
ComponentOne

FlexGrid for Silverlight

Copyright © 1987-2012 GrapeCity, Inc. All rights reserved.

ComponentOne, a division of GrapeCity

201 South Highland Avenue, Third Floor

Pittsburgh, PA 15206 • USA

Internet: info@ComponentOne.com

Web site: <http://www.componentone.com>

Sales

E-mail: sales@componentone.com

Telephone: 1.800.858.2739 or 1.412.681.4343 (Pittsburgh, PA USA Office)

Trademarks

The ComponentOne product name is a trademark and ComponentOne is a registered trademark of GrapeCity, Inc. All other trademarks used herein are the properties of their respective owners.

Warranty

ComponentOne warrants that the original CD (or diskettes) are 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 CD (or disk) 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 a defective CD (or disk) by sending it and a check for \$25 (to cover postage and handling) to ComponentOne.

Except for the express warranty of the original CD (or disks) 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. We are 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.

This manual was produced using [ComponentOne Doc-To-Help™](#).

Table of Contents

ComponentOne FlexGrid for Silverlight Overview	1
Help with ComponentOne Studio for Silverlight.....	1
The C1.Silverlight.FlexGrid.dll Assembly	1
Studio for Silverlight Samples	1
FlexGrid for Silverlight Samples	1
XAML Quick Reference	3
Using FlexGrid for Silverlight	7
Overview	7
Creating the C1FlexGrid.....	7
Populating the grid.....	8
Grouping Data	9
Aggregating Data	12
Sorting Data	14
Filtering Data (using ICollectionView).....	15
Filtering Data (using C1FlexGridFilter)	16
Unbound Mode.....	19
Cell Merging.....	21
Multi-cell Row and Column Headers	22
Selection and Selection Modes.....	23
Monitoring the Selection	24
Selecting cells and objects using code.....	25
Customizing the Selection Display.....	27
Custom Cells.....	27
Custom Cells in code: CellFactory class	27
Custom Cells in XAML: CellTemplate and CellEditingTemplate	28
Editing Features	29
Auto-complete and mapped columns.....	29
Data-Mapped columns.....	30
Using Custom Editors	30
Configuring Editors	31
Frozen Rows and Columns.....	31

Printing Support	32
Basic Printing	32
Advanced Printing	32
Localization Sample	37
iTunes Sample	38
Creating the grid	39
Loading the Data	40
Grouping	41
Searching and Filtering	42
Custom Cells	44
Financial Application Sample	47
Generating the data	48
Searching and Filtering	49
Custom Cells	50
Performance	53
Differences in C1FlexGrid for Silverlight/WPF	53
Object Model comparison	54
C1FlexGrid.Methods	58
C1FlexGrid.Events	59
We want your feedback!	59

ComponentOne FlexGrid for Silverlight Overview

ComponentOne FlexGrid® for Silverlight is a DataGrid control with a lightweight, flexible object model. Modeled after the popular WinForms version, **FlexGrid for Silverlight** offers many unique features such as unbound mode, flexible cell merging, and multi-cell row and column headers. **FlexGrid** also includes a WPF version of the control.

Help with ComponentOne Studio for Silverlight

Getting Started

For information on installing ComponentOne Studio for Silverlight, licensing, technical support, namespaces and creating a project with the control, please visit [Getting Started with Studio for Silverlight](#).

What's New

For a list of the latest features added to **ComponentOne Studio for Silverlight**, visit [What's New in Studio for Silverlight](#).

The C1.Silverlight.FlexGrid.dll Assembly

The **C1.Silverlight.FlexGrid.x.dll** assembly includes **ComponentOne FlexGrid for Silverlight**, a DataGrid control with a lightweight, flexible object model.

Main Classes

The following main classes are included in the **C1.Silverlight.FlexGrid.x.dll** assembly:

- **C1FlexGrid**: Modeled after the popular WinForms version, **FlexGrid for Silverlight** offers many unique features such as unbound mode, flexible cell merging, and multi-cell row and column headers.

Studio for Silverlight Samples

If you just installed ComponentOne Studio for Silverlight, open Visual Studio and load the Samples.sln solution located in the C:\Documents and Settings\<username>\My Documents\ComponentOne Samples\Studio for Silverlight or C:\Users\<username>\Documents\ComponentOne Samples\Studio for Silverlight folder. This solution contains all the samples that ship with this release. Each sample has a readme.txt file that describes it and two projects named as follows:

<SampleName>	Silverlight project (client-side project)
<SampleName>Web	ASP.NET project that hosts the Silverlight project (server-side project)

To run a sample, right-click the **<SampleName>Web** project in the Solution Explorer, select **Set as Startup Project**, and press F5.

FlexGrid for Silverlight Samples

The following tables describe each of the **C1FlexGrid** samples included:

Visual Basic Samples

Sample	Description
CollectionViewFilter	Shows an easy way to create ICollectionView filters based on string expressions.
ColumnPicker	Shows how to implement a column picker context menu in the C1FlexGrid .
ConditionalFormatting	Shows how you can implement conditional formatting using a C1FlexGrid .
CustomColumns	Shows how you can combine custom columns with regular bound columns.
CustomMerging	This sample shows how to implement a custom IMergeManager that creates merged ranges with multiple rows and columns. The sample creates a TV schedule and merges programs across weekdays (columns) and show times (rows).
DragCells	Shows how to perform cell drag and drop operations within the grid.
ExcelDragDrop	Shows how you can implement Excel-style drag and drop features in the C1FlexGrid .
ExcelStyleMerge	This sample implements Excel-style merging using a custom MergeManager class that keeps a list of merged ranges. The application calls the merge manager's AddMergedRange and RemoveMergedRange to add or remove arbitrary merged ranges to the list.
GridTooltips	Shows how to use hit testing to add cell-specific tooltips.
GroupAggregates	Demonstrates how to display dynamic aggregates in group rows.
HitTestTemplate	Shows how to use the HitTest method to determine where the mouse is with respect to the grid.
MainTestApplication	Shows the main features in the C1FlexGrid control.
MultiGridPdf	Shows how to create a PDF document that contains multiple C1FlexGrid controls.
MultiGridPrinting	Shows how you can print multiple grids into a single document.
Printing	Demonstrates the printing features of the C1FlexGrid control.
TraditionalIndexing	Shows how you can emulate the traditional indexing method used in the C1FlexGrid for WinForms .
VerticalHeaders	Shows how to use a CellFactory to render column headers in the vertical direction.

C# Samples

Sample	Description
CollectionViewFilter	Shows an easy way to create ICollectionView filters based on string expressions.
ColumnFilter	Demonstrates the use of the C1FlexGridFilter extension class to implement column filters in a C1FlexGrid control.
ColumnPicker	Shows how to implement a column picker context menu in the C1FlexGrid .
ConditionalFormatting	Shows how you can implement conditional formatting using a C1FlexGrid .
CustomColumns	Shows how you can combine custom columns with regular bound columns.
CustomMerging	This sample shows how to implement a custom IMergeManager that creates merged ranges with multiple rows and columns. The sample creates a TV schedule and merges programs across weekdays (columns) and show times (rows).
DragCells	Shows how to perform cell drag and drop operations within the grid.
DynamicConditionalFormatting	Shows how to apply conditional formatting to a grid based on dynamic value ranges.
ExcelDragDrop	Shows how you can implement Excel-style drag and drop features in the C1FlexGrid .
ExcelGrid	Shows a class that extends the C1FlexGrid to provide Excel-like features.

ExcelStyleMerge	This sample implements Excel-style merging using a custom MergeManager class that keeps a list of merged ranges. The application calls the merge manager's AddMergedRange and RemoveMergedRange to add or remove arbitrary merged ranges to the list.
GridTooltips	Shows how to use hit testing to add cell-specific tooltips.
GroupAggregates	Demonstrates how to display dynamic aggregates in group rows.
HitTestTemplate	Shows how to use the HitTest method to determine where the mouse is with respect to the grid.
MainTestApplication	Shows the main features in the C1FlexGrid control.
MultiGridPdf	Shows how to create a PDF document that contains multiple C1FlexGrid controls.
MultiGridPrinting	Shows how you can print multiple grids into a single document.
OData	Demonstrates how to use OData data sources in Silverlight applications.
Printing	Demonstrates the printing features of the C1FlexGrid control.
SalesAnalysis	Demonstrates using a C1FlexGrid in a Silverlight RIA Services Business Application.
TraditionalIndexing	Shows how you can emulate the traditional indexing method used in the C1FlexGrid for WinForms .
UnboundConditionalFormatting	Shows how you can implement conditional formatting with unbound grids.
VerticalHeaders	Shows how to use a CellFactory to render column headers in the vertical direction.

XAML Quick Reference

This topic is dedicated to providing a quick overview of the XAML used to create a C1FlexGrid control.

To get started developing, add a **c1** namespace declaration in the root element tag:

```
xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml"
```

Here is an example of C1FlexGrid taken from the **ColumnPicker** sample:

C1FlexGrid

Right-click column headers to select which columns are displayed.
Right-click cells to cut/copy/paste/clear the selection.

Save Current Layout Load Saved Layout

Line	Color	Name	Price	Weight
Computers	Green	Computer C0	337.00	54.00
Washers	Blue	Washer W1	943.00	24.00
Washers	Green	Washer W2	5.00	39.00
Computers	Red	Computer C3	959.00	61.00
Washers	White	Washer W4	910.00	20.00
Computers	Green	Computer C5	220.00	83.00
Washers	White	Washer W6	968.00	31.00
Washers	White	Washer W7	260.00	20.00
Washers	Green	Washer W8	721.00	83.00
Washers	Blue	Washer W9	662.00	96.00
Washers	Blue	Washer W10	978.00	19.00
Computers	Red	Computer C11	772.00	70.00
Washers	Red	Washer W12	437.00	12.00
Computers	Green	Computer C13	866.00	22.00

Below is the XAML for the sample:

```
<UserControl x:Class="ColumnPicker.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    FontFamily="Segoe UI" FontSize="13"
    d:DesignHeight="300" d:DesignWidth="600">

    <Grid x:Name="LayoutRoot" Background="White">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition />
        </Grid.RowDefinitions>

        <StackPanel Orientation="Vertical">
            <TextBlock Text="C1FlexGrid" FontSize="14"
                FontWeight="Bold"/>
            <TextBlock Text="Right-click column headers to select which
                columns are displayed." />
            <TextBlock Text="Right-click cells to cut/copy/paste/clear
                the selection." />
            <StackPanel Orientation="Horizontal">
                <Button Content="Save Current Layout"
                    Click="SaveLayout_Click" Margin="4 0" Padding="6 2"/>
            </StackPanel>
        </StackPanel>
    </Grid>
</UserControl>
```



```
        <Button Content="Load Saved Layout"
Click="LoadLayout_Click" Margin="4 0" Padding="6 2"/>
    </StackPanel>
</StackPanel>
    <c1:C1FlexGrid Name="_flex" Grid.Row="1" />
</Grid>
</UserControl>
```


Using FlexGrid for Silverlight

This document introduces the **C1FlexGrid** control for Silverlight and WPF. The control was introduced in the V2/2010 release of the corresponding ComponentOne Studios and has been substantially improved since then.

Overview

ComponentOne has a powerful **DataGrid** control for Silverlight and WPF. The **DataGrid** control has an object model based on Microsoft's **DataGrid** control, which makes it easy for developers to migrate to the ComponentOne grid when they need additional power and features such as built-in filtering, grouping, and much more.

So why add are we adding a new grid to the equation? There are two answers to that question:

1. The **C1FlexGrid** is one of the most popular and powerful grids for WinForms platform. We have a huge number of users who rely on the **C1FlexGrid**'s simple and flexible object model, and on its unique features that include unbound mode, flexible cell merging, multi-cell row and column headers, and so on. Many of these users requested a Silverlight/WPF version of the **C1FlexGrid**, and we are happy to comply.
2. The grids are very different. The **C1FlexGrid** is a simpler, smaller, lighter, grid. The **DataGrid** control is a full-featured control with many features that are not available in the **C1FlexGrid**, including built-in grouping UI, hierarchical support and more. You can pick the best grid depending on your application's requirements.

The **C1FlexGrid** is light in size (the Silverlight version is only about 100k) and also in dependencies (none other than the basic .NET assemblies for the corresponding platform). Despite its small size, the grid does include all the features you would expect from a professional grid and a few more (great speed, excel-style editing, easy styling, multi-cell row and column headers, cell merging, and much more).

The **C1FlexGrid** is available for Silverlight and for WPF. This article describes the control and illustrates by walking you through a comprehensive sample that covers most of the functionality in the control. The sample is called **MainTestApplication**.

There are two versions of the sample, one for Silverlight and one for WPF. They share most of the code, which illustrates how easy it is to develop desktop (WPF) and web (Silverlight) versions of applications using a single code base.

Most pages in the **MainTestApplication** application show two grids side by side. The grid on the left is always a **C1FlexGrid**. The one on the right is the Microsoft grid (**DataGrid** in the Silverlight version **WPFToolkit** grid in the WPF version). The Microsoft grids are shown only for comparison.

Creating the C1FlexGrid

Adding a **C1FlexGrid** control to your application requires the exact same steps as adding any custom control. There's nothing special about the **C1FlexGrid** in this regard. You start by adding a reference to the **C1FlexGrid** assembly to your project, and then add the control using XAML:

```
<UserControl x:Class="MainTestApplication.MainPage"
..
  xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml"
>
<Grid x:Name="LayoutRoot">
  <c1:C1FlexGrid/>
</Grid>
</UserControl>
```

Of course, you can also create the **C1FlexGrid** in code if you prefer:

```
var fg = new Cl.Silverlight.FlexGrid.ClFlexGrid();
LayoutRoot.Children.Add(fg);
```

Populating the grid

Once you have added the grid to your application, you will normally populate it using the **ItemsSource** property (like most other grids). The **ItemsSource** property expects an object that implements the **IEnumerable** interface, but in most cases you will work at a slightly higher level and use an object that implements the **ICollectionView** interface.

The **ICollectionView** interface is the main data-binding interface in Silverlight and WPF (in WinForms, that role was played by the **IBindingList** interface).

ICollectionView is a rich interface. In addition to enumerating the data items, it provides sorting, filtering, paging, grouping, and currency management services.

Silverlight provides a **PagedCollectionView** class that implements the **ICollectionView** interface. The **PagedCollectionView** constructor takes an **IEnumerable** object as a parameter and automatically provides all **ICollectionView** services. WPF provides similar classes, such as **ListCollectionView**.

For example, to display a list of customer objects in a **ClFlexGrid** Silverlight application, you would use code such as this:

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

You could also bind the grid directly to the customer list, of course. But binding to an **ICollectionView** is usually a better idea because it retains a lot of the data configuration for the application, and that can be shared across controls.

If many controls are bound to the same **ICollectionView** object, they will all show the same view. Selecting an item in one control will automatically update the selection on all other controls. Filtering, grouping, or sorting will also be shared by all controls bound to the same view.

The code above causes the grid to scan the data source and automatically generate columns for each public property of the items in the data source. Automatically created columns can be customized using code, you may disable the automatic column generation altogether and create the columns yourself, in code or in XAML.

For example, the XAML below disables the automatic column generation and specifies the columns in XAML instead:

```
<!-- create columns on a ClFlexGrid -->
<fg:ClFlexGrid x:Name="_flexiTunes"
    AutoGenerateColumns="False" >
    <fg:ClFlexGrid.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}" Width="200"
            HorizontalAlignment="Center" />
    </fg:ClFlexGrid.Columns>
</fg:ClFlexGrid>
```

This is similar to the XAML you would use to accomplish the same task using the Microsoft DataGrid or the original ComponentOne DataGrid controls:

```
<!-- create columns on an MSDataGrid (or ClDataGrid) -->
<ms:DataGrid Name="_msSongs"
    AutoGenerateColumns="False" >
    <ms:DataGrid.Columns>
```

```

<ms:DataGridTextColumn Binding="{Binding Name}" Header="Title"
    CanUserReorder="False" Width="300" />
<ms:DataGridTextColumn Binding="{Binding Duration}" />
<ms:DataGridTextColumn Binding="{Binding Size}" />
<ms:DataGridTextColumn Binding="{Binding Rating}" Width="200" />
</ms:DataGrid.Columns>
</ms:DataGrid>

```

As you can see, the syntax is virtually identical. Notice that you can use the binding as an indexer into the grid's **Columns** collection. For example, if you wanted to make the "Rating" column 300 pixels wide using code, you could write this:

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

This syntax should be familiar to **C1FlexGrid** users. It is exactly the same command you would use when working with the **WinForms** version of the control.

Grouping Data

The **ICollectionView** interface includes support for grouping, which means you can easily create hierarchical views of your data. For example, to group the customers in the sample above by country and city you would simply change the code above as follows:

```

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"));
}
_flexGrid.ItemsSource = view;

```

The "using (view.DeferRefresh())" statement is optional. It improves performance by suspending notifications from the data source until all the groups have been set up.

The image below shows the result:

Country	Active	ID	Name	First
Country: Brazil (19 items)				
Active: True (7 items)				
Brazil	<input checked="" type="checkbox"/>	0	Ed Ulam	Ed
Brazil	<input checked="" type="checkbox"/>	10	Rich Stevens	Rich
Brazil	<input checked="" type="checkbox"/>	91	Herb Cole	Herb
Brazil	<input checked="" type="checkbox"/>	275	Fred Ambers	Fred
Brazil	<input checked="" type="checkbox"/>	333	Ted Trask	Ted
Brazil	<input checked="" type="checkbox"/>	481	Andy Orsted	Andy
Brazil	<input checked="" type="checkbox"/>	495	Rich Richards	Rich
Active: False (12 items)				
Brazil	<input type="checkbox"/>	67	Quince Richards	Quince
Brazil	<input type="checkbox"/>	74	Paul Paulson	Paul
Brazil	<input type="checkbox"/>	113	Quince Ulam	Quince
Brazil	<input type="checkbox"/>	122	Steve Quaid	Steve
Brazil	<input type="checkbox"/>	124	Ulrich Heath	Ulrich

The data items are grouped by country and by 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.

If you want to disable grouping at the grid level, set the grid's **GroupRowPosition** property to **GroupRowPosition.None** (other options are **AboveData** and **BelowData**).

The grouping mechanism offered by the **ICollectionView** class is simple yet powerful. Each level of grouping is defined by a **PropertyGroupDescription** object. This object allows you to select the property that should be used for grouping and also a **ValueConverter** that determines how the property value should be used when grouping.

For example, if we wanted to group by country initials instead of by country, we would modify the code above as follows:

```
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();
}
_flexGrid.ItemsSource = view;
```

The **CountryInitialConverter** class implements the **IValueConverter** interface. It simply returns the first letter of the country name causing it to be used for grouping instead of the full country name:

```
// converter used to group countries by their first initial
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();
    }
}
```

After making this small change, customers will be grouped by their country's initial instead of by the full 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	<input type="checkbox"/>	1	Rich Frommer	Rich
Thailand	<input type="checkbox"/>	2	Ted Lehman	Ted
Thailand	<input type="checkbox"/>	23	Steve Cole	Stev
Turkey	<input type="checkbox"/>	79	Ulrich Trask	Ulric
Thailand	<input type="checkbox"/>	126	Dan Bishop	Dan
Turkey	<input type="checkbox"/>	166	Larry Frommer	Larr
Thailand	<input type="checkbox"/>	175	Mark Stevens	Mar
Turkey	<input type="checkbox"/>	177	Oprah Heath	Opr
Thailand	<input type="checkbox"/>	185	Mark Jammers	Mar
Turkey	<input type="checkbox"/>	212	Quince Frommer	Quir

Notice that the group rows display some information about the group they represent (property and value being grouped on, and item count).

You can customize that information by creating a new **IValueConverter** class and assigning it to the grid's **GroupHeaderConverter** property.

For example, the default group header converter (the one that shows the information in the image) is implemented as follows:

```
// 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;
    }
}
```

Aggregating Data

Once you have grouped the data, it is often useful to compute aggregate values for the groups. For example, if you grouped sales data by country or product category, you may want to have the grid display the totals sales for each country and product category.

This is easy to accomplish using the **C1FlexGrid**. Simply set the **GroupAggregate** property on the columns that you want to aggregate, and the grid will automatically calculate and display the aggregates. The aggregates are automatically recalculated when the data changes.

Note that the aggregates are shown in the group header rows. For them to be visible, the grid's **AreGroupHeadersFrozen** property must be set to false.

For example, consider the following grid definition:

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

If you set the grid's **ItemsSource** property to an **ICollectionView** object configured to group the data by “Line”, “Color”, and “Name”, you will get a grid that looks like this:

	Line	Color	Name	Price	Cost	Weight	Volume
▲ Total: (200 items)							
▲ Line: Washers (55 items)							
▲ Color: Green (16 items)							
▲ Price: Medium (1 items)							
	Washers	Green	P 0	68.00	233.00	3.00	1,171.00
▲ Price: Very High (8 items)							
	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)							
	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

The grid shows the groups in a collapsible outline format and automatically displays the number of items in each group. This is similar to the view that would be displayed by the Microsoft DataGrid control.

The **C1FlexGrid** lets you go one step further and display aggregate values for the columns. For example, if you wanted to display totals for the “Price”, “Cost”, “Weight” and “Volume” columns, you would modify the XAML above as follows:

```
<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>
```

This XAML code contains two changes:

- 1) It sets the grid’s **AreGroupHeadersFrozen** to false. This is necessary because the group aggregates are shown in the group header rows, and if the headers are frozen then the aggregates will not be visible.
- 2) It sets the **GroupAggregate** property on several columns to “Sum”. This causes the grid to calculate and display the aggregate in the group header rows. Several aggregates are available, including “Sum”, “Average”, “Count”, “Minimum”, “Maximum”, etc.

After making this change, the grid will look like this:

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

Notice how the group headers now display the aggregate value for the groups. The aggregate values are automatically recalculated when the data changes.

Sorting Data

Like most grids, the **C1FlexGrid** supports sorting. Clicking on a column header will sort the data in ascending or descending order. When the grid is sorted, a triangle is displayed in the corresponding column to indicate the current sort direction.

In addition to the regular sort toggle behavior, the **C1FlexGrid** also allows users to remove the sorting by control-clicking column headers. This removes the sorting applied to the column and causes the data to be displayed in the original order.

Like grouping, sorting is actually performed by the **ICollectionView** used as a data source. The grid simply detects mouse clicks and defers the sorting to the data source object. You can also sort the data directly using code.

For example, the code below sorts the data by name and country:

```
// start clean
view.SortDescriptions.Clear();

// sort by name
view.SortDescriptions.Add(
    new SortDescription("Name", ListSortDirection.Ascending));

// and then by country
view.SortDescriptions.Add(
    new SortDescription("Country", ListSortDirection.Ascending));
```

Note that after this code is invoked, the grid will automatically show the data in the new sort order and will also show the triangle sort indicator in the country column header.

You can disable grid sorting by setting the **AllowSorting** property to false, or disable it for specific columns by setting the **Column.AllowSorting** property to false. You can also prevent the grid from showing the sort triangle by setting the **ShowSort** property to false.

Filtering Data (using ICollectionView)

The **ICollectionView** interface also includes support for filtering data through its **Filter** property. The **Filter** property specifies a method that is called 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*).

The **MainTestApplication** sample included with this document includes a **SearchBox** control that consists of a **TextBox** control where the user types a value to search for and a timer. The timer provides a small delay to allow users to type the values to search for without re-applying the filter after each character.

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

```
_view.Filter = null;
_view.Filter = (object item) =>
{
    var srch = _txtSearch.Text;
    if (string.IsNullOrEmpty(srch))
    {
        return true;
    }
    foreach (PropertyInfo pi in _propertyInfo)
    {
        var value = pi.GetValue(item, null) as string;
        if (value != null &&
            value.IndexOf(srch, StringComparison.OrdinalIgnoreCase) > -1)
        {
            return true;
        }
    }
    return false;
};
```

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

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 were of type "Song" and the properties specified were "Title", "Album", and "Artist", then the function would return true if the string being searched for were 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) will reflect the result of the filter by showing only the items selected by the filter.

Note that filtering and grouping work perfectly well 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 shown was taken when the filter was 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 displayed above the grid. It is automatically updated whenever the list changes, so when the filter is applied the status is updated 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:

```
// 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.

Filtering Data (using C1FlexGridFilter)

The **C1FlexGrid** ships with “extender assemblies” called **C1.Silverlight.FlexGridFilter** (and **C1.WPF.FlexGridFilter**) that provide Excel-style filtering. To use these assemblies, add them to your project and then create a **C1FlexGridFilter** object attached to an existing grid. For example:

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

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

The decision to use extender assemblies instead of adding the functionality directly to the control has two reasons:

- 1) It allows us to keep grid assembly small. Developers can choose which extensions they want to use with each project (additional extender assemblies are currently under development).
- 2) We will provide the source code for the extender assemblies so developers can customize them if they want.

You can also enable filtering in the XAML file where the grid is declared. The syntax for that is shown below:

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

Once filtering is enabled, the grid will display a drop-down icon when the mouse is over the column headers. The drop-down shows an editor that allows users to specify how the data should be filtered on the column. Users may choose between two types of filter:

- 1) **Value filter:** This filter allows users to select the values that should be displayed from a list.
- 2) **Condition filter:** This filter allows users to specify two conditions composed of an operator (greater than, less than, etc) and a parameter. The conditions themselves are combined with an AND or an OR operator.

The image below shows what the filters look like while being edited:

Country	Active	ID	Name	Country I
Pakistan	<input checked="" type="checkbox"/>			5
Philippines	<input checked="" type="checkbox"/>			11
Philippines	<input type="checkbox"/>			11
Pakistan	<input type="checkbox"/>			5
Philippines	<input type="checkbox"/>			11
Indonesia	<input type="checkbox"/>			3
Iran	<input type="checkbox"/>			16
India	<input type="checkbox"/>			1
India	<input type="checkbox"/>			1
Italy	<input type="checkbox"/>	35	Noah Jammers	22

Value Filter: Users create the filter by selecting values from a list.

	Country	Active	ID	Name	✓	Country I
	Pakistan	<input checked="" type="checkbox"/>				5
	Philippines	<input checked="" type="checkbox"/>				11
	Philippines	<input type="checkbox"/>				11
	Pakistan	<input type="checkbox"/>				5
	Philippines	<input type="checkbox"/>				11
	Indonesia	<input type="checkbox"/>				3
	Iran	<input type="checkbox"/>	24	Herb Lehman		16

Show items where the value:

Contains

a

☒ And
☐ Or

(none)

Values >>
Apply
Clear
Cancel

Condition Filter: Users create the filter by setting one or two conditions.

The default filter settings should be adequate for most applications, but you may customize the filter in several ways.

Selecting the filter mode

The filter operates in two modes, determined by the setting of the **UseCollectionView** property.

If you set the filter's **UseCollectionView** to false, rows that do not satisfy the filter are hidden (the filter sets their **Visible** property is set to false). In this mode, the filter has no effect on the row count. This mode can be used in bound and unbound grids.

If you set the filter's **UseCollectionView** property to true, then the filter is applied to the data source instead (using the **ICollectionView.Filter** property). 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 mode can only be used in bound mode.

For example:

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

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

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

Or, in XAML:

```
<c1:C1FlexGrid Name="_flex" >

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

</c1:C1FlexGrid>
```

Customizing the 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** filters.

You may use the **FilterType** property to change this behavior and specify the type of filter that should be enabled 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 don't work with the filters.

For example, columns containing images cannot be filtered **Value** or **Condition** filters. In this case, you would disable the filter by setting the **FilterType** property to **None**.

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

The code below shows how you would accomplish this:

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

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

// 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;
```

Specifying filters in code

In most cases, filters are set by the users. But the **ColumnFilter** class exposes a full object model that allows developers to examine and modify the filter conditions using 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":

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

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

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

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

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

Persisting filters

The **C1FlexGridFilter** class contains a **FilterDefinition** property that gets or sets the current filter state as an XML string. This string can be used to persist the filter state when the user quits the application, so 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. Filter definitions may also be saved and restored to streams using the **SaveFilterDefinition** and **LoadFilterDefinition** methods.

Unbound Mode

The **C1FlexGrid** was designed to work with **ICollectionView** data sources, and to take full advantage of the features it provides.

But it can also be used in unbound mode. If you simply add rows and columns to the grid, you can get or set values in the cells using the familiar indexing notation shown below:

```
// add rows/columns to the unbound grid
for (int i = 0; i < 20; i++)
{
    fg.Columns.Add(new Column());
}
for (int i = 0; i < 500; i++)
{
    fg.Rows.Add(new Row());
}

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

The indexing notation should also be familiar to **C1FlexGrid** users. It is the same notation implemented by the WinForms version of the control. You can specify cells by the row and column indices, by row index and column name, or by row index and **Column** object.

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.

One important difference between the WinForms **C1FlexGrid** control and the Silverlight/WPF version is that in the WinForms version of the control, the indices included fixed rows and columns. In the Silverlight/WPF version, fixed rows and columns are not included in the count.

The diagram below shows the cell indexing scheme used in the **WinForms** version of the grid:

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3

The diagram below shows the new cell indexing scheme used in the **Silverlight/WPF** version of the grid:

	0,0	0,1	0,2
	1,0	1,1	1,2
	2,0	2,1	2,2

The new notation makes indexing easier because the indices match the index of the data items (row zero contains item zero) and the column count matches the number of properties being displayed.

The drawback of course is that a new method is required to access the content of the fixed cells, which are not accessible using the standard indexing scheme. This new method consists of additional properties called **RowHeaders** and **ColumnHeaders**.

These properties return an object of type **GridPanel** which can be seen as a 'sub-grid' with their own set of rows and columns.

For example, you could use this code to customize the row headers:

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

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

// set the width and content of the row headers
for (int c = 0; c < rh.Columns.Count; c++)
{
    // width of this column
    rh.Columns[c].Width = 60;
    for (int r = 0; r < rh.Rows.Count; r++)
    {
        // content of this cell
        rh[r, c] = string.Format("hdr {0},{1}", r, c);
    }
}
```

Notice how the **GridPanel** class exposes **Rows** and **Columns** collections just as the main grid does, and supports the same indexing notation. You can customize and populate the row and column headers using the same object model and techniques you use when working with the content area of the grid (the scrollable part).

Cell Merging

Of course we could not call a grid **C1FlexGrid** if it didn't support cell merging.

The **AllowMerging** property on the grid enables cell merging at the grid level. Once you have enabled merging at the grid level, use the **Row.AllowMerging** and the **Column.AllowMerging** properties to select the specific rows and columns that should be merged.

For example, the code below causes the grid to merge cells that contain the same country:

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

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

That is all it takes to create the grid shown below:

Country ▲	Active	ID	Name	First
▲ Country: M (39 items)				
▲ Active: False (15 items)				
Mexico	<input type="checkbox"/>	9	Ben Myers	Ben
	<input type="checkbox"/>	110	Steve Orsted	Steve
	<input type="checkbox"/>	246	Zeb Myers	Zeb
	<input type="checkbox"/>	257	Steve Heath	Steve
	<input type="checkbox"/>	261	Mark Stevens	Mark
	<input type="checkbox"/>	410	Noah Quaid	Noah
	<input type="checkbox"/>	448	Karl Stevens	Karl
Myanmar	<input type="checkbox"/>	490	Larry Stevens	Larry
	<input type="checkbox"/>	80	Fred Ulam	Fred
	<input type="checkbox"/>	262	Zeb Frommer	Zeb
	<input type="checkbox"/>	314	Karl Ulam	Karl
	<input type="checkbox"/>	324	Ulrich Orsted	Ulrich
	<input type="checkbox"/>	341	Oprah Griswold	Oprah
	<input type="checkbox"/>	427	Noah Neiman	Noah

The default merging behavior consists of merging adjacent cells that have the same content.

You can customize this behavior to create custom merging modes by creating a class that implements the **IMergeManager** interface and assigning an instance of that class to the grid's **MergeManager** property.

The **IMergeManager** interface is very simple (it contains only one method):

```
public interface IMergeManager
{
    CellRange GetMergedRange(
        C1FlexGrid grid, CellType cellType, CellRange range);
}
```

The **GetMergedRange** method takes parameters that specify the owner grid, the type of cell (scrollable, column header, etc.), and the single-cell range that may be merged with adjacent cells. The return value is a **CellRange** object that contains the range passed in and any adjacent cells that should be merged with it.

Note that although the interface is simple, the actual implementation of this method can be fairly complicated. A custom implementation that returns overlapping ranges for example may cause the grid to enter an infinite loop. Also, inefficient implementations will hurt performance.

Multi-cell Row and Column Headers

Most grids support row and column header cells, used to display a header over columns and to indicate the status of the row that contains the selection.

The **C1FlexGrid** takes this concept a little further and supports multi-cell headers. You could for example have two rows of column headers, one showing the year and one showing the quarter. Here's some code that shows how you could set that up:

```
// add an extra column header row
var ch = fg.ColumnHeaders;
ch.Rows.Add(new Row());

// populate the header rows
for (int c = 0; c < ch.Columns.Count; c++)
{
    ch[0, c] = 2009 + c / 4; // row 0: year
```

```

    ch[1, c] = string.Format("Q {0}", c % 4 + 1); // row 1: quarter
}

```

This code would produce a grid that looks like this:

		2009	2009	2009	2009	2010	2010
		Q 1	Q 2	Q 3	Q 4	Q 1	Q 2
hdr 0,0		level 0					
hdr 0,0		level 1					
hdr 1,0		level 2					
hdr 1,0	hdr 3,1	cell [0,0]	cell [0,1]	cell [0,2]	cell [0,3]	cell [0,4]	cell [0,5]
hdr 2,0	hdr 4,1	cell [1,0]	cell [1,1]	cell [1,2]	cell [1,3]	cell [1,4]	cell [1,5]
hdr 2,0	hdr 5,1	cell [2,0]	cell [2,1]	cell [2,2]	cell [2,3]	cell [2,4]	cell [2,5]

Notice the two rows used to display the column headers. You could achieve a similar effect in a traditional grid using column headers with line breaks. The difference becomes apparent when we add cell merging to the top fixed row, so columns that refer to the same year are automatically merged. It only takes two lines of code:

```

// add an extra column header row
var ch = fg.ColumnHeaders;
ch.Rows.Add(new Row());

// populate the header rows
for (int c = 0; c < ch.Columns.Count; c++)
{
    ch[0, c] = 2009 + c / 4; // row 0: year
    ch[1, c] = string.Format("Q {0}", c % 4 + 1); // row 1: quarter
}

// merge the top fixed row
fg.AllowMerging = AllowMerging.All;
ch.Rows[0].AllowMerging = true;

```

The result is displayed in the image below:

		2009				2010	2010
		Q 1	Q 2	Q 3	Q 4	Q 1	Q 2
hdr 0,0		level 0					
hdr 0,0		level 1					
hdr 1,0		level 2					
hdr 1,0	hdr 3,1	cell [0,0]	cell [0,1]	cell [0,2]	cell [0,3]	cell [0,4]	cell [0,5]
hdr 2,0	hdr 4,1	cell [1,0]	cell [1,1]	cell [1,2]	cell [1,3]	cell [1,4]	cell [1,5]
	hdr 5,1	cell [2,0]	cell [2,1]	cell [2,2]	cell [2,3]	cell [2,4]	cell [2,5]
hdr 3,0	hdr 6,1	cell [3,0]	cell [3,1]	cell [3,2]	cell [3,3]	cell [3,4]	cell [3,5]

Notice how cells that refer to the same year are merged in the top fixed row, making for a much clearer display. You can apply the same mechanism to merge row headers, as the image shows.

Selection and Selection Modes

In addition to showing data in a tabular format, most grid controls allow users to select parts of the data using the mouse and the keyboard.

The **C1FlexGrid** has a rich selection model controlled by the **SelectionMode** property. The following selection modes are available:

- **Cell:**
Selection corresponds to a single cell.
- **CellRange:**
Selection corresponds to a cell range (block of adjacent cells).
- **Row:**
Selection corresponds to a single whole row.
- **RowRange:**
Selection corresponds to a set of contiguous rows.
- **ListBox:**
Selection corresponds to an arbitrary set of rows (not necessarily contiguous).

The default **SelectionMode** is **CellRange**, which provides a familiar 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 grid cells.

Regardless of the selection mode, the grid exposes the current selection with the **Selection** property. This property gets or sets the current selection as a **CellRange** object.

Monitoring the Selection

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 outputs information to the console when the selection changes:

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

static string GetClipString(ClFlexGrid fg, 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", fg[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 performed the selection (dragging the mouse up or down while selecting).

You can easily extract a lot of useful information from the **Selection** using the **Rows.GetDataItems** method. This method returns a collection of data items associated with a **CellRange**. 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** method for a grid bound to a collection of **Customer** objects:

```
void _flex_SelectionChanged(object sender, CellRangeEventArgs e)
{
    // get customers in the selected range
    var customers =
        _flex.Rows.GetDataItems(_flex.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());
}
```

Notice how the 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.

Selecting cells and objects using code

The **Selection** property is read-write, so you can easily select cell ranges using code. You can also perform selections using the **Select** method, just like in the **C1FlexGrid** for WinForms. The **Select** method allows you to select cells or ranges, and optionally scroll the new selection into view so the user can see it.

For example, to select the first cell in the grid and make sure it's visible to the user, you could use this code:

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

The selection methods all work based on row and column indices. But you can use them to make selections based on cell content as well. For example, the code below will select the first row that contains a given string in the grid's "Name" column:

```
bool SelectName(string name)
{
    // find row that contains the given string in the "Name" column
    int col = _flexGroup.Columns["Name"].Index;
    int row = FindRow(_flex, name, _flex.Selection.Row, col, true);
    if (row > -1)
    {
        _flex.Select(row, col);
        return true;
    }

    // not found...
    return false;
}
```

The code uses a **FindRow** helper method defined below:

```
// look for a row that contains some text in a specific column
int FindRow(C1FlexGrid flex, string text,
            int startRow, int col, bool wrap)
{
    int count = flex.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 = flex[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 given above implements the functionality provided by the **FindRow** method in the WinForms version of the **C1FlexGrid**. That method is not built into the Silverlight version of the grid in order to keep the control footprint small. The method searches a given 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 it will work 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:

```

var customer = GetSomeCustomer;

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

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

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

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

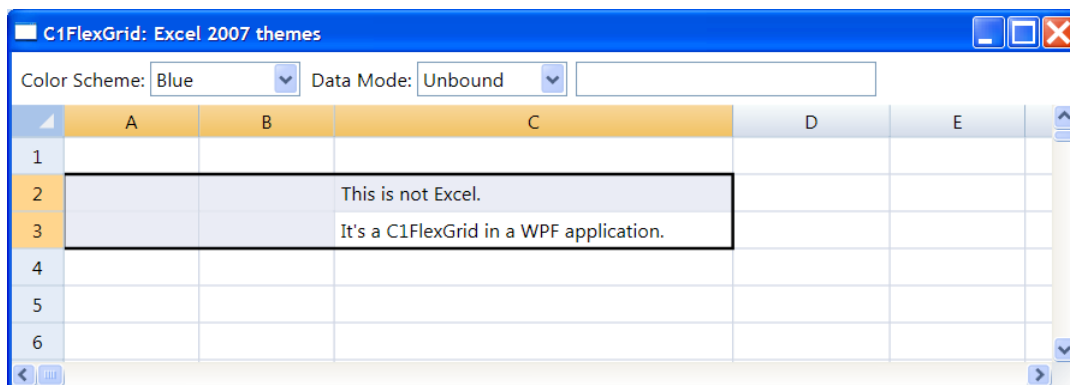
```

Customizing the Selection Display

The **C1FlexGrid** 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 will automatically draw 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 will highlight the headers that correspond to selected cells, making it easy for users to see which rows and columns contain the selection.

Together, these properties make it easy to implement grids that have the familiar Excel look and feel. The image below shows an example:



The **C1FlexGrid** designer has a context menu with options that allow you to select Excel-like preconfigured color schemes (Blue, Silver, Black). You can easily copy the XAML generated by the designer into reusable style resources.

Custom Cells

If you have used any of the Microsoft data grid controls (WinForms, Silverlight, or WPF), you probably know that in order to do any significant customization you have to create custom **Column** objects, override several methods, then add the custom columns to the grid using code. This is not a bad approach (the ComponentOne **DataGrid** control for Silverlight/WPF also follows this model, mainly to keep compatibility with the Microsoft grids).

Custom Cells in code: CellFactory class

The **C1FlexGrid** control uses a very different approach. The grid has a **CellFactory** class that is responsible for creating every cell shown on the grid. To create custom cells, you have to create a class that implements the **ICellFactory** interface and assign this class to the grid's **CellFactory** property. Like custom columns, custom **ICellFactory** classes can be highly specialized and application-specific, or they can be general, reusable, configurable classes. In general, custom **ICellFactory** classes are a lot simpler than custom columns since they deal directly with cells (columns, by contrast, need to deal with the columns themselves and also with the cells and other objects contained in the column).

The **ICellFactory** interface is very simple:

```
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**, is responsible for creating **FrameworkElement** objects used to represent cells. The parameters include the grid that owns the cells, the type of cell to create, and the **CellRange** to be represented by the cells. 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 second method, **CreateCellEditor**, is analogous to the first but creates a cell in edit mode.

The last method, **DisposeCell**, is called after the cell has been removed from the grid. It gives the caller a chance to dispose of any resources associated with the cell object.

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 Silverlight and WPF applications. Without virtualization, a grid would typically have to create several thousand visual elements at the same time, which would ruin performance.

Implementing custom **ICellFactory** classes is fairly easy because you can inherit from the default **CellFactory** class included with the **C1FlexGrid**. 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 sections describe examples of grids that use custom **ICellFactory** classes used to implement functionality we hope you will find interesting. The examples are part of the **MainTestApplication** included with this document, which includes both Silverlight and WPF versions of every sample. The descriptions in this document focus on key implementation points. For details, please refer to the sample application source code.

Custom Cells in XAML: CellTemplate and CellEditingTemplate

If you prefer to create custom cells in XAML instead of writing code, you can do that as well. The **C1FlexGrid** **Column** object 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:

```

<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>

```

Editing Features

Editing is enabled by default. You can disable editing for the whole grid by setting the **IsEditable** property to false. You can also disable editing for specific rows and columns by setting the **IsEditable** property on the row and column objects.

Editing in the **C1FlexGrid** is similar to editing in Excel:

- Pressing F2 or double-clicking a cell puts the grid in *full-edit* mode. In this mode, the cell editor remains active until the user presses Enter, Tab, or Escape, or until he moves the selection with the mouse. In full-edit mode, pressing the cursor keys does not cause the grid to exit edit mode.
- Typing text directly into a cell puts the grid in *quick-edit* mode. The cell editor remains active until the user presses Enter, Tab, or Escape, or any arrow keys. In quick-edit mode, pressing the cursor keys causes the grid to exit edit mode.

While editing, the user can toggle between full and quick modes by pressing the F2 key.

Entering and editing data in a **C1FlexGrid** is easy, efficient, and familiar.

Auto-complete 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:

```

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:

```

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.

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:

```
// 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;
```

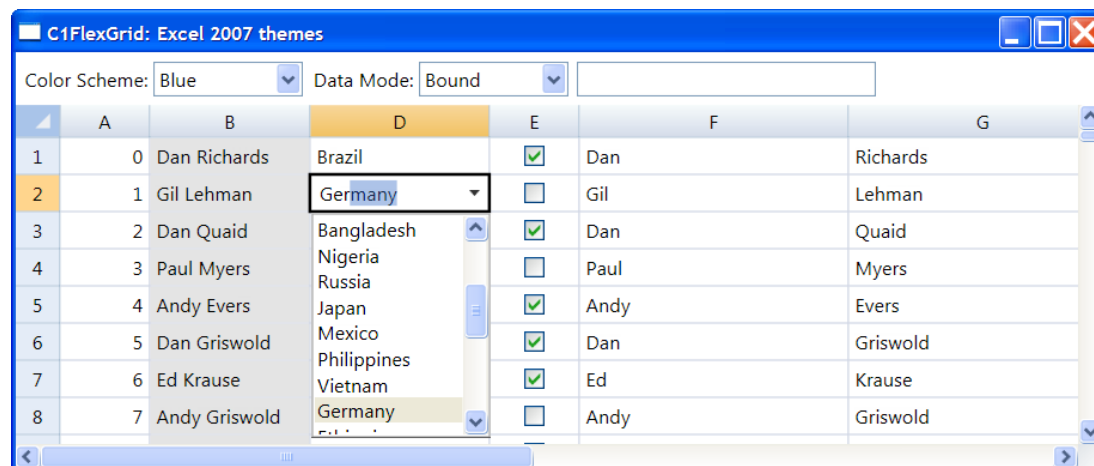
The code starts by building a dictionary that maps country ID values (integers) to country names (strings).

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”).



Using Custom Editors

The **C1FlexGrid** provides two single built-in editors: a checkbox for Boolean values and the **C1FlexComboBox** that extends a regular **TextBox** with autocomplete and list selection as described above.

You can create and use your own editors using the same mechanisms used to create custom cells that were described earlier in this document:

- 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.

Configuring Editors

Whether you are using built-in or custom editors, you can take advantage of the **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:

```
// 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);
}
```

Frozen Rows and Columns

When showing tables with many columns, it is often convenient to ‘freeze’ the first few rows or columns so they remain visible when the grid scrolls.

This can be achieved easily by setting the **Rows.Frozen** and **Columns.Frozen** properties. By default, the **C1FlexGrid** will show black lines between the fixed and scrollable areas of the grid (as Excel does). You can use the **FrozenLinesBrush** property to remove the divider lines or change their color.

The code below shows how you could implement a ‘freeze panes’ command similar to the one in Excel:

```
// freeze/unfreeze panes
void _chkFreezePanels_Click(object sender, RoutedEventArgs e)
{
    if (_chkFreezePanels.IsChecked.Value)
    {
        _flexGroup.Rows.Frozen = _flexGroup.Selection.Row;
        _flexGroup.Columns.Frozen = _flexGroup.Selection.Column;
    }
    else
    {
        _flexGroup.Rows.Frozen = 0;
        _flexGroup.Columns.Frozen = 0;
    }
}
```

When the user checks the **_chkFreezePanels** checkbox, the event handler sets the **Rows.Frozen** and **Columns.Frozen** properties to keep the rows and columns above and to the left of the current selection always in view. The image below shows the effect:

ID	Name	Country	Country I	Active	Fin
0	Vic Bishop	Myanmar	23	<input checked="" type="checkbox"/>	Vic
35	Dan Ulam	Myanmar	23	<input checked="" type="checkbox"/>	Da
77	Ulrich Quaid	Mexico	10	<input checked="" type="checkbox"/>	Ulr
81	Herb Jammers	Mexico	10	<input checked="" type="checkbox"/>	He
117	Vic Trask	Myanmar	23	<input checked="" type="checkbox"/>	Vic
176	Steve Lehman	Mexico	10	<input checked="" type="checkbox"/>	Ste
206	Paul Krause	Mexico	10	<input checked="" type="checkbox"/>	Pai
213	Karl Krause	Myanmar	23	<input checked="" type="checkbox"/>	Kar
218	Fred Paulson	Mexico	10	<input checked="" type="checkbox"/>	Fre

Printing Support

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 that 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 is rendered in actual size, scaled to fit onto a single page, or onto 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, 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 **C1FlexGrid** using the **GetPageImages** onto a print document in either platform.

Printing a C1FlexGrid in Silverlight 4

Printing documents in Silverlight requires the following steps:

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 has been rendered. The code below shows a sample implementation of this mechanism.

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

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

And here is the handler called to print the document:

```
// 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 is responsible for doing all 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.

```
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:

```
<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>
```

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.

```
// 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 is responsible for providing the page images and is conceptually similar to the **PrintPage** event handler used in Silverlight. It is implemented as follows:

```

/// <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:

```

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

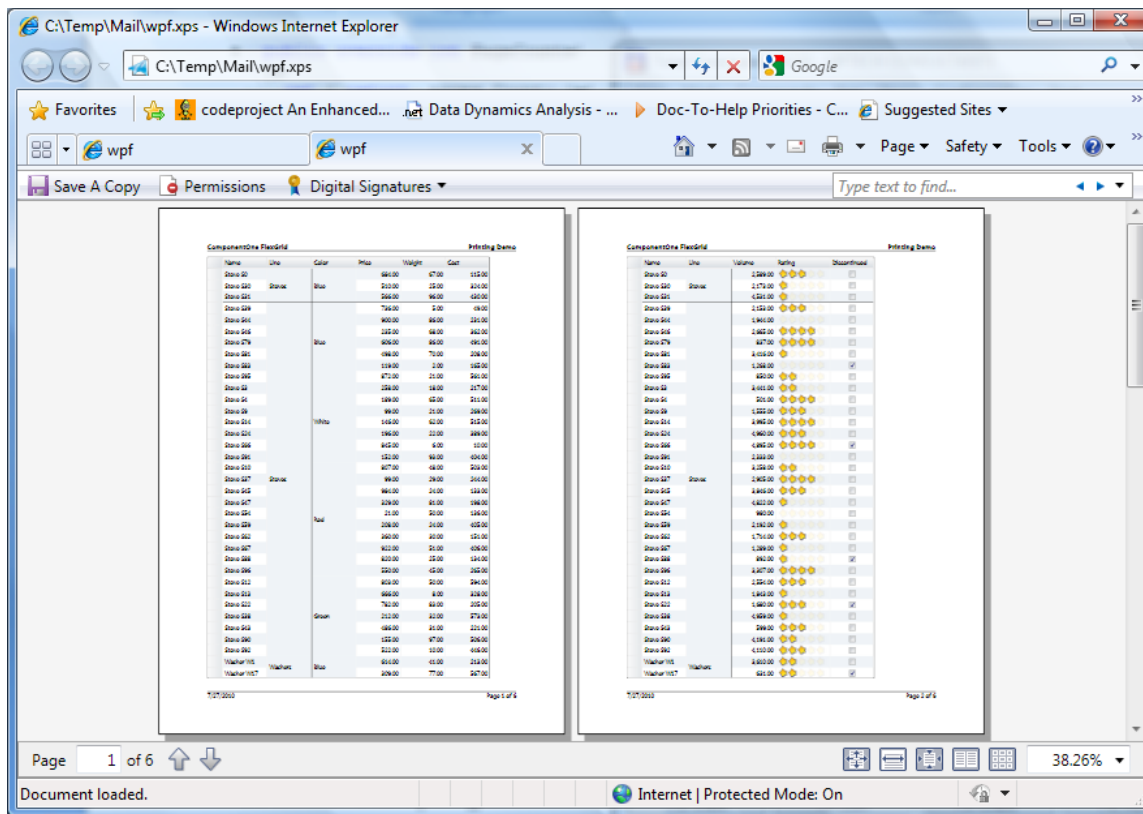
The remaining paginator methods have trivial implementations:

```

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.



Localization Sample

The **FlexGridLocalization** sample shows how you can localize the **FlexGrid** filter to different languages.

The **ComponentOne Studio for Silverlight** product ships with satellite .dlls that provide control localization to several languages. In addition to English, the ComponentOne controls support Arabic, Danish, German, Spanish, Finnish, French, Hebrew, Italian, Japanese, Dutch, Norwegian, Portuguese, Russian, and Swedish.

This sample shows how you can configure your Silverlight application to use the strings in any of these languages or to define your own in case the language you want to use is not on the above list or in case you want to customize the strings.

To localize your Silverlight application, follow these steps:

1. Choose the language(s) you want to support and find the corresponding two-letter **CultureInfo** code. For example, German is "de", Spanish is "es", and French is "fr".
2. Edit your project file and add all the two-letter codes to the "SupportedCultures" section of the file. For example, this sample's project file (**FlexGridLocalization.csproj**) contains the following line:

```
<pre>
    <SupportedCultures>de,es,fr,it,tr</SupportedCultures>
</pre>
```

This line instructs Visual Studio to add to the application's "xap" file the satellite .dlls that contain resources for German, Spanish, French, Italian, and Turkish.

3. If your list includes any languages not on the above list, then you have to create those satellite .dlls yourself. In our example, we specified that the Turkish language will be supported, so we have to add those resources. To do this, follow these steps:

- Add a new folder called "Resources" to your application.
 - Add a new item to the of type "**Resources File**" to the **Resources** folder.
 - Name the new file **C1.Silverlight.FlexGridFilter.tr.resx**. Notice that the name of the file specifies the item that will be localized and also the language that it will contain.
 - Double-click the new file and enter the names and values for all the resources. This sample contains the names of the resources used by the **FlexGridFilter**. Since I don't speak Turkish, I edited the file content so all content strings have a "tr" prefix.
4. Rebuild the application and check that the "xap" file contains all the resources you specified. You can do this by copying the "xap" file with a "zip" extension and checking that the file contains folders with the two-letter codes for the cultures specified.
 5. At this point, the application contains all the resources required, and the only item left is for you to specify the language that you want to use. This can be done in the **Application_Startup** method (in the App.xaml.cs file) or on the page itself. This sample specifies the culture to use in the MainPage.xaml.cs file:

```
<code>
    public MainPage()
    {
        // select the culture (de, it, fr, tr are supported)
        var ci = new System.Globalization.CultureInfo("tr-TR"); // de-DE
etc.

        // this controls the UI strings
        System.Threading.Thread.CurrentThread.CurrentUICulture = ci;

        // this controls number and date formatting (optional in this
sample)
        //System.Threading.Thread.CurrentThread.CurrentCulture = ci;

        // initialize the component
        InitializeComponent();
    }
</code>
```

iTunes Sample

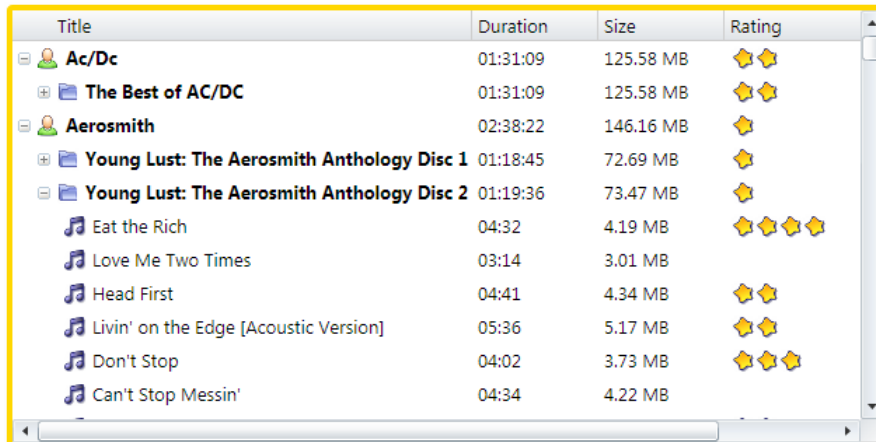
The iTunes sample application displays a library of about 10,000 songs grouped by artist and album. Albums and artists are represented by collapsible node rows. The application includes a search box that allows users to find songs, albums, or artists quickly and easily. Whenever a filter is applied, 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.

Without further ado, this is what the iTunes application looks like (the grid on the left is the **C1FlexGrid**; the one on the right is a standard Microsoft **DataGrid**, shown for comparison only):



Creating the grid

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

```
<!-- 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="#ffeaef4"
    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.

Loading the 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. Loading the data into an **ICollectionView** object is easy. The code looks like this:

```
// 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 would probably replace this with a web service that would load the song catalog from a server. Here is the implementation of the **LoadSongs** method:

```
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 simple .NET format string that will convert 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):

```
// 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/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

You probably noticed that 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 will 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 added three buttons above the grid: *Artists*, *Albums*, and *Songs*. The event handlers for these buttons are implemented as follows:

```
// 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);
}
```

As you can see, 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:

```
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 is very simple. It starts by retrieving the grid's **RowCollection** class implements the same **DeferNotifications** mechanism used by the **ICollectionView** interface. This mechanism is similar to the **BeginUpdate/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

As most typical song catalogs, 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 should be used 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:

```
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, you need to 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:

```
// 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 tackle the most interesting part of the application. Creating an **ICellFactory** object that will create 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.

All these tasks are accomplished by a custom **MusicCellFactory** class that implements the **ICellFactory** interface. To use the custom cell factory, we simply assign it to the grid's **CellFactory** property:

```
_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 is responsible for providing the cell's background and borders, and a **CellRange** object that specifies the cell that should be created (this may be a single cell or multiple cells if they are merged).

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

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

Regular cells are easy to handle. The method simply 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 being represented by the cell.

Cells in group rows are a little more complicated. In our application, group rows are used to 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 has been 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:

```
// 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(
```



```

        ClFlexGrid 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/expand button in addition to the regular cell content.

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

```

// bind cells in group rows
void BindGroupRowCell(ClFlexGrid 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.

```

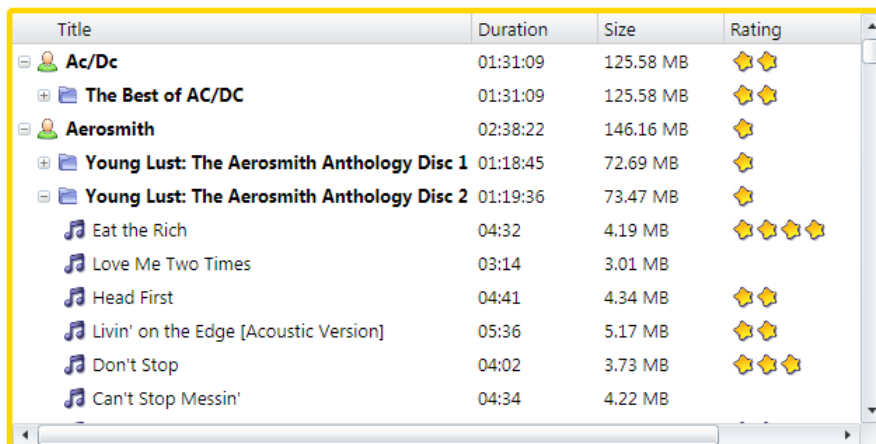
// build fake Song objects to represent groups (artist or album)
Song BuildFakeSong(GroupRow gr)
{
    var gs = gr.GetDataItems().OfType<Song>();
    return new Song()
    {
        Name = gr.Group.Name.ToString(),
        Size = (long)gs.Sum(s => s.Size),
        Duration = (long)gs.Sum(s => s.Duration),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

```

///
/// 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 the ratings will not change, so we don't need any dynamic bindings.

Financial Application Sample

This section describes a sample application inspired in the 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 you run the samples when you have a chance.

☒ OwnerDraw
 ☒ Auto Update
 Update Interval:
 Batch Size:

Financial Info: 3,690 companies selected.

	Symbol	Name	Bid	Ask
	A	Agilent Technologies	15,397.28 (4.0%) ▲	4,488.30 (4.8%) ▲
	AA	Alcoa Inc.	9,357.99 (-4.3%) ▼	9,664.74 (-1.8%) ▼
	AACC	Asset Acceptance Capital Corp.	72,045.64 (3.8%) ▲	4,363.96 (3.8%) ▲
	AAME	Atlantic American Corporation	10,813.49 (1.0%) ▲	14,528.03 (4.1%) ▲
	AANB	Abigail Adams National Bancorp, Inc.	2,561.69 (-2.7%) ▼	13,042.09 (4.9%) ▲
	AAON	AAON, Inc.	5,957.59 (-2.1%) ▼	2,431.11 (-2.1%) ▼
	AAPL	Apple Inc.	636.57 (1.8%) ▲	669.78 (-0.8%) ▼
	AATI	Advanced Analogic Technologies, Inc.	4,014.33 (-3.2%) ▼	3,779.54 (3.4%) ▲
	AAUK	Anglo American plc	4,227.45 (-2.2%) ▼	1,716.33 (3.0%) ▲
	AAWW	Atlas Air Worldwide Holdings	2,963.13 (5.9%) ▲	2,108.33 (4.6%) ▲
	ABAX	ABAXIS, Inc.	5,405.78 (-2.5%) ▼	6,994.02 (2.6%) ▲
	ABBC	Abington Bancorp, Inc.	4,344.10 (-1.1%) ▼	12,315.05 (5.8%) ▲
	ABBI	Abraxis BioScience, Inc.	8,310.55 (1.5%) ▲	13,702.40 (1.4%) ▲
	ABC	AmerisourceBergen Corp.	1,011.35 (0.0%) ▲	615.15 (-3.8%) ▼
	ABCB	Ameris Bancorp	9,730.66 (-3.6%) ▼	2,956.78 (4.7%) ▲

Generating the data

Our financial application uses a data source that tries to simulate 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 should be updated and how many should be updated 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/filtering services for us. Here is the code:

```
// 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:

```
<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 will probably not be interested in looking at all the records all the time, so we need some filter/search mechanism.

A real application would allow 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 above. Instead of selecting specific entries, our users will type something like “bank” or “electric” in the search box to filter the data.

Hooking up our **SearchBox** control to the financial data source is easy:

```
// 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, you will see that 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, you will notice that although the updates are happening, they are hard to understand. There are just too many numbers flashing at random positions on the screen.

We will use custom cells to 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 (is the value 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:

```

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

// custom cell factory definition
public class FinancialCellFactory : CellFactory
{
    public override void CreateCellContent(
        ClFlexGrid 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 being 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 is responsible for displaying 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 will 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** is responsible for gradually changing the control **Background** from its current value to transparent:

```

<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:

```

/// <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:

```
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.

```
// 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:

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

// update text
ticker._txtValue.Text = value.ToString(ticker._format);
ticker._txtChange.Text = string.Format("{0:0.0}% ", change * 100);
```

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.

```
// 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._arrow.Fill = null;
    ticker._txtChange.Foreground = ticker._txtValue.Foreground;
}
```



```

        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 will scale automatically to fit the space available.

```

// 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 changed and the control hasn't just been created, the code flashes the cell by calling the **Storyboard.Begin** method.

```

// 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 should immediately see a much more informative display, with cells flashing as their values change and sparklines providing a quick indication of value trends.

Performance

While running the demo, try changing the *Update Interval* and *Batch Size* parameters to see how those affect the application performance. You may also scroll the grid and select ranges to verify that the grid keeps updating.

It is nice to see that the rich custom cells with additional information, graphics, flashing and sparklines do not affect grid performance perceptibly, either in the Silverlight or WPF versions of the demo.

Differences in C1FlexGrid for Silverlight/WPF

The **C1FlexGrid** for Silverlight/WPF has many things in common with the WinForms version, including much of the object model, but they are not identical. There are two main reasons for the differences:

1) The controls target different platforms

Mouse and keyboard handling for example are different in WinForms and WPF/SL, and so is the imaging model, data binding, dependency properties, etc.

We did map all the properties/methods/events that made sense and 'felt right' in the new platform, but we had to strike a balance between making the control feel like "the WPF/SL version of the C1FlexGrid" (which we wanted) and not "a WinForms grid that works in WPF/SL" (which we did not want).

We worked hard to allow developers to take advantage of new platform features, but not force them to change the way they work. For example, you can change the look and feel of the grid by working with a designer and changing the control template in Microsoft Blend. But you can also do it by implementing a simple CellFactory class in C# or VB.

2) **We are always learning and improving**

The original WinForms C1FlexGrid is a stable, mature control that has been around for 10 years now (or a lot more if you count the ActiveX version that came before that). Over this time, we learned a lot about what features are important and how they should be implemented.

We took all this experience and used it to make the new C1FlexGrid the simplest and most powerful version yet.

Object Model comparison

The tables below summarize the main properties, methods and events in the **C1FlexGrid** for WinForms and the corresponding members in the **C1FlexGrid** for Silverlight/WPF.

C1FlexGrid Properties

WinForms	Silverlight/WPF	Comments
AllowAddNew		Supported at the data source level, no built-in UI.
AllowDelete		Supported at the data source level, no built-in UI.
AllowDragging	AllowDragging	
AllowEditing	IsReadOnly	Renamed for consistency with most other Silverlight/WPF controls.
AllowFiltering		Supported at the data source level, no built-in UI.
AllowFreezing		See the Rows.Frozen and Columns.Frozen properties.
AllowMerging	AllowMerging	
AllowResizing	AllowResizing	
AllowSorting	AllowSorting	
AutoClipboard		See ClipboardCopyMode and ClipboardPasteMode properties, Copy and Paste methods.
AutoGenerateColumns	AutoGenerateColumns	
AutoResize		See AutoSizeRows and AutoSizeColumns methods.
AutoSearch		Not supported.
AutoSearchDelay		Not supported.
BackColor	Background	
BottomRow	ViewRange.BottomRow	
ClipboardCopyMode	ClipboardCopyMode	
Cols	Columns	
DataSource	ItemsSource	
DrawMode		See CellFactory property.
EditOptions		See CellFactory property, PrepareCellForEdit event.
Enabled	IsEnabled	
ExtendLastCol		Set the width of the last column to “*”. Star sizing is a lot more flexible and powerful because it allows you to resize multiple columns and to specify their relative sizes.
FocusRect		Not supported.
HighLight		Not supported.
KeyActionEnter	KeyActionEnter	
KeyActionTab	KeyActionTab	
LeftColumn	ViewRange.LeftColumn	
NewRowWatermark		Not supported

RightColumn	ViewRange.RightColumn	
Rows	Rows	
Scrollbars	HorizontalScrollbarVisibility VerticalScrollbarVisibility	
ScrollOptions		Not supported
ScrollPosition	ScrollPosition	
SelectionMode	SelectionMode	
ShowButtons		Not supported
ShowCellLabels		Not supported
ShowCursor		Not supported
ShowErrors		Not supported
ShowSort	ShowSort	
Subtotal	Column. GroupAggregate	Use the data source's GroupDescriptors collection to group the data; set the GroupAggregate property on individual columns to show aggregates such as sum or average for each group.
SubtotalPosition		Not supported
TopRow	ViewRange.TopRow	
Tree.Indent	TreeIndent	
Tree.Show(level)	CollapseGroupsToLevel (level)	

C1FlexGrid.Column Properties

WinForms	Silverlight/WPF	Comments
AllowDragging	AllowDragging	
AllowEditing	IsReadOnly	
AllowFiltering		Supported at the data source level, no built-in UI.
AllowMerging	AllowMerging	
AllowResizing	AllowResizing	
AllowSorting	AllowSorting	
Caption	Header	
ComboList		Create a ColumnValueConverter object and assign it to the column's ValueConverter property.
DataMap		Create a ColumnValueConverter object and assign it to the column's ValueConverter property.
DataType	DataType	
EditMask		Not supported
Editor		Not supported (the PrepareCellForEdit event provides access to the cell editor being activated)
Filter		Not supported (filtering can be done at the data source level, but the FlexGrid has no UI to support it)
Format	Format	
ImageMap		Not supported
Index	Index	
IsCollapsed	IsCollapsed	
IsVisible	IsVisible	Whether this row is visible (Visible == true && IsCollapsed == false)
Left	Left	
Name	ColumnName	
Right	Right	
Selected	Selected	
UserData	Tag	
Visible	Visible	
Width	Width	The Width property is of type GridLength, and supports star-sizing for dynamic column widths (making the old ExtendLastCol property obsolete)
WidthDisplay	ActualWidth	

C1FlexGrid.Methods

WinForms	Silverlight/WPF	Comments
BeginUpdate() // perform updates EndUpdate()	using (Rows.DeferNotifications()) { // perform updates }	
FindRow(...)		Not supported (very easy to do using code)
FinishEditing(bool cancel)	FinishEditing(bool cancel)	
LoadExcel(string file)		Not supported (in the to-do list)
SaveExcel(string file)		Not supported (in the to-do list)
Select(int row, int col, bool scrollIntoView)	Select(int row, int col, bool scrollIntoView)	
ShowCell(int row, int col)	ScrollIntoView	
Sort(order, int col, int col2)		Not supported (should do at data source level)
StartEditing(row, col)	StartEditing(bool fullEdit, row, col)	The 'fullEdit' parameter determines whether the user can exit edit mode using the arrow keys.

C1FlexGrid.Events

WinForms	Silverlight/WPF	Comments
AfterScroll	ScrollPositionChanged	
AfterSelChange	SelectionChanged	
AfterSort		
AfterSubtotal		Not supported
BeforeScroll	ScrollPositionChanging	
BeforeSelChange	SelectionChanging	
BeforeSubtotal		Not supported
EnterCell	SelectionChanged	
LeaveCell	SelectionChanging	
RowValidated	OnRowEditEnded	
RowValidating	OnRowEditEnding	
SelChange	SelectionChanged	
SetupEditor	PrepareCellForEdit	
StartEdit	BeginningEdit	

We want your feedback!

If you have any feedback about the **C1FlexGrid** control, please send it to bernardoc@componentone.com. I look forward to hearing from you. Thanks!