Print** method shows a print dialog. If the user clicks OK, the document fires the **BeginPrint** event once, then **PrintPage** once for each page, and finally **EndPrint** when the last page renders. The code below shows a sample implementation of this mechanism.

We use two variables to hold the page images and to keep track of the page being rendered:

C#

```
List<FrameworkElement> _pages;
int _currentPage;
```

Here is the handler called to print the document:

C#

```
// print the grid
void _btnPrint_Click(object sender, RoutedEventArgs e)
{
  // create a PrintDocument
  var pd = new System.Windows.Printing.PrintDocument();

  // prepare to print
  _pages = null;
  pd.PrintPage += pd_PrintPage;

  // print the document
  pd.Print("C1FlexGrid");
}
```

The **PrintPage** method does all of the work. It generates all the page images the first time it is called, then renders the images onto the pages as they are created.

C#

```
void pd_PrintPage(object sender, PrintPageEventArgs e)
{
  if (_pages == null)
  {
    // calculate page size, discount margins
    var sz = e.PrintableArea;
    sz.Width -= 2 * 96; // one inch left/right margins
    sz.Height -= 2 * 96; // one inch top/bottom margins

    // generate the page images
    _currentPage = 0;
    _pages = _flex.GetPageImages(ScaleMode.ActualWidth, sz, 100);
  }

  // create visual element that represents this page
  var pageTemplate = new PageTemplate ();

  // apply margins to the page template
  pageTemplate.SetPageMargin(new Thickness(_margin));

  // add content to page template
  pageTemplate.PageContent.Child = _pages[_currentPage];

  // apply footer text
  pageTemplate.FooterRight.Text = string.Format("Page {0} of {1}",
    _currentPage + 1, _pages.Count);
  // render the page
  e.PageVisual = pageTemplate;

  // move on to next page
  _currentPage++;
  e.HasMorePages = _currentPage < _pages.Count;
}
```

Instead of rendering the grid images directly onto the pages, the sample uses a custom auxiliary class called **PageTemplate.** This class provides the margins, headers, footers, and a **ViewBox** control that hosts the actual grid images. Rendering the grid images directly onto the page would also work, but the template adds a lot of flexibility. (Note that the **PageTemplate** class is implanted in the sample and is not part of the C1FlexGrid assembly).

Here is the XAML that defines the **PageTemplate** class:

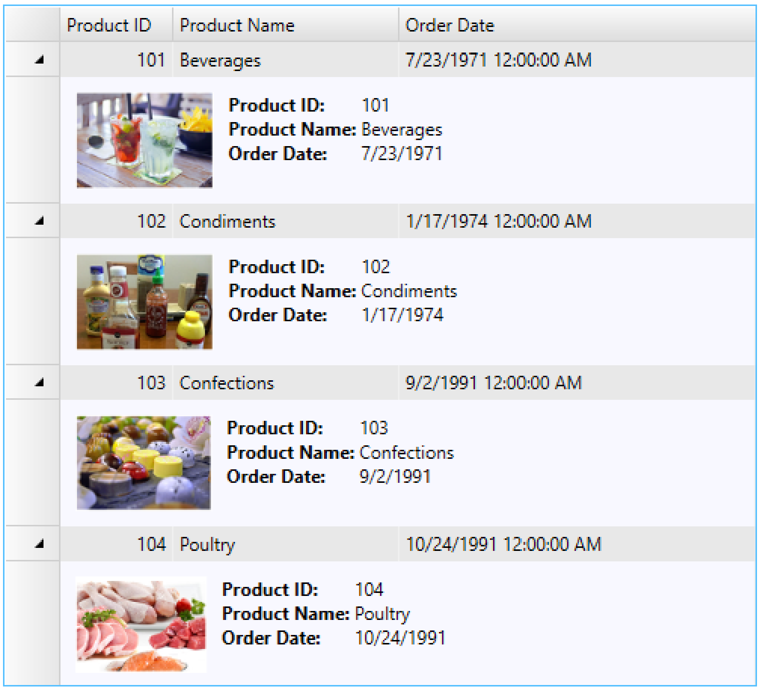
XAML

```
<Grid x:Name="LayoutRoot" Background="White">  <Grid.RowDefinitions>
<RowDefinition Height="96" />    <RowDefinition Height="*"/>    <RowDefinition
Height="96" />  </Grid.RowDefinitions>  <Grid.ColumnDefinitions>    <ColumnDefinition
Width="96"/>    <ColumnDefinition Width="*" />    <ColumnDefinition Width="96"/>
</Grid.ColumnDefinitions>

  <!-- header -->  <Border Grid.Column="1" HorizontalAlignment="Stretch"
VerticalAlignment="Bottom" Margin="0 12"        BorderBrush="Black"
BorderThickness="0 0 0 1" >    <Grid>      <TextBlock Text="ComponentOne FlexGrid"
FontWeight="Bold" FontSize="14"        VerticalAlignment="Bottom"
HorizontalAlignment="Left" />      <TextBlock Text="Printing Demo"
FontWeight="Bold" FontSize="14"        VerticalAlignment="Bottom"
HorizontalAlignment="Right" />    </Grid>  </Border>

  <!-- footer -->  <Border Grid.Column="1" Grid.Row="2" HorizontalAlignment="Stretch"
VerticalAlignment="Top"  Margin="0 12"       BorderBrush="Black" BorderThickness="0 1
0 0" >    <Grid>      <TextBlock x:Name="FooterLeft" Text="Today"
VerticalAlignment="Bottom" HorizontalAlignment="Left" />      <TextBlock
x:Name="FooterRight" Text="Page {0} of {1}"         VerticalAlignment="Bottom"
HorizontalAlignment="Right" />    </Grid>  </Border>

  <! -- page content -->  <Viewbox x:Name="PageContent" Grid.Row="1" Grid.Column="1"
VerticalAlignment="Top" HorizontalAlignment="Left" /></Grid>
```

Row Details Template

Row details template is a data panel that can be added to each row for displaying details. FlexGrid provides the flexibility to show information about each row through a template. You can embed text, UI elements, and data-bound controls, such as InputPanel, in the row details template. For each row, you can insert a data template to present its summary and show/provide details in other controls, such as text box, without affecting the dimensions of the grid. You can also use this template to create hierarchical grids displaying grouped data. In this example, we use row details template to display product-related information in FlexGrid.

The following image shows details of each row displayed through a row details template.



Product ID	Product Name	Order Date
101	Beverages	7/23/1971 12:00:00 AM

Product ID: 101
Product Name: Beverages
Order Date: 7/23/1971

Product ID	Product Name	Order Date
102	Condiments	1/17/1974 12:00:00 AM

Product ID: 102
Product Name: Condiments
Order Date: 1/17/1974

Product ID	Product Name	Order Date
103	Confections	9/2/1991 12:00:00 AM

Product ID: 103
Product Name: Confections
Order Date: 9/2/1991

Product ID	Product Name	Order Date
104	Poultry	10/24/1991 12:00:00 AM

Product ID: 104
Product Name: Poultry
Order Date: 10/24/1991

To add row details template in FlexGrid

1. Drag a grid control onto the XAML designer in a WPF application.
2. Create three columns in the grid to display **Product ID**, **Product Name** and **Order Date** data fields.

XAML copyCode

```
<c1:C1FlexGrid.Columns>
    <c1:Column Header="Product ID" Binding="{Binding ProductId}" Width="75" />
    <c1:Column Header="Product Name" Binding="{Binding ProductName}" Width="150" />
    <c1:Column Header="Order Date" Binding="{Binding OrderDate}" Width="300" />
</c1:C1FlexGrid.Columns>
```

3. Create a row details template in XAML that embeds an image control and six text block controls within a DockPanel.

XAML copyCode

```
<c1:C1FlexGrid.RowDetailsTemplate>
      <DataTemplate>
          <DockPanel Background="GhostWhite">
              <Image DockPanel.Dock="Left" Name="img" Source="{Binding ImgSource}"
                     Height="64" Margin="10" />
              <Grid Margin="0, 10">
                  <Grid.ColumnDefinitions>
                      <ColumnDefinition Width="Auto" />
                      <ColumnDefinition Width="*" />
                  </Grid.ColumnDefinitions>
                  <Grid.RowDefinitions>
                      <RowDefinition Height="Auto" />
                      <RowDefinition Height="Auto" />
                      <RowDefinition Height="Auto" />
                  </Grid.RowDefinitions>
                  <TextBlock Text="Product ID: " FontWeight="Bold" />
                  <TextBlock Text="{Binding ProductId}" Grid.Column="1" />
                  <TextBlock Text="Product Name: " FontWeight="Bold" Grid.Row="1" />
                  <TextBlock Text="{Binding ProductName}" Grid.Column="1" Grid.Row="1" />
                  <TextBlock Text="Order Date: " FontWeight="Bold" Grid.Row="2" />
                  <TextBlock Text="{Binding OrderDate, StringFormat=d}"
                             Grid.Column="1" Grid.Row="2" />
              </Grid>
          </DockPanel>
      </DataTemplate>
  </c1:C1FlexGrid.RowDetailsTemplate>
```

4. In the MainWindow.xaml.cs file, create a class, User, to add data in the grid.
   - **Visual Basic**

```
Public Class User
    Public Property ProductId() As Integer
        Get
            Return m_ProductId
        End Get
        Set
            m_ProductId = Value
        End Set
    End Property
    Private m_ProductId As Integer

    Public Property ProductName() As String
        Get
            Return m_ProductName
        End Get
        Set
            m_ProductName = Value
        End Set
    End Property
    Private m_ProductName As String

    Public Property OrderDate() As DateTime
        Get
            Return m_OrderDate
        End Get
        Set
            m_OrderDate = Value
        End Set
    End Property
    Private m_OrderDate As DateTime

    Public Property ImgSource() As ImageSource
        Get
            Return m_ImgSource
        End Get
        Set
            m_ImgSource = Value
        End Set
    End Property
    Private m_ImgSource As ImageSource

End Class
```

   - **C#**

```
public class User
{
    public int ProductId { get; set; }

    public string ProductName { get; set; }

    public DateTime OrderDate { get; set; }

    public ImageSource ImgSource{ get; set; }

}
```

5. Populate the grid by creating a list, and bind the grid to the list using the ItemsSource property.
   - **Visual Basic**

```
Public Sub New()
    InitializeComponent()

    'Create a list
    Dim users As New List(Of User)()

    'Add items to the list
    users.Add(New User() With {
    .ProductId = 101,
    .ProductName = "Beverages",
    .OrderDate = New DateTime(1971, 7, 23),
    .ImgSource = New BitmapImage(New Uri("pack://application:,,,/Resources/Beverage.png"))
})
    users.Add(New User() With {
    .ProductId = 102,
    .ProductName = "Condiments",
    .OrderDate = New DateTime(1974, 1, 17),
    .ImgSource = New BitmapImage(New Uri("pack://application:,,,/Resources/Condiments.png"))
})
    users.Add(New User() With {
    .ProductId = 103,
    .ProductName = "Confections",
        .OrderDate = New DateTime(1991, 9, 2),
        .ImgSource = New BitmapImage(New Uri("pack://application:,,,/Resources/Confections.png"))
    })
        users.Add(New User() With {
        .ProductId = 104,
        .ProductName = "Poultry",
        .OrderDate = New DateTime(1991, 10, 24),
        .ImgSource = New BitmapImage(New Uri("pack://application:,,,/Resources/Poultry.png"))
    })

        'Populate the grid
        grid.ItemsSource = users
    End Sub
```

- **C#**

```
public MainWindow()
{
    InitializeComponent();

    //Create a list
    List<User> users = new List<User>();

    //Add items to the list
    users.Add(new User() { ProductId = 101, ProductName = "Beverages", OrderDate = new DateTime(1971, 7, 23),
        ImgSource = new BitmapImage(new Uri(@"pack://application:,,,/Resources/Beverage.png")) });
    users.Add(new User() { ProductId = 102, ProductName = "Condiments", OrderDate = new DateTime(1974, 1, 17),
        ImgSource = new BitmapImage(new Uri(@"pack://application:,,,/Resources/Condiments.png")) });
    users.Add(new User() { ProductId = 103, ProductName = "Confections", OrderDate = new DateTime(1991, 9, 2),
        ImgSource = new BitmapImage(new Uri(@"pack://application:,,,/Resources/Confections.png")) });
    users.Add(new User() { ProductId = 104, ProductName = "Poultry", OrderDate = new DateTime(1991, 10, 24),
        ImgSource = new BitmapImage(new Uri(@"pack://application:,,,/Resources/Poultry.png")) });

    //Populate the grid
    grid.ItemsSource = users;
}
```

Layout and Appearance: ClearStyle Technology

ComponentOne ClearStyle technology is a new, quick and easy approach to providing Silverlight and WPF control styling. ClearStyle allows you to create a custom style for a control without having to deal with the hassle of XAML templates and style resources.

Currently, to add a theme to all standard WPF controls, you must create a style resource template. In Microsoft Visual Studio, this process can be difficult; this is why Microsoft introduced Expression Blend to make the task a bit easier. Having to jump between two environments can be a bit challenging to developers who are not familiar with Blend or do not have the time to learn it. You could hire a designer, but that can complicate things when your designer and your developers are sharing XAML files.

That's where ClearStyle comes in. With ClearStyle the styling capabilities are brought to you in Visual Studio in the most intuitive manner possible. In most situations you just want to make simple styling changes to the controls in your application so this process should be simple. For example, if you just want to change the row color of your data grid this should be as simple as setting one property. You shouldn't have to create a full and complicated-looking template just to simply change a few colors.

Structure of ClearStyle

Each key piece of the control's style is surfaced as a simple color property. This leads to a unique set of style properties for each control. For example, a **Gauge** has **PointerFill** and **PointerStroke** properties, whereas a **DataGrid** has **SelectedBrush** and **MouseOverBrush** for rows.

Let's say you have a control on your form that does not support ClearStyle. You can take the XAML resource created by ClearStyle and use it to help mold other controls on your form to match (such as grabbing exact colors). Or let's say you'd like to override part of a style set with ClearStyle (such as your own custom scrollbar). This is also possible because ClearStyle can be extended and you can override the style where desired.

ClearStyle is intended to be a solution to quick and easy style modification but you're still free to do it the old fashioned way with ComponentOne's controls to get the exact style needed. ClearStyle does not interfere with those less common situations where a full custom design is required.

ClearStyle Property

FlexGrid for WPF and **Silverlight** support ComponentOne's new ClearStyle technology that allows you to easily change control colors without having to change control templates. By just setting a few color properties you can quickly style the control.

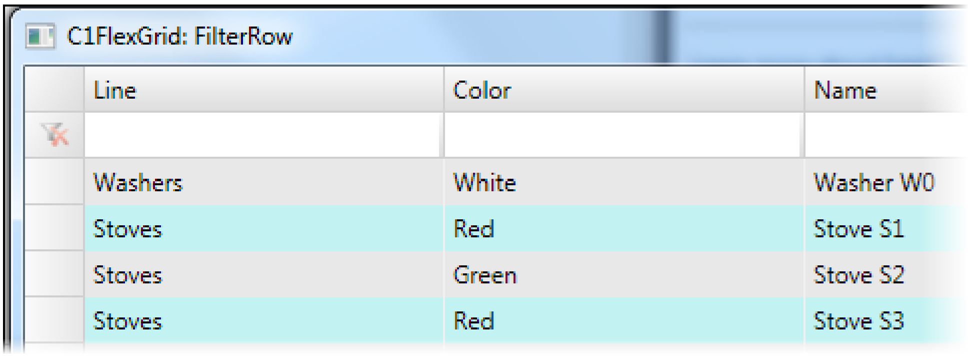
The following table outlines the brush properties of the **C1FlexGrid** control:

Brush	Description
Background	Gets or sets the brush of the control’s background.
AlternatingRowBackground	Gets or sets the System.Windows.Media.Brush that is used to paint the background of odd-numbered rows.
BottomRightCellBackground	Gets or sets the System.Windows.Media.Brush that is used to paint the background of the cell at the bottom right corner of the grid.
ColumnHeaderBackground	Gets or sets the System.Windows.Media.Brush that is used to paint column header backgrounds.
ColumnHeaderForeground	Gets or sets the System.Windows.Media.Brush that is used to paint column header content.
ColumnHeaderSelectedBackground	Gets or sets the System.Windows.Media.Brush that is used to paint column header backgrounds for selected cells.
CursorBackground	Gets or sets the System.Windows.Media.Brush that is used to paint the background of the cursor cell.
CursorForeground	Gets or sets the System.Windows.Media.Brush that is used to paint the foreground of the cursor cell.
EditorBackground	Gets or sets the System.Windows.Media.Brush that is used to paint the background of cells in edit mode.
EditorForeground	Gets or sets the System.Windows.Media.Brush that is used to paint the foreground of cells in edit mode.
FrozenLinesBrush	Gets or sets the System.Windows.Media.Brush that is used to paint the lines between frozen and scrollable areas of the grid.
GridLinesBrush	Gets or sets the System.Windows.Media.Brush that is used to paint the lines between cells.
GroupRowBackground	Gets or sets the System.Windows.Media.Brush that is used to paint the background of group rows.
GroupRowForeground	Gets or sets the System.Windows.Media.Brush that is used to paint the foreground of group rows.
HeaderGridLinesBrush	Gets or sets the System.Windows.Media.Brush that is used to paint the lines between row and column header cells.
RowBackground	Gets or sets the System.Windows.Media.Brush that is used to paint row backgrounds.
RowHeaderBackground	Gets or sets the System.Windows.Media.Brush that is used to paint row header

	backgrounds.
RowHeaderForeground	Gets or sets the System.Windows.Media.Brush that is used to paint row header content.
RowHeaderSelectedBackground	Gets or sets the System.Windows.Media.Brush that is used to paint row header backgrounds for selected cells.
SelectionBackground	Gets or sets the System.Windows.Media.Brush that is used to paint the background of selected cells excluding the cursor cell.
SelectionForeground	Gets or sets the System.Windows.Media.Brush that is used to paint the foreground of selected cells excluding the cursor cell.
TopLeftCellBackground	Gets or sets the System.Windows.Media.Brush that is used to paint the background of the cell at the left top corner of the grid.

Note that the reference links to the properties described above are referring to WPF version; for Silverlight version, please refer to the member with same name in Silverlight namespace.

You can completely change the appearance of the C1FlexGrid control by setting one or more properties, For example, if you set the **AlternatingRowBackground** property to "#FFC3F2F2", the C1FlexGrid control appears similar to the following:



Line	Color	Name
Washers	White	Washer W0
Stoves	Red	Stove S1
Stoves	Green	Stove S2
Stoves	Red	Stove S3

FlexGrid Samples

With the C1Studio installer, you get FlexGrid samples that help you understand the implementation of the product. The C# and VB samples are available at the default installation folder:

Documents\ComponentOne Samples\WPF\C1.WPF.FlexGrid\

The C# samples are available in **CS** folder at the default installation location.

Sample	Description
CheckBoxes	This sample demonstrates how you can use a custom CellFactory to display check boxes in unbound grids.
CollectionViewFilter	This sample demonstrates an easy way to create ICollectionView filters based on string expressions.
ColumnFilter	This sample demonstrates the use of C1FlexGridFilter extension class to implement column filters in FlexGrid.
ColumnFooters	This sample demonstrates adding column footer cells that remain visible when the user scrolls FlexGrid vertically.
ColumnPicker	This sample demonstrates how to implement a column picker context menu in the FlexGrid.
ComboBox	This sample demonstrates how to bind a FlexGrid column using a CellEditingTemplate with a ComboBox.
ConditionalFormatting	This sample demonstrates how to implement conditional formatting in FlexGrid.
CustomColumns	This sample demonstrates how to combine custom columns with regular bound columns.
CustomMerging	This sample demonstrates how to implement merging logic to create a TV-guide display.
CustomTypeDescriptor	This sample demonstrates support for ICustomTypeDescriptor interface in FlexGrid.
DataAttributes	This sample demonstrates how FlexGrid supports data attributes defined in the **System.ComponentModel.DataAnnotations** namespace.
DragCells	This sample demonstrates performing cell drag and drop operation within FlexGrid.
DynamicConditionalFormatting	This sample demonstrates applying conditional formatting to a grid based on dynamic value ranges.
DynamicImages	This sample demonstrates how to use custom cell factory to implement cells with dynamic images.
ExcelBook	This sample demonstrates various Excel-like features in FlexGrid.
ExcelDragDrop	This sample demonstrates Excel-style drag-and-drop feature in FlexGrid.
ExcelExport	This sample demonstrates exporting FlexGrid to Excel using C1Excel.
ExcelGrid	This sample demonstrates how to achieve an Excel-like look and feel in FlexGrid.

ExcelStyleMerge	This sample demonstrates Excel-style merging with a custom CellFactory.
ExtendLastColumn	This sample demonstrates how to implement ExtendLastCol functionality in the FlexGrid.
FilterRow	This sample demonstrates adding a FilterRow to FlexGrid.
Financial	This sample demonstrates how to display a simulated live data feed of stock quotes for over 3,000 companies.
FlexGridLocalization	This sample demonstrates localizing FlexGrid filter to different languages.
FlexGridSamples	This sample demonstrates specific features such as grouping, filtering, editing, unbound mode, etc. in FlexGrid.
GapAnalysis	This sample demonstrates hierarchical, form-style and traditional features in FlexGrid.
GridToolTips	This sample demonstrates how to use hit testing to add cell-specific tooltips in FlexGrid.
GridTreeView	This sample demonstrates how to implement a bound hierarchical TreeView in FlexGrid.
GroupAggregates	This sample demonstrates how to display dynamic aggregates in group rows.
GroupPanel	This sample demonstrates custom grouping in FlexGrid through FlexGridGroupPanel control.
HeaderTooltips	This sample demonstrates how to attach tooltips to column headers.
HitTestTemplate	This sample demonstrates usage of the HitTest method which gets information about the grid at any given location.
ImageColumnTemplate	This sample demonstrates how to display images in a bound grid column.
Invalidate	This sample demonstrates how to use the Invalidate method to update cells dynamically.
LabResults	This sample demonstrates how to build grouped, filtered views of data in an MVVM pattern.
MultiColumnCombo	This sample demonstrates how to implement a multi-column combo box using FlexGrid.
MultiGridPdf	This sample demonstrates creating a PDF document that contains multiple FlexGrid controls.
MultiGridPrinting	This sample demonstrates printing multiple grids into a single document.
OData	This sample demonstrates using OData data sources in WPF applications.
PrintingWPF	This sample demonstrates printing features of FlexGrid control.
SalesAnalysis	This sample demonstrates using FlexGrid and Chart controls in a WPF business application.
ScrollBarStyles	This sample demonstrates customizing the appearance of scrollbars within FlexGrid.
TraditionalIndexing	This sample demonstrates how to emulate traditional indexing method used in FlexGrid for WinForms.

TreeFactory	This sample demonstrates creating a Tree with connector lines using a custom cell factory.
UnboundCellFactory	This sample demonstrates using a custom CellFactory to create custom cells in an unbound FlexGrid.
UnboundCellFactoryWPF	This sample demonstrates creating custom cells in an unbound FlexGrid.
UnboundConditionalformatting	This sample demonstrates implementing conditional formatting in unbound grids.
Validation	This sample demonstrates how FlexGrid supports validation when the ShowErrors property is set to true.
VerticalHeaders	This sample demonstrates how to use CellFactory to render column headers in vertical direction.

The Visual Basic samples are available in **VB** folder at the default installation location.

Samples	**Description**
CheckBoxes	This sample demonstrates how you can use a custom CellFactory to display check boxes in unbound grids.
CollectionViewFilter	This sample demonstrates an easy way to create ICollectionView filters based on string expressions.
CustomTypeDescriptor	This sample demonstrates support for ICustomTypeDescriptor interface in FlexGrid.
ExcelGrid	This sample demonstrates how to achieve an Excel-like look and feel in FlexGrid.
FilterRow	This sample demonstrates adding a FilterRow to FlexGrid.
Financial	This sample demonstrates how to display a simulated live data feed of stock quotes for over 3,000 companies.
ImageColumnTemplate	This sample demonstrates how to display images in a bound grid column.
Invalidate	This sample demonstrates how to use the Invalidate method to update cells dynamically.
PrintingWPF	This sample demonstrates printing features of FlexGrid control.
UnboundCellFactoryWPF	This sample demonstrates creating custom cells in an unbound FlexGrid.

The following sections walk you through two FlexGrid samples namely **iTunes** and **Financial**. The iTunes sample shows implementing custom cells to change the look and feel of the grid. This sample also demonstrate how to customize group rows to display subtotals on each cell, and how to implement a search box using the ICollectionView filter feature. The Financial sample shows a simulated live data feed of stock quotes for over 3,000 companies.

- iTunes Sample
- Financial Sample

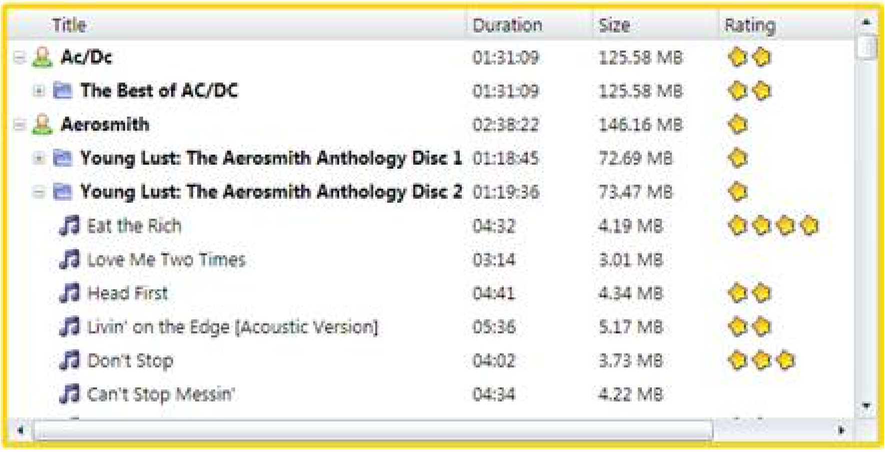
iTunes Sample

The iTunes sample application displays a library of about 10,000 songs grouped by artist and album. Albums and artists are represented by collapsible node rows. The application includes a search box that allows users to find songs, albums, or artists quickly and easily. Whenever a filter is applied, a status indicator lists the number of artists, albums, and songs selected, as well as the total size and duration for the selection.

Each item on the grid represents an artist, album, or song and includes a title, duration, size, and rating. The rating data (an integer between zero and five) is displayed graphically, using a series of zero to five stars. Group rows use the traditional plus and minus icons for collapsing and expanding buttons (instead of the standard triangular icons). The duration and size displayed for artists and albums are calculated dynamically, by adding all the songs in the corresponding group (album or artist). The rating for artists and albums is calculated as the average rating for the songs in the corresponding group (album or artist).

There are buttons above the grid that allow users to collapse all group rows on the grid to display only the artists, albums, or to expand the entire grid.

This is what the iTunes application looks like:



Title	Duration	Size	Rating
⊟ Ac/Dc	01:31:09	125.58 MB	★★
⊞ The Best of AC/DC	01:31:09	125.58 MB	★★
⊟ Aerosmith	02:38:22	146.16 MB	★
⊞ Young Lust: The Aerosmith Anthology Disc 1	01:18:45	72.69 MB	★
⊟ Young Lust: The Aerosmith Anthology Disc 2	01:19:36	73.47 MB	★
♫ Eat the Rich	04:32	4.19 MB	★★★★
♫ Love Me Two Times	03:14	3.01 MB	
♫ Head First	04:41	4.34 MB	★★
♫ Livin' on the Edge [Acoustic Version]	05:36	5.17 MB	★★
♫ Don't Stop	04:02	3.73 MB	★★★
♫ Can't Stop Messin'	04:34	4.22 MB	

Creating Grid

Here is the XAML that defines the **C1FlexGrid**.

XAML

```
<!-- show songs in a FlexGrid -->
<fg:C1FlexGrid x:Name="_flexiTunes" Grid.Row="1"
  AreRowGroupHeadersFrozen="False"
  HeadersVisibility="Column"
  GridLinesVisibility="None"
  Background="White"
  RowBackground="White"
  AlternatingRowBackground="White"
  GroupRowBackground="White"
  MinColumnWidth="30"
  SelectionBackground="#a0eaeff4"
  CursorBackground="#ffeaeff4"
  AutoGenerateColumns="False" >
  <fg:C1FlexGrid.Columns>
    <fg:Column Binding="{Binding Name}" Header="Title"
      AllowDragging="False" Width="300" />
    <fg:Column Binding="{Binding Duration}"
      HorizontalAlignment ="Right" />
    <fg:Column Binding="{Binding Size}"
```

```
      HorizontalAlignment ="Right" />
    <fg:Column Binding="{Binding Rating}"
      HorizontalAlignment="Center" Width="200" />
  </fg:C1FlexGrid.Columns>
</fg:C1FlexGrid>
```

The XAML sets a few style-related properties. Notice how the grid exposes several properties that allow you to customize its appearance considerably without resorting to custom cells or XAML templates.

Notice also that the code sets the **AutoGenerateColumns** property to false and defines the columns explicitly. This gives us a little extra control over the columns appearance and behavior.

Finally, notice that the XAML code sets the **AreRowGroupHeadersFrozen** property to false. Normally, group rows do not scroll horizontally. This allows users to see the group information at all times. But our iTunes application will display information on all columns of the group rows, so we do want them to scroll horizontally like all other rows. Setting the**AreRowGroupHeadersFrozen** property to false achieves this.

Reading Data

The data in the grid was originally copied from an actual iTunes library, and then serialized to an XML file which is included in the project as a resource. We load the data into an ICollectionView object with code that looks like this:

C#

```
// load the data
List <Song> songs = LoadSongs();

// create ICollectionView
var view = new PagedCollectionView(songs);

// group songs by album and artist
using (view.DeferRefresh())
{
  view.GroupDescriptions.Clear();
  view.GroupDescriptions.Add(new PropertyGroupDescription("Artist"));
  view.GroupDescriptions.Add(new PropertyGroupDescription("Album"));
}

// use converters to format song duration and size
fg.Columns["Duration"].ValueConverter = new SongDurationConverter();
fg.Columns["Size"].ValueConverter = new SongSizeConverter();

// bind data to grid
fg.ItemsSource = view;
```

The code uses a **LoadSongs** helper method that loads the songs from the XML file stored as a resource. In a real application, you can replace this with a web service that loads the song catalog from a server. Here is the implementation of the **LoadSongs** method:

C#

```
public static List<Song> LoadSongs()
{
  // find assembly resource
  var asm = Assembly.GetExecutingAssembly();
  foreach (var res in asm.GetManifestResourceNames())
  {
    if (res.EndsWith("songs.xml"))
    {
      using (var stream = asm.GetManifestResourceStream(res))
      {
        // load song catalog using an XmlSerializer
        var xmls = new XmlSerializer(typeof(List<Song>));
        return (List<Song>)xmls.Deserialize(stream);
      }
    }
  }
  return null;
}
```

The **Song** class stores song durations in milliseconds and sizes in bytes. This is not a convenient format to show to users, and there is not a simple .NET format string that converts the values to what we want. Instead of setting the **Column.Format** property, the code above sets the **Column.ValueConverter** property for the **Duration** and **Size** columns instead.

The **Column.ValueConverter** property specifies an IValueConverter object used to convert raw data values into display values for the column. Here is our sample implementation of value converters for song durations (expressed in milliseconds) and size (expressed in bytes):

C#

```
// converter for song durations (stored in milliseconds)
class SongDurationConverter : IValueConverter
{
  public object Convert(object value, Type targetType,
    object parameter,
    System.Globalization.CultureInfo culture)
  {
    var ts = TimeSpan.FromMilliseconds((long)value);
    return ts.Hours == 0
      ? string.Format("{0:00}:{1:00}", ts.Minutes, ts.Seconds)
      : string.Format("{0:00}:{1:00}:{2:00}",
             ts.Hours, ts.Minutes, ts.Seconds);
  }
  public object ConvertBack(object value, Type targetType,
    object parameter,
    System.Globalization.CultureInfo culture)
  {
    throw new NotImplementedException();
  }
}

// converter for song sizes (returns x.xx MB)
class SongSizeConverter : IValueConverter
{
  public object Convert(object value, Type targetType,
    object parameter,
    System.Globalization.CultureInfo culture)
  {
    return string.Format("{0:n2} MB", (long)value / 1024.0 / 1024.0);
  }
  public object ConvertBack(object value, Type targetType,
    object parameter,
    System.Globalization.CultureInfo culture)
  {
    throw new NotImplementedException();
  }
}
```

The first value converter uses distinct formats for durations that last over an hour; the second divides the byte counts to get megabytes. **IValueConverter** objects are flexible, simple, and very common in Silverlight and WPF programming.

That's all we have to do as far as data binding. We now have a proper **ICollectionView** data source and columns that show song name, duration, and size using a convenient format.

Grouping

The data binding code already took care of the basic grouping functionality by populating the **GroupDescriptions** property on our data source. This is enough for the grid to group the songs by album and artist and present collapsible group rows with some basic information about the group.

Let's take the grouping functionality a step further by adding buttons that automatically collapse or expand the catalog to show only artists (fully collapsed groups), artists and albums (intermediate state), or artists, albums and songs (fully expanded groups).

To accomplish this, we add three buttons above the grid: *Artists*, *Albums*, and *Songs*. The event handlers for these buttons are implemented as follows:

C#

```
// collapse/expand groups
void _btnShowArtists_Click(object sender, RoutedEventArgs e)
{
  ShowOutline(0);
}
void _btnShowAlbums_Click(object sender, RoutedEventArgs e)
{
  ShowOutline(1);
}
void _btnShowSongs_Click(object sender, RoutedEventArgs e)
{
  ShowOutline(int.MaxValue);
}
```

All event handlers use the same **ShowOutline** helper method. The first collapses all level zero group rows (artists); the second expands level zero groups (artists) and collapses level one (albums); the last expands all group rows. Here is the implementation of the **ShowOutline** method:

C#

```
void ShowOutline(int level)
{
  var rows = _flexiTunes.Rows;
  using (rows.DeferNotifications())
  {
    foreach (var gr in rows.OfType<GroupRow>())
    {
      gr.IsCollapsed = gr.Level >= level;
    }
  }
}
```

The method starts by retrieving the grid's **RowCollection** class and implements the same **DeferNotifications** mechanism used by the ICollectionView interface. This mechanism is similar to the **BeginUpdate** and **EndUpdate** pattern common in WinForms applications, and improves performance significantly.

Next, the code uses the LINQ **OfType** operator to retrieve all **GroupRow** objects from the **Rows** collection. This automatically excludes regular rows and allows us to check the level of every group row and update its **IsCollapsed** state based on the level of each group row.

Searching and Filtering

Like any song catalog, the one in the sample includes thousands of songs. They are grouped by artist and by album, but scrolling to find a particular song would be impractical.

The sample deals with this by implementing a search box similar to the one used in the real iTunes application. The user types a string into the search box, and the content is automatically filtered to show only songs, artists, or albums with titles that contain the specified string. Here is a brief description of what the **SearchBox** control looks like and how it works:

The control has two properties:

- **View**: an ICollectionView object that contains the data to be filtered.
- **FilterProperties**: A list of PropertyInfo objects that specify which properties in the data items to use when applying the filter.

When the user types into the **SearchBox control**, a timer starts ticking. When he stops typing for a short while (800ms), the filter is applied to the data source. The filter is not applied immediately after each character is typed because that would be inefficient and annoying to the user.

The timer class used to handle this is defined in the **C1FlexGrid** assembly. The **C1.Util.Timer** class is a utility that works in Silverlight and in WPF, making it a little easier to develop applications that share code and run under both platforms.

To apply the filter, the **SearchBox** control sets the **View.Filter** property to a method that checks each data item and keeps only those where at least one of the properties specified by the **FilterProperties** member contains the search string typed by the user.

This is what the code looks like:

C#

```
void _timer_Tick(object sender, EventArgs e)
 {
   // stop the timer
   _timer.Stop();

   // check that we have a filter to apply
   if (View != null && _propertyInfo.Count > 0)
   {
     // apply the filter
    View.Filter = null;
     View.Filter = (object item) =>
     {
       // get search text
       var srch = _txtSearch.Text;

       // no text? show all items
       if (string.IsNullOrEmpty(srch))
       {
         return true;
       }

       // show items that contain the text in any
       // of the specified properties
       foreach (PropertyInfo pi in _propertyInfo)
       {
        var value = pi.GetValue(item, null) as string;
         if (value != null &&
         value.IndexOf(srch, StringComparison.OrdinalIgnoreCase) > -1)
         {
           return true;
         }
       }

       // exclude this item...
       return false;
     };
   }
 }
```

The interesting part of the code is the block that applies the filter. It uses a lambda function instead of a regular method, but the effect is the same. The lambda function uses reflection to retrieve the value of each property specified and checks whether it contains the search string. If a match is found, the item is kept in view. Otherwise, it is filtered out.

To connect the **SearchBox** control to an application, set the **View** and **FilterProperties** properties. In our sample, we set the **View** property to our song data source and add the *Artist*, *Album*, and *Name* properties to the **FilterProperties** collection so the user can search for any of these elements. The code that connects the **SearchBox** to the application looks like this:

C#

```
// configure search box
 _srchTunes.View = view;
 foreach (string name in "Artist|Album|Name".Split('|'))
 {
   _srchTunes.FilterProperties.Add(typeof(Song).GetProperty(name));
 }
```

Custom Cells

We are now ready to create an **ICellFactory** object that creates custom cells to display the following elements:

- Images next to artists, albums, and songs;
- Custom images for collapsing and expanding groups;
- Custom images to display song, album, and artist ratings.

We accomplish all of these tasks using a custom **MusicCellFactory** class that implements the **ICellFactory** interface. To use the custom cell factory, we assign it to the grid's **CellFactory** property:

C#

```
_flexiTunes.CellFactory = new MusicCellFactory()
```

The **MusicCellFactory** class inherits from the default **CellFactory** class and overrides the **CreateCellContent** method to create the elements used to represent the content in each cell.

The **CreateCellContent** method takes as parameters the parent grid, a **Border** element that is created by the base class and provides the cell's background and borders, and a **CellRange** object that specifies the cell to create (this may be a single cell or multiple cells if they are merged).

Note that **CreateCellContent** only creates the cell content. If we want to take over the creation of the entire cell (including border and background), we override the **CreateCell** method instead.

In our example, the **CreateCellContent** method handles two main cases: regular cells and cells in group rows.

To handle regular cells, the method returns new **SongCell** or **RatingCell** elements depending on the column being created. These are custom elements that derive from **StackPanel** and contain images and text which are bound to the data item represented by the cell.

Cells in group rows are a little more complicated. In our application, group rows represent artists and albums. These items do not correspond to data items in the source collection. Our **CreateCellContent** method deals with this by creating fake **Song** objects to represent artists and albums. These fake **Song** objects have properties calculated based on the songs contained in the group. The **Duration** of an album is calculated as the sum of the **Duration** of each song in the album, and the **Rating** is calculated as the average **Rating** of the songs in the album. Once this fake **Song** object is created, we can use the **SongCell** and **RatingCell** elements as usual.

Here is a slightly simplified version of the **MusicCellFactory** class and its **CreateCellContent** implementation:

C#

```
// cell factory used to create iTunes cells
 public class MusicCellFactory : CellFactory
 {
   // override this to create the cell contents (the base class already
   // created a Border element with the right background color)
   public override void CreateCellContent(
          C1FlexGrid grid, Border bdr, CellRange range)
   {
     // get row
     var row = grid.Rows[range.Row];
     var gr = row as GroupRow;
```

```
    // special handling for cells in group rows
    if (gr != null && range.Column == 0)
    {
      BindGroupRowCell(grid, bdr, range);
      return;
    }

    // create a SongCell to show artist, album, and song names
    var colName = grid.Columns[range.Column].ID;
    if (colName == "Name")
    {
      bdr.Child = new SongCell(row);
      return;
    }

    // create a RatingCell to show artist, album, and song ratings
    if (colName == "Rating")
    {
      var song = row.DataItem as Song;
      if (song != null)
      {
        bdr.Child = new RatingCell(song.Rating);
        return;
      }
    }

    // use default binding for everything else
    base.CreateCellContent(grid, bdr, range);
  }
```

The **BindGroupRowCell** method creates the fake **Song** objects mentioned earlier, assigns them to the row's **DataItem** property so they can be used by the **CreateCellContent** method, and provides special handling for the first cell in the group row. The first cell in each group row is special because it contains the group's collapse and expand button in addition to the regular cell content.

Here is the code that handles the first item in each group row:

C#

```
// bind cells in group rows
 void BindGroupRowCell(C1FlexGrid grid, Border bdr, CellRange range)
 {
   // get row, group row
   var row = grid.Rows[range.Row];
   var gr = row as GroupRow;

   // first cell in the row contains custom collapse/expand button
   // and an artist, album, or song image
   if (range.Column == 0)
   {
     // build fake Song object to represent group
     if (gr.DataItem == null)
     {
       gr.DataItem = BuildFakeSong(gr);
     }

     // create the first cell in the group row
     bdr.Child = gr.Level == 0
       ? (ImageCell)new ArtistCell(row)
       : (ImageCell)new AlbumCell(row);
   }
 }
```

Finally, here is the code that creates the fake **Song** objects that represent artists and albums. The method uses the **GroupRow.GetDataItems** method to retrieve a list of all data items contained in the group. It then uses a LINQ statement to calculate the total size, duration, and average rating of the songs in the group.

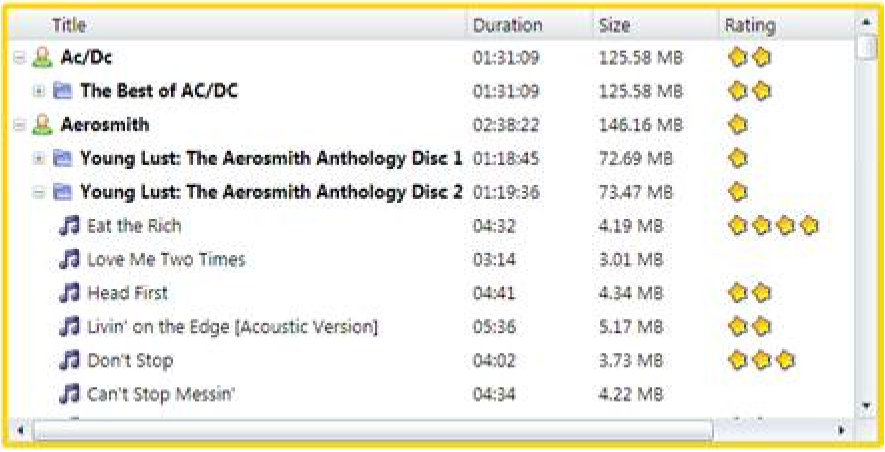
C#

```
// build fake Song objects to represent groups (artist or album)
 Song BuildFakeSong(GroupRow gr)
 {
   var gs = gr.GetDataItems().OfType<Song>();
   return new Song()
   {
     Name = gr.Group.Name.ToString(),
     Size = (long)gs.Sum(s => s.Size),
     Duration = (long)gs.Sum(s => s.Duration),
     Rating = (int)(gs.Average(s => s.Rating) + 0.5)
   };
 }
```

That is most of the work required. The only part left is the definition of the custom elements used to represent individual cells. The elements are:

- **SongCell**: Shows a song icon and the song name.
- **ArtistCell**: Shows a collapse/expand icon, an artist icon, and the artist name.
- **AlbumCell**: Shows a collapse/expand icon, an album icon, and the artist name.
- **RatingCell**: Shows a rating (integer between zero and five) as a graphical element, one star for each rating point.

The image below shows how the elements appear in the grid:



Title	Duration	Size	Rating
Ac/Dc	01:31:09	125.58 MB	★★
The Best of AC/DC	01:31:09	125.58 MB	★★
Aerosmith	02:38:22	146.16 MB	★
Young Lust: The Aerosmith Anthology Disc 1	01:18:45	72.69 MB	★
Young Lust: The Aerosmith Anthology Disc 2	01:19:36	73.47 MB	★
Eat the Rich	04:32	4.19 MB	★★★★
Love Me Two Times	03:14	3.01 MB	
Head First	04:41	4.34 MB	★★
Livin' on the Edge [Acoustic Version]	05:36	5.17 MB	★★
Don't Stop	04:02	3.73 MB	★★★
Can't Stop Messin'	04:34	4.22 MB	

These are all regular Silverlight and WPF Framework elements. They can be created with Microsoft Blend or in code.

The code below shows the implementation of the **RatingCell** element. The other elements are similar; you can review the implementation details by looking at the sample source code.

C#

```
///
 /// Cell that shows a rating value as an image with stars.
 ///
 public class RatingCell : StackPanel
{
   static ImageSource _star;

   public RatingCell(int rating)
   {
     if (_star == null)
     {
       _star = ImageCell.GetImageSource("star.png");
     }
     Orientation = Orientation.Horizontal;
     for (int i = 0; i < rating; i++)
     {
       Children.Add(GetStarImage());
     }
   }
   Image GetStarImage()
   {
     var img = new Image();
     img.Source = _star;
     img.Width = img.Height = 17;
     img.Stretch = Stretch.None;
     return img;
   }
}
```

The **RatingCell** element is very simple. It consists of a **StackPanel** with some **Image** elements in it. The number of **Image** elements is defined by the rating being represented, a value passed to the constructor. Each **Image** element displays a star icon. This is a static arrangement that assumes that the ratings do not change, so we do not need any dynamic bindings.

Financial Sample

This section describes a sample application created on a scenario suitable for financial industry. Financial applications typically rely on vast amounts of data, often obtained in real time from powerful, dedicated servers.

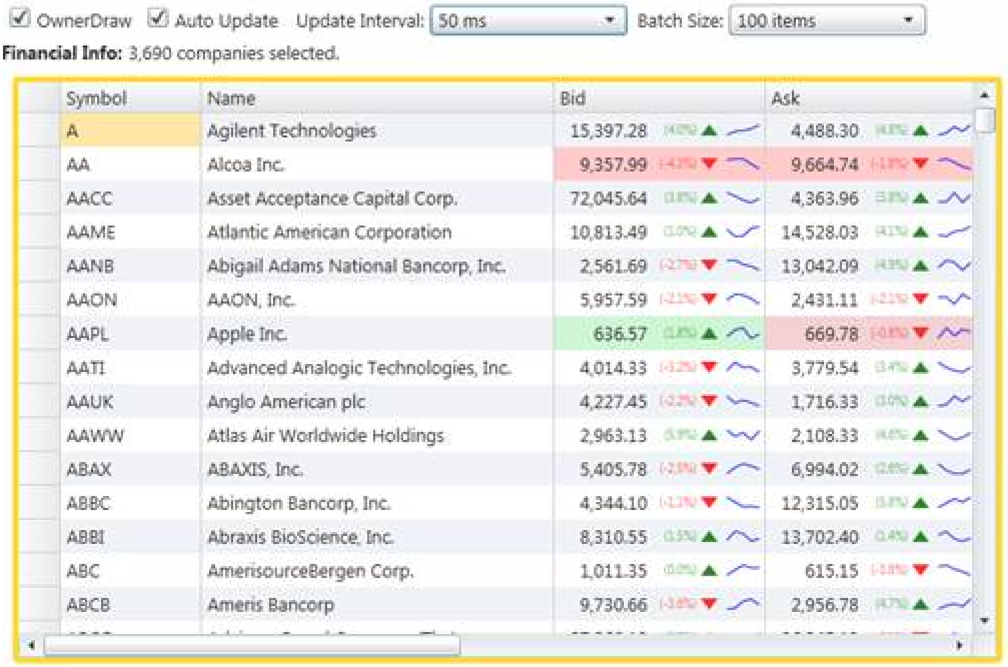
The real-time nature of the information in these applications requires fast updates (the application has to keep up with the data stream coming from the server) and mechanisms to convey the changes to users in a clear, efficient manner.

One common approach used to highlight changes in real time is the use of flashing elements. For example, a grid cell may flash in a different color when its value changes. The flash only lasts for a short time, enough to call the user’s attention to the change.

Another increasingly popular mechanism used to convey rich information in a quick and compact form is the *sparkline*. A sparkline is a mini-chart that shows trends and summary information more clearly and efficiently than long rows of numbers.

This section describes a sample financial application with real-time data updates, flashing cells, and sparklines. The sample highlights the performance that the **C1FlexGrid** provides both in Silverlight and in WPF.

The image below shows our sample financial application in action. The image cannot convey the dynamic nature of the application, which shows values changing constantly, so we suggest that you run the samples when you have a chance.



☑ OwnerDraw ☑ Auto Update Update Interval: 50 ms Batch Size: 100 items

Financial Info: 3,690 companies selected.

Symbol	Name	Bid		Ask	
A	Agilent Technologies	15,397.28	▲	4,488.30	▲
AA	Alcoa Inc.	9,357.99	▼	9,664.74	▼
AACC	Asset Acceptance Capital Corp.	72,045.64	▲	4,363.96	▲
AAME	Atlantic American Corporation	10,813.49	▲	14,528.03	▲
AANB	Abigail Adams National Bancorp, Inc.	2,561.69	▼	13,042.09	▲
AAON	AAON, Inc.	5,957.59	▼	2,431.11	▼
AAPL	Apple Inc.	636.57	▲	669.78	▼
AATI	Advanced Analogic Technologies, Inc.	4,014.33	▼	3,779.54	▲
AAUK	Anglo American plc	4,227.45	▼	1,716.33	▲
AAWW	Atlas Air Worldwide Holdings	2,963.13	▲	2,108.33	▲
ABAX	ABAXIS, Inc.	5,405.78	▼	6,994.02	▲
ABBC	Abington Bancorp, Inc.	4,344.10	▼	12,315.05	▲
ABBI	Abraxis BioScience, Inc.	8,310.55	▲	13,702.40	▲
ABC	AmerisourceBergen Corp.	1,011.35	▲	615.15	▼
ABCB	Ameris Bancorp	9,730.66	▼	2,956.78	▲

Generating the data

Our financial application uses a data source that simulates an actual server providing dynamic data with constant updates.

Not being finance professionals ourselves, we got some inspiration from WikiPedia (http://en.wikipedia.org/wiki/Market_data).

Our data source consists of **FinancialData** objects that represent typical equity market data message or business objects furnished from NYSE, TSX, or Nasdaq. Each **FinancialData** object contains information such as this:

Ticker Symbol	IBM
Bid	89.02
Ask	89.08
Bid size	300
Ask size	1000
Last sale	89.06
Last size	200
Quote time	14:32:45
Trade time	14.32.44
Volume	7808

In actuality, this information is usually an aggregation of different sources of data, as quote data (bid, ask, bid size, ask size) and trade data (last sale, last size, volume) are often generated over different data feeds.

To capture the dynamic nature of the data, our data source object provides a **FinancialDataList** with about 4,000 **FinancialData** objects and a timer that modifies the objects according to a given schedule. The caller can determine how often the values update and how many update at a time.

Binding the **FinancialDataList** to a grid and changing the update parameters while the program runs allows us to check how well the grid performs by keeping up with the data updates.

To check the details of the data source implementation, please see the **FinancialData.cs** file in the sample source.

Binding the grid to the financial data source is straightforward. Instead of binding the grid directly to our **FinancialDataList**, we create a PagedCollectionView to serve as a broker and provide the usual currency, sorting, grouping, and filtering services for us. Here is the code:

C#

```
// create data source
var list = FinancialData.GetFinancialData();
var view = new PagedCollectionView(list);

// bind data source to the grid
_flexFinancial.ItemsSource = view;
```

As in the previous sample, we set the **AutoGenerateColumns** property to false and use XAML to create the grid columns:

XAML

```
<fg:C1FlexGrid x:Name="_flexFinancial"
  MinColumnWidth="10"
  MaxColumnWidth="300"
  AutoGenerateColumns="False" >
  <fg:C1FlexGrid.Columns>
    <fg:Column Binding="{Binding Symbol}"    Width="100" />
    <fg:Column Binding="{Binding Name}"      Width="250" />
    <fg:Column Binding="{Binding Bid}"       Width="150"
        Format="n2" HorizontalAlignment="Right" />
    <fg:Column Binding="{Binding Ask}"       Width="150"
        Format="n2" HorizontalAlignment="Right" />
    <fg:Column Binding="{Binding LastSale}"  Width="150"
        Format="n2" HorizontalAlignment="Right" />
    <fg:Column Binding="{Binding BidSize}"   Width="100"
        Format="n0" HorizontalAlignment="Right" />
    <fg:Column Binding="{Binding AskSize}"   Width="100"
        Format="n0" HorizontalAlignment="Right" />
    <fg:Column Binding="{Binding LastSize}"  Width="100"
        Format="n0" HorizontalAlignment="Right" />
    <fg:Column Binding="{Binding Volume}"    Width="100"
        Format="n0" HorizontalAlignment="Right" />
    <fg:Column Binding="{Binding QuoteTime}" Width="100"
        Format="hh:mm:ss" HorizontalAlignment="Center" />
    <fg:Column Binding="{Binding TradeTime}" Width="100"
        Format="hh:mm:ss" HorizontalAlignment="Center" />
  </fg:C1FlexGrid.Columns>
</fg:C1FlexGrid>
```

Searching and Filtering

Much like iTunes users, financial analysts are probably not interested in looking at all of the records all of the time, so we need some filter or search mechanism.

A real application allows the analyst to pick specific sets of papers to look at, and probably to save these views and switch between them. Our sample takes a simpler approach by re-using the **SearchBox** control described in the iTunes Sample. Instead of selecting specific entries, our users type something like "bank" or "electric" in the search box to filter the data.

Hook up the **SearchBox** control to the financial data source with code like this:

C#

```
// create data source
FinancialDataList list = FinancialData.GetFinancialData();
var view = new PagedCollectionView(list);

// bind data source to the grid
_flexFinancial.ItemsSource = view;

// hook up the SearchBox (user can search by company name or symbol)
_srchBox.View = view;
var props = _srchCompanies.FilterProperties;
props.Add(typeof(FinancialData).GetProperty("Name"));
props.Add(typeof(FinancialData).GetProperty("Symbol"));
```

Custom Cells

If you run the sample now, the grid already works and updates the data as expected. You can change the update parameters to make them more or less frequent, scroll the grid while it updates, and so on.

However, although the updates are happening, they are hard to understand. There are just too many numbers flashing at random positions on the screen.

Custom cells provide a better user experience with flashes and sparklines.

Flashes temporarily change the cell background when the value they contain changes. If a value increases, the cell instantly turns green and then gradually fades back to white.

Sparklines are micro-charts displayed in each cell. The micro-chart shows the last five values of the cell so users can instantly identify trends (the value is going up, down, or stable).

To use custom cells, we proceed as in the previous sample. Start by creating a **FinancialCellFactory** class and assign an instance of that class to the grid's **CellFactory** property:

C#

```
// use custom cell factory
_flexFinancial.CellFactory = new FinancialCellFactory();

// custom cell factory definition
public class FinancialCellFactory : CellFactory
```

```
{
  public override void CreateCellContent(
               C1FlexGrid grid, Border bdr, CellRange range)
  {
    // get cell information
    var r = grid.Rows[range.Row];
    var c = grid.Columns[range.Column];
    var pi = c.PropertyInfo;

    // check that this is a cell we want
    if (r.DataItem is FinancialData &&
      (pi.Name == "LastSale" || pi.Name == "Bid" || pi.Name == "Ask"))
    {
      // create StockTicker element and add it to the cell
      var ticker = new StockTicker();
      bdr.Child = ticker;

      // bind StockTicker to the row’s FinancialData object
      var binding = new Binding(pi.Name);
      binding.Source = r.DataItem;
      binding.Mode = BindingMode.OneWay;
      ticker.SetBinding(StockTicker.ValueProperty, binding);

      // add some info to the StockTicker element
      ticker.Tag = r.DataItem;
      ticker.BindingSource = pi.Name;
    }
    else
    {
      // use default implementation
      base.CreateCellContent(grid, bdr, range);
    }
  }
}
```

Our custom cell factory starts by checking that the row data is of type **FinancialData** and that the column is bound to the **LastSale**, **Bid**, or **Ask** properties of the data object. If all these conditions are met, the cell factory creates a new **StockTicker** element and binds it to the data.

The **StockTicker** element displays the data to the user. It consists of a **Grid** element with four columns that contain the following child elements:

Element Description	Type	Name
Current value	TextBlock	_txtValue
Last percent change	TextBlock	_txtChange
Up/Down icon	Polygon	_arrow
Sparkline	Polyline	_sparkLine

These elements are defined in the **StockTicker.xaml** file, which we do not list here in its entirety.

The most interesting part of the **StockTicker.xaml** file is the definition of the **Storyboard** used to implement the flashing behavior. The **Storyboard** gradually changes the control **Background** from its current value to transparent:

XAML

```xml
<UserControl.Resources>
  <Storyboard x:Key="_sbFlash" >
    <ColorAnimation
      Storyboard.TargetName="_root"
      Storyboard.TargetProperty= "(Grid.Background).(SolidColorBrush.Color)"
      To="Transparent"
      Duration="0:0:1"
    />
  </Storyboard>
</UserControl.Resources>
```

The implementation of the **StockTicker** control is contained in the **StockTicker.cs** file. The interesting parts are commented below:

C#

```csharp
/// <summary>
/// Interaction logic for StockTicker.xaml
/// </summary>
public partial class StockTicker : UserControl
{
  public static readonly DependencyProperty ValueProperty =
    DependencyProperty.Register(
      "Value",
      typeof(double),
      typeof(StockTicker),
      new PropertyMetadata(0.0, ValueChanged));
```

We start by defining a DependencyProperty called **ValueProperty** that will be used for binding the control to the underlying data value. The **Value** property is implemented as follows:

C#

```csharp
public double Value
{
  get { return (double)GetValue(ValueProperty); }
  set { SetValue(ValueProperty, value); }
}
private static void ValueChanged(DependencyObject d,
            DependencyPropertyChangedEventArgs e)
{
  var ticker = d as StockTicker;
  var value = (double)e.NewValue;
  var oldValue = (double)e.OldValue;
```

In order to implement the sparkline, the control needs access to a more than just the current and previous values. This is accomplished by storing the **FinancialData** object in the control’s **Tag** property, then calling the **FinancialData.GetHistory** method.

In fact, the previous value provided by the event’s **OldValue** property is not reliable in this case anyway. Because the grid virtualizes the cells, the **StockTicker.Value** property may have changed because the control was just created as its cell scrolled into view. In this case, the control had no previous value. Getting the previous values from the **FinancialData** object takes care of this problem also.

C#

```csharp
// get historical data
var data = ticker.Tag as FinancialData;
var list = data.GetHistory(ticker.BindingSource);
if (list != null && list.Count > 1)
{
  oldValue = (double)list[list.Count - 2];
}
```

Once the values are available, the control calculates the latest change as a percentage and updates the control text:

C#

```csharp
// calculate percentage change
var change = oldValue == 0 || double.IsNaN(oldValue)
  ? 0
  : (value - oldValue) / oldValue;

// update text
ticker._txtValue.Text = value.ToString(ticker._format);
```

The percent change is also used to update the up/down symbol and the text and flash colors. If there is no change, the up/down symbol is hidden and the text is set to the default color.

If the change is negative, the code makes the up/down symbol point down by setting its **ScaleY** transform to -1, and sets the color of the symbol, text, and flash animation to red.

If the change is positive, the code makes the up/down symbol point up by setting its **ScaleY** transform to +1, and sets the color of the symbol, text, and flash animation to green.

C#

```csharp
// update symbol and flash color
var ca = ticker._flash.Children[0] as ColorAnimation;
if (change == 0)
{
  ticker._arrow.Fill = null;
  ticker._txtChange.Foreground = ticker._txtValue.Foreground;
}
else if (change < 0)
{
  ticker._stArrow.ScaleY = -1;
  ticker._txtChange.Foreground = ticker._arrow.Fill = _brNegative;
  ca.From = _clrNegative;
}
else
{
  ticker._stArrow.ScaleY = +1;
  ticker._txtChange.Foreground = ticker._arrow.Fill = _brPositive;
  ca.From = _clrPositive;
  }
```

Next, the code updates the sparkline by populating the **Points** property of the sparkline polygon with the value history array provided by the early call to the **FinancialData.GetHistory** method. The sparkline polygon's **Stretch** property is set to **Fill**, so the line scales automatically to fit the space available.

C#

```
  // update sparkline
  if (list != null)
  {
    var points = ticker._sparkLine.Points;
    points.Clear();
    for (int x = 0; x < list.Count; x++)
    {
      points.Add(new Point(x, (double)list[x]));
    }
  }
```

Finally, if the value actually changes and the control has not just been created, the code flashes the cell by calling the **StoryBoard.Begin** method.

C#

```
  // flash new value (but not right after the control was created)
  if (!ticker._firstTime)
  {
    ticker._flash.Begin();
  }
  ticker._firstTime = false;
}
```

That concludes the **StockTicker** control. If you run the sample application now and check the "Custom Cells" checkbox, you immediately see a much more informative display, with cells flashing as their values change and sparklines providing a quick indication of value trends.